J'assure aux concours

INFORMATIQUE

MPSI PCSI PTSI TSI TPC
MP PC PT PSI

- Le cours : l'essentiel à retenir
- Les méthodes pas à pas
- Tests de connaissances
- Exercices d'entraînement et d'approfondissement
- Programmation Python
- Corrigés détaillés et expliqués

Nicolas Audfray • Jean-Loup Carré • Stéphane Legros
Vojislav Petrov • Marc Rezzouk

DUNOD

Avec la collaboration scientifique de Isabelle Ecollan, Véronique Guilbert et Cédric Carlier

Conception et création de couverture : Dominique Raboin

11 rue Paul Bert, 92240 Malakoff
www.dunod.com

ISBN 978-2-10-076846-2

Avant-propos

Cet ouvrage s'adresse aux élèves des classes préparatoires scientifiques aux grandes écoles (filières MPSI, PCSI, PTSI, TSI, BCPST, TPC, MP, PC, PSI et PT). Il pourra également intéresser les étudiants préparant le CAPES de Mathématiques Option Informatique.

Le livre est divisé en dix chapitres, couvrant l'intégralité des programmes d'Informatique des deux années de classes préparatoires, les six premiers chapitres contenant le programme de première année et les quatre derniers celui de seconde année. Cet ouvrage est rédigé en utilisant le langage Python (et les modules `numpy` et `scipy`), outils nécessaires et suffisants pour la préparation aux concours. Les chapitres et exercices sont numérotés à partir de zéro pour respecter les conventions de Python.

Chaque chapitre commence par une partie nommée *L'essentiel du cours*. On y présente les points les plus importants du cours à la manière de fiches. La première chose à faire est de connaître cette partie.

On trouve ensuite une partie nommée *Les méthodes à maîtriser*. Elle présente les méthodes en rapport avec le chapitre de cours, illustrée d'un ou plusieurs exemples, et à savoir mettre en pratique.

La plupart des chapitres comprennent un *questionnaire à choix multiples* ou un *questionnaire vrai/faux*. Ceux-ci permettront d'identifier rapidement d'éventuelles lacunes.

Un large choix d'*exercices*, de niveaux variés, est ensuite présenté. Une correction est proposée pour chacun d'entre eux.

Les chapitres proposent ensuite des *Travaux Pratiques*, dont la difficulté est progressive. Ils portent sur des thèmes très variés, englobant des problèmes issus de l'informatique, mais également des mathématiques, de la physique, de la chimie, des sciences de l'ingénieur et de la biologie. Progresser en informatique demande de la pratique, et le lecteur est invité à réaliser ces travaux sur machine, essayer des codes, déchiffrer les messages d'erreurs que pourrait lui envoyer la machine pour progresser dans sa maîtrise syntaxique du langage, effectuer des tests en cherchant davantage les limites de ses programmes qu'une validation superficielle avant de consulter le corrigé. Certaines corrections de ces Travaux Pratiques sont incluses dans le présent ouvrage, et d'autres se trouvent sur le page dédiée à celui-ci sur le site de Dunod.

Nous avons utilisé certains pictogrammes tout au long de cet ouvrage :

 pour attirer l'attention du lecteur sur une remarque spécifique,

 pour attirer l'attention du lecteur sur des pièges potentiels,

 pour apporter au lecteur des précisions sur des points de syntaxe utiles dans le cadre de l'exemple traité mais non essentiels à retenir.

 pour indiquer une question ou un exercice assez difficile.

Table des matières

CHAPITRE 0

Architecture de l'ordinateur

L'essentiel du cours

■ 0 Généralités

Nous sommes aujourd'hui entourés d'ordinateurs de toutes tailles et de toutes puissances. Le premier qui vient à l'esprit est l'ordinateur de bureau, c'est essentiellement de celui-ci dont nous parlerons ici. On peut néanmoins citer les exemples représentés ci-dessous qui constituent une infime partie des ordinateurs présents dans notre environnement.
L'échelle de taille des ordinateurs s'étend de la simple puce d'une taille de quelques millimètres (voire moins) aux super-ordinateurs de calcul de la NASA par exemple.

Quelques exemples d'ordinateurs classés du plus petit au plus grand

Ce qui permet de regrouper ces différents appareils sous le même nom d'ordinateur, est leur fonction : traiter une information numérique.

Définition

Un **ordinateur** est une machine qui traite des données numériques à partir d'instructions organisées en **programmes**.

Définition

Le **programme** est la suite d'instructions (opérations logiques et arithmétiques) compréhensibles par l'ordinateur.

On peut écrire des programmes dans une multitude de langages (C, Fortran, Python, Scilab, Ocaml...), mais le langage réellement exécuté par le processeur de l'ordinateur est un langage qui n'est constitué que de mots binaires : le **langage machine**. Ce langage est constitué d'une suite de

mots qui spécifient les opérations à réaliser sur les bits de la mémoire de l'ordinateur. Le langage standardisé de plus bas niveau [1] est le **langage assembleur**. Il est compréhensible à la lecture par un humain. Il spécifie par exemple quelle valeur numérique l'ordinateur doit placer dans une case mémoire spécifiée.

■ 1 Stockage de l'information

Définition

L'information dans un ordinateur est stockée et transite sous forme de **bits** (*BInary digiTS*). Un bit ne peut prendre que 2 valeurs : 0 ou 1.
Un **multiplet** (*byte* en anglais) est une cellule mémoire élémentaire et est constitué de plusieurs bits (sauf exception 8 bits).
Un **octet** est un multiplet (*byte*) de 8 bits. Les *bytes* ne faisant pas 8 bits étant rares, dans le langage courant, « *byte* » et « octet » sont considérés comme synonymes.

Ces unités sont notées b (bit), B (*byte*) et o (octet). Pour mesurer la capacité de stockage d'un disque dur, on utilise comme unité le kilooctet (ko), le mégaoctet (Mo), le gigaoctet (Go) voire le teraoctet (To). En remarquant que $10^3 = 1000 \approx 1024 = 2^{10}$, certains (mais pas tous) utilisent parfois une définition alternative de ko dans laquelle 1ko = 1024o au lieu de 1ko = 1000o (voir le tableau).

Unité	Valeur standard	Variante binaire
ko	10^3o	2^{10}o = 1024o $\approx 1.02 \times 10^3$o
Mo	10^6o	2^{20}o = 1048576o $\approx 1.05 \times 10^6$o
Go	10^9o	2^{30}o = 1073741824o $\approx 1.74 \times 10^9$o
To	10^{12}o	2^{40}o = 1099511627776o $\approx 1.10 \times 10^{12}$o

Ces deux conventions peuvent engendrer des confusions. Ainsi, si vous achetez un disque dur de 60To, le fabriquant utilise le standard. Mais si votre système d'exploitation utilise la variante binaire, votre disque dur apparaîtra sur votre ordinateur comme faisant « seulement » 54.6To.
Pour lever toute ambiguïté, la Commission électrotechnique internationale [**IEC**] propose de nouveaux noms pour la variante binaire : kibioctet (kio), mébioctet (Mio), gibioctet (Gio) et tébioctet (Tio). Ces noms sont recommandés par différents organismes, mais ne sont pas encore rentrés dans le langage courant.

1. « Bas niveau » signifie proche du langage machine.

Définition

L'information est caractérisée par :

- son **adresse** : valeur numérique désignant l'emplacement de l'information ;
- sa **capacité** (taille mémoire) : exprimée en octets et ses puissances ;
- son **temps d'accès** : durée entre une demande (lecture / écriture) et sa réalisation effective ;
- son **temps de cycle** : durée entre 2 demandes ;
- son débit : nombre d'informations (lecture / écriture) par unité de temps.

La mémoire est organisée en un tableau d'octets, chacun identifié par une adresse. On y accède par l'intermédiaire de bus (ensemble de liaisons physiques) dont le bus d'adresse et le bus de données. Un bus de contrôle permet de définir l'action à réaliser (lecture / écriture).

Définition

Il existe 2 principaux types de mémoires :

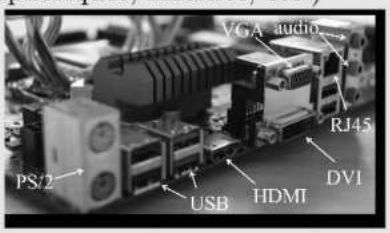
- la **mémoire vive**, volatile ou **RAM** (*Random Access Memory*), dont deux technologies prédominent :
  - statique : SRAM, réalisée à base de bascules ;
  - dynamique : DRAM, réalisée à base de condensateurs. C'est celle que l'on retrouve dans nos PC.
- la **mémoire morte**, non volatile ou **ROM** (*Read Only Memory*), qui se décline sous différentes forme qui ne sont d'ailleurs pas toutes en lecture seule :
  - PROM, mémoire programmable ;
  - EPROM, mémoire programmable effaçable (*Erasable*)
  - EEPROM, mémoire effaçable électriquement (*Electrical*), dont une catégorie connue est la mémoire **Flash** qui est pratique à utiliser mais assez lente. On les utilise dans les BIOS, les cartes électroniques et les cartes mémoire (SD, USB, CompactFlash) et également les disques durs SSD (*Solid-State Drive*).
  - les mémoires de masse magnétiques (disque dur classique) ;

La mémoire morte sert notamment à stocker des données qui ne doivent en aucun cas être effacées (numéro de série, adresse MAC d'une carte réseau, etc.) ainsi que les données ne devant pas être effacées sans une demande expresse (disque dur, carte mémoire, etc.).

2 Architecture de l'ordinateur de bureau

Définition

L'ordinateur est doté de **ports d'entrée-sortie** de communication. Ils permettent via des **protocoles de communication** de réaliser les échanges entre l'ordinateur et son environnement extérieur (utilisateur, périphériques, internet, etc.)



Connectiques d'entrées-sorties d'une carte mère d'ordinateur de bureau

On peut citer quelques-uns des **supports physiques** sur la carte mère représentée ci-dessus (de gauche à droite) :

- port PS/2 pour les souris et claviers (entrées)
- ports USB 2.0 (noir) et USB 3.0 (bleu) (entrée-sortie de périphériques tels que souris, claviers, imprimantes, etc.) ;
- sortie HDMI (*High Definition Multimedia Interface*), pour connecter un écran ou un projecteur en haute définition avec du son ;
- sorties VGA (*Video Graphic Array*) et DVI (*Digital Visual Interface*), pour connecter un écran ou un vidéo projecteur sans le son ;
- port éthernet RJ45 (entrée-sortie réseau). Ce port sert à connecter l'ordinateur par cable au réseau (de l'entreprise, de la maison, etc.). La prise RJ45 permet d'avoir un meilleur débit que le Wifi. Les nouveaux logements qui se contruisent aujourd'hui doivent être équipés de prises RJ45 d'après un arrêté de 2016 [**ARRb**] et d'après la norme [**NFC**] respectée par les constructeurs.
- entrée micro et sorties audio (casque, enceintes, etc.).
- les entrées-sorties non filaires : Bluetooth, Wifi, infrarouge (de moins en moins utilisé), radio, RFID (badge d'accès), NFC (paiement sans contact).

Définition

La **carte mère** accueille l'ensemble des composants de base d'un ordinateur :

- le processeur ou microprocesseur ;
- le (les) disque(s) dur(s) ;
- les barrettes de mémoire RAM ;
- les ports d'entrée-sortie ;
- les différents périphériques (lecteur DVD, lecteur de carte mémoire, etc.)
- la carte réseau ;
- la carte graphique ;
- l'alimentation électrique ;
- les systèmes de refroidissement (ventilateurs).

Après achat d'un ordinateur on peut ajouter des périphériques (nouvelle carte graphique plus puissante, port de communication supplémentaire, etc.), il faut alors les connecter à la carte mère. Leur intégration n'est possible que si les connectiques nécessaires sont présentes sur la carte.

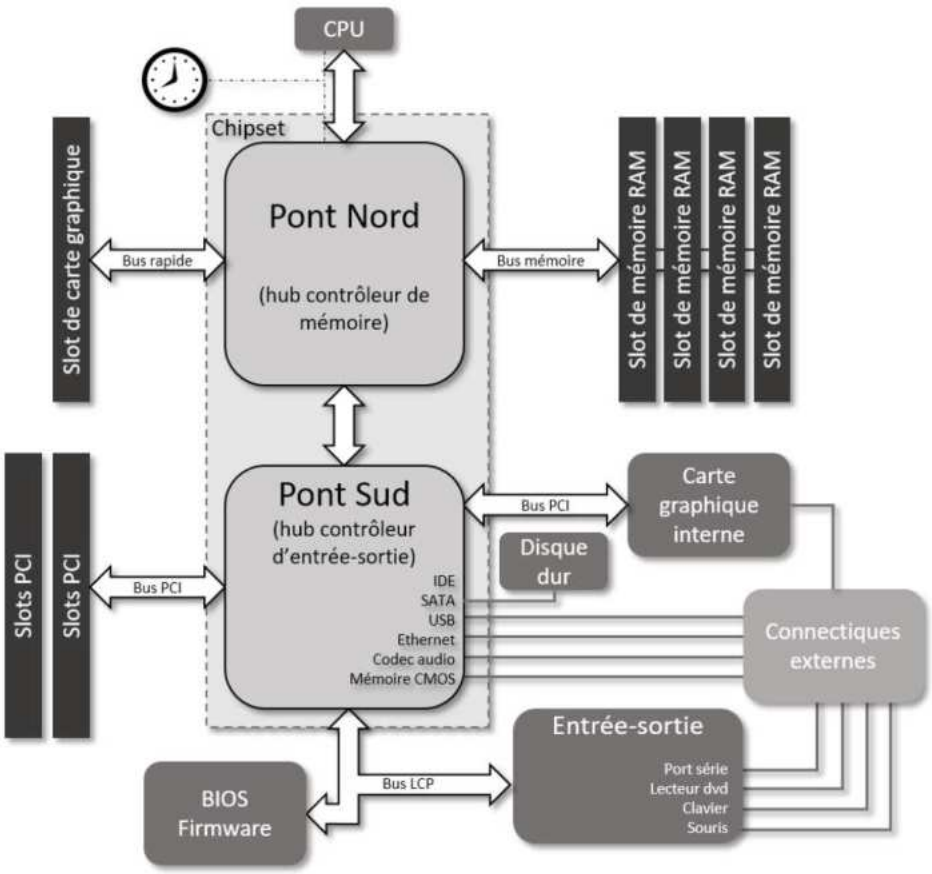
Définition

La carte mère réalise le lien entre ces différents composants par l'intermédiaire de **bus de communication**. On distingue généralement les bus d'instructions et les bus de données.

Définition

Le **chipset** est l'organe permettant de rythmer les échanges entre les différents composants. Il réalise un cadencement rapide avec le **pont nord** (*north bridge*) entre le processeur et la mémoire RAM principalement, et un cadencement plus lent avec les autres composants comme le disque dur.

Cela explique pourquoi l'ordinateur est beaucoup plus lent lorsque toute la mémoire RAM est utilisée et qu'il doit utiliser le disque dur pour réaliser le stockage de données temporaires.



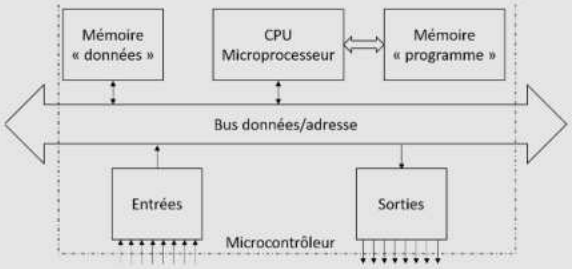
Définition

Le **processeur**, aussi appelé **microprocesseur** ou **CPU** (*Central Processing Unit*), est l'organe principal de l'ordinateur, il doit :

(0) chercher les instructions en mémoire ;
(1) décoder les instructions ;
(2) exécuter les instructions.

Définition

L'architecture **Harvard** est une des architectures historiques de processeurs. Son principe est que les données et le programme sont stockés dans deux mémoires différentes ayant chacune leur propre bus (resp. le bus de données et le bus d'instructions).
Cette architecture a inspiré la conception de nombreux microcontrôleurs embarqués (notamment des processeurs ARM) ; mais aussi certains processeurs d'ordinateurs de bureau, e.g., l'architecture Skylake d'Intel est une variante d'Harvard : les données et le programme sont dans la même unité mémoire, mais disposent de caches séparés ayant chacun leur propre bus.



Microcontrôleur d'architecture Harvard

Définition

Les ressources du processeur sont :

- un compteur ordinal (PC : *Program Counter*), il contient l'adresse de l'instruction ;
- un registre d'instruction, il contient le code de l'instruction ;
- différents registres :
  - les registres généraux (GPR : *General Purpose Register*) ;
  - les registres spécifiques (SFR : *Specific Function Register*) ;
  - les registres mémoire (RAM) ;
- un accumulateur (ou registre de travail) qui contient les résultats en cours ;
- une ou plusieurs **unités arithmétiques et logiques** (**ALU** *Arithmetic and Logic Unit*) constituées de portes logiques permettant de réaliser des opérations booléennes.
- un séquenceur (*scheduler*) qui coordonne les échanges entre les registres, produit et interprète les signaux de contrôle ;
- une **horloge** qui cadence les opérations à haute fréquence ;
- un ou plusieurs **caches** contenant des données récemment utilisées pour y accéder plus rapidement. Ce sont des zones mémoire de quelques kilooctects accessibles très rapidement. Le CPU vérifie d'abord si l'information est présente dans le cache avant de la redemander.

Un processeur peut être constitué d'un ou plusieurs cœurs c'est-à-dire plusieurs processeurs sur une même puce. Les architectures classiques actuelles comptent 4 cœurs (*QuadCore*), on trouve également encore beaucoup de *DualCore* qui contiennent 2 cœurs.
Sur des ordinateurs très puissants dédiés aux calculs avancés (simulation d'écoulements de fluides, calculs pour le lancement de fusées, etc.) on peut atteindre un nombre de cœurs très grand, qui ne cesse d'augmenter avec les évolutions de l'informatique.

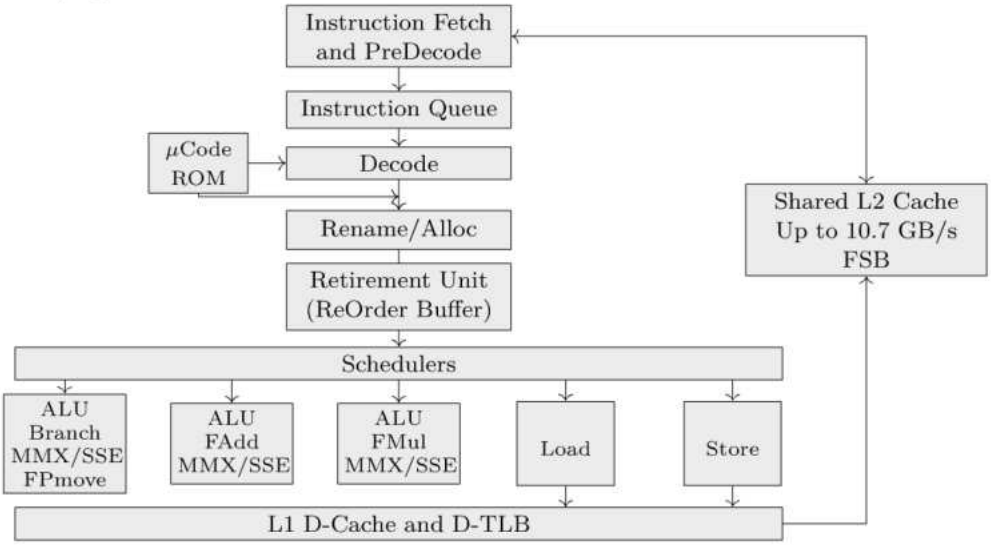
Un processeur cadencé à 3,00 GHz est capable d'exécuter 3.10^9 opérations élémentaires par seconde en théorie.

Exemple d'application

Architecture d'un processeur Intel® Core™ 2

Cet exemple est issu de la documentation Intel® [**INT16**] détaille la structure simplifiée d'un cœur de microprocesseur. Les processeurs actuels utilisent 2, 4 voire 8 cœurs tels que celui-ci. Le fonctionnement est le suivant :

- Chaque instruction est lue et prédécodée (détermination de la longueur de l'instruction par exemple) par le premier bloc (*Instruction Fetch and PreDecode*) qui utilise si besoin des données contenues dans la mémoire cache partagée (*L2*) avec les autres cœurs, puis mise dans la file d'instructions (*Instruction Queue*).
- Exécution du code :
  - déplacement et renommage des micro opérations dans le corps d'exécution (*Rename/Alloc*) ;
  - réordonnancement des micro opérations (*Retirement Unit*) afin de d'exécuter en premier les opérations qui sont prêtes à l'être ;
  - cadencement des ces micro opérations (*Schedulers*).
- Les différentes unités arithmétiques et logiques (*ALU*) exécutent les opérations qui leur sont dédiées (addition, multiplication, etc.), puis les résultats sont stockés dans la mémoire cache (*L1*)



3 Structure logicielle

Définition

Le **chargeur d'amorçage** est le micrologiciel qui se lance automatiquement au démarrage de l'ordinateur. Il exécute les tâches suivantes :

- initialisation des composants de la carte mère, du chipset et de certains périphériques ;
- identification des périphériques internes et externes connectés ;
- démarrage du système d'exploitation.

Ce micrologiciel est stocké dans une mémoire flash afin de ne pas être altérable facilement.

Les chargeurs d'amorçage les plus connus sont le **BIOS** (*Basic Input Output System* et plus récemment l'**UEFI** (*Unified Extensible Firmware Interface*) plus facilement modifiable, notamment pour les mises à jour. Ce dernier offre également une interface graphique haute définition plus agréable.

Définition

Le **système d'exploitation** ou **OS** (*Operating System*) gère les applications et logiciels de l'ordinateur. Il fait notamment le lien entre ces derniers et le matériel (processeur, mémoires, etc.).

Les systèmes d'exploitation les plus connus sont Windows, MacOS, Linux et Chrome OS pour les ordinateurs fixes et portables, Android, I-OS et Windows Phone pour les smartphones.

Le système d'exploitation gère également les multiples utilisateurs en assurant par exemple qu'un utilisateur ayant des droits limités ne puisse installer des logiciels.

Définition

Le **système de fichiers** définit la manière dont sont stockées, organisées et hiérarchisées les données en mémoire. On retrouve notamment un système de dossiers et sous-dossiers qui constituent une arborescence dans l'organisation des fichiers.

Les systèmes de fichiers que l'on retrouve principalement sous Windows sont **FAT32** (*File Allocation Table*) et **NTFS** (*New Technology File System*). Ce dernier est plus récent et permet notamment de stocker des fichiers plus volumineux. On peut formater un disque dur sous Windows dans l'un ou l'autre de ces formats. Cette opération, appelée **formatage**, conditionne le disque conformément au standard choisi et efface toutes les données présentes.

Définition

Les données utilisateur (**fichiers**) sont organisées par le système de fichiers sous la forme d'une arborescence partant d'un **dossier racine** (`C:`, `D:` sous Windows, `/` sous linux par exemple) dans lequel on trouve des **sous-dossiers** qui eux-mêmes contiennent d'autres sous-dossiers.

Cette organisation est gérée par le système de fichiers, elle ne correspond pas à des zones mémoires organisées comme telles. Ainsi si on déplace un dossier, la mémoire utilisée pour stocker les données n'est pas modifiée, seule la structure apparente (arborescence) l'est.

Définition

Le **chemin d'accès** (*path* en anglais) d'un fichier représente le parcours dans l'arborescence de la mémoire pour localiser le fichier (suite des dossiers à ouvrir), par exemple : `C:\WinPython-64bit-3.3.5.0\spyder.exe` pour lancer mon éditeur de code python.

Pour qu'un fichier puisse être exécuté sans préciser son chemin d'accès, celui-ci doit faire partie de la variable *path* qui contient l'ensemble des répertoires spécifiés où le système d'exploitation doit chercher le fichier.

Le système d'exploitation Windows affiche l'arborescence complète du dossier dans lequel on travaille (en mode graphique ou dans l'invite de commande). Sous Linux ou MacOS, dans le terminal, la commande `pwd` (*Path Working Directry*) permet de l'obtenir.

Une fois dans le dossier désiré, on peut obtenir la liste des fichiers et dossiers présents avec la commande `dir` sous Windows et `ls` sous linux et MacOS.

Le système de fichier gère également les **droits d'accès**. En effet pour des raisons de sécurité (de l'ordinateur ou de l'information) un utilisateur standard peut n'avoir accès à un fichier qu'en lecture pour éviter de le modifier.

■ 4 Hasard

En informatique, le hasard est parfois utile (voir notamment les TP 5.2 et 9.0). Pour générer des nombres « aléatoires », il existe principalement deux solutions :

(0) Les PRNG (*PseudoRandom Number Generator*), ce sont des programmes informatiques déterministes qui génèrent des suites de nombres « apparemmment » aléatoires.

(1) Les TRNG (*True Random Number Generator*), ce sont des dispositifs physiques (hardware) qui génèrent du hasard, soit en mesurant un phénomène physique chaotique comme une lampe à lave ou le bruit d'un capteur, soit en utilisant un phénomène quantique réputé aléatoire [**IDQ10**]. Les processeurs Intel actuels contiennent un TRNG [**INT12**]. La thèse [**San09**] référence de nombreux TRNG.

Si les TRNG garantissent un « vrai » hasard, ils sont plus lents et ne génèrent pas des lois uniformes, ce qui incite à utiliser des solutions mixtes : un TRNG est débiaisé et sert à initialiser un PRNG.

QCM sur le cours

(a) Quel est le rôle du processeur dans un ordinateur ?
☐ Il permet de stocker de manière temporaire les données de l'utilisateur.
☐ Il exécute les instructions et les calculs qui lui sont donnés par le système d'exploitation.
☐ Il permet de stocker de manière définitive les données de l'utilisateur.
☐ Il permet de relier les périphériques à l'ordinateur.

(b) Parmi les affirmations suivantes, lesquelles sont vraies ?
☐ En informatique la mémoire est un dispositif électronique qui sert à stocker des informations.
☐ La mémoire du disque dur doit pouvoir fonctionner en mode lecture mais pas en mode écriture pour ne pas être détériorée.
☐ La mémoire du disque dur doit pouvoir fonctionner en mode écriture mais pas en mode lecture pour des raisons de sécurité informatique.
☐ Dans la mémoire vive d'un ordinateur sont stockées définitivement des données importantes.

(c) Parmi les affirmations suivantes, lesquelles sont vraies ?
☐ La mémoire RAM est une mémoire accessible en lecture uniquement.
☐ La mémoire RAM est une mémoire accessible en écriture uniquement.
☐ La mémoire RAM est une mémoire accessible en lecture et en écriture.

(d) À quel endroit de la mémoire le BIOS d'un ordinateur est-il enregistré ?
☐ Dans la mémoire ROM (éventuellement flash).
☐ Dans la mémoire RAM.
☐ Dans la mémoire du disque dur.

(e) La répartition des fichiers dans la mémoire du disque dur suit-elle l'arborescence des dossiers ?
☐ Oui
☐ Non

(f) Parmi les affirmations suivantes, lesquelles sont vraies ?
☐ Les informations stockées dans un même dossier occupent des espaces mémoire voisins.
☐ Les données contenues dans un fichier texte de taille ordinaire sont stockées dans des cases mémoire voisines.
☐ Lorsque l'on déplace un fichier d'un dossier à un autre, on modifie la zone mémoire contenant les informations du fichier.

(g) Quel est le rôle de la carte mère ?
☐ Elle permet de stocker des données dans l'ordinateur.
☐ Elle permet de relier les différents organes de l'ordinateur entre eux.
☐ Elle alimente les périphériques en énergie électrique.
☐ Elle cadence les calculs du microprocesseur.

QCM sur le cours – corrigé

(a) Quel est le rôle du processeur dans un ordinateur ?
- ☐ Il permet de stocker de manière temporaire les données de l'utilisateur.
- ☑ Il exécute les instructions et les calculs qui lui sont donnés par le système d'exploitation.
- ☐ Il permet de stocker de manière définitive les données de l'utilisateur.
- ☐ Il permet de relier les périphériques à l'ordinateur.

(b) Parmi les affirmations suivantes, lesquelles sont vraies ?
- ☑ En informatique la mémoire est un dispositif électronique qui sert à stocker des informations.
- ☐ La mémoire du disque dur doit pouvoir fonctionner en mode lecture mais pas en mode écriture pour ne pas être détériorée.
- ☐ La mémoire du disque dur doit pouvoir fonctionner en mode écriture mais pas en mode lecture pour des raisons de sécurité informatique.
- ☐ Dans la mémoire vive d'un ordinateur sont stockées définitivement des données importantes.

(c) Parmi les affirmations suivantes, lesquelles sont vraies ?
- ☐ La mémoire RAM est une mémoire accessible en lecture uniquement.
- ☐ La mémoire RAM est une mémoire accessible en écriture uniquement.
- ☑ La mémoire RAM est une mémoire accessible en lecture et en écriture.

(d) À quel endroit de la mémoire le BIOS d'un ordinateur est-il enregistré ?
- ☑ Dans la mémoire ROM (éventuellement flash).
- ☐ Dans la mémoire RAM.
- ☐ Dans la mémoire du disque dur.

(e) La répartition des fichiers dans la mémoire du disque dur suit-elle l'arborescence des dossiers ?
- ☐ Oui
- ☑ Non

(f) Parmi les affirmations suivantes, lesquelles sont vraies ?
- ☐ Les informations stockées dans un même dossier occupent des espaces mémoire voisins.
- ☑ Les données contenues dans un fichier texte de taille ordinaire sont stockées dans des cases mémoire voisines.
- ☐ Lorsque l'on déplace un fichier d'un dossier à un autre, on modifie la zone mémoire contenant les informations du fichier.

(g) Quel est le rôle de la carte mère ?
- ☐ Elle permet de stocker des données dans l'ordinateur.
- ☑ Elle permet de relier les différents organes de l'ordinateur entre eux.
- ☐ Elle alimente les périphériques en énergie électrique.
- ☐ Elle cadence les calculs du microprocesseur.

CHAPITRE 1

Programmation

L'essentiel du cours

Dans cet ouvrage nous utilisons le langage Python.

■ 0 Expressions

Définition

Une **expression** est une formule qui renvoie un résultat.

Les opérations usuelles (plus, moins, fois, divisé, puissance, notées `+`, `-`, `*`, `/` et `**`) ainsi que des fonctions (par exemple `abs`) peuvent être utilisées dans les expressions. Exemples :

```
>>> (5 * 9) / 5 + 3**2
18.0

>>> abs(abs(-10)**3 + (-3)**7)
1187
```

Division euclidienne généralisée

Étant donné un réel $b > 0$, pour tout réel a, il existe deux nombres $q \in \mathbb{Z}$ et $r \in [0, b[$ tels que $a = bq + r$. En Python, le quotient q est calculé avec l'opération `//` et le reste r avec `%`.

■ 1 Types de base

Les types de base manipulés en Python sont :

- `int` : les entiers relatifs ;
- `float` : les nombres à virgule aussi appelés nombres flottants ;
- `complex` : les complexes, représentés par une paire de `float` ;
- `bool` : les booléens `True` et `False` ;
- `NoneType` : le type de `None`.

Le calcul est exact sauf pour les types `float` et `complex`. Pour ces deux types, les calculs peuvent mener à des erreurs d'arrondis.

Les opérations de comparaison `==`, `!=`, `<`, `<=`, `>`, `>=` renvoient un booléen. Les opérations usuelles sur les booléens (`and`, `or` et `not`) sont disponibles en Python.

Astuce : `2 < 7 < 5` est une abréviation de `2 < 7 and 7 < 5`.

La valeur `None` est renvoyée par les fonctions n'ayant aucun résultat. Par défaut, il n'est pas affiché dans la console. Par exemple, l'expression `print(2)` affiche `2` puis renvoie `None`.

■ 2 Instructions

Définition

Une instruction est un ordre donné à l'ordinateur, une instruction ne donne pas de valeur.

L'affectation ou assignation est une instruction qui associe une valeur à une variable. Les variables affectées peuvent ensuite être utilisées dans des expressions. Exemple :

```
>>> x = 3

>>> (x + 1) / 2
2.0
```

Ce qui est à gauche et ce qui est à droite du `=` n'ont pas le même rôle.
De plus, `=` et `==` ne doivent pas être confondus.

Le branchement conditionnel. Exemple :

```
if x > 0:
    print(1)
```

Ce code affiche `1` si `x` est strictement positif, et ne fait rien sinon.
Le corps du `if` (ici `print(1)`) est indenté, il n'est exécuté que si la condition du `if` (ici `x>0`) est évaluée à `True`. Il est aussi possible d'ajouter un `else`. Le corps du `else` ne sera exécuté que si la condition du `if` est évaluée à `False`. Le corps du `if` ou du `else` peut contenir plusieurs lignes. Exemple :

```
if x > 0:
    print(1)
else:
    x = 0
    print(2)
```

Définition

Les **boucles** permettent de répéter une suite d'instructions. Il en existe deux types : les boucles `while` et les boucles `for`.

La boucle `while` répète une suite d'instructions tant qu'une condition est vraie.

```
y = 123
n = 0
while y != 0:
    print(y)
    y = y // 10
```

Dans ce code, on affiche `y` puis on le quotiente par `10` tant qu'on n'a pas obtenu zéro. Le code va afficher `123` puis `12` puis `1`.

La boucle `for` permet de répéter une suite d'instructions un nombre prédéterminé de fois.

```
for k in range(5):
    print(k)
```

Le code précédent affiche tous les entiers de 0 inclus à 5 exclu.

■ 3 Séquence

Une séquence est une suite finie de valeurs. Il existe plusieurs types de séquences en Python, dont voici quelques exemples :

- les chaînes de caractères : `"azerty"` ou `'azerty'` ou `'''azerty'''` ou `"""azerty"""` ;
- les tuples : `(1, 7, 58)` ;
- les listes : `[1, 7, 58]`.

Une chaîne de caractères ne contient que des caractères. Un tuple ou une liste peut contenir des données de n'importe quel type (voire des données de types différents). Les opérations suivantes sont communes à toutes les séquences `s` et t :

- lecture de l'élément numéro `k`, `s[k]` (le premier élément porte le numéro 0) ;
- extraction de la sous-séquence de l'élément numéro `i` inclus à l'élément numéro `j` exclu, `s[i:j]` ;
- concaténation de deux séquences, `s + t` ;
- concaténation de `n` copies d'une même séquence, `s * n` ou `n * s`.

Lors de la lecture d'un élément, il est possible d'utiliser des numéros négatifs, `-1` pour le dernier élément, `-2` pour l'avant-dernier...
Lors de l'extraction d'une sous-séquence, il est possible d'ajouter un pas `k`, `s[i:j:k]`. Omettre `i` signifie commencer la sous-séquence au début, omettre `j` signifie finir la sous-séquence à la fin.

Au contraire des chaînes de caractères et des tuples, les listes sont modifiables, et disposent d'opérations supplémentaires :

- modification de l'élément numéro `k` de la liste : `s[k] = x` ;
- ajout d'un élément `x` en fin de liste : `s.append(x)` ;
- suppression d'un élément ou de toute une sous-séquence : `del s[k]` ou `del s[i:j:k]` ;
- suppression du dernier élément (ou du numéro `k`) en renvoyant sa valeur : `s.pop()` ou `s.pop(k)`.

Les opérations `s[i:j:k] = t` et `s.extend(t)` peuvent être utiles à connaître.

L'expression `s + t` ne modifie pas `s` mais crée une nouvelle liste. L'instruction `s += t` modifie `s` et lui ajoute `t` à la fin.

■ 4 Bibliothèques (modules) python

Définition

Les **modules** (ou **bibliothèques** ou encore **librairies**) regroupent des fonctions et constantes utilisables dès lors que le module a été importé dans le code à exécuter, ainsi que des types de données particuliers.

L'importation se fait avec l'instruction `import` qui peut être utilisée de différentes façons :

```
import monmodule_1
import monmodule_2 as modu
from monmodule_3 import *
from monmodule_4 import fonction_4_1, fonction_4_2

# On pourra ainsi à partir de ces différents chargements de bibliothèques
# utiliser les fonctions et variables suivantes :

monmodule_1.fonction_1_1(variable_1)
modu.fonction_2_1(variable_1, variable_2)
monmodule_3.constante_3_1
fonction_4_1(variable_1)
```

Certaines fonctions sont définies dans plusieurs modules. Pour maîtriser l'origine de l'utilisation d'une fonction, il est préférable de l'appeler en spécifiant sa bibliothèque. Par exemple la fonction sinus est définie dans les modules `numpy` et `math`. On l'appellera donc de la manière suivante :

```
import math
import numpy as np  # alias très classiquement utilisé

math.sin(x)  # utilisation de la fonction sinus du module math
np.sin(x)  # utilisation de la fonction sinus du module numpy plus complet : sin(np.array([...])) possible
```

Si on veut éviter ce genre de problèmes on peut importer le module `numpy` en entier par exemple (`from numpy import *`) et uniquement les fonctions utiles de l'autre bibliothèque.

■ 5 Définir sa propre fonction

L'instruction `def` permet de définir une fonction. Exemple :

```
def ma_fonction(arg1, arg2):
    s = arg1 + arg2
    p = arg1 * arg2
    return p % s
```

Une fonction doit avoir :

- zéro, un ou plusieurs arguments (ici `arg1` et `arg2`),
- des instructions à exécuter,
- une valeur de retour : la valeur de l'expression après le `return`.

L'instruction `return` arrête l'exécution de la fonction, même si elle est exécutée au milieu d'une boucle.
L'absence de `return` sous-entend un `return None` à la fin de la fonction.
L'instruction `return None` peut être abrégée en `return`.

L'expression `lambda x : x+2` renvoie la fonction qui à `x` associe `x+2`.

■ 6 Variables locales, variables globales

Chaque instruction de la forme `nom = ...` ou `for nom in ...`, présente dans le code d'une fonction, crée, lors de l'appel de la fonction, une variable locale nommée `nom`. Une telle variable n'est définie que pendant l'exécution de la fonction et peut tout à fait posséder le même nom qu'une variable qui existait avant l'appel, sans qu'il y ait de confusion entre ces deux variables.

On observe également ce comportement si `nom` est le nom d'un des arguments de la fonction ; ainsi, Python accepte le code ci dessous, qui calcule le n-ième nombre harmonique $H_n = \sum_{k=1}^{n} \frac{1}{k}$:

```
def H(n):
    s = 0
    while n!= 0:
        s += 1 / n
        n = n - 1

    return(s)
```

```
>>> n = 100000

>>> H(n)
12.090146129863408

>>> n
100000
```

Si un nom de variable apparaît ailleurs dans le code de la fonction, le compilateur reconnait une variable globale, qui est donc une variable utilisée dans le corps d'une fonction mais non définie dans ce code. Ainsi, les différentes constantes utilisées dans la modélisation d'un problème sont en général des variables initialisées en début de programme, puis utilisées comme variables globales des différentes fonctions. Cependant, on peut parfois souhaiter qu'une fonction modifie une variable globale `nom`. Comme une instruction `nom = ...` crée automatiquement une variable locale `nom`, il faut préciser dans la définition de la fonction que la variable `nom` est globale, ce qui se fait en ajoutant la ligne `global nom` au début du code de la fonction. On trouvera des exemples d'utilisations de variables globales dans l'exercice **8.10**

La distinction entre variables locales et globales s'applique également aux variables d'une sous-fonction : une variable locale d'une fonction peut être déclarée comme variable « globale » d'une de ses sous-fonctions. On utilisera alors le mot clef `nonlocal` au lieu de `global`.

■ 7 Fichiers

Le chemin d'un fichier peut être défini à partir du répertoire courant (chemin relatif), ou à partir de la racine de l'ordinateur (chemin absolu). La fonction `getcwd` du module `os` renvoie le chemin absolu du répertoire courant. Il est également possible de modifier le répertoire courant, en utilisant la fonction `chdir` du module `os` (les exemples qui suivent correspondent à deux systèmes d'exploitation différents) :

```
>>> import os

>>> os.getcwd()
'/Users/OrdiFixe'

>>> os.chdir("Doc/Python")

>>> os.getcwd()
'/Users/OrdiFixe/Doc/Python'
```

```
>>> import os

>>> os.getcwd()
'C:/Users/OrdiFixe'

>>> os.chdir("Doc/Python")

>>> os.getcwd()
'C:\\/OrdiFixe/Doc/Python'
```

Pour ouvrir un fichier, on utilise la fonction `open` en lui indiquant le chemin (relatif ou absolu) du fichier. Nous utiliserons ici les trois options `"w"`, `"r"` et `"a"`, selon que l'on veut écrire dans le fichier, lire le fichier ou ajouter du texte au fichier. L'instruction

```
monfichier = open("exemple.txt", option)
```

crée un flux nommé `monfichier` qui va permettre, selon l'option choisie, d'écrire ou de lire dans le fichier `exemple.txt` du répertoire courant.
L'écriture et la lecture se font ensuite en utilisant les instructions :

```
monfichier.write("chaine de caractere")  # pour écrire à la suite du fichier

ligne = monfichier.readline()  # pour lire la ligne suivante du fichier

texte = monfichier.read()  # pour lire la totalité (restante) du fichier

lignes = monfichier.readlines() # pour obtenir la liste des lignes restantes dans le fichier

morceau = monfichier.read(n) # pour lire les n caractères qui suivent dans le fichier
```

Une fois le travail à effectuer terminé, il ne faut pas oublier de fermer le fichier, à l'aide de la méthode `close` :

```
monfichier.close()
```

C'est à ce moment qu'un fichier ouvert en écriture est physiquement créé. Si un fichier portant le même nom est déjà présent à l'endroit choisi, ce dernier sera écrasé sans avertissement.

L'encodage du fichier (ASCII, UTF8 ou encodage plus exotique) peut être précisé en argument optionnel de la fonction `open` pour que les caractères accentués (ou cyrilliques ou arabes) soient correctement traités.

Le fonctionnement de ces fonctions est explicité dans l'exemple suivant : on crée un fichier contenant quelques lignes de texte (tout d'abord en mode `"w"` puis en mode `"a"`, puis on ouvre ce fichier pour vérifier son contenu. On rappelle que le caractère « passage à la ligne » est `\n`.

```
>>> monfichier = open("Haïku.txt", "w",encoding='utf-8')
>>> monfichier.write(Un vieil étang et\nUne grenouille qui plonge,\n")
45 # nombre de caractères écrits
>>> monfichier.close()
>>> monfichier = open("Haïku.txt", "a")
>>> monfichier.write("Le bruit de l'eau.\n")
19 # nombre de caractères écrits
>>> monfichier.close()
>>> monfichier = open("Haïku.txt", "r",encoding='utf-8')
>>> monfichier.read(5)
'Un vi'
```

```
>>> monfichier.readline()
'eil étang et\n'
>>> monfichier.readlines()
['Une grenouille qui plonge,\n', 'Le bruit de l'eau.\n']
>>> monfichier.close()
```

Il est également possible d'utiliser un fichier ouvert en lecture comme un itérateur, l'itération se faisant alors sur les lignes du fichier :

```
>>> for L in open("Haïku.txt", "r",encoding='utf-8'): print(L)
Un vieil étang et
Une grenouille qui plonge,
Le bruit de l'eau.
```

■ 8 Hasard

Les bibliothèques `random` et `numpy.random` permettent de générer des nombres aléatoires selon diverses lois (uniforme, binomiale, etc.). Elles utilisent toutes les deux Mersenne Twister (un PRNG [1]). Ces bibliothèques sont conçues pour le calcul et pas pour le chiffrement.
Certaines fonctions sont communes à ces bibliothèques, comme `random()` (qui tire un flottant au hasard dans $[0, 1[$) ou `choice(L)` (qui tire au hasard un élément de `L`) ou `shuffle(L)` (qui mélange [2] `L`).

La fonction `randint(a, b)` tire un entier aléatoire (selon une loi uniforme) compris entre `a` inclus et `b` inclus pour la bibliothèque `random`. Mais elle tire un entier aléatoire compris entre `a` inclus et `b` **exclu** pour la bibliothèque `numpy.random`.

La bibliothèque `numpy.random` permet de simuler toutes les lois usuelles (voir la documentation de `numpy`).

```
import numpy.random as rd
# Simule une loi binomiale, somme de n Bernoulli indépendantes de paramètres $p=0.2$.
B = rd.binomial(10, 0.2)
# Simule une loi normale d'espérance (moyenne) 5 et d'écart type 1.
N = rd.normal(5, 1)
P = rd.poisson(7)  # Simule une loi de Poisson de paramètre 7.
```

Les bibliothèques `secret` et `os` (fonctions `getrandom` et `urandom`) génèrent du hasard de meilleure qualité utilisable, par exemple, pour de la cryptographie.

1. cf. chapitre 0 partie 4 page 13.
2. Le mélange « épuise » rapidement les générateurs pseudo-aléatoires (mais pas les TRNG). Pour une discussion sur ce sujet, voir la section 3.4.2 (Random Sampling and Shuffling) de [**Knu13**].

Les méthodes à maîtriser

Méthode 1.0 : Parcourir une liste

Lorsqu'on veut parcourir une liste (ou une chaîne de caractères, ou un tuple) on se demande : « *Est-il utile d'accéder aux indices de cette liste ?* »

- Si oui, on choisit d'itérer la liste par indices : `for i in range(len(L)):`
- Si non, on choisit d'itérer la liste par éléments : `for x in L:`

Lorsqu'on souhaite utiliser simultanément deux éléments successifs d'une liste, le parcours par indices est préférable. On doit faire attention à ne pas accéder à des éléments de la liste qui n'existeraient pas, en s'arrêtant avant le dernier indice le cas échéant.

Exemples d'application

- Calculer la somme des termes d'une liste `L` : l'accès aux indices est inutile, on choisit le parcours `for x in L:`
- Calculer l'espérance[1] d'une variable aléatoire X à valeurs dans $[\![0, n-1]\!]$ si la liste `L` contient les probabilités des évènements $X = i$ (`L[i]` contient $\mathbb{P}(X = i)$) : l'accès aux indices est indispensable, on choisit le parcours `for i in range(len(L)):`
- Tester si une liste L est triée dans l'ordre croissant : on a besoin d'accéder simultanément à un élément et à son successeur lors de notre parcours. On choisit un parcours par indices et on s'arrête à l'avant-dernier élément : `for i in range(len(L)-1):`

Méthode 1.1 : Créer une liste dont on connaît une expression du terme général

Une première solution est d'utiliser un simple `for`.

```
L = []
for k in range(n):
    L.append(f(k))
```

Une solution alternative consiste à utiliser une **liste par compréhension**.

```
L = [f(k) for k in range(n)]
```

Les deux codes font la même chose.

Exemple d'application

`[k**2 for k in range(5)]` renvoie `[0, 1, 4, 9, 16]`.

1. L'espérance $\mathbb{E}(X)$ d'une variable aléatoire X à valeurs dans $[\![0, n-1]\!]$ est définie par $\mathbb{E}(X) = \sum_{i=0}^{i=n-1} i\,\mathbb{P}(X = x_i)$ où $\mathbb{P}(X = i)$ désigne la probabilité que X vaille i.

Méthode 1.2 : Choisir entre une boucle while et une boucle for

Lorsque le nombre d'itérations est connu à l'avance ou lorqu'on connait la suite des valeurs à parcourir, on préfère une boucle `for`.
Lorsque le nombre d'itérations est inconnu, on choisit (par dépit !) une boucle while.

Avec `return`, il est possible d'interrompre une fonction au milieu d'un `for`. Le `for` est donc aussi adapté au cas où on connait un majorant du nombre d'itérations.

Exemples d'application

- Pour trouver l'ensemble des diviseurs d'un entier, on utilisera une boucle `for` : la borne supérieure à tester est connue (n, ou, en rusant un peu, $\sqrt{n}$).
- Pour connaître le plus petit entier n tel que la suite récurrente $u_{n+1} = u_n^2 + 1$ dépasse 10^7 pour un $u_0 > 0$ donné on utilisera une boucle `while`
- Pour tester si un élément est présent dans une liste, une boucle `while` devrait a priori être envisagée, étant donné qu'on interrompt le parcours de cette liste dès lors qu'on trouve l'élément ou lorsqu'on a épuisé tous les éléments. Néanmoins, une boucle `for` peut aussi être utilisée dans le cadre d'une fonction, le `return` ayant comme effet bénéfique d'interrompre le parcours de la liste en cas de succès.

Méthode 1.3 : Calculer une somme

Pour calculer une somme, on :

- définit une variable `s` que l'on initialise à zéro ; cette variable jouera le rôle d'accumulateur ;
- choisit un parcours par élément ou par indice ;
- écrit le corps de boucle, l'instruction `s = s + ...` étant répétée ;
- effectue un éventuel traitement de s en sortie de boucle.

Exemple d'application

Calculer $\dfrac{1}{n}\displaystyle\sum_{i=1}^{n}\dfrac{1}{i^2}$

```
def ma_somme(n):
    s = 0
    for i in range(1, n + 1):
        s = s + 1 / (i**2)
    return s / n
```

Méthode 1.4 : Écrire une boucle while

Pour écrire une boucle `while`, on prend gare à :

- bien définir la condition de poursuite de la boucle `while`. Il arrive que la condition d'arrêt soit plus simple à formuler, il est alors aisé de formuler la condition de poursuite de la boucle en écrivant `not(condition arrêt)` ;
- faire en sorte que la boucle finisse, en n'oubliant pas de modifier la variable sur laquelle porte la condition d'arrêt ;

- réfléchir si les variables en sortie de la boucle sont bien celles qu'on souhaitait avoir, et éventuellement les modifier, ou changer le moment où on modifie la variable sur laquelle porte la condition d'arrêt.

Exemple d'application

Étant donnée une liste $L = [p_0, \ldots, p_{n-1}]$ d'entiers naturels non nuls et un entier naturel N, calculer le plus petit indice k (s'il existe) tel que $N \leqslant \sum_{i=0}^{k} p_i$.

Par convention, k sera égal à -1 si $N > \sum_{i=0}^{n-1} p_i$.

```
def calcul_indice(L, N):
    n = len(L)
    S, k = L[0], 0 # S = L[0] et k=0
    while( S < N and k + 1 < n): # la seconde condition assure l'existence de L[k] à la ligne 7
        k += 1
        S += L[k] # on veut que S = L[0]+...+L[k]
    if S < N: # on est arrivé au bout de la liste sans dépasser N
        return -1
    else: return k # La somme a dépassé N à l'instant k
```

Méthode 1.5 : Tester ses fonctions ; débugger

Lorsqu'on écrit un programme (ou qu'on passe un oral de concours !) il est essentiel de tester – dans la mesure du possible – ses fonctions au fur et à mesure.
Pour ce faire, on appelle depuis la console les fonctions à tester avec des arguments simples et des retours que l'on peut facilement prévoir à la main.
Il ne faut pas hésiter à lancer plusieurs tests, en n'ommettant pas de tester des cas limites (comme une liste vide passée en argument).
Dans le cas où la fonction n'a pas le comportement attendu, si le premier temps – indispensable – de réflexion ne permet pas de trouver l'erreur, on peut soit faire appel aux fonctions de débuggage de l'éditeur employé, soit débugger la fonction à la main en traçant les valeurs des variables critiques, en ajoutant des affichages à l'écran à l'aide de la fonction `print`.

Exemple d'application

Tester si une chaîne de caractères contient la lettre 'e' ou la lettre 'E'.

Un élève produit le code suivant :

```
def test(s):
    for i in range(len(s)):
        if i == 'e' or i == 'E':
            return False
        else:
            return True
```

Une bonne première batterie de tests consiste à vérifier cette fonction avec quelques chaînes simples, e.g., `'info'`, `'ETE'`, `'the'`. La fonction renvoie `False` tout le temps. En plaçant un `print(i)` juste avant le test, on se rend compte qu'on itère sur des entiers et non des caractères. Ceci permet de corriger le premier problème.
La même batterie de tests, sur la fonction corrigée, révèle que certaines chaînes ne sont pas correctement traitées. On peut alors conclure quant à la seconde erreur commise ici (le `else`) et rectifier son programme :

```
def test(s):
    for i in range(len(s)):
        if s[i] == 'e' or s[i] == 'E':
            return False
    return True
```

QCM sur le cours

(a) Parmi les commandes suivantes, lesquelles sont des expressions ?

☐ `3` ☐ `x = 3` ☐ `x == 3` ☐ `return 3` ☐ `for k in range(3):`

(b) Parmi les fonctions suivantes, lesquelles permettent de trouver si le caractère `"e"` est présent dans la chaîne de caractères `s` ?

☐
```
def cherche(s):
    for x in s:
        if x == 'e':
            return True
    return False
```

☐
```
def cherche(s):
    for i in range(len(s)):
        if s[i] == e:
            return True
    return False
```

☐
```
def cherche(s):
    n = 0
    while n < len(s):
        if s[n] == "e":
            return True
    return False
```

☐
```
def cherche(s):
    for x in s:
        if x == "e":
            return True
        else:
            return False
```

(c) Quel(s) programme(s) permet(tent) de construire la liste `L=[0, 0.01, 0.02, ... ,1]` ?

☐
```
L = []
for i in range(101):
    L[i] = i * 0.01
```

☐
```
L = []
for i in range(100):
    L.append(i * 0.01)
```

☐
```
L = []
for i in range(101):
    L.append(i * 0.01)
```

(d) Quel programme est syntaxiquement correct ?

☐
```
if n % 3 == 0:
    print("A")
elif n % 3 == 1:
    print("B")
elif:
    print("C")
```

☐
```
if n % 3 == 0:
    print("A")
else n % 3 == 1:
    print("B")
else n % 3 == 2:
    print("C")
```

☐
```
if n % 3 == 0:
    print("A")
elif n % 3 == 1:
    print("B")
else:
    print("C")
```

(e) Quelle instruction permet de tester si `x` n'est pas dans $\{0, 1\}$?

☐ `if x =! 0 and x =! 1:`
☐ `if not (x == 0 or x == 1):`
☐ `if x != 0 or x != 1:`
☐ `if x != 0 and x != 1:`

(f) Quel(s) programme(s) affiche(nt) 9 ?

☐
```
def f(x):
    s = x**2
    return s
f(3)
print(s)
```

☐
```
def f(x):
    s = x**2
    return s
s = f(3)
print(s)
```

☐
```
def f(x):
    x**2
    return x
print(f(3))
```

QCM sur le cours – corrigé

(a) Parmi les commandes suivantes, lesquelles sont des expressions ?

☑ `3` ☐ `x = 3` ☑ `x == 3` ☐ `return 3` ☐ `for k in range(3):`

(b) Parmi les fonctions suivantes, lesquelles permettent de trouver si le caractère `"e"` est présent dans la chaîne de caractères `s` ?

☑
```
def cherche(s):
    for x in s:
        if x == 'e':
            return True
    return False
```

☐
```
def cherche(s):
    for i in range(len(s)):
        if s[i] == e:
            return True
    return False
```

☐
```
def cherche(s):
    n = 0
    while n < len(s):
        if s[n] == "e":
            return True
    return False
```

☐
```
def cherche(s):
    for x in s:
        if x == "e":
            return True
        else:
            return False
```

(c) Quel(s) programme(s) permet(tent) de construire la liste `L=[0, 0.01, 0.02, ... ,1]` ?

☐
```
L = []
for i in range(101):
    L[i] = i * 0.01
```

☐
```
L = []
for i in range(100):
    L.append(i * 0.01)
```

☑
```
L = []
for i in range(101):
    L.append(i * 0.01)
```

(d) Quel programme est syntaxiquement correct ?

☐
```
if n % 3 == 0:
    print("A")
elif n % 3 == 1:
    print("B")
elif:
    print("C")
```

☐
```
if n % 3 == 0:
    print("A")
else n % 3 == 1:
    print("B")
else n % 3 == 2:
    print("C")
```

☑
```
if n % 3 == 0:
    print("A")
elif n % 3 == 1:
    print("B")
else:
    print("C")
```

(e) Quelle instruction permet de tester si `x` n'est pas dans $\{0, 1\}$?

☐ `if x =! 0 and x =! 1:`
☑ `if not (x == 0 or x == 1):`
☐ `if x != 0 or x != 1:`
☑ `if x != 0 and x != 1:`

(f) Quel(s) programme(s) affiche(nt) 9 ?

☐
```
def f(x):
    s = x**2
    return s
f(3)
print(s)
```

☑
```
def f(x):
    s = x**2
    return s
s = f(3)
print(s)
```

☐
```
def f(x):
    x**2
    return x
print(f(3))
```

Exercices

Applications directes du cours

Exercice 1.0

0. Écrire une fonction `somme_n_premiers_entiers` qui calcule la somme S_n des n premiers entiers naturels non nuls en utilisant une boucle `for`. Cette fonction doit admettre en entrée l'entier n et renvoyer la somme S_n. Tester sur plusieurs valeurs de n pour valider le résultat :

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

1. Écrire une fonction `factorielle_n` qui calcule la valeur $n!$ avec une boucle `while`. Cette fonction doit admettre en entrée l'entier n et renvoyer $n!$.

2. Écrire une fonction qui fait appel à la précédente pour calculer le coefficient binomial $\binom{n}{k} = \dfrac{n!}{k!(n-k)!}$. La fonction admettra en entrée n et k.

Exercice 1.1 Indice du maximum, sous-suites croissantes maximales

On se donne une liste aléatoire de N entiers entre 1 et 100 sous forme d'une liste :

```
>>> import random
>>> N = 15
>>> L = [random.randint(1, 100) for i in range(N)]
>>> print(L)
[57, 72, 59, 26, 71, 81, 48, 77, 60, 40, 86, 89, 15, 5, 7]
>>> max(L) #L'instruction max renvoie un élément maximal de cette liste
89
>>> a = L.index(max(L))  # boîte noire python...
>>> print("élément maximal d'indice {}, valeur: {}".format(a, L[a]))
élément maximal d'indice 11, valeur: 89
```

0. Écrire (sans utiliser la méthode `index`) la fonction `indicemax(L)` qui à partir d'une liste `L` renvoie l'indice d'un élément maximal.

1. Écrire une fonction `estcroissante(L)` qui teste si une liste `L` de nombres est croissante.

2. Une partition d'une liste L en sous-suites croissantes est une suite $(L_0, L_1, \ldots, L_{k-1})$ de listes croissantes telles que $L = L_0 + L_1 + \cdots + L_{k-1}$. La partition maximale de L en sous-suites croissantes est l'unique partition pour laquelle le nombre k de sous-suites est minimale.
Écrire une fonction `listecroissmax` qui, appliquée à une liste `L`, renvoie la liste ordonnée des couples d'indices `(debut, fin exclue)` correspondant à la partition maximale de `L` en sous-suites croissantes.
Par exemple :

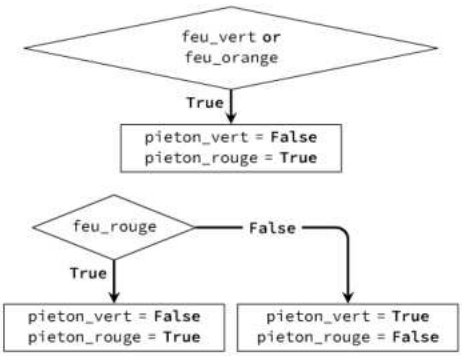
```
>>> L = [1, 2, 4, 5, 7, 2, 4, 5, 1, 0, 2, 5, 6, 8, 9, 3, 4, 5, 4, 9, 1, 5, 8, 9, 3, 0, 4]
>>> Lss = listecroissmax(L)
>>> for deb, fin in Lss:
        print(L[deb:fin], end='')
[1, 2, 4, 5, 7][2, 4, 5][1][0, 2, 5, 6, 8, 9][3, 4, 5][4, 9][1, 5, 8, 9][3][0, 4]
```

3. Écrire une fonction `soussuitemax(L)` qui renvoie une sous-suite croissante de longueur maximale.

Exercice 1.2 Feux tricolores

Cet exercice est basé sur l'automate qui gère les feux de signalisation d'un carrefour. Le comportement réel est loin de celui proposé ici, mais le contexte permet de comprendre rapidement les questions. On ne tient pas compte des différentes temporisations du système réel par exemple.
Les variables `feu_vert`, `feu_orange`, `feu_rouge`, `pieton_vert` et `pieton_rouge` sont de type `bool`.

0. L'organigramme ci-contre représente de manière schématique le branchement conditionnel qui impose que le feu piéton soit au rouge si le feu est vert ou orange. Écrire ce branchement conditionnel en Python en utilisant un `if`.



1. Dans le même esprit, écrire la condition qui impose que le feu piéton soit rouge si le feu n'est pas rouge et que le feu piéton soit vert sinon (voir organigramme ci-contre).

2. Écrire la boucle `while` qui spécifie que tant que le feu piéton est au vert, le feu tricolore doit rester au rouge.

3. Écrire la fonction `passage_feu_vert(fv, fo, fr)` qui allume le feu vert et éteint les autres feux. Cette fonction admet en entrée trois variables booléennes (`feu_vert`, `feu_orange` et `feu_rouge`) et renvoie la nouvelle valeur de ces trois variables.

4. Sur le même principe, écrire les fonctions `passage_feu_orange`, `passage_feu_rouge`, `passage_pieton_vert`, `passage_pieton_rouge`.

5. Proposer une fonction `changement_feux` qui permette de passer à l'état suivant, c'est-à-dire au feu orange si le feu était vert, etc. On exécutera cette fonction manuellement pour chaque changement d'état du feu.

Exercice 1.3 Variance

On considère une liste L de nombres ; dans les formules qui suivront on notera n la longueur de L.
La moyenne d'une liste L de nombres est définie par :

$$m = \frac{1}{n} \sum_{i=0}^{i=n-1} L[i]$$

0. Écrire une fonction `moyenne(L)` qui prend en argument une liste L et qui retourne sa moyenne.

La variance d'une liste L de nombres est définie par :

$$V(L) = \frac{1}{n} \sum_{i=0}^{n-1} (L[i] - m)^2$$

1. Écrire une fonction `variance(L)` qui prend en argument une liste L et qui retourne sa variance.

La littérature spécialisée cite souvent la relation de König, qui facilite les calculs de variance à la main. La relation de König donne la variance d'une liste L de moyenne m :

$$VP(L) = \frac{1}{n} \sum_{i=0}^{i=n-1} \left(L[i]^2 - m^2\right)$$

2. Écrire une fonction `variancekonig(L)` qui prend en argument une liste L et qui retourne sa variance telle que calculée avec la relation de König.

3. Définir les listes $L_1 = [1, 2, 4, 8]$ et $L_2 = [2^{27} + 1, 2^{27} + 2, 2^{27} + 4, 2^{27} + 8]$.
Tester les calculs de variance avec ces listes avec les deux méthodes de calcul précédentes.
Que constate-t-on ? Expliquer le phénomène.

Exercice 1.4 Simulation d'une loi de Poisson

Nous souhaitons simuler une variable aléatoire X qui suit une loi de Poisson de paramètre $\lambda > 0$, c'est-à-dire une variable aléatoire à valeurs dans $\mathbb{N}$ telle que pour tout $k \in \mathbb{N}$, la probabilité de l'évènement $(X = n)$ soit égale à $p_n = e^{-\lambda} \frac{\lambda^n}{n!}$.

Utilisation de la fonction de répartition de la loi de Poisson
Pour tout $n \in \mathbb{N}$, on note $S_n = \sum_{k=0}^{n} p_k$. Si U suit une loi uniforme sur $[0, 1[$, on définit la variable aléatoire X à valeur dans $\mathbb{N}$:

$$X = \sup\{n \in \mathbb{N},\ U \leqslant S_n\}$$

Nous avons :

$$P(X = 0) = P(U \leqslant p_0) = p_0 \text{ et } \forall n \geqslant 1,\ P(X = n) = P(S_{n-1} < U \leqslant S_n) = S_n - S_{n-1} = p_n$$

donc X suit une loi de Poisson de paramètre λ.

0. Écrire une fonction `Poisson` qui, appliquée à un réel $\lambda > 0$, simule la variable X. On commencera par définir U grâce à la fonction `random` du module `numpy.random`, puis on calculera les S_n successifs jusqu'à dépasser la valeur U.

Utilisation d'un produit de variables uniformes et indépendantes
Soient $(U_n)_{n\geqslant 0}$ une suite de variables aléatoires indépendantes et de même loi uniforme sur $[0, 1]$. On note Y le premier instant n où $U_0 U_1 \dots U_n \leqslant e^{-\lambda}$. On peut montrer et nous admettrons que Y est une variable aléatoire presque sûrement définie et qu'elle suit une loi de Poisson de paramètre λ.

1. Écrire une fonction `Poisson_bis` qui, appliquée à un réel $\lambda > 0$, simule la variable Y.

2. Estimer empiriquement les espérances et variances des variables renvoyées par `Poisson` et `Poisson_bis`. Comparer les résultats obtenus aux valeurs théoriques et les commenter.

Exercice 1.5 Retour à l'origine dans une marche aléatoire

Soit $d \in \{1, 2, 3\}$. Une puce se déplace dans $\mathbb{Z}^d$ de façon aléatoire :

- elle se trouve à l'origine O à l'instant $n = 0$;
- entre les instants n et $n + 1$, elle fait un saut de longueur 1 dans l'une des $2d$ directions (Nord-Sud si $d = 1$, Nord-Sud-Est-Ouest quand $d = 2$ et Nord-Sud-Est-Ouest-Haut-Bas quand $d = 3$). On suppose que les sauts sont indépendants et que les $2d$ directions sont équiprobables.

On note M_n la position de la puce à l'instant n : elle sera représentée par une liste de longueur d.

0. Écrire une fonction M qui, quand on l'applique à un couple (d, n), simule cette marche aléatoire et renvoie le point M_n. On utilisera la fonction `randint` du module `numpy.random`.

1. On note T le premier instant où la puce revient à son point de départ O (avec $T = +\infty$ s'il elle n'y revient pas). Écrire une fonction qui simule le calcul de T. Appliquer un grand nombre de

fois cette fonction, avec $d = 1$, $d = 2$ puis $d = 3$. Que peut-on conjecturer quant à la probabilité de l'évènement $T < +\infty$?

Pour aller plus loin

Exercice 1.6

Pour tout entier $n \geqslant 2$, on définit :

$$u_n = \sqrt{1+\sqrt{2+\cdots+\sqrt{n-1+\sqrt{n}}}} \quad \text{et} \quad v_n = \sqrt{1+\sqrt{2+\cdots+\sqrt{n-1+\sqrt{2n+1}}}}.$$

0. Écrire des fonctions `u` et `v` qui, appliquées à un entier $n \geqslant 2$, renvoient respectivement des valeurs approchées de u_n et v_n. Utiliser ces fonctions pour conjecturer le comportement des suites $(u_n)_{n\geqslant 2}$ et $(v_n)_{n\geqslant 2}$.

1. Écrire une fonction `approximation` qui, appliquée à un flottant $\varepsilon > 0$, renvoie une approximation de la limite commune de ces deux suites à ε près.

Exercice 1.7 Méthode des moindres carrés et application en chimie

Nous reprenons dans cet exercice les fonctions de l'exercice 1.3.
La régression linéaire consiste à trouver pour une liste $X = [x_0, \ldots, x_{n-1}]$ et une liste $Y = [y_0, \ldots, y_{n-1}]$ de valeurs réelles, la « meilleure » droite $y = ax + b$ approchant le nuage des points $(x_i, y_i)_{0\leqslant i<n}$. On convient de choisir cette meilleure droite de manière optimale au sens des moindres carrés en ceci qu'elle minimise la somme des écarts au carré entre les y_i et les $ax_i + b$.

On note m_X et m_Y les moyennes respectives des listes X et Y.

La covariance de X et de Y est définie par $\mathrm{Cov}(X,Y) = \dfrac{1}{n}\displaystyle\sum_{i=0}^{n-1}(x_i - m_X)(y_i - m_Y)$.

La variance de X est donc donnée par $V(X) = \mathrm{Cov}(X,X)$.

Les coefficients a et b sont donnés par les formules suivantes :

$$a = \frac{\mathrm{Cov}(X,Y)}{V(X)} \text{ et } b = m_Y - am_X$$

Le coefficient de corrélation linéaire, compris entre -1 et 1, mesure si la régression linéaire est de bonne qualité ou non. Un coefficient proche de 1 en valeur absolue dénote un bon ajustement ; plus ce coefficient est faible en valeur absolue et moins la régression linéaire est adaptée. Il est donné par la formule :

$$\rho = \frac{\mathrm{Cov}(X,Y)}{\sqrt{V(X)V(Y)}}.$$

0. Écrire une fonction `covariance(L1, L2)` qui prend en argument deux listes `L1` et `L2` et qui retourne leur covariance.

1. Écrire une fonction `reglin(L1, L2)` qui prend en argument deux listes `L1` et `L2` et qui renvoie les coefficients a,b et ρ obtenus par les formules de la régression linéaire par la méthode des moindres carrés.

On étudie dans les deux exemples suivants l'évolution de la concentration d'un réactif en fonction du temps pour une réaction chimique de la forme A+B $\rightarrow$ C.

Dans certains cas, on peut écrire $v = -\dfrac{\mathrm{d}[A]}{\mathrm{d}t}$ sous la forme $k[A]^{\alpha}$

On dit alors que la réaction admet un ordre α par rapport au réactif A.

Un exemple classique de réaction pharmaco-chimique d'ordre 0 est l'élimination de l'alcool dans le sang. On note $f(t)$ le taux d'alcool dans le sang d'un sujet après son réveillon du Nouvel An, où il n'a pas eu une consommation responsable, mais s'est arrêté de boire bien avant minuit, qui est associé à $t = 0$.

On fournit le relevé suivant :

t (en h)	$f(t)$
0	2.3
1	2.17
3	1.84
5	1.56
7	1.27
10	0.81

2. Vérifier que la réaction est bien d'ordre 0, i.e. que que $|\rho| \approx 1$

3. Sur une même figure, représenter les points mesurés (on utilisera la fonction `plt.scatter`) et la droite obtenue avec la méthode des moindres carrés.

Certaines réactions sont d'ordre 1, on peut alors à l'aide de la méthode des moindres carrés déterminer une courbe approchée interpolant la concentration en fonction du temps. Cette interpolation ne sera pas toutefois optimale au sens de la méthode des moindres carrés, d'autres méthodes comme celle de Gauss-Newton donnant de meilleures interpolations.

La réaction de Landolt s'écrit :

$$H_2O_2 + 2I^- + 2H^+ \rightarrow I_2 + 2H_2O$$

Dans le cas d'un large excès d'ions I^- et d'ions H^+, la réaction de Landolt est d'ordre 1.

On étudie l'évolution de $[H_2O_2]$ en fonction du temps. On pose $f(t) = [H_2O_2](\mathrm{t})\ 10^{-4}$.

Voici des valeurs expérimentales établies à l'Université d'Aix-Marseille :

t (en s)	$f(t)$
0	296
29	281
58	266
91	251
127	236
170	221
203	207

On peut justifier, à l'aide du cours de mathématiques, que $\ln(f(t)) = At + B$.

4. Construire des listes correspondant à $\ln(f(t))$ et à t

5. En déduire une fonction approchant $f(t)$.

6. Sur une même figure, représenter les points mesurés [1] et la courbe représentative déduite par précédemment.

1. Utiliser la fonction `plot` de la bibliothèque `matplotlib.pyplot`. Cf chapitre 4 page 119.

Exercices type TP SI Banque PT

Exercice 1.8 Traitement de données expérimentales

On considère l'axe linéaire EmeriCC représenté ci-contre. C'est un système qui modélise les axes linéaires industriels nécessitant une grande vitesse de déplacement ainsi qu'une grande précision de position (robot de manutention par exemple).

Lors d'un essai on récupère le fichier de points suivant que vous pourrez récupérer sur la page dédiée à l'ouvrage du site de Dunod. Outre quelques lignes qui donnent le nombre de points, le pas de temps, etc. le fichier de points donne sur chaque ligne le numéro du point, la position du chariot, sa vitesse ainsi que la consigne du variateur électrique.

```
Nom    Position    Vitesse    Consigne variateur
Unite Ox    s    s    s
Unite Oy    mm    mm/s
Delta (Ox)    0,01    0,01    0,01
Delta (Oy)    -1,00    -1,00    -1,00
Nombre de points    100    100    100
0    0,00    0,00    54,00
1    0,68    67,27    54,00
2    1,40    71,31    54,00
3    2,17    76,02    54,00
```

On souhaite dans un premier temps lire les données pour les renseigner dans des listes de flottants (`type float`). On remarque que le fichier est constitué d'un certain nombre de lignes d'en-tête contenant du texte puis de lignes contenant les données acquises.
Nous allons pour cela écrire pas à pas un programme Python permettant d'ouvrir le fichier, lire les données, les enregistrer puis les tracer sous forme de courbe. Dans ce fichier les colonnes sont séparées par des tabulations (`"\t"` pour les chaînes de caractère Python). Les 6 premières lignes constituent l'en-tête et contiennent des informations que l'on doit stocker.

0. Compléter la fonction `lecture_emericc` (remplacer les `...`) qui prend comme argument une chaîne de caractères correspondant au nom du fichier de points, et qui renvoie une liste de listes `entete` contenant l'en-tête du fichier ainsi que les listes de flottants `pts`, `pos`, `vit` et `var` contenant respectivement les données de numéro de point, position, vitesse et consigne variateur.

```
def lecture_emericc(filename):
    monfichier = ...  # ouverture du fichier en mode lecture
    entete = []
    for i in range(6):
        ligne = ...  # lecture de la ligne suivante
        # manipulation sur les chaînes de caractères
        entete.append(ligne.replace(",", ".").strip("\n").split("\t"))
    pts = []
    pos = []
    vit = []
    var = []
    for ligne ...  # itération sur les lignes de monfichier jusqu'à la dernière
        var0, var1, var2, var3 = ligne.replace(
            ",", ".").strip("\n").split("\t")  # manipulation sur les chaînes de caractères
        pts.append(float(var0))
        pos.append(float(var1))
        vit.append(float(var2))
        var.append(float(var3))
```

```
    ...  # fermeture du fichier précédemment ouvert
    return entete, pts, pos, vit, var
```

1. Écrire les lignes de code permettant de récupérer les données du fichier en utilisant l'appel de la fonction ainsi définie.

Le période d'acquisition (pas de temps) est inscrite dans la variable `entete` dans la 4^{e} ligne, 2^{e} colonne.

2. Écrire les lignes de code permettant de stocker dans une liste de flottants `temps` les valeurs du temps pour chaque point.

On souhaite à présent lisser les données de vitesse bruitées par la mesure par une méthode de moyenne mobile, où pour chaque point i on modifie sa vitesse v_i par la moyenne définie ci-dessous faisant intervenir tous les points entre $i - N$ et $i + N$.

$$\bar{v}_n = \frac{1}{2N+1} \cdot \left(v_n + \sum_{k=1}^{N} (v_{n-k} + v_{n+k}) \right)$$

3. Proposer une fonction `lissage_moyenne_mobile(liste, N)` permettant de lisser une liste de valeurs en considérant pour chaque point les N précédents et les N suivants. On pourra supprimer les N premiers et derniers points.
4. Tracer sur un même graphique les données de vitesse brutes, lissées avec $N = 3$ et $N = 5$. On pourra utiliser la syntaxe suivante :

```
plt.figure(1)
plt.plot(temps, vitesse, '-', label="vitesse brute en mm/s")
plt.plot(temps, vitesse_lissee_3, '--', label="vitesse lissée N=3 en mm/s")
...
plt.xlabel('temps en s')
plt.legend()
plt.show()
```

On se rend compte que si l'on souhaite prendre un N grand (si la fréquence d'acquisition est élevée) on tronque assez fortement les données en supprimant les N premières et les N dernières.

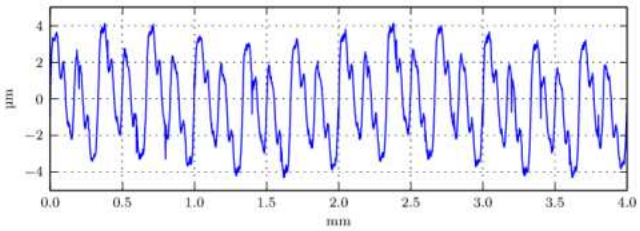
5. Créer la fonction `lissage_moyenne_mobile_affinee(liste, N)` qui permette de ne supprimer aucun point. Pour les premiers et derniers on prendra en compte tous les points disponibles (de $i - k$ à $i + N$ pour le k^{e} point par exemple).
6. Tracer [1] sur un même graphique les données brutes, lissées par la première méthode avec $N = 3$ et lissées par la seconde méthode avec $N = 3$.

Exercice 1.9 Métrologie des états de surface

Nous étudions dans cet exercice le `filtre à phase correcte` défini dans la norme ISO 11562 [**NF 98**] pour l'évaluation des défauts d'états de surfaces des pièces mécaniques, et permettant de filtrer les données périodiques en agissant comme un filtre basse fréquence.
La figure ci-dessous donne le profil anamorphosé (échelles différentes en abscisse et en ordonnée) d'une pièce mécanique, mesuré à l'aide d'un rugosimètre mécanique. On donne également les lignes de code ayant permis de récupérer les données et tracer la figure.

1. Utiliser la fonction `plot` de la bibliothèque `matplotlib.pyplot`. Cf chapitre 4 page 119.

```
x = []
y = []
monfichier = open("mesure_rugosite.txt", "r")
for L in monfichier:
    xx, yy = L.strip("\n").split("\t")
    x.append(float(xx))
    y.append(float(yy))

plt.plot(x, y)
plt.xlabel('mm')
plt.ylabel('µm')
plt.grid()
plt.show()
```



Ce profil contient une partie dite de rugosité, partie haute fréquence et une partie de défaut de forme et d'ondulation, partie basse fréquence. Le traitement de ces données doit permettre de séparer ces défauts. Le filtre défini dans la norme consiste à remplacer chaque point y_k par le point $\bar{y}_k$ tel que :

$$\bar{y}_k = \frac{\sum_{i=1}^{N} s(x_k - x_i) \cdot y_i}{\sum_{i=1}^{N} s(x_k - x_i)}$$

avec $s(x)$ la fonction de pondération suivante :

$$s(x) = \frac{1}{\alpha\lambda_{co}} \cdot e^{-\pi\left(\frac{x}{\alpha\lambda_{co}}\right)^2}$$

avec x la distance entre un point voisin et le point à déplacer et $\alpha = \sqrt{\frac{\ln 2}{\pi}}$.

λ_{co} est la longueur d'onde de coupure de ce filtre.

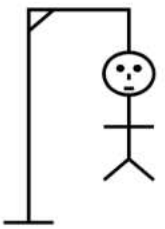
0. Écrire une fonction très simple `s(x, lambda_co)` qui renvoie la valeur de la pondération pour un point situé à une distance x du point considéré. On pourra utiliser les fonctions et constantes mathématiques de la bibliothèque `math` (`math.pi`, `math.log(2)`, par exemple).

1. Proposer une fonction `filtrage_phase_correcte(x, y, lambda_co)` qui prend en argument la liste des abscisses, la liste des ordonnées des points à filtrer et la longueur d'onde de coupure du filtre. La fonction doit retourner une liste des ordonnées de basses fréquences `y_bf` et une liste des ordonnées des hautes fréquences `y_hf`.

2. Tester différentes valeurs de longueurs d'onde de coupure pour ce filtre.

Travaux pratiques

TP 1.0 – Le Pendu



L'objectif de ce TP est de programmer le jeu du PENDU. Le principe du jeu est le suivant : un joueur doit deviner un mot secret. Au début, il ne connait que la longueur du mot (le mot est représenté par une suite de _). Le joueur va proposer successivement des lettres. Si la lettre proposée par le joueur est présente dans le mot, elle est révélée, c'est-à-dire que les _ correspondants sont remplacés. Si le joueur révèle toutes les lettres, il a gagné, si le joueur commet trop d'erreurs[1], il a perdu.

Dans l'exemple ci-contre, la première lettre proposée par le joueur est le E, le E avec accent est donc révélé. Le joueur trouve ensuite le A puis commet une première erreur avec le I. Il propose alors le R, ce qui révèle les deux R du mot secret. Après une seconde erreur (le B), le joueur gagne avec le C.

Lettre proposée	mot révélé	nombre d'erreurs
	_ _ _ _ _ _	0
E	_ _ _ _ _ É	0
A	_ A _ _ É	0
I	_ A _ _ É	1
R	_ A R R É	1
B	_ A R R É	2
C	C A R R É	2

Nous utiliserons 4 variables globales :

- `mot_secret` : une chaîne de caractères (que des minuscules) représentant le mot à deviner.
- `mot_revele` : une liste de caractères représentant le mot découvert, initialement, c'est une liste de « _ ».
- `erreur_max`, `erreur` : deux entiers représentant respectivement le nombre maximal d'erreurs autorisées et le nombre total d'erreurs commises par le joueur

Pour simplifier, nous supposons dans un premier temps que le mot ne contient pas de signe diacritique (accent, cédille, etc.) contrairement à l'exemple (la dernière lettre de CARRÉ est diacritée).

0. Initialisez les variables globales en choisissant arbitrairement un mot secret.

1. Écrire une fonction `affiche(mot)` qui, étant donné une liste de caractères, affiche les caractères contenus dans la liste les uns à la suite des autres.
Par exemple `affiche(['_', 'o', '_', 'p'])` doit afficher `'_o_p'`. L'argument optionnel `end=""` de la fonction `print` peut être utile.

2. Écrire une fonction `revelation(l)` qui remplace par des `l`, dans le mot découvert, tous les _ correspondant à des `l` dans le mot secret.

3. Écrire une fonction `victoire()` qui renvoie `True` si le joueur a gagné, et `False` sinon.

4. Écrire une fonction `jouer_une_lettre()` qui :
— demande au joueur de jouer une lettre (en utilisant `input`),
— révèle la lettre jouée par le joueur avec la fonction `revelation`,
— augmente le nombre d'erreurs si le joueur a joué une lettre qui n'est pas dans le mot secret.

5. Écrire une fonction `main` qui permet de faire une partie complète.

Maintenant, nous considérons que le mot secret peut contenir des diacritiques. Nous voulons que lorsque le joueur joue une lettre, les versions diacritées de la lettre soient aussi révélées. Ainsi, en jouant E, les lettres É, È, Ë, Ê sont révélées.
Dans ce but, nous introduisons la fonction `normaliser` qui retire tous les signes diacritiques d'un texte écrit en lettres latines.

1. Le joueur commet une erreur lorsqu'il propose une lettre qui n'est pas présente dans le mot. Il existe plusieurs variantes du jeu. Ici, nous considèrerons que présenter deux fois la même lettre présente dans le mot ne constitue pas une erreur, par contre, nous considèrerons que proposer deux fois la même lettre absente du mot constitue deux erreurs.

```
from unicodedata import normalize

def normaliser(s):
    return normalize('NFD', s).encode('utf8').decode('ascii', 'ignore')
```

6. Modifier les fonctions précédemment définies pour atteindre cet objectif.

Nous souhaitons à présent que le mot secret soit tiré au hasard dans une liste de mots et non plus fixé au départ. La liste de mots sera stockée dans un fichier texte `vocabulaire.txt` qui contient un mot par ligne.

7. Écrire une fonction `lire(nom)` qui étant donné un nom de fichier `nom` contenant un mot par ligne renvoie la liste des mots de ce fichier.
On pourra utiliser la fonction `open` et les méthodes `.readline` et `.strip`.

8. Modifier la fonction `init` pour qu'elle tire au hasard un mot dans le fichier `vocabulaire.txt`.
On pourra utiliser la fonction `randint` ou la fonction `choice` de la bibliothèque `random`.

9. Modifier la fonction `main` pour qu'elle commence par appeler la fonction `init`.

10. Modifier votre programme pour que les scores (nombre de victoires, nombre de défaites) soient stockés dans un fichier, et qu'après l'affichage de « GAGNE » ou « PERDU » les scores soient affichés.

TP 1.1 – Modèle des urnes d'Ehrenfest

Le modèle des urnes est un modèle « stochastique » introduit en 1907 [**EEA07**] par les époux Ehrenfest pour illustrer certains des paradoxes apparus dans les fondements de la mécanique statistique naissante. Le mathématicien Mark Kac a écrit à son propos qu'il était « ... probablement l'un des modèles les plus instructifs de toute la physique ... ». Il étudie l'évolution d'un système complexe, où les relations de récurrence sont régies par des phénomènes aléatoires.
On considère deux urnes A et B, ainsi que N boules, numérotées de 0 à $N-1$. Initialement, toutes les boules se trouvent dans l'urne A. Le processus stochastique associé consiste à répéter l'opération suivante :

« Choisir au hasard un numéro `i` compris entre 0 et $N-1$, prendre la boule `i`, changer la boule `i` d'urne. »

Nous allons utiliser une représentation informatique de cette situation à l'aide d'une liste `L` de longueur N composée de 0 et de 1 ; à un moment donné si la boule numéro `i` est dans l'urne A alors on a `L[i] = 1` ; si la boule numéro `i` est dans l'urne B alors on a `L[i] = 0`.

Implémentation du modèle

0. Écrire une fonction `initial()` qui renvoie la liste `L0` représentant la situation des boules à l'instant initial (toutes les boules sont dans l'urne A).

1. Écrire une fonction `transition(L)` qui prend en entrée la liste `L` représentant un état des urnes et renvoie la liste `L` après le choix d'un nombre au hasard et transfert de la boule associée.

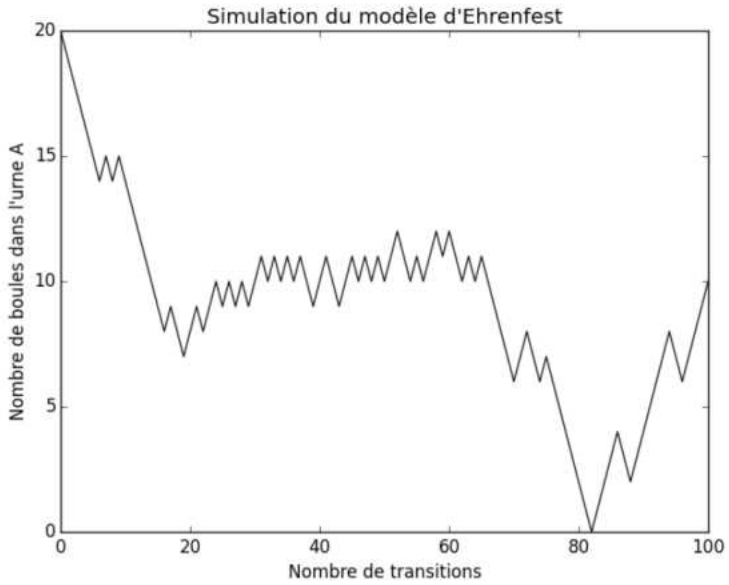
2. Écrire une fonction `nombreA(L)` qui prend en entrée la liste `L` représentant un état des urnes et renvoie le nombre de boules présentes dans l'urne A.

3. Écrire une fonction `evolution(k,N)` qui, à l'aide des fonctions précédentes :
- crée la liste `L0` correspondant à l'état initial ;
- répète `k` fois les transitions en stockant dans une liste `NA` le nombre de boules dans l'urne A après chaque transition (on aura `NA[0]=N` et `len(NA)=k+1`) ;
- renvoie la liste `NA`.

La librairie `matplotlib.pyplot`, décrite au chapitre 4 page 119, permet de tracer des courbes. Dans ce TP les fonctions `plot`, `xlabel`, `ylabel`, `title` de cette bibliothèque nous serons utiles.

4. Donner des instructions permettant d'effectuer une représentation graphique d'une simulation du modèle de Ehrenfest avec $N = 20$ et $k = 100$. Le nombre de transitions effectuées apparaîtra en abscisses, et le nombre de boules dans l'urne A apparaîtra en ordonnées.

On obtiendra par exemple :



Théorème de Kac

Dans ce modèle, on obtient une courbe qui part initialement de N et commence par décroître vers la valeur moyenne $N/2$, comme on pourrait s'y attendre intuitivement pour un système tendant vers l'équilibre.
Mais cette décroissance est irrégulière : il existe des fluctuations autour de la valeur moyenne $N/2$, qui peuvent devenir parfois très importantes.
En particulier, quel que soit le nombre de boules N fini, il existe toujours des retours à l'état initial, pour lesquels toutes les boules sont dans l'urne A. Mais le temps moyen entre deux retours à l'état initial consécutifs croît très rapidement avec N, ce qui ne les rend pas facilement observables.

Formellement, on introduit une suite d'instants $\{t_n\}_{n=1,2,\ldots}$ (finis) pour lesquels toutes les boules reviennent dans l'urne A (par convention, on pose $t_0 = 0$). On peut alors définir une nouvelle suite $\tau_n = t_n - t_{n-1}$ des durées finies entre deux retours à l'état initial consécutifs.

Le théorème de Kac (1947, [**Kac47**]) affirme que cette durée moyenne vaut 2^N

$$\langle\, \tau \,\rangle \;=\; \lim_{p\to\infty}\; \frac{1}{p} \sum_{n=1}^{p} \tau_n = 2^N.$$

5. Écrire une fonction `chercheN(L,N)` qui pour une liste `L` donnée en argument renvoie une liste contenant tous les indices `i` pour lesquels `L[i] == N`.

6. Écrire une fonction `diff(positions)` qui pour une liste `positions` donnée, contenant les indices pour lesquels `positions[i] == N`, renvoie une liste contenant les différences entre deux indices consécutifs de cette liste. Ainsi, `diff([0, 5, 16, 20])` renverra `[5, 11, 4]`
7. Écrire une fonction `differences(L,N)` qui pour une liste `L` donnée, contenant le nombre de boules dans l'urne A à chaque instant, renvoie une liste contenant les différences entre deux indices consécutifs où `L[i] == N` de cette liste.
8. On pose $k =$ `len(L)`. Quel est l'ordre de grandeur, en fonction de k, du nombre d'opérations que réalise la fonction précédente ?
9. Écrire une fonction `moyennerec(L)` qui réalise le calcul de la durée moyenne entre deux retours consécutifs à l'état initial. L'argument est une liste `L` qui contient le nombre de boules dans l'urne A à chaque instant. On se servira des fonctions précédentes.
10. Proposer une démarche qui vous permettrait de vérifier expérimentalement la pertinence du théorème de Kac pour $N = 10$.
11. Expliquer pourquoi la démarche précédente échouerait pour la vérification expérimentale de ce théorème pour $N = 60$.

TP 1.2 – Machine à voter

Une machine à voter est une machine devant laquelle les électeurs passent un par un pour voter, qui compte les votes et qui, à la fin de l'élection donne les scores obtenus par chaque candidat.
Dans ce TP, nous allons écrire une version très simplifiée d'une machine à voter, sans les sécurités usuelles.
Commençons par écrire le code d'une machine à voter honnête, dans le fichier `machine.py`.

0. Écrire une fonction `lire_candidats()` qui lit le fichier `candidats.txt` qui contient un nom de candidat par ligne et qui renvoie la liste des candidats. Un exemple de fichier est donné par la figure 0.

```
Huguette Bouchardeau
Jacques Chirac
Michel Crépeau
Michel Debré
Marie-France Garaud
Valéry Giscard d'Estaing
Arlette Laguiller
Brice Lalonde
Georges Marchais
François Mitterrand
Michel Colucci
```

FIGURE 0. Le fichier `candidats.txt` pour l'élection de 1981.

1. Écrire une fonction `resultats(Lcandidats, Lvotes)` qui :
   - prend en argument la liste des noms des candidats `Lcandidats`
   - prend en argument la liste des votes `Lvotes` : le candidat `Lcandidats[k]` a obtenu `Lvotes[k]` suffrages

- écrit les résultats dans un fichier `csv`, appelé `resultats.csv` : sur chaque ligne, il doit y avoir le nom d'un candidat, puis un point-virgule, puis le nombre de voix du candidat.

```
Huguette Bouchardeau ; 321
```

FIGURE 1. Exemple de ligne possible pour le fichier `resultats.csv`.

2. Écrire une fonction `affiche_candidats(Lcandidats)` qui prend en argument la liste des candidats et qui affiche (avec `print`) la liste des candidats avec leur numéro.

```
0 : Huguette Bouchardeau
1 : Jacques Chirac
2 : Michel Crépeau
```

FIGURE 2. Les premières lignes affichées par `affiche_candidats(Lcandidats)`.

3. Écrire une fonction `unvote(Lcandidats, Lvotes)` qui prend en argument la liste des candidats et la liste des votes (le candidat `Lcandidats[k]` a obtenu, pour le moment, `Lvotes[k]` suffrages) qui :
- affiche la liste des candidats avec la fonction `affiche_candidats`,
- demande à l'électeur de voter pour le numéro de son candidat avec la fonction `input`,
- ajoute une voix au candidat correspondant dans la liste `Lvotes`,
- renvoie `False` si l'électeur a voté `FIN`, et `True` sinon.

La valeur `FIN` est une valeur spéciale qui sert à indiquer que l'élection est terminée. Si la chaîne de caractères entrée par l'électeur n'est pas un numéro de candidat valide (par exemple si elle contient une lettre ou si c'est un nombre trop grand), le vote est considéré comme nul, et aucune voix n'est ajoutée à aucun candidat. Pour vérifier si le vote est nul, la méthode `str.isdecimal` des chaînes de caractères peut être utile.

4. Écrire une fonction `votez(Lcandidats)` qui :
- initialise la liste des votes `Lvotes` avec que des zéros : chaque candidat a, au début de l'élection, zéro voix,
- rappelle la fonction `unvote` jusqu'à ce qu'un électeur vote `FIN`,
- renvoie la liste `Lvotes`.

5. Écrire une fonction `machine_a_voter()` qui combine les fonctions précédentes et permet de voter. La fonction va lire la liste des candidats, demander aux électeurs de voter, puis, quand c'est fini, créer le fichier des résultats.

Dans la suite du TP, nous écrivons des programmes qui s'effacent eux-mêmes. Pour éviter les pertes de données, mettre le fichier `machine.py` en lecture seule.

Considérons le script suivant :

```
for k in range(10):
    print("RIP")
```

FIGURE 3. Le contenu attendu du fichier.

```
def guerrier_glorieux(nom_fichier):
    f = open(nom_fichier, 'w')
    f.close()
```

6. Que se passe-t-il si on applique la fonction `guerrier_glorieux` au fichier `victime.txt` qui contient le texte suivant : `Bonjour` ?

7. Modifiez la fonction `guerrier_glorieux` pour, qu'après avoir été exécutée, le fichier donné en argument contienne le texte de la figure 3.

8. Importez la bibliothèque `sys`. Observez la valeur de `sys.argv`. Quel est son type ?

9. Créer un fichier Python `seppuku.py` qui, lorsqu'il est exécuté, se transforme et devient le programme de la figure 3.

Le script du fichier `seppuku.py` doit fonctionner même si le fichier est renommé, d'où l'intérêt de `sys.argv` pour connaître le nom du fichier.

10. Tester ce que donne deux exécutions successives du fichier `seppuku.py`.

Nous écrivons maintenant le programme d'une machine à voter malhonnête dans un nouveau fichier `machine2.py`. Cette machine va avantager le candidat numéro zéro, puis, pour ne pas se faire prendre, va effacer son propre code pour le remplacer par le code d'une machine à voter honnête.

11. Reprendre le code de `machine.py`, le mettre dans `machine2.py`, puis modifier le code pour qu'à chaque fois qu'un électeur vote, il y ait une chance sur cinq que son vote soit transformé en un vote pour le candidat numéro zéro.

12. Modifier le code de votre machine pour qu'une fois qu'elle a publié les résultats truqués dans le fichier `resultats.txt`, elle efface son code et le remplace par le code d'une machine à voter honnête (c'est-à-dire par le code de `machine.py`).

13. Commentez l'exigence 45 du règlement technique des machines à voter publié par le ministère de l'intérieur en annexe de l'arrêté [**ARRa**].

> Exigence 45 : Les programmes nécessaires à la réalisation de ces fonctions doivent être des modules indépendants et stockés sous forme inaltérable. Les mémoires destinées au stockage des informations propres au scrutin doivent être amovibles, avec verrouillage physique d'accès durant le scrutin, afin d'éviter toute manipulation frauduleuse.

TP 1.3 – Chiffrages de César et de Vigenère. Analyse fréquentielle.

Chiffre de César

En cryptographie, le chiffrement par décalage, aussi connu comme le **chiffre de César**, est une méthode de chiffrement très simple utilisée par Jules César dans ses correspondances secrètes (ce qui explique le nom « chiffre de César »).
Le texte chiffré s'obtient en remplaçant chaque lettre du texte clair original par une lettre à distance fixe, toujours du même côté, dans l'ordre de l'alphabet. Pour les dernières lettres (dans le cas d'un décalage à droite), on reprend au début. Par exemple avec un décalage de 3 vers la droite, A est remplacé par D, B devient E, et ainsi jusqu'à W qui devient Z, puis X devient A etc. Il s'agit d'une permutation circulaire de l'alphabet. La longueur du décalage, 3 dans l'exemple évoqué, constitue la clé du chiffrement qu'il suffit de transmettre au destinataire — s'il sait déjà qu'il s'agit d'un chiffrement de César — pour que celui-ci puisse déchiffrer le message. Dans le cas de l'alphabet latin, le chiffre de César n'a que 26 clés possibles (y compris la clé nulle, qui ne modifie pas le texte).

On utilisera des textes encodés en majuscules, sans ponctuation et sans accent. Par ailleurs on n'écrira pas les espaces dans les textes codés et décodés.
Les objets manipulés seront d'une part des chaînes de caractères, dont on rappelle qu'elles ne sont pas **mutables**, et d'autre part des entiers qui représenteront un code ASCII/Unicode [1] d'un caractère (c'est le même code dans les deux systèmes pour les majuscules latines sans signe diacritique).

0. Écrire une fonction qui prend en entrée un caractère (majuscule) et un décalage et renvoie un caractère.
Cette fonction peut :
- récupérer le code Unicode du caractère (à l'aide de la fonction `ord(caractere)`), appliquer le décalage souhaité, calculer le code Unicode du caractère modulo 26 dans l'intervalle $[65, 90]$ puis retransformer le code Unicode en caractère (s'inspirer du I en cas de problème syntaxique) ;
- ou alors travailler sur l'indice d'un caractère dans la chaîne `s = "AB..Z"`. Cette chaîne est déjà définie dans la bibliothèque `string` sous le nom `ascii_uppercase`.

On suppose que l'on veut chiffrer un texte qui respecte les conventions présentées.
Ainsi « Je programme tous les jours pour m'améliorer » sera écrit sous la forme
« JEPROGRAMMETOUSLESJOURSPOURMAMELIORER ».

1. Écrire une fonction de chiffrage d'une chaîne de caractères. Elle renverra une chaîne de caractères et aura pour en-tête `def ChiffrageCesar(chaine_a_coder, decalage):`
2. Chiffrer le texte suivant avec un décalage de 9, « Oublier les conventions n'amène rien de bon ».
3. Écrire une fonction qui déchiffre une chaîne de caractères qui respecte les conventions présentées.
4. Décoder le texte précédemment encodé avec le même décalage à des fins de vérifications.
Décoder NYHCL avec un décalage de 7 et de 20.

1. Cf. la section 3 du chapitre 2 page 64 sur la représentation des caractères.

Analyse fréquentielle du chiffre de César

Le chiffre de César n'est malheureusement pas très utile pour réellement chiffrer des données. Même lorsque la clé n'est pas connue, tester à la main les 26 possibilités (25 en réalité...) permet à un opérateur humain d'en déduire dans la plupart des cas le texte décodé.
Nous allons étudier une autre méthode ici, qui s'appuie sur une détection automatique de la clé la plus probable de déchiffrement. Les lettres ont des fréquences moyennes d'apparition en français qui, bien que dépendant de la langue utilisée, des corpus de textes étudiés, sont approximativement connues. Voici un tableau résumant celles-ci :

Lettre	f	Lettre	f	Lettre	f	Lettre	f
a	0.0768	h	0.0064	o	0.0534	v	0.0127
b	0.0080	i	0.0723	p	0.0324	w	0.0000
c	0.0332	j	0.0019	q	0.0134	x	0.0054
d	0.0360	k	0.0000	r	0.0681	y	0.0021
e	0.1776	l	0.0589	s	0.0823	z	0.0007
f	0.0106	m	0.0272	t	0.0730		
g	0.0110	n	0.0761	u	0.0605		

Notre objectif sera de casser un code de César en regardant les 26 décalages possibles, et en cherchant, à l'aide de la minimisation d'une quantité, quel est le décalage le plus probable. Nous allons minimiser la somme des écarts des fréquences au carré.
Plus exactement, si $f_t(c)$ désigne la fréquence d'apparition théorique moyenne d'un caractère c dans la langue française, et si $f(c)$ désigne sa fréquence d'apparition dans un texte donné, l'écart des carrés est défini par :

$$S(e) = \sum_{c \in \mathscr{A}} (f_t(c) - f(c))^2$$

où $\mathscr{A}$ désigne l'alphabet.
En notant f_d les fréquences obtenues dans un texte par un décalage de d caractères dans le déchiffrage de César, nous allons chercher d qui minimise :

$$\sum_{c \in \mathscr{A}} (f_t(c) - f_d(c))^2$$

5. Pour un texte donné (encodé selon nos conventions), calculer les fréquences d'apparition des lettres du texte. On renverra le résultat sous la forme d'une liste (de 26 nombres). On pourra commencer par initialiser une liste contenant 26 zéros.
6. Pour une liste de fréquences donnée, écrire une fonction qui calcule la quantité S.
7. Pour un texte et un décalage donné, écrire une fonction qui calcule la quantité S.
8. Pour une chaîne de caractères, pour laquelle le décalage est inconnu, écrire une fonction qui renvoie une liste contenant les quantités $S(d)$ pour les différents décalages d possibles (d prend les valeurs entières de 0 à 25)
9. Écrire une fonction qui cherche le minimum d'une liste de nombres et renvoie la position de ce minimum.
10. Écrire une fonction qui renvoie la chaîne de caractères la plus probable dans le cas d'un chiffrage de César de décalage inconnu à l'aide de la méthode de l'analyse fréquentielle
11. Décoder le texte suivant en utilisant les fonctions précédentes. Quel était le décalage employé ?
« SJRVNVJWPNAMNBVJAAXWBUNBXRAJDLXRWMDOND »

La Disparition est un roman de Georges Perec écrit avec la contrainte très particulière de ne pas contenir la lettre e. Un fichier externe `ladisparitioncodee.txt` est disponible sur la page dédiée à cet ouvrage sur le site de Dunod.

12. Après avoir ouvert ce fichier en lecture, et avoir stocké la chaîne de caractères contenue dans ce fichier, essayez de décoder le texte par analyse fréquentielle.

Chiffre de Vigenère

Le **chiffre de Vigenère** est un système de chiffrement polyalphabétique, c'est un chiffrement par substitution, mais une même lettre du message clair peut, suivant sa position dans celui-ci, être remplacée par des lettres différentes, contrairement à un système de chiffrement monoalphabétique comme le chiffre de César (qu'il utilise cependant comme composant). Cette méthode résiste ainsi à l'analyse de fréquences, ce qui est un avantage décisif sur les chiffrements monoalphabétiques.
Ce chiffrement introduit la notion de clé. Une clé se présente généralement sous la forme d'un mot ou d'une phrase. Pour pouvoir chiffrer notre texte, à chaque caractère nous utilisons une lettre de la clé pour effectuer la substitution. Évidemment, plus la clé sera longue et variée et mieux le texte sera chiffré.
Dans la clé, le caractère en position i va déterminer le décalage à effectuer dans le texte à coder à la position i. Si le caractère dans la clé est un 'A' on effectuera un décalage de 0. Si le caractère est un 'B' on effectuera un décalage de +1. Si c'est un 'S' un décalage de +2 ... si c'est un 'Z' un décalage de +25.
Nous utiliserons les mêmes conventions sur les clés que sur les textes : majuscules ; pas d'accent ; pas de ponctuation et pas d'espace.

Ainsi, 'KEBAB' avec la clé 'ADANA' se code 'KHBNB' (les 'A' dans « ADANA » n'ont pas d'effet de décalage ; le 'D' a pour effet de décaler de +3 donc transforme le 'E' en 'H' ; le 'N' a un effet +13, donc transforme un 'A' en 'N')
Lorsque la clé utilisée est plus courte que le texte, elle est répétée autant de fois que nécessaire. Ainsi 'KEBABBIENCUIT' avec la clé 'ADANA' est codé comme si la clé était 'ADANAADANAADA' et est codé 'KHBNBBLEACULT'.

13. Décoder à la main le texte « OPEIPZEPXETWAFIPWMS » codé avec la clé « ACE ».
14. Écrire une fonction qui connaissant le caractère de la clé utilisée renvoie le décalage à faire.
15. Écrire une fonction qui prend en argument la position dans le texte à coder et renvoie le caractère utilisé dans la clé.
16. Écrire une fonction `CodageVigenere(texte,cle)` de codage d'un texte selon une clé.
17. Écrire une fonction qui déchiffre un texte codé selon le chiffre de Vigenère connaissant la clé.
18. Vérifier votre fonction à l'aide de la question 13.

Corrections des exercices

Corrigé exo 1.0

0. Le `range(1, n + 1)` permet de décrire tous les entiers entre 1 et n.

```
def somme_n_premiers_entiers(n):
    s = 0
    for i in range(1, n + 1):
        s += i
    return s
```

1. Le while est exécuté jusqu'à ce que i soit égal à n. Il faut faire attention à la condition que l'on met et où l'on place le `i += 1`.

```
def factorielle(n):
    f = 1
    i = 1
    while i <= n:
        f *= i
        i += 1
    return f
```

2. Une fonction peut faire appel à d'autres fonctions :

```
def c_n_k(n, k):
    return factorielle(n) / (factorielle(k) * factorielle(n - k))
```

Corrigé exo 1.1

0. On parcourt la liste en mettant à jour le maximum local (`maxi`) trouvé ainsi que son indice (`ind`).

```
def indicemax(L):
    ind = 0
    maxi = L[0]
    for i in range(1, len(L)):
        if maxi < L[i]:
            ind, maxi = i, L[i]  # on met à jour le max local

    return ind
```

1. On s'arrête dans la boucle si on trouve deux éléments consécutifs mal ordonnés.

```
def estcroissante(L):
    n = len(L)
    for i in range(n - 1):
        if L[i] > L[i + 1]:
            return False
    return True
```

2. On reprend l'idée de la question précédente tout en mettant à jour la liste des sous-suites trouvées. Attention à bien rajouter la dernière sous-liste après la boucle.

```
def listecroissmax(L):
    """prend une liste L en paramètre et renvoie la liste ordonnée des couples
d'indices correspondant au partitionnement en scm de L."""
    deb = 0
    Lsoussuites = []
    n = len(L)

    for i in range(n - 1):
        if L[i] > L[i + 1]:
            Lsoussuites.append((deb, i + 1))
            deb = i + 1

    Lsoussuites.append((deb, n))

    return Lsoussuites
```

3. On adapte l'algorithme de recherche de l'indice d'un maximum à la structure de données (couple d'indices `(debut, fin exclue)`).

```
def soussuitemax(L):
    Lss = listecroissmax(L)
    n = len(Lss)
    debmax, finmax, maxi = 0, 0, 0
    for deb, fin in Lss:
        long = fin - deb
        if long > maxiprov:
            maxi = long
            debmax, finmax = deb, fin
    return debmax, finmax
```

```
debmax, finmax = soussuitemax(L)
print(L[debmax:finmax])
```

```
[0, 2, 5, 6, 8, 9]
```

Corrigé exo 1.2

0. Pour spécifier un ou logique on utilise le `or` entre les conditions.

```
if feu_vert or feu_orange:
    pieton_rouge = True
    pieton_vert = False
```

1. Pour utiliser le non logique on utilise le mot `not` devant la condition.

```
if not feu_rouge:
    pieton_rouge = True
    pieton_vert = False
else:
    pieton_rouge = False
    pieton_vert = True
```

2. La boucle `while` est exécutée de manière itérative tant que la condition d'entrée est `True`

```
while pieton_vert:
    feu_vert, feu_orange, feu_rouge = False, False, True
```

Pour cette application cela n'a aucun intérêt et ralentit considérablement le calcul car l'action « Mettre le feu au rouge » est réitérée à chaque cycle, c'est à dire des millions de fois en attendant que le feu piéton passe au rouge.

3. La fonction ne prend pas d'argument et elle renvoie un `None` qu'il est inutile d'écrire. Les variables sont globales, elles doivent avoir été définies en tant que telles auparavant.

```
def passage_feu_vert():
    global feu_vert, feu_orange, feu_rouge
    feu_vert, feu_orange, feu_rouge = True, False, False
    return
```

4. Voici le code complet.

```
feu_vert, feu_orange, feu_rouge = True, False, False

if not feu_rouge:
    pieton_vert, pieton_rouge = False, True
else:
    pieton_vert, pieton_rouge = True, False


def passage_feu_vert(fv, fo, fr):
    fv, fo, fr = True, False, False
    return fv, fo, fr


def passage_feu_orange(fv, fo, fr):
    fv, fo, fr = False, True, False
    return fv, fo, fr


def passage_feu_rouge(fv, fo, fr):
    fv, fo, fr = False, False, True
    return fv, fo, fr


def passage_pieton_vert(pv, pr):
    pv, pr = True, False
    return pv, pr


def passage_pieton_rouge(pv, pr):
    pv, pr = False, True
    return pv, pr


def changement_feux(fv, fo, fr, pv, pr):
    if fv:
        fv, fo, fr = passage_feu_orange(fv, fo, fr)
    elif fo:
        fv, fo, fr = passage_feu_rouge(fv, fo, fr)
        pv, pr = passage_pieton_vert(pv, pr)
    else:
        if pieton_vert:
            pv, pr = passage_pieton_rouge(pv, pr)
        else:
            fv, fo, fr = passage_feu_vert(fv, fo, fr)
    return fv, fo, fr, pv, pr
```

On appelle ensuite la fonction `changement_feux` avec la console. Le \ permet de continuer une instruction trop longue sur la ligne suivante.

```
>>> feu_vert, feu_orange, feu_rouge, pieton_vert, pieton_rouge = \
... changement_feux(feu_vert, feu_orange, feu_rouge, pieton_vert, pieton_rouge) \
... # le feu passe au orange
>>> feu_vert, feu_orange, feu_rouge, pieton_vert, pieton_rouge = \
... changement_feux(feu_vert, feu_orange, feu_rouge, pieton_vert, pieton_rouge) \
... # le feu passe au rouge et le piéton au vert
>>> ... # on peut continuer indéfiniment
```

Corrigé exo 1.3

0. À la main :

```
def moyenne(L):
    s = 0
    for x in L:
        s = s + x
    return s / len(L)
```

Et en utilisant `sum` :

```
def moyenne(L):
    return sum(L) / len(L)
```

1. On prend garde à n'appeller la fonction `moyenne` qu'une seule fois. En effet, l'appeler plusieurs fois demande plus de temps de calcul à l'ordinateur. Nous verrons au chapitre 3 comment estimer ce temps de calcul supplémentaire[1].

```
def variance(L):
    m = moyenne(L)
    s = 0
    for x in L:
        s += (x - m)**2
    return s / len(L)
```

2.

```
def variancekonig(L):
    s = 0
    for x in L:
        s += x**2
    return s / len(L) - moyenne(L)**2
```

3.

```
L1 = [2**i for i in range(4)]
L2 = [2**27 + x for x in L1]
print(variance(L1), variancekonig(L1))
print(variance(L2), variancekonig(L2))
```

4. On peut sans peine vérifier que $V(L_1) = V(L_2)$. Pourtant si le calcul de la variance est correct à l'aide de la première formule, il est faux dans le cas du calcul de la variance de L_2 par la formule de König.

1. Dans le cas présent, appeler plusieurs fois la fonction `moyenne` entraîne une *complexité* en $\mathcal{O}(n^2)$.

Cela s'expliquera mieux après le prochain chapitre : si $2^{27}+1$ est correctement codé comme flottant en machine, ce n'est pas le cas de $(2^{27}+1)^2$ qui nécessiterait une mantisse de 54 bits pour être codé exactement.

Corrigé exo 1.4

0. Il faut ici calculer les S_n successifs et nous utilisons la relation $p_{n+1} = p_n \frac{\lambda}{n+1}$ pour éviter les calculs répétés des puissances de λ et des factorielles. Après avoir simulé le tirage aléatoire de U, nous initialisons trois variables :

$$S = e^{-\lambda} = S_0,\ p = e^{-\lambda}\lambda = p_1 \text{ et } n = 0.$$

Tant que $S < U$, nous incrémentons n en nous assurant que les propriétés suivantes restent [1] vraies : $S = S_n$ et $p = p_{n+1}$. À la sortie de la boucle, nous avons $U \leqslant S$, ce qui donne $S_{n-1} < U \leqslant S_n$ (en posant $S_{-1} = -1$ pour le cas $n = 0$) : le compteur n contient alors la valeur de X.

```
import numpy.random as rd

from math import exp

def poisson(l):
    U, S, n, p = rd.random(), exp(-l), 0, exp(-l)*l
    while(U > S):
        S += p
        n += 1
        p *= l / (n + 1)
    return k
```

1. L'analyse est élémentaire : on définit $m = e^{-\lambda}$ pour ne pas calculer cette valeur plusieurs fois, on initialise n à la valeur 0 et S à la valeur U_0 ; tant que $S > e^{-\lambda}$, on multiplie S par U_{n+1} et on incrémente n. Ainsi, quand on sort de la boucle `while`, n contient la valeur de la variable Y.

```
def poisson_bis(l):
    m, n, S = exp(-l), 0, rd.random()
    while(S > m):
        S *= rd.random()
        n += 1
    return k
```

2. On effectue un assez grand nombre M de tirages indépendants simulant X (ou Y) et on estime l'espérance et la variance de façon classique :

$$E(X) \simeq \frac{1}{M}\sum_{i=1}^{M} x_i = \overline{x} \text{ et } V(X) \simeq \frac{1}{M}\sum_{i=1}^{M} x_i^2 - \overline{x}^2.$$

```
def estimation(l, M):
    S, S2, Sbis, S2bis = 0, 0, 0, 0
    for i in range(M):
        X = poisson(l) # on calcule x_i
        Y = poisson_bis(l) # on calcule y_i
        S += X # on somme les x_i
        S2 += X**2 # on somme les x_i^2
```

1. Ces propriétés sont appelées « invariants de boucle », cf. le chapitre 3 p. 85.

```
        Sbis += Y # on somme les y_i
        S2bis += Y**2  # on somme les y_i^2
    return S / M, S2 / M - (S / M)**2, Sbis / M, S2bis / M - (Sbis / M)**2 # on renvoie les estimations.
```

La loi de Poisson de paramètre λ a une espérance et une variance égales à λ, et nous obtenons des résultats cohérents :

```
>>>estimation(2, 10000)
(2.0034, 2.0249884399999996, 2.0088, 2.0245225600000003)
>>> estimation(200, 10000)
(200.3554, 198.0336908399986, 199.6793, 199.79105150999385)
>>> estimation(300, 10000)
(299.9778, 302.5977071600064, 300.1755, 295.95529974999954)
```

Corrigé exo 1.5

0. On initialise une liste L de longueur d ne contenant que des 0, puis on simule n déplacements aléatoires :

```
def M(d, n):
    L = [0 for i in range(d)]
    for t in range(n):
        i = rd.randint(d)  # on choisit la coordonnée à modifier
        if rd.randint(2) == 0:
            L[i] -= 1  # avec probabilité 1/2, on soustrait 1
        else:
            L[i] += 1  # avec probabilité 1/2, on ajoute 1
    return L
```

1. On reprend la méthode précédente, en s'arrêtant dès que $L = [0, \dots, 0]$. Nous avons ajouté une variable N qui permet de sortir de la boucle `while` si la puce n'est pas encore revenue à l'origine à l'instant N, et ainsi forcer la sortie quand T est très grand ou infini.

```
def T(d, N):  # la borne N permet de sortir de la boucle si T est trop grand (ou infini)
    O, L, retour = [0 for i in range(d)], [0 for i in range(d)], False
    n = 0
    while (not retour) and n < N:  # tant que l'on n'est pas revenu à l'origine
        i = rd.randint(d)  # la puce saute
        if rd.randint(2) == 0:
            L[i] -= 1
        else:
            L[i] += 1
        retour = (L == O)  # est-ce que la puce est revenue à l'origine ?
        n += 1  # et on incrémente n
    if retour:
        return n  # la puce est revenue à l'origine à l'instant n <= N
    else:
        return -1  # la puce n'est toujours pas retournée en O à l'instant N
```

La fonction `a`, appliquée à (d, N, M), estime la probabilité de l'évènement $T \leqslant N$ en effectuant M simulations :

```
def a(d, N, M):
    c = 0
    # c = nombre de fois T <= N
    for i in range(M):
        if T(d, N) != -1:


            c += 1
    return c / M
```

```
>>> a(1, 1000, 1000), a(1, 5000, 1000), a(1, 10000, 1000)
(0.978, 0.984, 0.991)
>>> a(2, 50000, 1000), a(2, 100000, 1000), a(2, 1000000, 1000)
(0.773, 0.793, 0.82)
>>> a(3, 20000, 10000), a(3, 50000, 10000)
(0.3367, 0.3356)
```

Quand $d = 1$, on peut conjecturer que la puce va presque sûrement revenir à l'origine, puisqu'au cours de 1000 simulations, on est revenu à l'origine dans plus de 99% des cas avant l'instant $N = 10000$. Les choses sont moins claires pour $d = 2$ et $d = 3$, les temps de retour à l'origine étant très grands et les temps de calcul trop longs. La théorie nous apprend que pour $d = 1$ ou $d = 2$, on revient presque sûrement à l'origine (et même que l'on repasse presque sûrement une infinité de fois par l'origine : la marche aléatoire est **récurrente**) ; par contre, pour $d = 3$, la probabilité de l'évènement $T < +\infty$ est proche de 0,34 (et on ne repasse presque sûrement qu'un nombre fini de fois par l'origine : la marche aléatoire est **transiente**).

Corrigé exo 1.6

0. Nous allons calculer u_n (resp. v_n) à l'aide d'une boucle : nous initialisons au début du calcul la variable `a` à la valeur $\sqrt{n}$ (resp. $\sqrt{2n+1}$) puis, pour un indice i variant de $n-1$ à 1 par pas de -1, nous remplaçons a par $\sqrt{i+a}$. Cela donne :

```
from math import sqrt


def u(n):
    a = sqrt(n)
    for i in range(n - 1, 0, -1):
        a = sqrt(i + a)
    return a


def v(n):
    a = sqrt(2 * n + 1)
    for i in range(n - 1, 0, -1):
        a = sqrt(i + a)
    return a
```

Le calcul des premières valeurs des deux suites permet de conjecturer qu'elles sont adjacentes. La fonction suivante vérifie que $u_2 \leqslant u_3 \leqslant \cdots \leqslant u_n \leqslant v_n \leqslant \cdots \leqslant v_3 \leqslant v_2$, et renvoie la liste des valeurs approchées des $v_i - u_i$:

```
def verif(n):
    U = u(2)
    V = v(2)
    b = u(2) <= v(2)
    L = [V - U]
    i = 2
    while(b and i < n):
        NU = u(i + 1)  # on calcule les termes suivants
        NV = v(i + 1)  # des suites u et v
        b = U <= NU <= NV <= V  # le booléen b prend la valeur false s'il y a un problème
        U, V = NU, NV
        L.append(V - U)  # on ajoute l'écart à la liste L
        i += 1
```

```
    if b:
        return L
    else:
        return 'les suites ne semblent pas adjacentes'
```

Nous obtenons des résultats qui confortent la conjecture :

```
>>> verif(11)
[0.06407963669631855, 0.014581088998520508, 0.0029870200777970535, 0.0005607263974747312,
 9.767377139469069e-05, 1.5940648317336326e-05, 2.45600719828154e-06, 3.594075530521934e-07,
 5.0202848544955714e-08]
```

En admettant que les suites sont bien adjacentes, nous pouvons calculer une approximation de leur limite commune en utilisant la fonction `approximation` : les variables U et V contiennent les valeurs successives u_n et v_n, qui sont calculées tant que $V - U > \varepsilon$. Nous ajoutons le garde-fou $n < 10000$ pour être certain de sortir de la boucle `while`.

```
def approximation(epsilon):
    n = 2
    U = u(n)
    V = v(n)
    while(n < 10000 and V - U > epsilon):
        n += 1
        U = u(n)
        V = v(n)
    if n == 10000:
        return 'la convergence est trop lente'
    else:
        return U
```

Corrigé exo 1.7

0. On importe une bibliothèque comme `math` ou `numpy` pour le logarithme :

```
import numpy as np
T=[0,29,58,91,127,170,203]
F=[296,281,266,251,236,221,207]
Y=[np.log(x) for x in F]
```

1. On réutilise les fonctions de l'exercice précédent :

```
a,b,rho=reglin(T,Y)
def f(t):
    return np.exp(a*t+b)
```

On peut alternativement utiliser la syntaxe `lambda` :

```
a,b,rho=reglin(T,Y)
f = lambda t:np.exp(a*t+b)
```

2. On calcule les images des éléments de `T` par `f` :

```
import matplotlib.pyplot as plt

plt.plot(T, F, "o")
```

```
Z=[f(t) for t in T]
plt.plot(T, Z)
plt.title("Valeurs expérimentales et ajustées pour la réaction de Landolt")
```

Corrigé exo 1.8

0. On retrouve les commandes classiques d'ouverture, fermeture et lecture des lignes de fichiers.

```
def lecture_emericc(filename):
    monfichier = open(filename, "r")  # Ouverture du fichier
    entete = []
    for i in range(6):  # Itération sur les 6 premières lignes d'en-tête
        ligne = monfichier.readline()  # Lecture de la ligne suivante
        entete.append(ligne.replace(",", ".").strip("\n").split("\t"))
        # Travail sur les chaînes de caractères
    pts = []
    pos = []
    vit = []
    var = []
    for ligne in monfichier:  # Itération sur les lignes successives jusqu'à la dernière
        var0, var1, var2, var3 = ligne.replace(
            ",", ".").strip("\n").split("\t")
        pts.append(float(var0))
        pos.append(float(var1))
        vit.append(float(var2))
        var.append(float(var3))
    monfichier.close()  # Fermeture du fichier
    return entete, pts, pos, vit, var
```

1. L'appel de la fonction est réalisé de cette manière :

```
entete, points, position, vitesse, variateur = lecture_emericc(
    "test_emericc.txt")
```

2. Le pas de temps est défini en `str`, il ne faut pas oublier de le convertir en `float`.

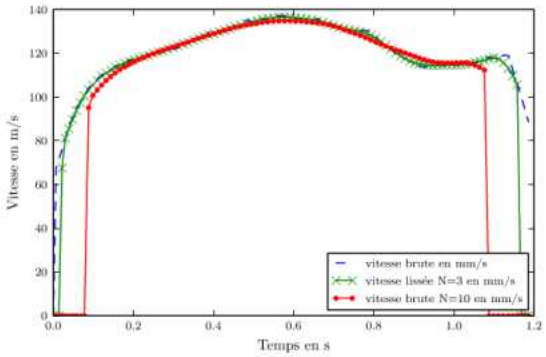
```
pas_temps = float(entete[3][1])
temps = []
for i in range(len(points)):
    temps.append(position[i] * pas_temps)
```

3. Fonction `lissage_moyenne_mobile` :

```
def lissage_moyenne_mobile(liste, N):
    liste_lissee = []
    for i in range(N):
        liste_lissee.append(0)  # on supprime les N premiers points
    for i in range(N, len(liste) - N):
        xi = liste[i]
        for k in range(1, N + 1):
            xi += liste[i - k] + liste[i + k]
        liste_lissee.append(xi / (2 * N + 1))
    for i in range(len(liste) - N, len(liste)):
        liste_lissee.append(0)  # on supprime les N derniers points
    return liste_lissee
```

4. Le tracé est le suivant, on remarque que l'on tronque beaucoup les données et que l'on filtre trop avec $N = 10$.

```
# Lissage des données
vit_lissee_3 = lissage_moyenne_mobile(vitesse, 3)
vit_lissee_10 = lissage_moyenne_mobile(vitesse, 1

# Tracé de la figure
plt.figure(1)
plt.plot(temps, vitesse, '-',
         label="vitesse brute en mm/s")
plt.plot(temps, vit_lissee_3, 'x-',
         label="vitesse lissée N=3 en mm/s")
plt.plot(temps, vit_lissee_10, '.-',
         label="vitesse brute N=10 en mm/s")
plt.ylabel('Vitesse en m/s')
plt.xlabel('Temps en s')
plt.legend(loc=4)
plt.show()
```

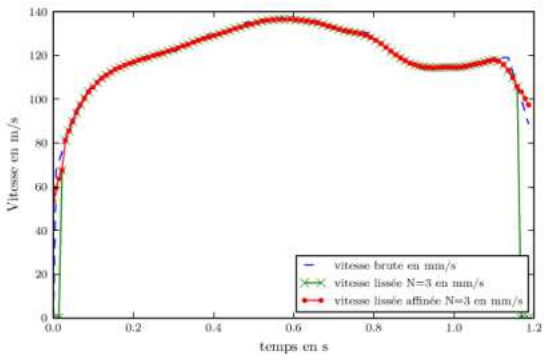


5. On affine en mettant un test sur la présence ou non des voisins

```
def lissage_moyenne_mobile_affinee(liste, N):
    liste_lissee = []
    for i in range(len(liste)):  # parcours de tous les points
        xi = liste[i]
        nb_voisins = 1  # on compte les voisins pour le calcul de la moyenne
        for k in range(1, N + 1):
            if i - k < 0:  # test de présence du voisin i - k
                nb_voisins += 1
                xi += liste[i + k]
            elif i + k > len(liste) - 1:  # test de présence du voisin i + k
                nb_voisins += 1
                xi += liste[i - k]
            else:
                nb_voisins += 2
                xi += liste[i - k] + liste[i + k]
        liste_lissee.append(xi / nb_voisins)
    return liste_lissee
```

6. On observe sur le tracé que les données non supprimées sur les bords ne sont pas de bonne qualité non plus. Il serait plus judicieux de garder les données initiales pour ces points.

```
vit_lissee_aff_3 =
    lissage_moyenne_mobile_affinee(vitesse, 3)

plt.figure(3)
plt.plot(temps, vitesse, '--',
         label="vitesse brute en mm/s")
plt.plot(temps, vit_lissee_3, 'x-',
         label="vitesse lissée N=3 en mm/s")
plt.plot(temps, vit_lissee_aff_3, '.-',
    label="vitesse lissée affinée N=3 en mm/s")
plt.xlabel('temps en s')
plt.ylabel('Vitesse en m/s')
plt.legend(loc=4)
plt.show()
```



Corrigé exo 1.9

0. On peut également définir `alpha` à l'intérieur de la fonction `s`.

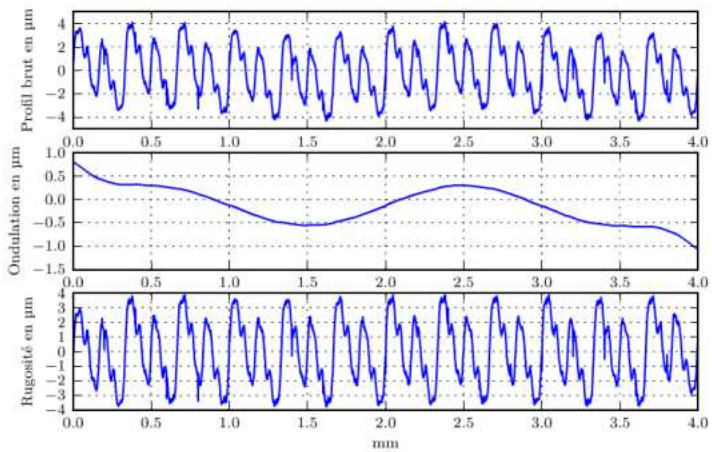
```
alpha = math.sqrt(math.log(2) / math.pi)


def s(x, lambda_co):
    return 1 / (alpha * lambda_co) * math.exp(-math.pi * (x / (alpha * lambda_co))**2)
```

1. La fonction de filtrage est donnée ci-après. On notera que les deux boucles imbriquées deviennent coûteuses en temps lorsqu'on augmente le nombre de points.

```
def filtrage_phase_correcte(x, y, lambda_co):
    y_bf = []
    y_hf = []
    for i in range(len(x)):
        y_temp = y[i]
        somme = 0
        for j in range(len(x)):
            somme += s(x[j] - x[i], lambda_co)
            y_temp += y[j] * s(x[j] - x[i], lambda_co)
        y_temp /= somme
        y_bf.append(y_temp)
        y_hf.append(y[i] - y_temp)
    return y_bf, y_hf

lambda_coupure = 1
point_bf, point_hf = filtrage_phase_correcte(x, y, lambda_coupure)
```

2. Le résultat pour une longueur d'onde de coupure de 1 mm est convenable, on peut d'ailleurs l'estimer visuellement sur les tracé des données brutes.

```
plt.figure(1)
plt.subplot(311)
plt.plot(x, y)
plt.ylabel('Profil brut en µm')
plt.grid()

plt.subplot(312)
plt.plot(x, point_bf)
plt.ylabel('Ondulation en µm')
plt.grid()

plt.subplot(313)
plt.plot(x, point_hf)
plt.xlabel('mm')
plt.ylabel('Rugosité en µm')
plt.grid()

plt.show()
```



Corrections des TP

Corrigé TP 1.1

0. Il suffit de créer une liste de N fois le nombre 1 :

```
def initialisation(N):
    return [1 for i in range(N)]
```

1. On importe la bibliothèque `random` permettant de générer du hasard :

```
import random as rd

def transition(L):
    n=rd.randint(0,len(L)-1)
    L[n]=1-L[n] #astuce qui permet de transformer un 0 en 1 et réciproquement
    return L #pas indispensable l'action sur L est globale mais on respecte l'énoncé
```

2. Il suffit de sommer tous les éléments de `L` (ou de compter le nombre de 1) :

```
def nombreA(L):
    return sum(L)
```

3. En utilisant les fonctions précédentes :

```
def evolution(k,N):
    NA=[N]
    L=initialisation(N)
    for i in range(k):
        L=transition(L)
        NA.append(compteA(L))
    return NA
```

4.

```
import matplotlib.pyplot as plt

k,N=20,100
T=list(range(N+1))
Y=evolution(k,N)
plt.plot(T,Y)
plt.xlabel("Nombre de transitions")
plt.ylabel("Nombre de boules dans l'urne A")
plt.title("Simulation du modèle d'Ehrenfest")
```

5. Le parcours par indices est adapté ici :

```
def chercheN(L,N):
    I=[]
    for i in range(len(L)):
        if L[i]==N:
            I.append(i)
    return I
```

6. Encore une fois le parcours par indices est nécessaire :

```
def diff(positions):
    L=[]
    for i in range(len(positions)-1):
        L.append(positions[i+1]-positions[i])
    return L
\end{monpyton}
\item Il suffit d'enchaîner les deux fonctions précédentes :
\begin{monpython}
def differences(L,N):
    pos=chercheN(L,N)
    return diff(pos)
```

7. Une simulation complète nécessite les exécutions des fonctions `evolution` puis `differences`. La première fonction nécessite k passages dans la boucle, la seconde aussi : cet algorithme nécessite $\mathscr{O}(k)$ opérations en ordre de grandeur.

8.

```
def moyennerec(L):
    N=L[0]
    tps=differences(L,N)
    return sum(tps)/len(tps)
```

9. Pour $N = 10$, le théorème de Kac affirme qu'il faut en moyenne 2^{10} itérations pour un retour à l'état initial. On pourrait donc tester ceci, en lançant une simulation avec un nombre nettement plus grand que cette durée, par exemple $k = 10^5$ puis en exécutant la fonction `moyennerec`. Le temps de calcul ne serait pas prohibitif.

10. Pour $N = 60$ la procédure précédente ne serait plus pratiquable car il faudrait lancer une simulation avec $100 \times 2^{60} \approx 1,15 \times 10^{20}$ itérations.

Corrigé TP 1.3

Voici quelques éléments de correction :

```
def decalageUnitairev1(c,d):
    return chr((ord(c)-65+d)%26+65) #Formule en passant par les codes ASCII

import string
Alphabet=string.ascii_uppercase
def decalageUnitairev2(c,d):
    n=Alphabet.index(c) #On récupère l'indice de c dans Alphabet..
    n=(n+d)%26 #.. c'est un bon exercice (de Sup) de l'implémenter sans index
    return Alphabet[n]

def ChiffrageCesar(chaine,decalage):
    s=""
    for ch in chaine:
        s+=decalageUnitairev2(ch,decalage) #Les str ne sont pas modifiables
    return s

def DechiffrageCesar(chaine,decalage):
    return ChiffrageCesar(chaine,-decalage) #De l'intérêt du modulo

FTh1=[768,80,332,360,1776,106,110,64,723,19,0,589,272,
      761,534,134,681,823,730,605,127,0,54,21,7]
Fth=[x/10000 for x in FTh1] #Plus lisible

def frequences(texte):
    L=[0. for i in range(26)]
    for ch in texte:
        i=Alphabet.index(ch) #ou i=ord(ch)-65
```

```
        L[i]+=1 #L[i] est le nombre de caractères d'indices i comptés
    return [x/len(texte) for x in L]

def qteS(L):
    return sum([(L[i]-Fth[i])**2 for i in range(26)]) #par compréhension

def minimaison(L):
    pos,val=0,L[0]
    for i in range(len(L)):
        if L[i]<val:
            pos,val=i,L[i] #ne pas oublier de remettre les deux vars à jour
    return pos #alternativement return L.index(min(L)) marche

def crackcesar(texte):
    L=[]
    for i in range(26):
        s=DechiffrageCesar(texte,i)
        L.append(qteS(frequences(s)))
    return minimaison(L) #renvoie la position du décalage le plus probable

def vraitexte(texte):
    return DechiffrageCesar(texte,crackcesar(texte)) #décode le texte

"""Q11 : "ONAINVENTERSADEPUIS"
... Il n'y a pas d'erreur d'orthographe il s'agit du codage RSA"""

def decalageVigenere(ch):
    return Alphabet.index(ch) #encore lui ...

def carVigenere(p,cle): #p est la position dans le texte
    return cle[p%len(cle)] #la clé est répétée ; modulo convient

def CodageVigenere(texte,cle):
    s=""
    for i in range(len(texte)): #parcours par indices pour connaître la position
        s+=decalageUnitairev2(texte[i],decalageVigenere(carVigenere(i,cle)))
    return s

def DechiffrageVigenere(texte,cle): #on peut aussi créer une "clé de décodage"
    s=""
    for i in range(len(texte)):
        s+=decalageUnitairev2(texte[i],-decalageVigenere(carVigenere(i,cle)))
    return s

print(DechiffrageVigenere("OPEIPZEPXETWAFIPWMS","ACE"))
```

CHAPITRE

2

Représentation des nombres

L'essentiel du cours

■ 0 Représentation mathématique des nombres

Définition

On appelle **base** tout entier naturel supérieur ou égal à deux.
On appelle **chiffre en base** b tout entier naturel compris entre zéro inclus et b exclu.

Définition

On appelle représentation en base $b \in \mathbb{N} \setminus \{0,1\}$ de l'entier $n \in \mathbb{N}$ une suite de chiffres $(a_k)_{k\in[\![0,p]\!]}$ telle que :

$$\sum_{k=0}^{p} a_k b^k = \sum_{k\in[\![0;p]\!]} a_k b^k = a_p b^p + a_{p-1} b^{p-1} \ldots + a_2 b^2 + a_1 b + a_0$$

Par exemple, dix-neuf s'écrit, en base dix, $19 = 1\times10^1+9\times10^0$ ou $019 = 0\times10^2+1\times10^1+9\times10^0$. En base deux, il s'écrit $10011_2 = 1\times2^4+0\times2^3+0\times2^2+1\times2^1+1\times2^0$ (pour éviter toute confusion, lorsque la base n'est pas dix, nous noterons la base en indice).
On appelle **nombre de chiffres en base** b **de** n le nombre minimal de chiffres nécessaire pour écrire n en base b. Par exemple, dix-neuf a deux chiffres en base dix.

Définition

On appelle représentation en base $b \in \mathbb{N}^*$ du réel $x \in \mathbb{R}^+$ une suite de chiffres $(a_k)_{k\in[\![-\infty,p]\!]}$ telle que :

$$\sum_{k\in]\!]-\infty;p]\!]} a_k b^k = a_p \times b^p + a_p \times b^{p-1} + \ldots + a_0 \times b^0 + a_{-1} \times b^{-1} + a_{-2} \times b^{-2} + \ldots$$

Pour lever toute ambiguïté les chiffres avec des puissances de b positives seront séparés des autres par une virgule (notation française) ou un point (notation anglaise).

Par exemple un quart s'écrit $0,250000\ldots$ en base dix, ce que nous écrirons plus simplement $0,25$. En base deux il s'écrit $0,01_2$. Nous dirons qu'un nombre a **un nombre fini de chiffres après la virgule** si la suite de ses chiffres après la virgule se termine par une infinité de zéros.

Règles de calcul

En base b :

- Multiplier par b décale la virgule d'un cran vers la droite.
- Diviser par b décale la virgule d'un cran vers la gauche.

Calcul du chiffre des unités

On appelle **chiffre des unités** en base b le chiffre associé à b^0.

- Le chiffre des unités d'un entier n est le reste de la division euclidienne de n par b.
- Le chiffre des unités d'un réel x est le chiffre des unités de sa partie entière $\lfloor x \rfloor$.

1 Représentation des entiers en machine

Dans l'ordinateur, toutes les valeurs vont être représentées par des suites de bits, c'est-à-dire par des suites de chiffres en base deux.

Définition

Les **entiers non-signés** sont des entiers « sans signe », c'est-à-dire nécessairement positifs.

En pratique, un entier va, généralement, être stocké sur un nombre fixé de bits (par exemple 32 ou 64). Pour représenter un entier non signé, on représente tous ses chiffres en base 2.

Tous les entiers ne sont pas représentables. Par exemple sur 32 bits, on peut représenter tous les entiers entre 0 et $2^{32} - 1 = 4294967295 \approx 4.3 \times 10^9$.

Le calcul se fait modulo $2^{\text{nombre de bits}}$. Ainsi, si le produit de deux entiers non-signés de 32 bits dépasse $2^{32} - 1$, on ne garde que le reste modulo 2^{32}. Par exemple, le calcul de $2^{31} * 2$ va donner 0.

Le codage des entiers est plus subtil en Python, il permet de représenter des entiers de taille arbitrairement grande. Il est possible d'avoir les entiers non-signés « classiques » avec les fonctions `uint32` et `uint64` de la bibliothèque `numpy`. La lettre `u` signifie ici « unsigned ».

Définition

Les **entiers signés** sont des entiers « avec signe », c'est-à-dire positifs ou négatifs.

Il existe plusieurs représentations des entiers signés, la plus courante est la représentation en complément à deux.

Définition

Dans la représentation en complément à deux, le premier bit indique le signe : 1 pour négatif, et 0 pour positif.
Si le premier bit est un 0, l'entier est interprété comme un entier non-signé.
Si le premier bit est un 1, l'entier est :

- interprété comme un entier non-signé,
- puis on lui retire $2^{\text{nombre de bits}}$.

La plage d'entiers représentable en signé sur p bits est $[\![-2^{p-1}, 2^{p-1}-1]\!]$.

Une autre représentation, plus rarement utilisée, consiste à ajouter un biais (une constante) aux entiers.

Définition

La **représentation biaisée** de l'entier n sur p bits est la représentation en non-signé de $n + \text{biais}$ où $\text{biais} = 2^{p-1} - 1$.

La plage d'entiers représentable avec ce biais est $[\![-2^{p-1}+1, 2^{p-1}]\!]$.
Les entiers dont le premier bit est 1 sont strictements positifs, les autres sont négatifs ou nuls. L'entier 0 est représenté par un zéro suivi de $n-1$ uns.

Il existe d'autres représentations des entiers en machine, par exemple le code de Gray (cf. exercice 2.11 p. 73), le BCD (*Binary Coded Decimal*, DCB en français, qui permet de représenter les entiers en base 10), le DPD (*Densely Packed Decimal*), etc.

■ 2 Représentation des réels en machine

Étant donné une base b, tout réel x non nul peut s'écrire sous la forme $(-1)^s \left(\sum_{k\in]\!]-\infty;e]\!]} a_k b^k\right)$ avec $a_e, a_{e-1}, \ldots$ des chiffres en base b et $a_e \neq 0$ et $s \in \{0,1\}$.
En factorisant par b^e et décalant l'indice de e, on arrive à $x = (-1)^s \left(\sum_{k\in]\!]-\infty;0]\!]} a_{k+e} b^k\right) \times b^e$.
Puis, en renumérotant les chiffres :

$$x = (-1)^s \left(\sum_{k\in\mathbb{N}} \alpha_k b^{-k}\right) \times b^p = (-1)^s \alpha_0, \alpha_1\alpha_2\alpha_3 \ldots \times b^e$$

Pour stocker une approximation d'un réel, il suffit alors de choisir une base puis de stocker s, e, et les premiers chiffres après la virgule.

Définition

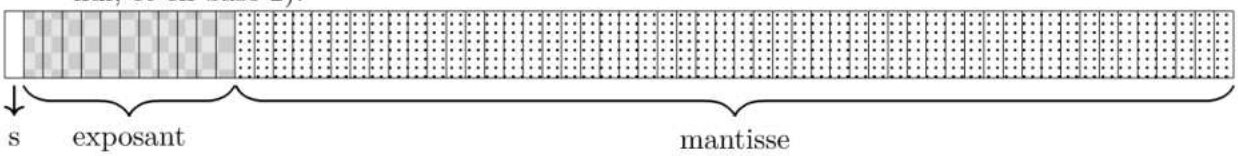
Les nombres ainsi représentés sont appelés **nombres flottants** ou **flottants**.

Comme seuls les premiers chiffres après la virgule sont conservés, une erreur d'approximation a lieu pour certains réels qui n'ont pas une écriture finie en base 2, (par exemple $\sqrt{2}$ ou 1.2).

En choisissant $b = 2$, sur 64 bits, un nombre réel peut être représenté comme suit :

- s est représenté sur 1 bit,
- e est représenté comme un entier biaisé sur 11 bits,

- Les 52 derniers bits représentent les 52 premiers chiffres après la virgule. Ils sont appelés significande ou mantisse. α_0 n'est pas représenté, car il vaut nécessairement 1 (il est non nul, et en base 2).



Dans cette représentation, e varie entre -1023 et $+1024$ (et non pas entre -1024 et 1023 comme ce serait le cas si e était codé en complément à deux). Les valeurs extrêmes $e = -1023$ et $e = +1024$ sont réservées pour coder des valeurs spéciales : les infinis ($+\infty$ et $-\infty$), le nombre zéro, nan (not a number), les flottants *subnormaux* (de très petits flottants).

Cette représentation s'appelle **`binary64`** dans la norme IEEE 754 - 2008[**IEE**]. Cette norme définit quatre autres représentations basiques des flottants : `binary32` et `binary128` ($b = 2$ avec 32 ou 128 bits), `decimal64` et `decimal128` ($b = 10$, ce qui permet de représenter de manière exacte certains nombres « usuels » comme 0.1 ou 0.2). Elle définit aussi d'autres représentations plus exotiques.

Voir le TP 4.1 p. 145 pour une utilisation astucieuse de la représentation des flottants.

Définition

On appelle **epsilon machine** l'écart entre le flottant 1.0 et le flottant juste supérieur à 1.0. Il vaut 2^{-52} ≈`2.22044604925e-16` pour des binary64.

L'epsilon machine ε correspond, grosso-modo, à l'erreur commise en approximant $1, b_1 b_2 b_3 \ldots$ par $1, b_1 b_2 b_3 \ldots b_{52}$ (pour les flottants 64 bits).
Comme $1, b_1 b_2 b_3 \ldots \approx 1$, l'erreur commise en approximant le réel x est d'environ $\varepsilon \times |x|$.

Les erreurs d'arrondis sur les flottants rendent dangereux les tests d'égalité. Par exemple, l'expression `a == 0` peut renvoyer `False` à cause d'une erreur d'arrondi sur `a`.
Pour tester si deux flottants sont égaux, nous testerons donc la « presque égalité », c'est -à-dire si la différence entre les flottants est « petite ». Par exemple, pour tester si `a` et `b` sont égaux, on écrit `abs(a-b) < 10**-14`. La constante (ici `10**-14`) doit être choisie raisonnablement petite.

■ 3 Représentation des caractères en machine

Les caractères sont représentés en machine par des suites de bits. Un des codages les plus utilisés est le codage ASCII, il permet de représenter l'alphabet latin de 26 lettres plus quelques caractères spéciaux sur 7 bits. Comme l'ordinateur traite les bits par paquets de 8 appelés octets, on complète ces 7 bits en ajoutant un zéro au début.
Les tables suivantes donnent la conversion des paquets de 4 bits en chiffres hexadécimaux, puis la conversion de l'hexadécimal vers l'ASCII. Par exemple, le caractère `A` a pour code hexadécimal `41` et va donc être représenté en machine par l'octet `01000001`.

Hex	Suite de bits
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1																
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

Les cases vides du tableau ASCII correspondent aux caractères spéciaux, par exemple au caractère de fin de ligne. Ces caractères peuvent être écrits grâce au caractère d'échappement \ appelé backslash ou antislash. Ainsi, le saut de ligne peut s'écrire `'\n'`.

Pour écrire un backslash dans une chaîne de caractères, il suffit de le doubler : `'\\'`. Il est aussi possible de préfixer la chaîne de caractères par un `r` (par exemple `r'C:\bidule \truc.txt'`), mais alors il n'est plus possible d'utiliser le backslash comme caractère d'échapement pour insérer des caractères spéciaux (comme le saut de ligne).
Dans les chemins de fichiers, il est possible de remplacer les backslash par des slash `/`, pour éviter les problèmes susmentionnés.

Cette représentation des caractères a toutefois ses limites. Représenter un caractère par 7 bits ne permet que de représenter $2^7 = 128$ caractères. C'est insuffisant pour représenter en même temps le français (é, è, ï, ...), le serbe (Б, Ж, ...), l'arabe (ا ب ت ث, ...), le chinois (数, 据, ...), etc.
L'Unicode est une norme qui attribue aux caractères de tous les alphabets actuels un numéro. Certains caractères ont un numéro (aussi appelé « point de code ») plus grand que 2^8, il est donc nécessaire d'avoir plusieurs octets pour représenter un caractère. l'Unicode est compatible avec l'ASCII, dans le sens où si un caractère existe en ASCII, il a le même numéro en Unicode.
La norme UTF-8 permet d'utiliser jusqu'à 4 octets pour un même caractère, ce qui permet de coder tous les caractères Unicode. UTF-8 utilise les principes suivants :

- Si le premier bit d'un octet est un « zéro », alors les 7 autres bits sont interprétés comme un code ASCII/Unicode.
- Si le premier bit d'un octet est un « un », alors le nombre de « un » consécutifs au début de l'octet (maximum 4) correspond au nombre total d'octets utilisés pour ce caractère. Les octets suivants commencent par `10`.

Représentation binaire	
`0xxxxxxx`	Un caractère ASCII codé sur 7 bits
`110xxxxx 10xxxxxx`	Un caractère Unicode codé sur 11 bits
`1110xxxx 10xxxxxx 10xxxxxx`	Un caractère Unicode codé sur 16 bits
`11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`	Un caractère Unicode codé sur 21 bits

Il est possible en Python de désigner un caractère par son code Unicode, on peut écrire dans une chaîne de caractères `\x` suivi de deux chiffres en base 16 ou `\u` suivi de 4 chiffres ou `\U` suivit de 8 chiffres. Par exemple `"\xF1\u0411"` donne la chaîne de caractères `'ñБ'`.

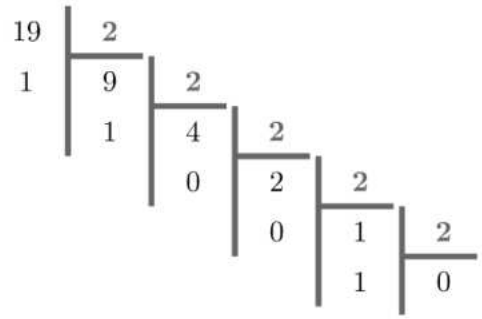
Les méthodes à maîtriser

Méthode 2.0 : Convertir un entier n en base b

- On détermine le chiffre des unités (ou chiffre de poids faible) en prenant le reste de n modulo b.
- On divise n par b et on recommence, tant que $n \neq 0$, pour calculer les chiffres suivants.

Exemple d'application

Calculer la représentation de 19 en base deux



L'écriture de dix-neuf en base deux est donc 10011_2.

Méthode 2.1 : Convertir un réel x de $[0,1[$ en base b

- On multiplie x par b pour que le chiffre juste après la virgule devienne le chiffre des unités.
- On lit le chiffre des unités de x (c'est sa partie entière).
- On retire à x sa partie entière et on recommence.

Exemple d'application

Convertir $1/5 = 0.2$ en base 2

$$
\begin{array}{llll}
0.2 & \times 2 = & 0.4 & 0 \\
0.4 & \times 2 = & 0.8 & 0 \\
0.8 & \times 2 = & 1.6 & 1 \\
0.6 & \times 2 = & 1.2 & 1 \\
0.2 & \times 2 = & 0.4 & 0 \\
0.4 & \times 2 = & 0.8 & 0 \\
\vdots & & \vdots & \vdots
\end{array}
$$

On remarque que les mêmes lignes vont se répéter à l'infini.
$1/5 = 0.001100110011\ldots_2 = 0.00\underline{1100}_2$ (la partie soulignée se répète infiniment).
On remarque que 0.2 tombe juste en base 10 mais pas en base 2. Ainsi, lorsqu'on entre 0.2 dans la console Python, une approximation est faite.

Le point important est qu'il est facile de déterminer le chiffre des unités. Pour trouver les autres chiffres, on les déplace au niveau des unités par des divisions ou des multiplications.

Méthode 2.2 : Convertir un réel en base b

- Déterminer son signe (+ ou −).
- Déterminer ses chiffres à gauche de la virgule avec la méthode 2.0.
- Déterminer ses chiffres à droite de la virgule avec la méthode 2.1.

Par exemple, 19.2 s'écrit en base deux : 10011.00110011 . . .

Méthode 2.3 : Convertir un entier de la base seize vers la base deux

On remplace chaque chiffre en base seize par les quatre chiffres en base deux qui lui correspondent (voir le tableau de la section 3 du cours).

Par exemple 61 en base seize s'écrit $\underbrace{0110}_{6}\underbrace{0001}_{1}$ en base deux.

Méthode 2.4 : Convertir un entier de la base deux vers la base seize

- Rajouter des zéros à gauche pour avoir un nombre de chiffres divisible par quatre.
- Regrouper les chiffres en base deux par quatre pour constituer des chiffres en base seize.

Par exemple, 111011 en base deux s'écrit aussi $\underbrace{0011}_{3}\underbrace{1011}_{B}$ soit 3B en base seize.

Méthode 2.5 : Déterminer la représentation en complément à deux

Si $n \leqslant 0$ est représentable, alors sa représentation en complément à deux est la même que sa représentation en non-signé.
Si $n = -m < 0$ est représentable, alors la représentation de n est complément à deux sur p bits est la représentation de $2^p - m$ en non-signé. On remarque que $2^p - m = (2^p - 1) - (m - 1)$. On pose alors cette soustraction en base deux. Comme $2^p - 1$ est représenté par une suite de 1, cette soustraction revient à inverser les bits de $m - 1$.

Exemple d'application

Représenter −19 sur 8 bits

On a $m - 1 = 18 = 10010_2 = 00010010_2$ donc −19 est représenté par 11101101.
La soustraction $(2^8 - 1) - 18$ est détaillée ci-après.

	1	1	1	1	1	1	1	1
−				1	0	0	1	0
	1	1	1	0	1	1	0	1

Méthode 2.6 : Tester l'égalité de deux flottants

Les erreurs d'approximation rendent peu pertinent le test `x==y` pour `x` et `y` deux flottants. Par exemple, `math.sin(math.pi)` renvoie `1.2246467991473532e-16` au lieu de zéro.
Usuellement, on remplace les tests d'égalité `x == y` par des tests de la forme `abs(x-y) < ε` avec `ε` raisonnablement petit (par exemple `ε = 1`).

Vrai/Faux sur le cours

0. C'est la même instruction du processeur qui fait le + sur les entiers et sur les flottants. ☐ **Vrai** ☐ **Faux**

1. Tout nombre entier peut être représenté par une suite finie de 0 et de 1. ☐ **Vrai** ☐ **Faux**

2. Tout nombre rationnel peut être représenté par une suite finie de 0 et de 1. ☐ **Vrai** ☐ **Faux**

3. Tout nombre réel peut être représenté par une suite finie de 0 et de 1. ☐ **Vrai** ☐ **Faux**

4. Quand on écrit `0.1` dans la console Python, il n'y a pas d'erreur d'approximation ☐ **Vrai** ☐ **Faux**

5. Informatiquement, on peut représenter de manière exacte 0.1. ☐ **Vrai** ☐ **Faux**

6. Peut-on représenter 2^{100} sur 32 bits ? ☐ **Vrai** ☐ **Faux**

7. En Python 3, le calcul de `14/2` donne l'entier `7`. ☐ **Vrai** ☐ **Faux**

8. `19 == 19.0` renvoie `True` en Python ☐ **Vrai** ☐ **Faux**

9. Il existe un nombre flottant qui représente exactement $\sqrt{3}$. ☐ **Vrai** ☐ **Faux**

10. Il existe un nombre flottant qui représente $+\infty$. ☐ **Vrai** ☐ **Faux**

11. Entre 0.5 et 1 il y a autant de flottants (`binary64`) qu'entre 1 et 2. ☐ **Vrai** ☐ **Faux**

12. Dans un ordinateur, les entiers sont forcément représentés en base 2. ☐ **Vrai** ☐ **Faux**

13. Le code suivant lève une exception car mathématiquement, $\tan\left(\frac{\pi}{2}\right)$ n'est pas défini. ☐ **Vrai** ☐ **Faux**

```
import math
math.tan(math.pi / 2)
```

14. La différence entre deux flottants plus grands que 1 est un multiple entier du epsilon machine. ☐ **Vrai** ☐ **Faux**

15. La différence entre deux flottants plus grands que 0.5 est un multiple entier de la moitié du epsilon machine. ☐ **Vrai** ☐ **Faux**

16. La différence entre deux flottants plus grands que 3 est un multiple entier du triple du epsilon machine. ☐ **Vrai** ☐ **Faux**

17. Si `n` est un entier (positif ou négatif) le calcul de `f(n)` renvoie le nombre de chiffres de `n` en base 3. ☐ **Vrai** ☐ **Faux**

```
def f(n):
    x = 0
    while n!= 0:
        n = n // 3
        x = x + 1
    return x
```

Exercices

Applications directes du cours

Exercice 2.0

On considère un entier $n \in \mathbb{N}^\star$ ayant p chiffres en base 2.

0. Quelle est la plage des valeurs des entiers à p chiffres ?
1. En déduire que $p = \lfloor \log_2(n) \rfloor + 1$ où $\log_2(n) = \ln(n)/\ln(2)$ est le logarithme en base 2 de n.
2. Montrer que $p = \lceil \log_2(n+1) \rceil$ où $\lceil x \rceil$ est le plafond de x (l'arrondi à l'entier supérieur).
3. Donner une formule similaire pour le nombre q_b de chiffres de n en base b.

Exercice 2.1

0. Écrire 42 puis -42 en entier signé (complément à deux) sur 8 bits.
1. Que représente `1000` en entier signé sur 4 bits ?
2. Quelle est la plage des entiers non-signés réprésentables sur 4 bits ? des entiers signés ?

Exercice 2.2

0. Écrire une fonction `base10(n)` qui prend en entrée un entier positif `n` et qui renvoie la liste de ses chiffres en base dix en commençant par le chiffre de poids faible.
Par exemple, `base10(1234)` devra renvoyer `[4, 3, 2, 1]`.
1. Même question avec la base onze.
2. Écrire une fonction `approx10(p, q, c)` qui, étant donné deux entiers naturels non nuls `p` et `q` tels que `p < q` et un entier `c` renvoie les `c` premiers chiffres après la virgule de `p / q` en base dix.

Exercice 2.3 D'après un oral ENSAM 2015

0. Écrire une fonction `binaire` qui prend en argument un entier `n` et renvoie la liste de ses chiffres en base 2, avec le bit de poids fort à gauche.
Exemple : $23 \to 10111$
1. Écrire une fonction `NombreDeUns` qui prend en argument un entier `n` et renvoie le nombre de 1 dans la décomposition en base 2 de `n`.
Exemple : $23 \to 4$
2. On appelle palindrome tout nombre dont l'écriture en base 2 est identique de gauche à droite et de droite à gauche.
Exemple : $9 \to 1001$
Écrire une fonction `palindrome` qui vérifie si un entier `n` est un palindrome.
3. Trouver tous les palindromes inférieurs à 100.
Remarque. On demande d'écrire un code Python permettant de trouver ces palindromes.

Exercice 2.4 Équations du second degré

Dans cet exercice, nous étudions les solutions de l'équation $ax^2 + bx + c = 0$ avec a, b et c trois réels.

0. Écrire une fonction `delta` qui, étant donnés les nombres a, b et c renvoie le discriminant $\Delta = b^2 - 4ac$.

1. Écrire une fonction `nombre_solutions` qui, étant donnés les nombres a, b et c renvoie le nombre de solutions réelles (soit zéro, soit un, soit deux) de l'équation $ax^2 + bx + c = 0$.

2. On peut calculer une racine carrée d'un nombre `r` réel ou complexe en Python grâce à l'opération `r**0.5`. Mathématiquement, l'équation $ax^2 + bx + c = 0$ a deux solutions complexes (éventuellement égales) : $x_1 = \frac{-b+\sqrt{\Delta}}{2a}$ et $x_2 = \frac{-b-\sqrt{\Delta}}{2a}$ où $\sqrt{\Delta}$ est une racine complexe de Δ. Écrire une fonction `sols` qui, étant donnés les nombres a, b et c renvoie les deux solutions complexes de l'équation $ax^2 + bx + c = 0$.

3. Tester ces trois fonctions avec les équations $x^2 + 1.4x + 0.49$ et $x^2 + 0.2x + 0.01$. Commentez.

4. Même question avec l'équation $x^2 + x + \frac{1}{4} + 10^{-20} = 0$.

Exercice 2.5 Associativité

Pour tous réels a, b et c, on a $(a + b) + c = a + (b + c)$. Cette propriété s'appelle l'associativité, et elle n'est pas vraie pour les nombres flottants.

0. Écrire une fonction `test` qui prend en argument 3 flottants a, b et c et qui vérifie si $(a + b) + c = a + (b + c)$.

1. Trouver par tâtonnements trois flottants `a`, `b` et `c` tels que `test(a, b, c)` renvoie `False`.

2. Parmi les flottants compris entre 0 et 10 et ayant au plus un chiffre après la virgule, on tire au hasard (uniformément) un triplet `(a, b, c)`. Quelle est la probabilité que `test(a, b, c)` renvoie `False` ?

On souhaite maintenant récupérer tous les triplets (`a`, `b`, `c`) tels que `test(a, b, c)` renvoie `False` dans un fichier csv. Le fichier csv est un fichier texte, sur chaque ligne un triplet doit être écrit. Les valeurs des triplets sont séparées par des points-virgules.

3. Écrire une fonction `NonAssoc` qui prend en argument un nom de fichier csv et qui écrit dans ce fichier tous les triplets recherchés.

Exercice 2.6

0. Quel est le plus grand nombre flottant codable sur 64 bits ? On rappelle que la valeur $e = 1024$ de l'exposant e est réservée pour coder des valeurs spéciales.

1. Quel est le deuxième plus grand nombre flottant codable sur 64 bit ?

2. Quel est l'écart absolu entre ces deux nombres ? Et l'écart relatif entre ces deux nombres ? Commenter.

Exercice 2.7 Flottants babyloniens

On appelle flottant babylonien une liste de huit chiffres en base soixante dont :

- Le premier s est 0 (positif) ou 1 (négatif).
- Le second est l'exposant e plus 30.

- Le troisième c_0 est le premier chiffre en base 60 avant la virgule.
- Les sept suivants $c_1, \ldots, c_5$ sont les chiffres après la virgule en base 60

Le flottant babylonien $b = [s, E, c_0, c_1, c_2, \ldots, c_5]$ représente le nombre $x = (-1)^s \times 60^{E-30} c_0, c_1 \ldots c_7$.

0. Écrire une fonction `bab2float(b)` qui, étant donné un flottant babylonien `b`, renvoie un flottant Python approximant `b`.
Par exemple `Bab2float([1,31,12,25,42,20,0,0])` devrait renvoyer `-745.7055555555556`.

1. Quel est le plus grand flottant babylonien ? Le plus petit ?

2. Écrire une fonction `int2bab` qui, étant donné un entier n renvoie sa représentation sous forme de flottant babylonien, on supposera que n est représentable.

3. Écrire une fonction `somme(b1, b2)` qui, étant donnés deux flottants babyloniens positifs, calcule leur somme arrondie à l'inférieur (sans passer par le type `float`, en utilisant l'algorithme de sommation vue en primaire).

Pour aller plus loin

Exercice 2.8 Patriot Missile Software Problem

Une batterie de missiles Patriot détecte les missiles ennemis et les intercepte avec un contre-missile. La batterie mesure le temps pour prévoir le déplacement des missiles ennemis.
Elle dispose d'un compteur (un entier) que nous appellerons `c` qui compte le nombre de dixièmes de secondes écoulées depuis sa mise en marche. Le temps écoulé `t` est calculé par l'opération suivante `t=c*0.1`. Nous nous intéressons à l'erreur de calcul commise lors de cette multiplication.
D'après un rapport du General Accounting Office [GAO92], le logiciel du Patriot utilise des nombres à virgule fixe ayant 24 chiffres après la virgule. Pour stocker un réel x, on stocke l'entier $\lfloor x \times 2^{24} \rfloor$ (la partie entière de 2 puissance 24), les chiffres au delà du 24ème après la virgule sont tronqués.
Sauf précision contraire, toutes les valeurs numériques demandées doivent être données en base 10.

0. Écrire en base 2 le nombre 0.1, on s'arrêtera à 24 chiffres après la virgule. On note y le nombre obtenu en tronquant 0.1 à 24 chiffres après la virgule.

1. Combien vaut $0.1 - y$?

Le calcul à virgule fixe induit une erreur de calcul sur le dernier chiffre. On note z le nombre obtenu en changeant le $24^{\text{ème}}$ bit après la virgule de y.

2. Combien valent $y - z$ et $0,1 - z$?

On note $\varepsilon = |0,1 - z|$. La batterie de missiles Patriot fait une erreur de ε en approximant 0.1.
Début février 1991, l'armée israélienne a empiriquement constaté qu'au bout de 8h, la précision des missiles est significativement réduite. Puis, le 25 février 1991, six batteries de missiles Patriot (un bataillon) ont été déployées à Dhahran, en Arabie Saoudite, pendant 100h.

3. Exprimer en fonction de ε l'erreur commise sur t par la batterie au bout de 8h puis au bout de 100h. Nous noterons e_8 et e_{100} ces erreurs.

4. Donnez une valeur approchée des deux erreurs précédentes.

Un Scud a une vitesse de croisière de Mach 5, soit environ 1702 m/s.

5. Pendant un temps de e_{100} de combien de mètres se déplace un Scud ?

Suite à cette imprécision, un Scud irakien ne fut pas intercepté et causa 28 morts parmi les soldats américains.

Exercice 2.9 Ordre de sommation

On sait que mathématiquement $S = \sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}$. Informatiquement, on peut approcher S par $S_n = \sum_{k=1}^{n} \frac{1}{k^2} = \frac{\pi^2}{6}$ pour n assez grand (par exemple $n = 100$ ou $n = 1000$).
Dans cet exercice, nous travaillerons uniquement avec des flottants de 16 bits, que l'on obtient avec la fonction `float16` de la bibliothèque `numpy` que nous importons via l'instruction `import numpy as np`.

0. Écrire une première fonction calculant S_n en faisant la somme dans l'ordre des k croissants. Par exemple, le calcul de S_3 se fera comme suit : $S_3 = \left(\frac{1}{1^2} + \frac{1}{2^2}\right) + \frac{1}{3^2}$

1. Écrire une première fonction calculant S_n en faisant la somme dans l'ordre des k décroissants. Par exemple, le calcul de S_3 se fera comme suit : $S_3 = \frac{1}{1^2} + \left(\frac{1}{2^2} + \frac{1}{3^2}\right)$

Si mathématiquement l'ordre n'a pas d'importance, informatiquement, l'erreur de calcul ne sera pas la même dans les deux cas Nous appelons `f` l'une de ces deux fonctions (peut-être la première, à moins que ce ne soit la deuxième ?) et `g` l'autre. En calculant $\frac{\pi^2}{6}$ grâce à l'expression `np.float16(np.pi**2/6)`, on obtient les résultats suivants :

$\frac{\pi^2}{6} - f(100)$	0.0097656
$\frac{\pi^2}{6} - g(100)$	0.017578

2. Identifier quelle est la fonction `f` (la première ? la seconde ?) et quelle est la fonction `g`.

3. Que se passe-t-il si on utilise des flottants de 64 bits au lieu de 16 bits ?

Exercice 2.10 Loi du maximum de n dés

On représente la loi d'une variable aléatoire X à valeurs dans $[\![0, N]\!] = \{0, \ldots, N\}$ par un tableau T de $N + 1$ valeurs tel que $T[i]$ contient la probabilité que X vaut i.
Ainsi, la loi d'un dé à 6 faces[1] est `[0.0, 0.167, 0.167, 0.167, 0.167, 0.167, 0.167]` et la loi du maximum de deux dés est `[0.0, 0.028, 0.083, 0.139, 0.194, 0.25, 0.306]`.

0. Écrire des instructions Python permettant de calculer la loi du maximum de deux dés.

On code le résultat $x_0, \ldots, x_{n-1}$ de n dés par l'entier $N = \sum_{k=0}^{n-1}(x_k - 1) \times 6^k$. Ainsi le résultat $2, 1, 3$ sera codé par l'entier $1 + 2 \times 6^2 = 73$.

1. Écrire une fonction `dés(N, n)` qui, étant donné un entier `N`, renvoie la suite $[x_0, \ldots, x_{n-1}]$ de `n` dés codés par l'entier `N`.

2. Écrire une fonction `loi_max_dés(n)` en Python qui renvoie la loi du maximum de n dés (en analysant tous les résultats possibles des dés).

On souhaite à présent généraliser la fonction précédente, et pouvoir calculer d'autres fonctions que le maximum (par exemple la médiane, ou le second dé ou autre). Plus formellement, On veut calculer la loi de probabilité de $f([x_1, \ldots, x_{n_1}])$ où f est une fonction qui prend en argument le résultats du lancé de n dés et qui renvoie un entier.

3. Écrire une fonction `loi_f(n, f, M)` qui prend en argument une fonction f à valeurs dans $[\![0, M]\!]$ et qui renvoie la loi de $f([x_1, \ldots, x_{n-1}])$.

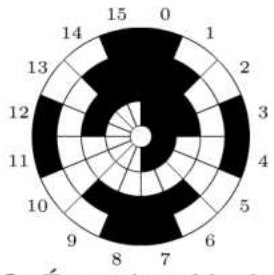
1. Dans cet exercice, tous les dés ont 6 faces numérotées de 1 à 6 et les résultats des jets de dés suivent une loi de probabilité uniforme.

Exercice 2.11 Code Gray

Définition

Le **code Gray** ou code de Gray est un code binaire également appelé **code binaire réfléchi** qui a pour intérêt de ne modifier qu'un seul bit entre deux valeurs codées consécutives.

Il est utilisé principalement dans les capteurs angulaires (appelés codeurs absolus) afin d'éviter les erreurs. En effet, si plusieurs bits doivent simultanément changer de valeur, il y a un risque non négligeable qu'un bit change de valeur avant l'autre, ce qui a pour conséquence de donner une évaluation de l'angle du capteur erronée durant un court instant et qui peut conduire le système à réagir d'une manière non prévue.



Pour présenter le fonctionnement, on se place dans le cas d'un capteur permettant de mesurer 16 positions angulaires, notées 0 à 15, comme le montre la figure ci-contre. Le capteur est composé d'un disque, comportant des informations de position réparties sur 4 pistes, et d'un lecteur optique muni de 4 capteurs qui renvoient 4 signaux (x, y, z, t) en fonction de la position du disque (blanc = 1 et noir = 0). Un transcodeur transforme ensuite ce code de position (x, y, z, t) en code binaire naturel (d, c, b, a), d étant le bit de poids fort.

0. Écrire la table de vérité de ce capteur en code Gray (celui représenté par les cases noires et blanches) et en code binaire naturel. On constate qu'il n'y a effectivement qu'un bit qui est modifié à chaque incrémentation angulaire.

Pos	x	y	z	t	d	c	b	a
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	1	0	0	1	0
3								
4								
⋮								

Table de vérité à compléter

1. Si le capteur possède N pistes, quelle est sa résolution, c'est-à-dire la plus petite valeur angulaire qu'il peut mesurer ?

2. Le cahier des charges impose une résolution maximale de 0,05°. Combien de pistes vont être nécessaires pour ce capteur ?

Le code Gray peut être obtenu à partir du code binaire naturel de plusieurs manières différentes. La plus simple à mettre en œuvre en informatique consiste à prendre la valeur binaire naturelle du nombre à coder et de réaliser un ou exclusif (xor) avec le même nombre binaire décalé d'un rang à droite comme l'exemple suivant :

$$0101 \oplus 0010 = 0111$$

Le code Gray du nombre 5_{10} (0101_2) est 0111_{2gray}.

3. Proposer un algorithme en Python permettant de traduire un nombre binaire naturel en code Gray.

On pourra utiliser l'opérateur $\wedge$ (ou exclusif) ainsi que `>>` (décalage vers la droite). On remarquera également qu'en Python un nombre est un nombre, quelle que soit la base dans laquelle on l'a exprimé. Par exemple `0b101` et 5 représentent la même chose.

```
>>> print(0b101)
5
```

Il n'y a *a priori* pas de manière simple de coder la fonction inverse permettant de passer du code Gray au binaire naturel, mais vous pouvez vous y essayer...

Exercice 2.12 Base ternaire balancée

Inspiré du sujet de l'ENS Lyon 2012 (fillière universitaire)
La base ternaire balancée ressemble à la base ternaire :

- Il y a trois chiffres différents.
- La suite de chiffres $t_p t_{p-1} \dots t_3 t_2 t_1 t_0$ représente l'entier $\sum_{k=0}^{p} t_k \times 3^k$.

La différence est que les chiffres sont 0, 1 et -1. Pour faciliter l'écriture, le -1 sera noté z.

0. Quel entier est représenté par $z110$?
1. Donner une représentation pour chaque nombre entier de $[-5, 5]$.
2. Comment est représenté 10 dans cette base ? et 19 ?

Dans la suite de cet exercice, les suites de chiffres sont représentées par des chaînes de caractères ne contenant que les caractères `0`, `1` ou `z` et ne commençant pas par `0`. Le nombre zéro sera représenté par la chaîne vide `""`. On admet que chaque entier a une représentation unique de ce type.

3. Écrire une fonction `positif` qui prend en argument une chaîne de caractères `t` représentant un entier en base ternaire balancée et qui renvoie `True` s'il est positif ou nul et `False` sinon.
4. Écrire une fonction `opposee` qui prend en argument une chaîne représentant un entier et qui renvoie la chaîne représentant son opposé.
5. Écrire une fonction `plus` qui prend en entrée deux chaînes de caractères représentant deux entiers en base ternaire balancée et qui renvoie la chaîne de caractères représentant leur somme. On reprogrammera l'algorithme d'addition sans repasser par les entiers.
6. Écrire une fonction `convert` qui prend en entrée une chaîne de caractères `t` et qui renvoie l'entier représenté par `t` en base ternaire balancée.
7. Écrire une fonction `b3b` qui prend en argument un entier `n` et qui renvoie sa représentation en base ternaire balancée.

Vrai/Faux sur le cours – corrigé

0. Non, les représentations sont complètement différentes et le calcul à faire sur les bits n'est pas le même. ☐ **Vrai** ☑ **Faux**

1. En prenant un nombre p suffisamment grand de bits (tel que $-2^{-p+1} \leqslant n < 2^{p-1}$), l'entier n est représentable en signé sur p bits. ☑ **Vrai** ☐ **Faux**

2. Un nombre rationnel peut être représenté par un couple d'entiers. En `Python`, la bibliothèque `fractions` implémente les rationnels. ☑ **Vrai** ☐ **Faux**

3. Pour représenter tous les nombres réels, on a besoin d'une suite infinie de bits. Ceci est lié au fait que $\mathbb{R}$ n'est pas dénombrable. ☐ **Vrai** ☑ **Faux**

4. Python utilise par défaut des flottants binaires (`binary64`) et `0.1` ne tombe pas juste en base deux. ☐ **Vrai** ☑ **Faux**

5. Oui, par exemple avec un `decimal64` (ou avec la bibliothèque `decimal` de Python). Mais 0.1 n'a pas de représentation exacte en `binary64` ☑ **Vrai** ☐ **Faux**

6. Par exemple, avec un flottant `binary32`. ☑ **Vrai** ☐ **Faux**

7. Le calcul renvoie le flottant 7.0. En Python 2, le comportement est différent. ☐ **Vrai** ☑ **Faux**

8. Le test d'égalité renvoie `True` même si les types sont différents. ☑ **Vrai** ☐ **Faux**

9. $\sqrt{3}$ est irrationnel, dans toutes les bases son nombre de chiffres après la virgule est infini. Il ne peut pas être représenté exactement par un flottant. ☐ **Vrai** ☑ **Faux**

10. On l'obtient avec l'expression `float("inf")`. ☑ **Vrai** ☐ **Faux**

11. Pour chaque exposant, il y a autant de mantisses possibles. Entre 0.5 et 1 c'est l'exposant -1, entre 1 et 2 c'est l'exposant 0. La « densité » de flottants est donc deux fois plus grande entre 0.5 et 1. ☑ **Vrai** ☐ **Faux**

12. Ils peuvent aussi être représentés dans d'autres bases (par exemple en base 10 avec DCB). ☐ **Vrai** ☑ **Faux**

13. Informatiquement, avec l'erreur d'approximation, l'expression `math.pi/2` renvoie une valeur proche de $\pi/2$ dont la valeur est bien définie. ☐ **Vrai** ☑ **Faux**

14. L'écart est un entier fois l'epsilon machine fois une puissance de deux. Ici, comme les deux flottants sont plus grands que 1, la puissance de deux vaut au moins 2^0. ☑ **Vrai** ☐ **Faux**

15. Ici, comme les deux flottants sont plus grands que 0.5, la puissance de deux vaut au moins 2^0. ☑ **Vrai** ☐ **Faux**

16. Deux flottants entre 2 et 4 peuvent avoir une différence de seulement deux fois le epsilon machine. La phrase devient vraie si on remplace 3 par 4 et « triple » par « quadruple ». ☐ **Vrai** ☑ **Faux**

17. On utilise la convention que zéro a zéro chiffre. ☑ **Vrai** ☐ **Faux**

Corrections des exercices

Corrigé exo 2.0

0. Les entiers à p chiffres sont compris entre 2^{p-1} (un 1 suivi de $p-1$ zéros en base deux) et 2^p exclus. Ce qui donne comme plage de valeurs $[\![2^{p-1}, 2^p[\![= [\![2^{p-1}, 2^p - 1]\!]$.

1. On applique le logarithme en base deux à l'inégalité $2^{p-1} \leqslant n < 2^p$. On obtient $\log_2(2^{p-1}) \leqslant \log_2(n) < \log_2(2^p)$ d'où $p - 1 \leqslant \log_2(n) < p$ et donc $p - 1 = \lfloor \log_2(n) \rfloor$.

2. On ajoute 1 à l'inégalité $2^{p-1} \leqslant n < 2^p$. On obtient $2^{p-1} + 1 \leqslant n + 1 < 2^p + 1$. En utilisant le fait que tous les termes sont entiers, on transforme ces inégalités en $2^{p-1} < n + 1 \leqslant 2^p$ puis on applique le logarithme en base deux, ce qui donne $p - 1 < n + 1 \leqslant p$ d'où le résultat voulu.

3. On montre de la même manière que $q_b = \lfloor \log_b(n) \rfloor + 1 = \lceil \log_b(n+1) \rceil$.

Corrigé exo 2.1

0. Pour 42, on obtient 00101010 et pour -42 on retire 1 (ce qui change les deux derniers bits), puis on remplace chaque 1 par un 0 et vice versa, ce qui donne 11010110.

1. On reconnait 8 en entier non-signé. Comme le premier bit vaut 1, on retire $2^4 = 16$ à cette valeur, ce qui donne -8.

2. Pour les entiers signés, la plage est de $[\![-8, 7]\!]$ et pour les non-signés de $[\![0, 15]\!]$.

Corrigé exo 2.2

0. La solution classique avec un `while`.

```
def base10(n):
    L = []
    while n != 0:
        L.append(n % 10)
        n //= 10
    return L
```

On peut aussi utiliser `str` qui renvoie une chaîne de caractères (pour un nombre, c'est la chaîne de ses chiffres en base dix), et un slicing (pour renverser l'ordre de la liste).

```
def base10(n):
    return [int(k) for k in str(n)[::-1]]
```

1. On utilise la première solution de la question précédente en remplaçant les deux `10` par des `11`.

2. On utilise la fonction précédemment définie.

```
def approx10(p, q, c):
    return base10(p * 10**c // q)
```

Corrigé exo 2.3

Les trois premiers programmes demandés sont des applications directes du cours.

```
def binaire(n):
    L = []
    while n != 0:
        L = [n % 2] + L
        n = n // 2
    return L
```

```
def NombredeUns(n):
    L = binaire(n)
    s = 0
    for k in range(len(L)):
        s += L[k]
    return s
```

```
def palindrome(n):
    L = binaire(n)
    M = L[::-1]
    return L == M
```

Pour la fonction `palindrome` on utilise un slicing pour inverser les éléments de la liste `L`.
Pour trouver les palindromes, on les énumère et on les stocke au fur et à mesure dans une liste `pal`.

```
pal = []
for k in range(101):
    if palindrome(k):
        pal.append(k)
print(pal)
```

Corrigé exo 2.4

0. On applique la formule.

```
def delta(a, b, c):
    return b**2 - 4 * a * c
```

1. On distingue 3 cas selon le signe de Δ.

```
def nombre_solutions(a, b, c):
    d = delta(a, b, c)
    if d > 0:
        return 2
    elif d == 0:
        return 1
    else:
        return 0
```

2. On applique directement les formules. Pour renvoyer les deux valeurs, on renvoie un couple.

```
    def sols(a, b, c):
        rd = delta(a, b, c)**0.5
        r1 = (-b + rd) / 2 / a
        r2 = (-b - rd) / 2 / a
        return r1, r2
```

3. L'équation $x^2 + 1.4x + 0.49$ peut se factoriser en $(x + 0.7)^2 = 0$, on a donc, mathématiquement, une unique solution $x = 0.7$ et un discriminant nul.
On calcule `delta(1, 1.4, 0.49)`, on obtient `-2.220446049250313e-16`, on reconnaît -2^{52}.
Le calcul du discriminant a engendré une petite erreur, égale au epsilon machine.
Lors du calcul de `NombreSolutions(1, 1.4, 0.49)`, le test `d == 0` renvoie `False` car `d` n'est pas tout à fait nul. À cause de cette erreur epsilonesque le résultat final n'est pas celui attendu : on obtient `0` au lieu de `1`.

Le calcul des solutions donne :

```
((-0.7+7.450580596923828e-09j), (-0.7-7.450580596923828e-09j))
```

La petite erreur sur Δ engendre une petite erreur sur la partie imaginaire des solutions.

Lorsqu'un calcul renvoie un résultat complexe avec une partie imaginaire « très petite », il s'agit peut-être d'un réel.

L'équation $x^2 + 0.2x + 0.01$ peut se factoriser en $(x + 0.1)^2 = 0$, on a donc aussi une seule solution $x = 0.1$ et un discriminant nul. Mais cette fois, le calcul de `delta(1, 0.2, 0.01)` donne une erreur dans l'autre sens, on obtient une valeur strictement positive : `6.938893903907228e-18`. Cette valeur est strictement plus petite que l'epsilon machine, car les réels manipulés sont strictement plus petits que 1.
`nombre_solutions(1, 0.2, 0.01)` donne alors `2` au lieu du `1` attendu ; et le calcul des solutions renvoie :

```
(-0.09999999868291098, -0.10000000131708903)
```

Cette fois-ci, il n'y a pas de partie imaginaire, l'erreur se fait sur la partie réelle.

4. Le calcul de `1/4+10**(-20)` donne `0.25` car le `10**(-20)` est trop petit. Le calcul va donc se faire sur l'équation $x^2 + x + \frac{1}{4} = 0$. Tous les calculs sont exacts car $1/4$ tombe juste en base deux. On obtient alors une unique solution (`-0.5`) au lieu de zéro.

Corrigé exo 2.5

0. Il suffit d'appliquer la formule. On évite d'écrire un `if`.

```
def test(a, b, c):
    return (a + b) + c == a + (b + c)
```

1. Par exemple $a = 0.1$, $b = 0.2$ et $c = 0.3$.

2. On teste tous les triplets possibles. On compte avec la variable `d` le nombre de triplets qui conviennent, et on divise le résultat par le nombre total de triplets.

```
d = 0
for a in range(100):
    for b in range(100):
        for c in range(100):
            if test(a / 10, b / 10, c / 10) == False:
                d += 1
print(d / 100**3)
```

Ce code affiche `0.247486`. Il y a environ une chance sur 4 de tomber sur des flottants contredisant l'associativité.
Si on cherche des triplets avec 10 inclus et non exclus, on arrive à une probabilité de `0.253726`.

3. On fait attention à convertir les flottants en chaîne de caractères pour pouvoir les écrire dans le fichier. On n'oublie pas le `"\n"` qui permet de passer à la ligne suivante.

```
def NonAssoc(nf):
    f = open(nf, "w")
    for a in range(100):
        for b in range(100):
            for c in range(100):
```

```
                if test(a / 10, b / 10, c / 10) == False:
                    f.write(str(a / 10) + "; " + str(b / 10) +
                            "; " + str(c / 10) + "\n")
    f.close()
```

Les premières lignes du fichier obtenu sont [1] :

```
0.1
0.1
0.4
0.1
0.1
0.6
0.1
0.1
1.0
0.1
0.1
1.1
0.1
0.1
1.2
```

Corrigé exo 2.6

0. Ce nombre est bien sûr positif et possède un exposant maximal, donc $e = 1023$ ainsi qu'une mantisse constituée uniquement de 1. Il vaut donc

$$x_{\max} = 2^{1023} \sum_{k=0}^{k=52} 2^{-k} = 2^{1023}(2 - 2^{-52}) = 2^{1024} - 2^{971} \approx 8,988 \times 10^{307}$$

1. Ce nombre est toujours positif. Il possède toujours un exposant maximal. La mantisse est désormais composée de 51 chiffres 1 et d'un chiffre 0. On a donc :

$$x_{\max 2} = 2^{1023} \sum_{k=0}^{k=51} 2^{-k} = 2^{1023}(2 - 2^{-51}) = 2^{1024} - 2^{972}$$

2. L'écart absolu est de $\mid x_{\max} - x_{\max 2} \mid = 2^{971}$. Il s'agit de la considérable imprécision intrinsèque au codage des flottants dans cette gamme de nombres.

L'écart relatif est de $\dfrac{\mid x_{\max} - x_{\max 2} \mid}{x_{\max}} = \dfrac{2^{971}}{2^{1024} - 2^{971}} \approx 2^{-53}$.

On retrouve une erreur de l'ordre du epsilon machine. Ce n'est guère surprenant, l'écart relatif entre deux nombres consécutifs étant presque constant et de l'ordre de epsilon machine.

Corrigé exo 2.7

0. On commence par écrire une fonction qui étant donnés les chiffres d'un entier en base soixante retrouve cet entier.

```
def b60(L):
    R = 0
    for k in L:
        R = R * 60 + k
    return R
```

```
def b60(L):
    R = 0
    for k in range(len(L)):
        R = R * 60 + L[k]
    return R
```

1. Selon le mode d'arrondi des flottants, les valeurs peuvent varier

On peut alors écrire la fonction voulue en utilisant la formule donnée dans l'exercice.

```
def bab2float(b):
    return (-1)**b[0] * 60**(b[1] - 35) * b60(b[2:])
```

Le -30 devient -35 car il faut faire passer les cinq derniers chiffres de `b60(b[2:])` après la virgule.

1. Le plus grand flottant babylonien est `[0, 60, 60, 60, 60, 60, 60, 60]` ce qui fait exactement $1.34892561182641002647667081216 \times 10^{55}$.

2. On commence par écrire une fonction `base60` sur le modèle de `base10` et `base11` de l'exercice 2.2.
Ensuite, on peut écrire la fonction.

```
def int2bab(n):
    # On ajoute 6 zéros au cas où n ai moins de 6 chiffres.
    chiffres = base60(n)[::-1] + [0] * 6
    # -7 car il y a les 6 zéros rajoutés et 1 chiffre à gauche de la virgule.
    e = len(chiffres) - 7
    s = 0 if n > 0 else 1  # si n>0 alors s=0 sinon s=1
    return [s, e + 30] + chiffres[0:6]
```

3. La principale difficulté consiste à aligner les deux nombres (ils n'ont pas nécessairement le même exposant). On rajoute des zéros devant le nombre le plus petit. Ensuite, on applique l'algorithme de l'école primaire.

```
def somme(b1, b2):
    ecart = b1[1] - b2[1]  # L'écart entre les exposants
    d1 = max(0, -ecart)  # Le décalage de chiffres à faire sur b1
    d2 = max(0, ecart)
    bm1 = [0] * d1 + b1[2:]  # La mantisse décalée de b1
    bm2 = [0] * d2 + b2[2:]
    e3 = max(b1[1], b2[1])
    retenue = 0  # La retenue à propager
    bm3 = [0] * 6  # La mantisse de la somme (initialisée à zéro)
    for k in range(5, -1, -1):
        calcul = bm1[k] + bm2[k] + retenue
        retenue = calcul // 60
        bm3[k] = calcul % 60
    if retenue != 0:  # Si les deux chiffres tout à gauche provoquent une retenue
        bm3 = [retenue] + bm3[:5]  # La retenue crée un nouveau chiffre
        e3 += 1  # La somme gagne une puissance de 60
    return [0, e3] + bm3
```

Corrigé exo 2.8

0. 0.00011001100110011001100 1

1. 3.58×10^{-8}

2. $y - z = 2^{-24} \approx 5.96 \times 10^{-8}$ et $0,1 - z \approx 9.54 \times 10^{-8}$.

3. On compte le nombre de dixièmes de secondes qu'il y a dans huit heures.
$e_8 = 8 \times 60 \times 60 \times 10\varepsilon = 288000\varepsilon$ et $\mathrm{e}_{100} = 3.6 \times 10^6$

4. $e_8 \approx 0.027$ et $e_{100} \approx 0.34$

5. 584m

Corrigé exo 2.9

0. On fait attention à n'utiliser que des flottants de 16 bits.

```
def S1(n):
    S = np.float16(0)
    for k in range(1, n + 1):
        S += np.float16(1 / k**2)
    return S
```

1. On utilise un `range` décroissant avec un pas de `-1`.

```
def S2(n):
    S = np.float16(0)
    for k in range(n, 0, -1):
        S += np.float16(1 / k**2)
    return S
```

2. L'erreur d'une somme est proportionnelle au résultat de la somme. La fonction `S1` commence par sommer les termes les plus gros de la somme, elle fait donc une plus grosse erreur de calcul que `S2`. On en déduit que `f` est `S2` et que `g` est `S1`. math.pi**2/6-S1b(100)

3. On constate toujours une différence entre `f(100)` et `g(100)` mais elle est beaucoup plus petite (elle est de l'ordre du epsilon machine).

Corrigé exo 2.10

0. On imbrique deux boucles `for` (une par dé).

```
L = [0] * 7
for k in range(1, 7):
    for p in range(1, 7):
        L[max(k, p)] += 1
Loi = [t / 36 for t in L]
```

1. On adapte l'algorithme classique de changement de base.

```
def dés(N, n):
    L = []
    for k in range(n):
        L.append(N % 6 + 1)
        N = N // 6
    return L
```

2. On teste tous les cas.

```
def loi_max_dés(n):
    L = [0] * 7  # Les valeurs vont de 0 (valeur interdite) à 6 inclus.
    for k in range(6**n):
        # On compte le nombre d'issues favorables pour chaque valeur de 1 à 6.
        L[max(des(k, n))] += 1
    # On divise le nombre d'issues favorables par le nombre total d'issues
    return [x / 6**n for x in L]
```

3. On généralise la fonction précédente.

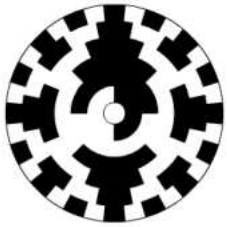
```
def loi_f(n, f, M):
    L = [0] * (M+1)
    for k in range(6**n):
        L[f(des(k, n))] += 1
    return [x / 6**n for x in L]
```

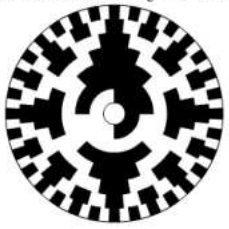
Corrigé exo 2.11

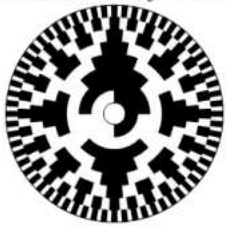
0. La table de vérité est la suivante, un seul bit ne change à chaque fois. On remarque une symétrie au niveau du code de Gray entre la ligne 1 et la ligne 8 si on enlève le bit de poids fort. On retrouve cette symétrie verticale sur le codeur, quel que soit le nombre de bits présents.

Pos	x	y	z	t	d	c	b	a
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	1	0	0	1	0
3	0	0	1	0	0	0	1	1
4	0	1	1	0	0	1	0	0
5	0	1	1	1	0	1	0	1
6	0	1	0	1	0	1	1	0
7	0	1	0	0	0	1	1	1
8	1	1	0	0	1	0	0	0
9	1	1	0	1	1	0	0	1
10	1	1	1	1	1	0	1	0
11	1	1	1	0	1	0	1	1
12	1	0	1	0	1	1	0	0
13	1	0	1	1	1	1	0	1
14	1	0	0	1	1	1	1	0
15	1	0	0	0	1	1	1	1


Codeur Gray 4 bits


Codeur Gray 6 bits


Codeur Gray 7 bits


Codeur Gray 8 bits

1. Le capteur possède N pistes, il y a donc 2^N valeurs pour 360°. La résolution du capteur est donc $\dfrac{360}{2^N}$.

2. On souhaite une résolution $r = 0,05 = \dfrac{360}{2^N}$, on trouve donc :

$$2^N > \frac{360}{0,05} = 7200 \Rightarrow N = 13\text{bits}$$

En effet, $2^{12} = 4096 < 7200 < 2^{13} = 8192$. On aura donc une résolution réelle de $0,044°$.

3. Voici deux fonctions qui affichent le code de Gray d'un nombre (décimal, binaire ou autre).

```
def bintogray(x):
    print(bin(x ^ (x >> 1)))
    return
```

```
def bintogray(x):
    print(bin(x ^ (x // 2)))
    return
```

Corrigé exo 2.12

0. -15

1. Pour les entiers de zéro à 5, on a les représentations `''`, `'1'`, `'1z'`, `'10'`, `'11'`, `'1zz'`.
Pour les entiers de -1 à -5, on a les représentations `'z'`, `'z1'`, `'z0'`, `'zz'`, `'z11'`.

2. 10 est représenté par `'101'` et 19 par `'1z01'`.

3. Un entier positif est soit nul (chaîne de caractères vide) soit strictement positif auquel cas son premier chiffre est 1.

```
def positif(t):
    return t == "" or t[0] == "1"
```

4. Il suffit de remplacer les `"z"` par des `"0"` et vice-versa.

```
def oppose(t):
    t2 = ""
    for k in t:
        if k == "1":
            t2 += "z"
        elif k == "z":
            t2 += "1"
        else:
            t2 += "0"
    return t2
```

5. On écrit une fonction `chiffre(c)` qui prend un chiffre (`"0"`, `"1"` ou `"z"`)

```
def chiffre(c):
    return ["z", "0", "1"].index(c) - 1


def convert(t):
    v = 0
    for k in t:
        v = v * 3 + chiffre(k)
    return v
```

6. On adapte la fonction `base11` de l'exercice 2.2 à la base trois. Dans le cas où le chiffre en base 3 « normale »vaut 2, on utilise le chiffre -1, ce qui crée un décalage de $2-(-1)$ avec `n`, on lui ajoute donc 3.

```
def b3b(n):
    s = ""
    while n!= 0:
        d = n % 3  # Le dernier chiffre en base 3 normale
        if d == 2:
            d = "z"
            n = n + 3
        n //= 3
        s = str(d) + s
    return s
```

CHAPITRE 3

Algorithmique

L'essentiel du cours

Un algorithme est l'idée d'un programme. Dans un algorithme, on fait abstraction du langage de programmation (Python, C, Ocaml,...) et des détails d'implémentation. Un algorithme doit être décrit avec suffisamment de précision en français pour pouvoir être traduit (on dit aussi implémenté) dans n'importe quel langage de programmation. Ce chapitre s'intéresse aux propriétés des algorithmes, son contenu ne dépend donc pas du fait que nous programmons en Python.

■ 0 Preuve d'algorithme

Dans cette partie, nous considérons l'exemple ci-après. C'est une fonction qui prend en argument une liste et qui remplace toutes ses valeurs par des zéros. On s'intéresse à prouver que cette fonction est « correcte », c'est-à-dire qu'elle ne va pas boucler à l'infini et qu'elle fait bien ce qu'on attend d'elle.

```
def zero(L):
    k = 0
    while k < len(L):
        L[k] = 0
        k += 1
```

Définition

Étant donnée une boucle `while`, on appelle **variant de boucle** toute quantité v telle que :

- $v \in \mathbb{N}$
- v décroit strictement à chaque passage dans la boucle.

Trouver un variant de boucle démontre que la boucle termine (car sinon il existerait une suite infinie strictement décroissante d'entiers naturels, ce qui est impossible).
Sur l'exemple `len(L) - k` est un variant de boucle.

Définition

On appelle **invariant de boucle** une propriété P telle que :

- P est vraie au début du premier passage dans la boucle.
- Si P est vraie au début d'un passage de la boucle, alors elle est encore vraie à la fin.

Avec ces conditions, il est évident que l'invariant sera encore vrai à la fin de la boucle. Un invariant de boucle bien choisi permet de démontrer qu'un programme fait bien ce qu'on attend de lui.
Sur l'exemple, la propriété P « La sous-liste `L[0:k]` ne contient que des zéros » est un invariant de boucle :

- Au début du premier passage de la boucle, la sous-liste est vide donc P est vraie.
- Si P est vraie, alors `L[0:k]` ne contient que des zéros. Après l'instruction `L[k] = 0`, la sous-liste `L[0:k+1]` ne contient que des zéros, et donc après l'instruction `k += 1`, P est vraie.

À la fin de la boucle, `k` vaut `len(L)`, on en déduit que `L[0:len(L)]` ne contient que des zéros et donc que `L` ne contient que des zéros. On a donc démontré que l'algorithme fait bien ce qu'il est censé faire.

■ 1 Complexité

La complexité est une mesure des ressources nécessaires à un calcul. Nous considèrerons ici deux ressources : le temps et la mémoire.
Nous considèrerons par la suite l'exemple suivant :

```
1  def somme(L):
2      S=0
3      for k in range(len(L)):
4          S = S + L[k]
5      return S
```

La fonction `somme` calcule la somme des éléments d'une liste.

Définition

La **complexité en temps** d'un calcul est le nombre d'opérations élémentaires nécessaires pour faire le calcul.

Le calcul de `somme([1, 2, 3])` nécessite à priori 4 opérations élémentaires (1 fois la ligne 2, et trois fois la ligne 4). Toutefois, la notion d'opération élémentaire n'est pas précisément définie : on pourrait considérer par exemple que la ligne 4 fait non pas une, mais 2 opérations élémentaires (une somme puis une affectation) et alors le calcul de `somme([1, 2, 3])` nécessiterait 7 opérations élémentaires.
Cette imprécision est sans conséquence car on estime la complexité « à la louche ». Pour définir formellement ce que signifie ce « à la louche », nous introduisons les trois notations suivantes :

Définition

Étant donné deux suites positives T_n et u_n, on dit que

- T_n est un grand $\mathcal{O}$ de u_n s'il existe une constante k_2 telle que pour tout n assez grand $T_n \leqslant k_2 \cdot u_n$; on note $T_n = \mathcal{O}(u_n)$.
- T_n est un grand Ω de u_n s'il existe une constante $k_1 > 0$ telle que pour tout n assez grand $k_1 \cdot u_n \leqslant T_n$; on note $T_n = \Omega(u_n)$
- T_n est un grand Theta de u_n s'il existe deux constantes $k_1 > 0$ et k_2 telles que pour tout n assez grand $k_1 \cdot u_n \leqslant T_n \leqslant k_2 \cdot u_n$; on note $T_n = \Theta(u_n)$.

Le grand $\mathcal{O}$ sert à majorer la complexité, et le grand Ω à la minorer. Nous utiliserons le plus souvent le grand $\mathcal{O}$.

Définition

Étant donné un algorithme, on définit T_n comme étant le maximum des complexités des calculs de l'algorithme sur une entrée de taille n.
T_n est appelée **complexité au pire en temps** de l'algorithme.

On définit la complexité au meilleur en remplaçant maximum par minimum dans la définition précédente.
La complexité sera généralement exprimée sous la forme $\mathcal{O}(u_n)$ avec u_n une suite simple, ce qui nous affranchit d'une définition précise de l'opération élémentaire.
Les complexités les plus fréquentes seront en $\mathcal{O}(n)$ (complexité linéaire), $\mathcal{O}(n^2)$ (complexité quadratique), $\mathcal{O}(\ln(n))$ (complexité logarithmique) et $\mathcal{O}(2^n)$ (complexité exponentielle).
Sur le même modèle que la complexité en temps, on définit la complexité en mémoire.

Définition

La **complexité en mémoire** d'un calcul est la taille de la mémoire (en octets ou en bits) prise par les valeurs intermédiaires du calcul (on exclut l'entrée et la sortie de l'algorithme du calcul, sauf si elles sont accédées en lecture et en écriture).

Définition

Étant donné un algorithme, on définit E_n comme étant le maximum des complexités en mémoire des calculs de l'algorithme sur une entrée de taille n.
E_n est appelée **complexité au pire en mémoire** (ou en espace) de l'algorithme.

On définit la complexité au meilleur en remplaçant maximum par minimum dans la définition précédente.

Taille des entiers

Dans la mémoire de l'ordinateur, un entier k a une taille proportionnelle au nombre de ses chiffres, c'est-à-dire de l'ordre de $\log_2(k)$. Mais, pour simplifier les calculs, on supposera pour la complexité en mémoire qu'un entier n'occupe qu'une place $\mathcal{O}(1)$.
De plus, lorsqu'un algorithme prend en entrée un entier k, on n'exprimera pas la complexité en fonction de la taille de k (c'est-à-dire en fonction de $n = \log_2(k)$) mais en fonction de k.

Les méthodes à maîtriser

Méthode 3.0 : Comment déterminer la complexité d'un algorithme ?

- Poser $n =$ « la taille des paramètres ».
- Déterminer la complexité de chaque instruction.
- Pour chaque boucle, déterminer combien de fois on passe dans la boucle.
- Conclure

Sauf précision contraire, la complexité demandée est la complexité au pire en temps.
Sur la fonction `somme` de la partie 1, on a $n =$ `len(L)`, les lignes 2 et 4 se font en temps constant (en $\mathcal{O}(1)$) et on passe n fois dans le `for`. La complexité est donc en $T_n = \underbrace{\mathcal{O}(1)}_{\text{ligne 2}} + \underbrace{n \times \underbrace{\mathcal{O}(1)}_{\text{ligne 4}}}_{\text{for}} = \mathcal{O}(n)$.

Considérons un exemple plus compliqué : la fonction `neufs` définie ci-à-côté.
Cette fonction calcule de manière fort peu astucieuse le plus grand nombre de 9 consécutifs dans l'écriture en base 10 de `n` ; mais nous n'avons pas besoin de le savoir pour estimer sa complexité.
On constate que chacune des instructions utilisées est en $\mathcal{O}(1)$ (temps constant). On utilise bien `max` qui a une complexité linéaire, mais on ne l'utilise que sur 2 valeurs.

```
def neufs(n):
    L = []
    while n!= 0:
        L.append(n % 10)
        n = n // 10
    M = 0
    for k in range(len(L)):
        i = k
        while i < len(L) and L[i] == 9:
            i += 1
        M = max(M, i - k)
    return M
```

On passe dans le premier `while` une fois par chiffre de `n` en base 10
(en effet, l'opération `n = n//10` fait perdre un chiffre à `n`), soit environ $\log_{10}(n)$ fois.
La liste `L` contient alors (à moins de 1 près) $\log_{10}(n)$ valeurs. On passe dans le `for` $\log_{10}(n)$ fois, idem pour le second `while`.
La complexité est donc de

$$T_n = \underbrace{\log_{10}(n) \times \mathcal{O}(1)}_{\text{premier while}} + \underbrace{\log_{10}(n) \times \log_{10}(n) \times \mathcal{O}(1)}_{\text{boucles imbriquées}} = \mathcal{O}(\log_{10}^2(n)) = \mathcal{O}(\ln^2 n).$$

Méthode 3.1 : Comment déterminer la complexité en mémoire d'un algorithme ?

- Poser $n =$ « la taille des paramètres ».
- Déterminer la place prise en mémoire par chaque variable.
- Conclure

Pour la fonction `somme` définie dans le cours, on constate que si la liste `L` contient des entiers ou des flottants, chaque variable intermédiaire (`k` et `S`) contient un type de base, donc demande une place en mémoire en $\mathcal{O}(1)$ et donc la complexité en mémoire au pire est en $\mathcal{O}(1)$.
Pour la fonction `neufs` définie précédemment, la complexité en mémoire vient de la variable `L` (toutes les autres variables demandent $\mathcal{O}(1)$ en mémoire). Or cette variable va contenir environ $\log_{10}(n)$ éléments d'où une complexité en mémoire de $\mathcal{O}(\log_{10}(n)) = \mathcal{O}(\ln(n))$.

Méthode 3.2 : Comment démontrer qu'un algorithme termine ?

- Exhiber, pour chaque boucle `while`, un variant de boucle.
- Démontrer, si ce n'est pas trivial, chaque variant de boucle (dans la plupart des cas cette étape n'est pas nécessaire)

Pour la fonction `neufs`, `n` est un variant pour le premier `while`, et `len(L) - i` est un variant pour le second.

Méthode 3.3 : Comment démontrer qu'un algorithme fait bien ce qu'on attend de lui ?

Pour chaque boucle :

- Exhiber un invariant de boucle qui permet de montrer le résultat voulu.
- Sauf si c'est trivial (ce qui est très souvent le cas) démontrer cet invariant.
- Exprimer ce que veut dire chaque invariant à la fin de la boucle.

Pour la fonction `somme` du cours, on constate $\mathtt{S} = \sum_{p=0}^{\mathtt{k}-1} \mathtt{L}[p]$. On en déduit qu'à la fin de l'algorithme, comme `k` vaut `len(L)`, on a $\mathtt{S} = \sum_{p=0}^{\mathtt{len(L)}-1} \mathtt{L}[p]$. La fonction calcule bien la somme des éléments de la liste.

Méthode 3.4 : Comment démontrer qu'un algorithme numérique calcule bien ce qu'on veut ?

Pour chaque boucle :

- Chercher une grandeur numérique qui reste constante à chaque passage dans la boucle.
- Exprimer cette grandeur au début et à la fin de la boucle.
- En déduire une relation entre les valeurs initiales et finales des variables.

Considérons le code suivant.

```
def euclide(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

On constate que le pgcd de `a` et `b` reste constant. De plus, à la fin, `b` vaut 0 et donc $\text{pgcd}(\mathtt{a}, \mathtt{b}) = \mathtt{a}$. On en déduit qu'à la fin de l'algorithme, la valeur de `a` renvoyée est égal au pgcd des valeurs initiales de `a` et `b`. Cette fonction calcule donc le pgcd.

Vrai/Faux sur le cours

0. Une complexité en $\Theta(n)$ est toujours mieux qu'une complexité en $\Theta(n^2)$. ☐ **Vrai** ☐ **Faux**

1. Une boucle `while` peut ne jamais terminer. ☐ **Vrai** ☐ **Faux**

2. Une boucle `for` a toujours une complexité en $\mathcal{O}(n)$. ☐ **Vrai** ☐ **Faux**

3. La copie d'une liste de taille n prend un temps $\mathcal{O}(n)$. ☐ **Vrai** ☐ **Faux**

```
def myst(n):
    N = n + 2
    while n!= 0:
        if N == n: n = n - 1
        else: N = N - 1
```

4. `n` est un variant de boucle du `while` de `myst`. ☐ **Vrai** ☐ **Faux**

5. `N` est un variant de boucle du `while` de `myst`. ☐ **Vrai** ☐ **Faux**

6. `N+n` est un variant de boucle du `while` de `myst`. ☐ **Vrai** ☐ **Faux**

7. `N*n` est un variant de boucle du `while` de `myst`. ☐ **Vrai** ☐ **Faux**

8. `myst` renvoie toujours `None`. ☐ **Vrai** ☐ **Faux**

9. `n <= N` est un invariant de boucle du `while` de `myst`. ☐ **Vrai** ☐ **Faux**

10. Il existe toujours un invariant de boucle. ☐ **Vrai** ☐ **Faux**

```
def test(L):
    b = True
    for k in range(1, len(L)):
        b = b and L[k - 1] < L[k]
    return b
```

11. La fonction `test` termine toujours. ☐ **Vrai** ☐ **Faux**

12. « `b` $\Leftrightarrow$ les éléments de `L[0:k]` sont classés dans l'ordre strictement croissant » est un invariant de boucle. ☐ **Vrai** ☐ **Faux**

```
import math

def f(n):
    S = 0
    for i in range(n**2):
        for j in range(int(math.log2(n))):
            S = S + (-1)**(i + j)
    return S
```

13. `f(n)` a une complexité en $\mathcal{O}(n^2 \ln(n))$. ☐ **Vrai** ☐ **Faux**

14. « $S = \sum_{i=0}^{n} \sum_{k=0}^{j} (-1)^{i+j}$ » est un invariant de la boucle extérieure. ☐ **Vrai** ☐ **Faux**

15. « $S = \sum_{k=0}^{i} \sum_{j=0}^{\lfloor \log_2(n) \rfloor} (-1)^{i+j}$ » est un invariant de la boucle extérieure. ☐ **Vrai** ☐ **Faux**

Exercices

Exercice 3.0

Considérons les deux fonctions suivantes.

```
def truc(n):
    for j in range(n):
        for k in range(n):
            pass
```

```
def machin(n):
    for j in range(n):
        pass
    for k in range(n):
        pass
```

Quelle est la complexité de chacune de ces fonctions ?

Exercice 3.1 Recherche du maximum

On souhaite écrire une fonction `maximum` renvoyant le maximum d'une liste (la fonction `max` existe déjà en Python, on souhaite ici la reprogrammer). Le code incomplet de la fonction est donné ci-après.

```
def maximum(L):
    M = L[0]
    for k in range(1, len(L)):
        # A compléter
    return M
```

0. Compléter le code en préservant l'invariant suivant : « `M` est le maximum de la liste `L[:k]` ».
1. Modifier la fonction pour remplacer `for k in range(1, len(L)) :` par `for x in L :`
2. Quelle est la complexité de la fonction `maximum` ?

Exercice 3.2 Multiplication égyptienne

Considérons le programme suivant, qui implémente un ancien algorithme égyptien. Dans cet exercice, `a` et `b` sont supposés être des entiers positifs.

```
def Egyptienne(a, b):
    t = 0
    while a > 0:
        if a % 2 == 1:
            t = t + b
        b = 2 * b
        a = a // 2
    return t
```

0. Montrer que la fonction `Egyptienne` termine toujours.
1. Détailler l'exécution de `Egyptienne(41, 3)`
2. Montrer, à l'aide d'un invariant de boucle, que la fonction renvoie le produit entre les deux arguments.
3. Quelle est la complexité de cette fonction ?

Exercice 3.3 Algorithme d'Euclide

Considérons l'algorithme suivant, qui prend en entrée deux entiers naturels a et b :

Tant que $b \neq 0$

(0) r prend pour valeur le reste de la division euclidienne de a par b.
(1) a prend la valeur de b.
(2) b prend la valeur de r.

Renvoyer a

0. Implémenter cet algorithme en Python.
1. Pourquoi cette fonction ne lève jamais l'exception `ZeroDivisionError` ?
2. Montrer que l'algorithme termine toujours.
3. Montrer à l'aide d'un invariant de boucle que l'algorithme calcule le pgcd de a et b.
4. Montrer qu'à la fin de l'étape (0), si $b < a$ alors $r \leqslant a/2$.
5. En déduire que la complexité de l'algorithme est en $\mathcal{O}(\ln(n))$ avec $n = \max(a, b)$.

Exercice 3.4

Dans cet exercice, nous considérons la fonction `neuf` donnée en exemple dans les méthodes.

0. Modifier la fonction pour avoir une complexité en temps en $\mathcal{O}(\ln(n))$.
1. Modifier la fonction pour avoir une complexité en mémoire en $\mathcal{O}(1)$.

Exercice 3.5

```
def cherche(s, m):
    for k in range(len(s) - len(m) + 1):
        b = True
        for i in range(len(m)):
            if s[k + i] != m[i]:
                b = False
        if b:
            return True
    return False
```

```
def cherche2(s, m):
    for k in range(len(s) - len(m) + 1):
        if s[k:k + len(m)] == m:
            return True
    return False
```

0. Quelle est la complexité de la fonction `cherche` ?
1. Quelle est la complexité de la fonction `cherche2` ?
2. Quelle est la complexité au meilleur de la fonction `cherche` ?
3. Que fait la fonction `cherche` ?
4. Le justifier à l'aide de deux invariants de boucle (un pour chaque boucle `for`).

Exercice 3.6 Algorithme S

L'algorithme S sert à tirer au hasard (uniformément) n entiers différents parmi l'ensemble $[\![1, N]\!]$ des entiers compris entre 1 et N. Il prend donc en argument deux entiers n et N tels que $0 \leqslant n \leqslant N$. Voici le détail de l'algorithme :

On met dans la variable R un ensemble vide.
Tant que $N > 0$:

(0) Tirer à pile ou face avec une probabilité de pile de n/N.
(1) Si vous avez obtenu pile, soustraire 1 à n et ajouter N à l'ensemble R.
(2) Soustraire 1 à N.

0. Implémenter cet algorithme en Python. Pour R, utiliser une liste. Pour tirer à pile ou face, utiliser la fonction `randint` ou la fonction `random` de la bibliothèque `random`.
1. Montrer que cet algorithme termine toujours.
2. Expliquer ce que fait cet algorithme.
3. Quelle est la complexité au pire et au meilleur en temps de cet algorithme ?

Pour poursuivre l'étude avec une version récursive, voir **7.14** p. 291

Exercice 3.7 Recherche dichotomique

0. Écrire, à l'aide de la bibliothèque `random`, une fonction `rand_list(N, t)` qui crée une liste aléatoire[1] de `t` entiers compris entre 0 et `N`, puis trie la liste (avec la fonction[2] `sorted`) avant de la renvoyer.
Par exemple, `rand_liste(60, 20)` pourrait renvoyer :

```
33 [2, 5, 7, 21, 23, 27, 28, 29, 29, 32, 33, 34, 49, 50, 50, 54, 54, 56, 59, 59]
```

Cette fonction sera utile pour tester les fonctions suivantes.
On se propose d'écrire une fonction `recherche(L, a)` qui recherche l'élément `a` dans la liste `L` et qui renvoie l'indice de l'élément `a` ou -1 si l'élément ne figure pas dans la liste.

1. Écrire cette fonction qui recherche l'élément linéairement en parcourant potentiellement **tous** les éléments de la liste. Cette fonction est utilisable même si la liste `L` n'est pas croissante.
2. On se propose d'écrire une fonction `recherchedicho(L, a)` qui recherche l'élément `a` dans la liste **triée** `L` en utilisant le procédé de **dichotomie** : on regarde si `a` est supérieur ou inférieur à l'élément (à peu près) au milieu de la liste et suivant le résultat on poursuit la recherche dans la sous-liste de gauche ou dans la sous-liste de droite de cet élément. On converge alors rapidement vers l'élément (ou bien vers son emplacement théorique s'il n'est pas présent dans la liste).
Écrire cette fonction. Elle devra renvoyer la valeur -1 si l'élement `a` n'est pas présent dans `L`.
3. Quelle est la complexité en nombre de tests sur les éléments de la liste des deux fonctions précédentes ?

Exercice 3.8 Exponentiation rapide

On considère le programme Python suivant :

```
def Puissance(a, n):
    A = a
    N = n
    R = 1
    while N > 0:
        if N\% 2 == 0:
            A = A**2
            N = N // 2
        else:
            R = R * A
```

1. La méthode 5.3 page 197 permet de résoudre plus facilement ce problème grâce à la bibliothèque `numpy.random`.
2. La fonction `sorted` utilise l'algorithme *tim sort* pour trier la liste. Cet algorithme est un mélange astucieux des algorithmes de tri rapide et de tri fusion décrits au chapitre 8.

```
        N = N - 1
    return R
```

0. Montrer que cette fonction s'arrête.
1. Prouver à l'aide d'un invariant de boucle que cette fonction calcule bien a^n
2. Quelle est la complexité (on ne comptera que les itérations de la boucle while) de cette fonction ? Justifier alors le nom de cet algorithme.

Exercice 3.9 Alignement de séquences génétiques (prog. dynamique)

Une séquence d'ADN est une suite de désoxyribonucléotides (A, C, G ou T), que l'on peut donc représenter par un mot écrit sur l'alphabet $\{A, C, G, T\}$. Une telle séquence peut subir trois mutations élémentaires :

- la *délétion* : suppression d'un nucléotide ;
- l'*insertion* : ajout d'un nucléotide ;
- la *substitution* : modification d'un nucléotide.

Étant donnés deux séquence d'ADN u et v, nous cherchons le scénario le plus vraisemblable expliquant la mutation de u en v. Ainsi, pour $u_0 = AACG$ et $v_0 = TAG$, un scénario possible est : substitution de la première lettre de u_0 en la lettre T, pour donner la séquence $TACG$, puis délétion de la lettre C. Chaque lettre du mot u ne doit subir qu'une mutation et un tel scénario peut être codé en alignant les séquences u et v, en introduisant des symboles $-$ pour représenter les délétions et les insertions. Voici quelques exemples de scénarios pour les mots u_0 et v_0 :

$$\begin{array}{cccc} A & A & C & G \\ T & A & - & G \end{array} \qquad \begin{array}{ccccc} - & A & A & C & G \\ T & A & G & - & - \end{array} \qquad \begin{array}{cccccc} A & A & C & G & - & - \\ - & T & - & - & A & G \end{array}$$

Autrement-dit, un scénario peut être vu comme un couple $(\tilde{u}, \tilde{v})$ de mots sur l'alphabet $\{A, B, C, T, -\}$ avec les règles suivantes :

- $\tilde{u}$ et $\tilde{v}$ sont de même longueur ;
- on retrouve u (resp. v) si on supprime de $\tilde{u}$ (resp. de $\tilde{v}$) les lettres $-$;
- les lettres $-$ de $\tilde{u}$ et de $\tilde{v}$ ne sont jamais alignées.

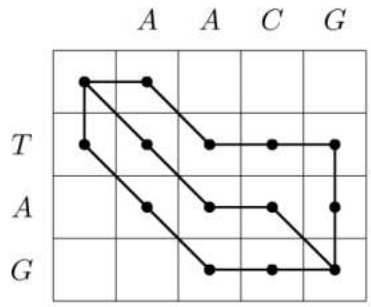
Nous allons définir le score d'un scénario, en utilisant le vocabulaire anglo-saxon usuel :

- un *match* désigne le cas où deux lettres alignées sont égales et ajoute 2 points au score ;
- un *mismatch* désigne le cas où deux lettres alignées sont distinctes ; il fait perdre un point.
- un *gap* désigne le cas où il y a délétion ou insertion et il fait perdre 2 points.

Ainsi, nos scénarios ont les scores respectifs 1 (deux matchs, un mismatch et un gap), -5 (un match, un mismatch et trois gaps) et -11 (un mismatch et 5 gaps). Le but de cet exercice est de construire un scénario optimal, c'est-à-dire de score maximal. Ce score maximal, noté $d(u, v)$, pourra jouer le rôle de distance du mot u au mot v.

0. Écrire une fonction `score` qui, appliquée à $\tilde{u}$ et $\tilde{v}$, renvoie le score du scénario $(\tilde{u}, \tilde{v})$.

Nous pouvons représenter les scénarios graphiquement, comme ci-contre, dans le cas particulier des séquences u_0 et v_0. Un scénario est codé par un chemin allant de la case en haut à gauche à la case en bas à droite, dans lequel trois déplacements élémentaires sont permis : vers la droite (délétion), vers le bas (insertion), ou en diagonale descendante (match ou mismatch). Les trois chemins tracés correspondent à nos trois scénarios.



1. Si n et p sont les longueurs respectives des séquences u et v, on note $s(n,p)$ le nombre de scénarios possibles pour passer de u à v. Montrer que l'on a :

$$\forall n, m \in \mathbb{N}, s(n,m) = \begin{cases} 1 & \text{si } n = 0 \text{ ou } m = 0 \\ s(n-1,m) + s(n-1,m-1) + s(n,m-1) & \text{sinon} \end{cases}$$

Écrire une fonction qui calcule $s(n,m)$. Cette fonction construira une matrice qui contiendra, à la fin du calcul, les valeurs $s(i,j)$ pour $0 \leqslant i \leqslant n$ et$0 \leqslant j \leqslant m$. Quelle est la complexité en temps et en place de votre fonction ?
Combien existe-t-il de scénarios différents pour les séquences u_0 et v_0 ? Et pour des séquences de longueur 30 et 45 ?

2. Modifier la fonction précédente pour que sa complexité en place soit en $\mathcal{O}(\min(n,m))$.

Le nombre de scénarios différents étant beaucoup trop grand, il n'est pas raisonnable de chercher un scénario optimal en testant tous les scénarios possibles. Nous allons mettre en place un algorithme efficace, de type **programmation dynamique**, décrit en 1970 par Needleman et Wunsch [**NW70**]. On fixe deux séquences $u = u_0u_1 \dots u_{n-1}$ et $v = v_0v_1 \dots v_{m-1}$ de longueurs respectives n et m et on pose, pour $0 \leqslant j \leqslant n$ et $0 \leqslant i \leqslant m$:

$$\delta(i,j) = d(u_0u_1 \dots u_{j-1}, v_0v_1 \dots v_{i-1})$$

Autrement-dit, $\delta(i,j)$ est la distance du préfixe $u[:j]$ de u au préfixe $v[:i]$ de v.

3. Que vaut $\delta(i,j)$ quand $i = 0$ ou $j = 0$?

4. Pour $1 \leqslant i \leqslant m$ et $1 \leqslant j \leqslant n$, donner une expression de $\delta(i,j)$ en fonction de $\delta(i-1,j)$, $\delta(i-1,j-1)$ et $\delta(i,j-1)$. Calculer $d(u_0,v_0)$.

5. Écrire une fonction `distance` qui, quand on l'applique à deux séquences u et v, renvoie la distance de u à v. Cette fonction construira une matrice δ qui contiendra, à la fin du calcul, les valeurs $\delta(i,j)$. Quelle est la complexité en temps et en place de votre fonction ?

Nous souhaitons maintenant construire l'alignement correspondant à un scénario optimal. Pour cela, on peut remonter dans la matrice δ en retrouvant le chemin correspondant aux scores $\delta(i,j)$. Plus précisément, partant de la case (m,n) de score $\delta(m,n)$, on sait que $\delta(m,n) = \delta(m-1,n-1)+1$ (ou $+2$), ou que $\delta(m,n) = \delta(m-1,n)-1$, ou que $\delta(m,n) = \delta(m,n-1)-1$; on peut donc construire les dernières lettres de $\tilde{u}$ et de $\tilde{v}$, et ainsi de suite. Il est cependant dommage de rechercher la direction à suivre alors que cela a déjà été fait au moment du remplissage de la matrice δ.

6. Écrire une fonction `alignement` qui, appliquée à deux séquences u et v, renvoie les mots $\tilde{u}$ et $\tilde{v}$ représentant un scénario optimal pour u et v, ainsi que la distance de u à v. On utilisera toujours une matrice δ, dans laquelle on stockera, en plus de $\delta(i,j)$, un élément indiquant la direction à suivre pour remonter dans la matrice.

Exercice 3.10 Orbitales atomiques

Les électrons d'un atome se répartissent en plusieurs couches électroniques qui peuvent elles-mêmes se décomposer en plusieurs sous-couches. Les couches sont caractérisées par le nombre quantique principal $n \in \mathbb{N}^\star$. Les sous-couches sont caractérisées en plus par un entier ℓ tels que $0 \leqslant \ell < n$. Chaque sous-couche peut contenir jusqu'à $2(2\ell + 1)$ électrons.

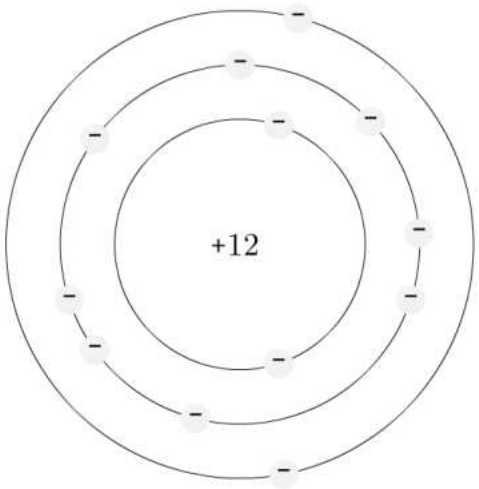


FIGURE 0. L'atome de magnésium et ses 3 couches électroniques.

Chaque sous-couche est notée avec la valeur de n suivi d'une lettre indiquant la valeur de ℓ. L'ordre des lettres est spdfgh, ainsi, 1s correspond à $n = 1$ et $\ell = 0$ et 4d correspond à $n = 4$ et $\ell = 2$.

La règle de Klechkowski dit que les sous-couches se remplissent dans l'ordre indiqué par la figure 1 : par $n + \ell$ croissant, et, en cas d'égalité, par n croissant. Chaque sous-couche est totalement remplie avant que la sous-couche suivante ne puisse obtenir un électron. La configuration d'un atome sera notée par les noms [1] des sous-couches suivis, chacun, du nombre d'électrons dans la sous-couche. Par exemple le silicium (14 électrons) a pour configuration 1s2 2s2 2p6 3s2 3p2 et le magnésium a pour configuration 1s2 2s2 2p6 3s2. On remarquera que la dernière sous-couche du silicium n'est pas totalement remplie, elle ne contient que 2 électrons pour un maximum de 6.

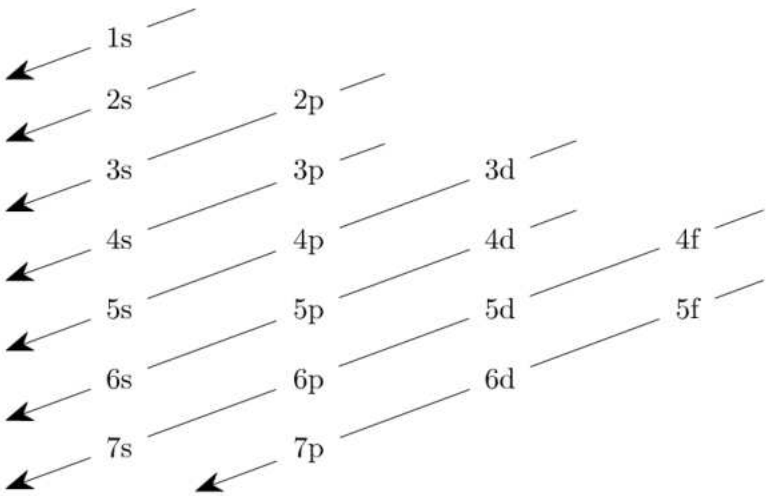


FIGURE 1. Ordre de remplissage des sous-couches

1. On écrira les sous-couches par n croissant puis par ℓ croissant.

0. Écrire une fonction `couche_suivante(n, l)` qui étant donnée une sous-couche `n,l` donne la sous-couche suivante. Ainsi `couche_suivante(4, 2)` renvoie `5, 1` et `couche_suivante(4, 0)` renvoie `3, 2`.

1. Écrire une fonction `souscouches(Z)` qui, étant donné un numéro atomique[1] Z, renvoie une liste de listes `C` telle que :

- `C[n-1][l]` vaut le nombre d'électrons dans la couche `n, l` si cette couche contient au moins un électron.
- `C[n-1][l]` n'est pas défini sinon.

Par exemple `souscouche(8)` renvoie `[[2], [2, 4]]`.

2. Écrire une fonction `to_string(C)` qui convertit une liste de listes représentant une configuration électronique en chaîne de caractères. Par exemple `to_string([[2], [2, 4]])` doit renvoyer `"1s2 2s2 2p4"`. Les sous-couches seront écrites par n croissant, et en cas d'égalité, par ℓ croissant.

Étant donné un atome de numéro atomique `Z`, on note $n_{\texttt{Z}}$ le nombre de ses couches électroniques. Par exemple, pour le magnésium, $n_{12} = 3$.

3. Estimez la complexité de `souscouches(Z)` en fonction de $n_{\texttt{Z}}$.

4. Montrer que $n_{\texttt{Z}}^3 = \mathcal{O}(\texttt{Z})$. Puis estimez la complexité de `souscouches(Z)` en fonction de `Z`.

1. Le numéro atomique est égal au nombre de protons de l'atome, et donc ici au nombre d'électrons.

Travaux pratiques

TP 3.0 – L'agence matrimoniale

Vous êtes directeur d'une agence matrimoniale qui se trouve face à la situation idéale suivante : vous avez exactement $2n$ clients, constitués de n femmes (notées $F_0, F_1, \ldots, F_{n-1}$) et de n hommes (notés $H_0, H_1, \ldots, H_{n-1}$), qui souhaitent se marier. Vous avez demandé à chaque personne de classer les n personnes du sexe opposé par ordre de préférence, et vous devez créer n couples. Vous devez donc choisir une permutation σ de l'ensemble $\{0, 1, \ldots, n-1\}$, correspondant à la proposition des couples $(F_0, H_{\sigma(0)}), (F_1, H_{\sigma(1)}), \ldots, (F_{n-1}, H_{\sigma(n-1)})$.
Votre proposition σ sera dite *instable* s'il existe deux entiers distincts i et j de $\{0, \ldots, n-1\}$ tels que F_i préfère $H_{\sigma(j)}$ à $H_{\sigma(i)}$ et $H_{\sigma(j)}$ préfère F_i à F_j. Un tel couple (i, j) sera qualifié d'instable pour σ. Une telle situation est évidemment dommageable, car les personnes F_i et $H_{\sigma(j)}$ n'accepteront pas votre proposition. Le but de ce TP est de démontrer qu'il existe toujours une proposition stable et d'en donner un algorithme de construction.

Les classements constituant la donnée du problème seront représentés par deux listes de taille n, notées CF et CH : la case i de la liste CF (resp. de la liste CH) contient la liste des vœux de la femme F_i (resp. de l'homme H_i), dans l'ordre croissant de ses préférences. Voici un exemple de données :

$$CF_0 = [[0, 2, 1], [2, 0, 1], [0, 2, 1]] \quad \text{et} \quad CH_0 = [[0, 1, 2], [1, 0, 2], [0, 2, 1]]$$

dans lequel la femme F_0 préfère H_1, puis H_2, et enfin H_0.
Une proposition σ sera représentée par le tuple $(\sigma(0), \ldots, \sigma(n-1))$. Ainsi, $\sigma_0 = (1, 0, 2)$ propose les appariements (F_0, H_1), (F_1, H_0) et (F_2, H_2). Pour les données CF_0 et CH_0, $(2, 1)$ est une instabilité pour σ_0 car F_2 préfère H_1 à H_2 et H_1 préfère F_2 à F_0. Par contre, $(1, 2)$ n'en est pas une car F_1 préfère H_0 à H_2 (même si H_2 préfère F_1 à F_2).

Quelques fonctions élémentaires.

0. Écrire une fonction qui, appliquée à CF et à trois indices $i, j_1, j_2 \in \{0, \ldots, n-1\}$, renvoie le booléen `True` si F_i préfère H_{j_1} à H_{j_2}, et `False` sinon. Quel est, dans le pire des cas, le temps de calcul de cette fonction ? On remarquera que cette fonction peut s'appliquer à CH pour tester si H_i préfère F_{j_1} à F_{j_2}.

1. Pour améliorer l'efficacité de cette fonction, nous pouvons remplacer CF et CH par des matrices carrées RF et RH d'ordre n : l'entrée $RF[i][j]$ (resp. $RH[i][j]$) contient le rang de l'homme H_j (resp. de la femme F_j) dans la liste des préférences de F_i (resp. de H_i). Par convention, la personne préférée aura le rang 0. Écrire une fonction `convertir_matrice` qui, quand on l'applique à la liste CF (resp. CH), renvoie la matrice RF (resp. RH).

2. Écrire une fonction `preference` qui, appliquée à RF et à trois indices $i, j_1, j_2 \in \{0, \ldots, n-1\}$, renvoie le booléen `True` si F_i préfère H_{j_1} à H_{j_2}, et `False` sinon. Quel est, dans le pire des cas, le temps de calcul de cette fonction ? Une nouvelle fois, cette fonction, appliquée à RH, permet également de voir si H_i préfère F_{j_1} à F_{j_2}.

3. Écrire une fonction qui, appliquée aux matrices CF, CH et à une proposition σ, renvoie la liste des couples (i, j) instables pour σ. Tester votre fonction sur CF_0, CH_0 et σ_0. Expliciter une proposition stable pour (CF_0, CH_0).

4. Écrire une fonction `est_stable` qui, appliquée aux matrices RF, RH et à une proposition σ, renvoie le booléen `True` si la proposition est stable, et `False` sinon.

Une méthode naïve.
La fonction `permutations` du module `itertools` crée un itérateur permettant de parcourir toutes les permutations d'une liste ou d'un tuple, comme dans l'exemple ci-dessous :

```
>>> import itertools
>>> for sigma in itertools.permutations([0, 1]):
    print(sigma)
(0, 1)
(1, 0)
```

On remarquera que les permutations ainsi renvoyées sont des tuples, même quand on a appliqué `itertools.permutations` à une liste.

5. En déduire une fonction `proposition_stable_naïve` qui, appliquée aux matrices CF et CH, renvoie une solution stable (s'il en existe une). Tester votre fonction avec CF_0, CH_0, puis avec

$$CF_1 = [[1,2,0,3],[2,1,0,3],[0,2,3,1],[1,2,0,3]]$$

$$CH_1 = [[3,0,2,1],[3,2,0,1],[2,3,0,1],[2,3,0,1]]$$

Une méthode efficace.
Nous allons maintenant faire évoluer dynamiquement une *proposition partielle stable*, en parlant plutôt de fiançailles. Comme ces fiançailles vont être modifiées au cours du calcul, nous les représenterons par une liste σ de longueur n, plutôt que par un tuple ; ainsi, si $j = \sigma[i]$, la femme F_i sera fiancée à l'homme G_j si $j \in \{0, 1, \ldots, n-1\}$, et ne sera pas fiancée si $j = n$. Au début du calcul, $\sigma = [n, \ldots, n]$, puisqu'aucun mariage n'a encore été envisagé. Les numéros des femmes non encore fiancées seront placées dans une pile P, initialisée à $P = [0, 1, \ldots, n-1]$. Tant que P est non vide, on en extrait un élément i : on regarde alors le premier vœu H_j de F_i (élément qui est en tête de $RF[i]$). Si cet homme n'est pas encore fiancé, on fiance F_i et H_j ; si H_j est déjà fiancé à F_k, on regarde si H_j préfère F_i à F_k. Si c'est le cas, on rompt les fiançailles de H_j (en remettant k dans la pile) et on le fiance avec F_i ; sinon, on remet i dans P. Pour que cela fonctionne, il faut que les listes des vœux des personnes évoluent dynamiquement : à chaque étape du calcul, quand la femme F_i teste son premier vœu (qu'il ait été accepté ou pas), on le supprime de la liste $CF[i]$.

6. Expliciter le fonctionnement de cet algorithme dans le cas particulier $(CF1, CH1)$. Montrer que l'on construit bien ainsi une proposition stable.

7. Écrire une fonction `choix` qui, appliquée à un entier n, calcule une liste de choix aléatoire C, c'est-à-dire une liste de longueur n telles que pour tout i, $C[i]$ est une liste contenant les n entiers compris entre 0 et $n-1$. Cette fonction permettra de simuler à la fois CF et CG. On pourra utiliser la méthode `shuffle` du module `numpy.random`.

Pour mettre en place cet algorithme de façon efficace, nous aurons besoin de répondre rapidement à deux questions : si l'homme H_j est déjà fiancé, quel est l'indice de sa fiancée et préfère-t-il F_i à sa fiancée ? La réponse à la seconde question se fera en utilisant la matrice RH et nous aurons une réponse efficace à la première question en définissant conjointement à σ une liste τ représentant σ^{-1} (avec $\tau(j) = n$ si H_j n'a pas encore été fiancé).

8. Écrire le code d'une fonction `proposition_stable` qui, appliquée à CF et CH, renvoie une proposition stable en utilisant cet algorithme. Nous ajouterons un peu d'aléa en extrayant de la pile P un élément i aléatoire.

9. Prouvez que cet algorithme est correct, c'est-à-dire qu'il renvoie bien dans tous les cas une proposition stable, et que le temps de calcul dans le pire des cas est un $\mathcal{O}(n^2)$. Nous avons ainsi

démontré qu'il existait dans tous les cas une proposition stable, que l'on pouvait calculer en temps quadratique.

10. Que remarque-t-on quand on applique plusieurs fois l'algorithme à un même (CF, CH) ?

Cet algorithme est asymétrique, puisque les femmes et les hommes n'y jouent pas le même rôle. Pour détecter un éventuel avantage des uns sur les autres, nous noterons, quand σ est une proposition stable associée à (CF, CH), m_F la moyenne du rang de $H_{\sigma[i]}$ dans la liste des préférences de F_i. De même, m_H sera la moyenne du rang de F_i dans la liste des préférences de $H_{\sigma[i]}$.

11. Écrire une fonction `test` qui, appliquée à un entier n, calcule un couple aléatoire (CF, CH) correspondant à des classements possibles fournis par nos $2n$ clients, calcule une solution stable σ pour ce couple (CF, CH), puis renvoie le couple (m_F, m_H) associé à (CF, CH, σ). Peut-on dire que l'algorithme utilisé favorise l'un des deux sexes par rapport à l'autre ?

Une situation plus réaliste.

Nous supposons maintenant que l'agence a des nombres différents de clients de chaque sexe : n femmes et p hommes. Les $n+p$ clients ne sont plus obligés de classer toutes les personnes du sexe opposé : les listes $CF[i]$ et $CH[j]$ sont donc de longueur variable. Nous dirons que F_i et H_j sont *compatibles* si F_i apparaît dans le classement de H_j et si H_j apparaît dans le classement de F_i.

Une proposition σ sera donc une liste de longueur n, proposant un certain nombre de fiançailles et vérifiant les conditions suivantes :

- pour tout $i \in \{0, 1, \ldots, n-1\}$, $j = \sigma[i]$ est un entier compris (au sens large) entre 0 et p ; si $j < p$, F_i et H_j sont compatibles et on fiance F_i et H_j ; sinon, F_i n'est pas fiancée ;
- on ne propose pas le même fiancé à deux femmes différentes ; autrement-dit, seul l'élément p peut apparaître plusieurs fois dans la liste σ.

Une telle proposition est dite instable dans trois cas :

a) il existe deux entiers distincts i et j dans $\{0, \ldots, n-1\}$ tels que $\sigma[i] \neq p$, $\sigma[j] \neq p$, F_i préfère $H_{\sigma(j)}$ à $H_{\sigma(i)}$ et $H_{\sigma(j)}$ préfère F_i à F_j ;

b) il existe deux entiers distincts i et j dans $\{0, \ldots, n-1\}$ tels que $\sigma[i] = p$, $\sigma[j] \neq p$ et $H_{\sigma[j]}$ préfère F_i à F_j ;

c) il existe deux entiers i et k dans respectivement $\{0, \ldots, n-1\}$ et $\{0, \ldots, p-1\}$ tels que F_i et H_k ne sont pas fiancés et sont compatibles.

Si σ est une proposition de mariages, nous noterons $\mathscr{F}(\sigma)$ (resp. $\mathscr{H}(\sigma)$) la liste croissante des indices des femmes (resp. des hommes) auxquelles (resp. auxquels) un fiancé a été proposé par σ.

12. Écrire une fonction `choix_bis` qui, appliquée à un couple d'entiers (n, p), renvoie une liste aléatoire pouvant représenter CF (cette même fonction appliquée au couple (p, n) permettra de simuler CH).

13. Modifier les fonctions `convertir_matrice` et `preference` pour les adapter à cette nouvelle situation.

14. Écrire une fonction `proposition_stable_bis` qui adapte la fonction `proposition_stable` pour construire une solution stable dans ce cadre plus général. La fonction renverra le triplet $(\sigma, \mathscr{F}(\sigma), \mathscr{H}(\sigma))$. Tester votre fonction sur quelques exemples.

On peut démontrer que les parties $\mathscr{F}(\sigma)$ et $\mathscr{H}(\sigma)$ ne dépendent pas de la proposition stable σ. Autrement-dit, les laissés-pour-compte sont les mêmes pour toutes les propositions stables.

Vrai/Faux sur le cours – corrigé

0. Une complexité en $\Theta(n)$ est mieux qu'une complexité en $\Theta(n^2)$ lorsque n est grand. Mais le Θ ne donne aucune information sur ce qui se passe quand n est petit. ☐ **Vrai** ☑ **Faux**

1. Si la condition vaut toujours vraie (par exemple un `while True`), le `while` va tourner indéfiniment. ☑ **Vrai** ☐ **Faux**

2. Non, cela dépend du nombre de passages dans le `for` et de la complexité du corps du `for`. ☐ **Vrai** ☑ **Faux**

3. Il faut recopier chaque élément, on ne peut pas avoir mieux que $\mathcal{O}(n)$. ☑ **Vrai** ☐ **Faux**

4. `n` décroît mais pas strictement. Si la condition du `if` n'est pas satisfaite, `n` reste constant. ☐ **Vrai** ☑ **Faux**

5. `N` décroît mais pas strictement. Si la condition du `if` est satisfaite, `N` reste constant. ☐ **Vrai** ☑ **Faux**

6. Soit `N` soit `n` diminue de 1, donc leur somme diminue toujours de 1. ☑ **Vrai** ☐ **Faux**

7. Soit `N` soit `n` diminue de 1, donc leur produit diminue toujours. ☑ **Vrai** ☐ **Faux**

8. `myst` termine toujours car on a trouvé un variant de boucle (questions précédentes) donc `myst` renvoie toujours une valeur. En l'absence de `return`, c'est la valeur `None` qui est renvoyée. ☑ **Vrai** ☐ **Faux**

9. `N` ne décroît que s'il est différent de `n`, comme au départ il est plus grand, il reste toujours plus grand. ☑ **Vrai** ☐ **Faux**

10. Il existe toujours un invariant de boucle inutile (par exemple la propriété qui est toujours vraie). Par contre, il n'existe pas toujours d'invariant de boucle qui permette de démontrer ce qu'on veut démontrer. L'invariant de boucle est une forme de récurrence, et certaines propriétés vraies ne peuvent pas être démontrées par récurrence. ☑ **Vrai** ☐ **Faux**

11. C'est évident, il n'y a aucune construction (pas de `while` notamment) qui risquerait de faire boucler indéfiniment la fonction. ☑ **Vrai** ☐ **Faux**

12. Cet invariant permet de démontrer que la fonction renvoie `True` si et seulement si les éléments de la liste sont classés dans l'ordre strictement croissant. ☑ **Vrai** ☐ **Faux**

13. On passe n^2 fois dans la boucle extérieure. Pour chaque passage dans la boucle extérieure, on passe $\lfloor \log_2(n) \rfloor$ fois dans la boucle intérieure. ☑ **Vrai** ☐ **Faux**

14. Cet invariant supposerait que la boucle extérieure est terminée mais pas la boucle intérieure, c'est absurde. ☐ **Vrai** ☑ **Faux**

15. Pour chaque passage dans la boucle extérieure, on accomplit en entier la boucle intérieure. ☑ **Vrai** ☐ **Faux**

Corrections des exercices

Corrigé exo 3.0

Lors de l'exécution de `truc(n)`, la boucle extérieure (`for j`) est accomplie n fois. À chaque passage dans la boucle extérieure, on passe n fois dans la boucle intérieure. Au total, on passe n^2 fois dans la boucle intérieure. In fine, la complexité est en $\mathcal{O}(n^2)$.
Lors de l'exécution de `machin(n)`, la première boucle `for` est exécutée n fois, puis la seconde boucle est exécutée n fois, soit une complexité en $\mathcal{O}(n) + \mathcal{O}(n) = \mathcal{O}(n)$.

Corrigé exo 3.1

0. Le maximum recherché est soit `M`, soit `L[k]`.

```
def maximum(L):
    M = L[0]
    for k in range(1, len(L)):
        if M < L[k]:
            M = L[k]
    return M
```

1. Il suffit de remplacer `L[k]` par `x`.

```
def maximum(L):
    M = L[0]
    for x in L:
        if M < x:
            M = x
    return M
```

2. $\mathcal{O}(n)$ où n est la longueur de la liste `L`.

Corrigé exo 3.2

0. `a` est un entier naturel, et est strictement décroissant : le `while` termine donc, ainsi que la fonction `Egyptienne`.

1. On décrit dans un tableau l'évolution des variables au cours du temps. Chaque ligne correspond à un passage à la ligne `while a > 0:`

a	b	t
41	3	0
20	6	3
10	12	3
5	24	3
2	48	27
1	96	27
0	192	123

2. On utilise la méthode 3.4. On remarque que la quantité `a*b+t` est constante. Elle vaut au début `a*b` et à la fin `t`. On en déduit que la valeur renvoyée est le produit des valeurs initiales de `a` et `b`.

3. Il y a plusieurs manières de trouver la complexité.
Première méthode. À chaque passage dans le `while` la variable `a` est divisée par deux donc perd un chiffre en base deux. Le nombre de passage dans le `while` est donc borné par le nombre de chiffres de `a` en base deux, donc borné par $\log_2(\mathtt{a}) + 1$, d'où une complexité en $\mathcal{O}(\ln(\mathtt{a}))$.

Seconde méthode. Appelons a_0 la valeur initiale de `a`. À chaque passage dans le `while` la variable `a` est divisée par deux, donc, au bout de k passages, `a` contient au plus $a_0/2^k$. On cherche alors k_0 tel que $a_0/2^{k_0} < 1$. Cela donne $a_0 < 2^{k_0}$ d'où $\log_2(a_0) < k_0$. La valeur $k_0 = \lfloor \log_2(a_0) \rfloor + 1$ convient. Le nombre de passages dans la boucle est borné par k_0, donc la complexité est en $\mathcal{O}(\ln(\mathtt{a}))$.

Corrigé exo 3.3

0. On traduit directement, presque mot-à-mot en Python.

```
def euclide(a, b):
    while b != 0:
        r = a % b
        a = b
        b = r
    return a
```

1. Cette exception pourrait éventuellement être levée par l'expression `a % b` mais on a vérifié avant que `b` est non nul, donc cette exception n'est jamais levée.

2. La suite des valeurs de `b` est une suite d'entiers naturels strictement décroissante[1], donc le `while` termine toujours, donc la fonction termine toujours.

3. On applique la méthode 3.4. On remarque que la quantité $\text{pgcd}(a, b)$ est constante, qu'à la fin b vaut zéro donc cette constante est égale à a. On en déduit que la valeur renvoyée est bien le pgcd des valeurs initiales de a et b.

4. Considérons deux cas.
Premier cas : $b \leqslant a/2$. Comme $a \leqslant b$ on a le résultat voulu.
Deuxième cas $b > a/2$: on a alors $a/2 < b < a$. On peut écrire $a = 1 \times b + (a - b)$ et on remarque que $0 \leqslant a - b < b$ (car $a < 2b$). Ainsi $a - b$ est le reste de la division euclidienne de a par b (le quotient vaut 1), d'où $r = a - b$. Comme $a/2 < b$ on a $a - b < a - a/2 = a/2$.
Dans tous les cas, on a $r \leqslant a/2$.

5. Si $b < a$ au début, alors cette propriété est un invariant de boucle, elle est préservée au cours de l'exécution de l'algorithme.
Lors de l'exécution, $r < a/2$, or r devient ensuite b qui devient ensuite a, donc en deux passages dans le `for`, a est divisé par deux (au moins), donc il perd au moins un chiffre en base deux.
Par conséquent, le nombre de passages dans le `for` est majoré par deux fois le nombre de chiffres en base deux de a plus un. Ce qui donne une complexité en $\mathcal{O}(1 + \log_2(a)) = \mathcal{O}(\ln(a))$.
Si $a < b$, alors $r = a$, et à la première étape, les valeurs de a et b sont échangées et on se ramène au cas précédent, d'où une complexité en $\mathcal{O}(\ln(b))$.
Dans tous les cas, la complexité est en $\mathcal{O}(\ln(n))$.

1. Ce n'est pas forcément le cas pour `a` qui peut augmenter lors du premier passage dans le `while`.

Corrigé exo 3.4

0. Le problème vient des deux boucles imbriquées. On va s'arranger pour n'avoir qu'une seule boucle.
On maintient dans le `for` l'invariant suivant : « Le nombre de 9 à la fin de `L[:k]` est `i` »

```
def neufs(n):
    L = []
    while n!= 0:
        L.append(n % 10)
        n = n // 10
    M = 0
    i = 0
    for k in range(len(L)):
        if L[k] == 9:
            i += 1
        else:
            i = 0
        M = max(M, i)
    return M
```

1. Pour diminuer la complexité en espace, on va éviter de stocker les chiffres dans une liste.

```
def neufs(n):
    M = 0
    i = 0
    while n!= 0:
        if n % 10 == 9:
            i += 1
        else:
            i = 0
        M = max(M, i)
        n //= 10
    return M
```

Corrigé exo 3.5

0. Dans la première boucle, on passe moins de n fois avec n la longueur de `s`. Dans la seconde, moins de p fois avec p la longueur de `m`.
À l'intérieur des deux boucles, le temps de calcul est en $\mathcal{O}(1)$.
Au final, la complexité est en $\mathcal{O}(np)$.

1. On passe au plus n fois dans l'unique boucle. À l'intérieur de la boucle, le test d'égalité entre deux chaînes de caractères est en $\mathcal{O}(p)$, d'où une complexité totale en $\mathcal{O}(np)$.

2. Au mieux `b` vaut `True` du premier coup (c'est possible si `m` est un préfixe de `s`. On a alors fait p passages dans la seconde boucle et un seul dans la première, d'où une complexité au meilleur en $\mathcal{O}(p)$.

3. `cherche(s,m)` teste si `m` est un sous-mot [1].

4. La boucle interne a pour invariant « `b` équivaut à `s[k:k+i]==m[:i]` » La boucle externe a pour invariant « `s[j:j+len(m)] != m` pour tout `j<k` » car si il y avait égalité, on serait sorti de la boucle avec le `return True`.
On en conclut que si la boucle n'est jamais interrompue par le `return True` alors `m` n'est pas un sous-mot de `s`. Si la boucle est interrompue, d'après l'invariant de la boucle intérieure, on a trouvé `m` dans `s`.

1. « Sous-mot » est là un anglicisme, une traduction littérale de *subword* de `s`. La traduction usuelle est facteur.

Corrigé exo 3.6

0. On traduit l'algorithme en Python.

```
import random


def S(n, N):
    R = []  # L'ensemble R est représenté par une liste
    while N > 0:
        # p vaut True avec une probabilité de n/N
        p = random.randint(1, N) <= n
        if p:
            R.append(N)
            n = n - 1
        N = N - 1
    return R
```

1. N contient un entier naturel, et la valeur de cet entier décroît strictement à chaque passage dans la boucle, donc l'algorithme termine.
On remarque que la condition $n > 0$ pourrait remplacer $N > 0$ car n est toujours plus petit que N.

2. On veut tirer n valeurs distinctes parmi $[\![1, N]\!]$. On a deux cas, soit N appartient soit N n'appartient pas au résultat.
Le premier cas arrive avec une probabilité n/M. Le tirage à pile ou face permet de savoir si N est dans le résultat. Ensuite, l'algorithme tire au hasard les $n-1$ éléments qui restent à tirer parmi $[\![1, N-1]\!]$.
Dans le second cas, on ne diminue par n, on tire donc n éléments dans $[\![1, N-1]\!]$.

3. L'algorithme a une complexité au meilleur et au pire en $\mathcal{O}(N)$. Avec la condition $n > 0$, la complexité au meilleur est seulement en $\mathcal{O}(n)$.

Corrigé exo 3.7

0.

```
import random

def rand_list(N, t):
    return sorted([random.randint(1, N) for i in range(t)])
```

1.

```
def recherche(L, a):
    '''
    recherche l'élément a dans L
    renvoie l'indice correspondant, -1 si pas trouvé
    '''
    n = len(L)
    for i in range(n):
        if a == L[i]:
            return i  # on sort directement
    return -1
```

```
i = recherche(L, a)
print('indice', i, 'élément', a)
print(L[:i], L[i], L[i + 1:])
```

```
indice 10 élément 33
[2, 5, 7, 21, 23, 27, 28, 29, 29, 32] 33 [34, 49, 50, 50, 54, 54, 56, 59, 59]
```

Bien sûr, on dispose d'outils Python de type boîte noire qui font la même chose.

```
if a in L:
    print(L.index(a))
else:
    print(-1)
```

```
10
```

2. Le principe est simple mais le test de fin est assez subtil.

```
def recherchedicho(L, a):
    '''
    recherche l'élément a dans L
    renvoie l'indice correspondant, -1 si pas trouvé
    '''
    deb, fin = 0, len(L) - 1    # debut, fin de la sous-liste
    milieu = (deb + fin) // 2
    while fin > deb:
        milieu = (deb + fin) // 2
        if L[milieu] >= a:
            fin = milieu
        else:
            deb = milieu + 1

    if a == L[deb]:
        return deb
    else:
        return -1
```

Lorsque `fin = deb + 1` par exemple il faut être sûr de renvoyer le bon indice. On a dans ce cas, `milieu = deb` quand on fait la moyenne
d'où l'inégalité large pour le test `if L[milieu] >= a`...

```
N = 25

for i in range(30000):
    a, L = creeliste(N)
    #L = [0, 1]
    #a = 0
    i = recherchedicho(L, a)
    if i == -1:
        print(L, a, i, L.index(a))
```

```
N = 25
a, L = creeliste(N)

#a = 40
i = recherchedicho(L, a)
print('indice', i, 'nombre', a)
print(L[:i], L[i], L[i + 1:])
```

```
indice 7 nombre 16
[2, 6, 7, 9, 12, 13, 15] 16 [17, 20, 22, 29, 35, 36, 39, 40, 40, 45, 49, 49, 50, 52, 67, 68, 72]
```

3. La première fonction parcourt potentiellement tous les éléments de la liste, elle a une complexité en $\mathscr{O}(n)$. La deuxième, qui agit par dichotomie, effectue pour $n = 2^p$, au maximum $p+2$ tests sur les éléments de la liste. On se convainc aisément que le nombre de tests est de l'ordre de $\log_2(n)$ ce qui est bien sûr bien meilleur sauf si l'élément recherché est dans les premiers de la liste...

Corrigé exo 3.8

0. Nous allons montrer que la quantité N est un variant de boucle.

N est toujours un entier, en effet il est initialement entier (par hypothèse), et par une récurrence simple on a bien N entier après k passages dans la boucle.

La quantité N est strictement décroissante (on note N' sa valeur après un passage dans la boucle) :

en effet si N est pair avant un passage dans la boucle, on a $N' = \dfrac{N}{2}$ et comme $N \geqslant 2$ on a bien $N' < N$.
Alternativement si N est impair avant un passage dans la boucle on a $N' = N - 1$ et donc $N' < N$

Enfin, la boucle s'arrête lorsque $N \leqslant 0$

N est bien un variant de boucle : cette boucle s'arrête effectivement.

1. Montrons que la propriété suivante est un invariant de boucle :
$(\mathscr{P}_n)$: (Après n exécutions de la boucle $a^n = R \times A^N$)

Initialisation : avant la première exécution de la boucle, par initialisation, on a $R = 1$, $A = a$ et $N = n$ et donc $R \times A^N = 1 \times a^n = a^n$.

Hérédité : Supposons pour un n donné $(\mathscr{P}_n)$ vraie (et $N \neq 0$). Deux cas sont à distinguer :
Si N est pair, on a alors $N' = \dfrac{N}{2}$, $R' = R$ et $A' = A^2$.
On a donc $R' \times A'^{N'} = R \times (A^2)^{\frac{N}{2}} = R \times A^{2\times\frac{N}{2}} = R \times A^N = a^n$ et $(\mathscr{P}_{n+1})$ est vraie.
Sinon N est impair. On a alors $N' = N - 1$, $R' = R \times A$ et $A' = A$.
On a donc $R' \times A'^{N'} = R \times A \times A^{N-1} = R \times A^N = a^n$ et $(\mathscr{P}_{n+1})$ est vraie.
Au final $(\mathscr{P}_{n+1})$ est vraie dans tous les cas, l'hérédité vient d'être prouvée.

Cette propriété est donc bien un invariant de boucle. Comme la boucle s'arrête lorsque $N = 0$ (on avait précédemment nécessairement $N = 1$ et $N = 2$), à l'arrêt de la boucle on a $a^n = R$ et donc cet algorithme calcule bien a^n (qu'il stocke dans R qui est la valeur renvoyée).

2. De manière intuitive, on va diviser la taille du problème par 2 au pire tous les deux passages de boucles, la boucle sera donc exécutée au pire $2\lceil\log_2(N)\rceil$ fois et l'algorithme aura une complexité en $\mathscr{O}(\ln n)$
Nous pouvons le démontrer de manière plus rigoureuse en notant b_n l'écriture de N en binaire après n passages dans la boucle.
Le dernier bit de b_n donne la parité de N. Si le bit le moins significatif de b_n est 1, alors N est impair et $b_{n+1} = b_n - 1$. Si le bit le moins significatif de b_n est 0, alors N est impair, et b_{n+1} est constitué de tous les bits de b_n sauf le dernier.
Par itération de l'algorithme, si le bit le moins significatif de b_n est un 1, alors b_{n+2} est constitué de tous les bits de b_n sauf le dernier et sinon c'est le cas de b_{n+1}.

Notons p le nombre de bits de b_0. L'algorithme s'arrêtera lorsque $b_k = 0$. Il faudra pour cela que tous les bits de b_0 aient disparu. Il faudra pour cela $p-1$ passages de boucles qui correspondront aux étapes où le bit le moins significatif de b_n est 0 et autant d'étapes qu'il y a de 1 dans l'écriture de b_0, qui correspondront aux étapes où le bit le moins significatif de b_n est 1.
Le nombre de passages dans la boucle est ainsi $p-1+u$ où u est le nombre de 1 dans b_0. Il est compris entre p (car b_0 commence nécessairement par un 1) et $2p-1$ (cas où b_0 n'est constitué que de 1).
Il reste à déterminer p. On a $2^{p-1} \leqslant n \leqslant 2^p - 1$. Par passage au logarithme en base 2, on a $p - 1 \leqslant \log_2(n) < p$. Et donc $p = \lfloor \log_2(n) \rfloor + 1$.
Il y a donc entre $\lfloor \log_2(n) \rfloor$ et $2\lfloor \log_2(n) \rfloor - 1$ passages dans la boucle : la complexité de cet algorithme est dans tous les cas en $\mathcal{O}(\ln n)$ opérations élémentaires.

Le calcul naïf de la puissance nécessite $\mathcal{O}(n)$ opérations. L'algorithme présenté ici, qui réalise le même calcul, le fait bien avec beaucoup moins d'opérations, d'où le nom d'exponentiation rapide.

Corrigé exo 3.9

0.

```
def score(u, v):
    s = 0
    for i in range(len(u)):
        if u[i] == '-' or v[i] == '-':
            s -= 2
        elif u[i] != v[i]:
            s -= 1
        else:
            s += 2
    return s
```

1. Si $n = 0$ ou $m = 0$, il existe un seul chemin (on se déplace toujours vers la droite, ou toujours vers le bas), donc $s_{n,m} = 1$. Ceci fonctionne même si $n = m = 0$, puisque quand u et v sont vides, il existe bien un unique scénario : $\tilde{u}$ et $\tilde{v}$ sont également vides.
Si $n, m \geqslant 1$, on obtient la relation demandée en remarquant que l'ensemble des chemins cherchés se décompose en :

- $s(n, m-1)$ chemins dont le dernier déplacement est vers le bas ;
- $s(n-1, m-1)$ chemins dont le dernier déplacement est diagonal ;
- $s(n-1, m)$ chemins dont le dernier déplacement est vers la droite.

Il ne faut surtout pas utiliser cette relation pour programmer le calcul de $s(n, m)$ récursivement.
En suivant l'indication, nous obtenons :

```
def calcul_s(n, m):
    S = [[1 for j in range(m + 1)] for i in range(n + 1)]
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            S[i][j] = S[i - 1][j] + S[i - 1][j - 1] + S[i][j - 1]
    return S[n][m]
```

Le temps de calcul et l'espace mémoire utilisés sont clairement de l'ordre de nm. Nous obtenons $s(4, 3) = 129$ et $s(35, 40) = 577216875504248378452264677$.

2. Au lieu de mémoriser toute la matrice S, on se contente de conserver en mémoire deux lignes (si $m \leqslant n$) ou deux colonnes (si $n < m$). Nous commençons par nous ramener, par symétrie, au

cas où $m \leqslant n$, puis nous créons deux listes de longueur m, « l'ancienne ligne » et la « nouvelle ligne » :

```
def calcul_s_bis(n, m):
    if m > n:
        return calcul_s_bis(m, n)
    else:
        AL = [1 for j in range(m + 1)]
        NL = [1 for j in range(m + 1)]
        for i in range(1, n + 1):
            for j in range(1, m + 1):
                NL[j] = AL[j] + AL[j - 1] + NL[j - 1]
            AL, NL = NL, AL  # se fait en temps et espace constant
        return AL[m]
```

3. Si l'un des mots est vide, on ne peut l'aligner qu'en utilisant des gaps, d'où $\delta(0,j) = -2j$ et $\delta(i,0) = -2i$ pour $0 \leqslant i \leqslant n$ et $0 \leqslant j \leqslant m$.

4. On peut arriver de trois façons à la case (i,j) : depuis la case $(i-1,j)$ en perdant deux points, depuis la case $(i,j-1)$ en perdant deux points ou depuis la case $(i-1,j-1)$ en perdant un point si $v[i-1] \neq u[i-1]$ ou en gagnant deux points si $v[i-1] = u[j-1]$ (attention au décalage d'indice). Comme les chemins arrivant aux points $(i,j-1)$, $(i-1,j)$ et $(i-1,j-1)$ sont de scores maximaux $\delta(i,j-1)$, $\delta(i-1,j)$ et $\delta(i-1,j-1)$, le chemin optimal arrivant à (i,j) est de score maximal :

$$\delta(i,j) = \max\Big(\delta(i,j-1) - 2, \delta(i-1,j) - 2, \delta(i-1,j-1) + g(i,j)\Big)$$

en posant $g(i,j) = -1$ si $v[j-1] \neq u[i-1]$ et $g(i,j) = 2$ sinon.

Avec u_0 et v_0, nous obtenons ainsi :

$$\delta(1,1) = \max\Big(\delta(1,0)-2, \delta(0,1)-2, \delta(0,0)-1\Big) = -1$$

$$\delta(1,2) = \max\Big(\delta(1,1)-2, \delta(0,2)-2, \delta(0,1)-1\Big) = -3$$

et ainsi de suite jusqu'à remplir la matrice ci-contre : $d(u_0,v_0) = \delta(3,4) = 1$.

		A	A	C	G
	0	−2	−4	−6	−8
T	−2	−1	−3	−5	−7
A	−4	0	1	−1	−3
G	−6	−2	−1	0	1

5.

```
def distance(u, v):
    n, m = len(v), len(u)
    d = [[0 for j in range(m + 1)] for i in range(n + 1)]
    # On remplit la première ligne et la première colonne
    for j in range(1, m + 1):
        d[0][j] = d[0][j - 1] - 2
    for i in range(1, n + 1):
        d[i][0] = d[i - 1][0] - 2
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            if v[i - 1] == u[j - 1]:
                b = 2   # match
            else:
                b = -1   # mismatch
            d[i][j] = max(d[i - 1][j] - 2, d[i][j - 1] -
                          2, d[i - 1][j - 1] + b)
    return d[n][m]  # delta(n,m) = d(u,v)
```

6. La fonction auxiliaire `plus_grand`, appliquée à trois entiers a, b et c, renvoie 0, 1 ou 2 suivant que $\max(a, b, c)$ vaut a, b ou c.

On commence par construire de la même façon la matrice δ, en stockant dans chaque case, en plus de la valeur $\delta(i, j)$, un caractère indiquant le dernier déplacement amenant à la case (i, j) : $'d'$ pour une délation (déplacement vers la droite), $'i'$ pour une insertion (déplacement vers le bas) et $'m'$ pour un match ou un mismatch (déplacement en diagonale). Une fois la matrice δ calculée, il reste à remonter de la case (n, m) à la case $(0, 0)$ en construisant les mots $\tilde{u}$ et $\tilde{v}$.

```
def plus_grand(a, b, c):
    if a >= b:
        if a >= c:
            return 0
        else:
            return 2
    else:
        if b >= c:
            return 1
        else:
            return 2


def alignement(u, v):
    n, m = len(v), len(u)
    # 'd' = délétion, 'i' = insertion, 'm' = match ou mismatch
    d = [[(0, 'd') for j in range(m + 1)] for i in range(n + 1)]
    # On remplit la première ligne et la première colonne
    for j in range(1, m + 1):
        d[0][j] = d[0][j - 1][0] - 2, 'd'
    for i in range(1, n + 1):
        d[i][0] = d[i - 1][0][0] - 2, 'i'
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            if v[i - 1] == u[j - 1]:
                b = 2   # match
            else:
                b = -1   # mismatch
            a = plus_grand(d[i - 1][j][0] - 2, d[i][j - 1]
                           [0] - 2, d[i - 1][j - 1][0] + b)
            if a == 0:
                d[i][j] = d[i - 1][j][0] - 2, 'i'
            elif a == 1:
                d[i][j] = d[i][j - 1][0] - 2, 'd'
            else:
                d[i][j] = d[i - 1][j - 1][0] + b, 'm'
    i, j, utilde, vtilde = n, m, "", ""
    while i != 0 and j != 0:
        if d[i][j][1] == 'd':  # on remonte vers la gauche
            utilde, vtilde = u[j - 1] + utilde, '-' + \
                vtilde  # on lit une lettre de u
            j = j - 1
        elif d[i][j][1] == 'i':  # on remonte vers le haut
            # on lit une lettre de v
            utilde, vtilde = '-' + utilde, v[i - 1] + vtilde
            i = i - 1
        else:  # on remonte en diagonale
            utilde, vtilde = u[j - 1] + utilde, v[i - 1] + \
                vtilde  # on lit une lettre de u et de v
            i, j = i - 1, j - 1
    return d[n][m][0], utilde, vtilde
```

Corrigé exo 3.10

0. Si ℓ est différent de zéro, le résultat est évident. Sinon le nouveau ℓ vaut la moitié de l'ancien n. Pour trouver le nouveau n, on utilise le fait que la somme $n + \ell$ a augmenté de 1.

```
def couche_suivante(n, l):
    if l!= 0:
        return n + 1, l - 1
    return n + 1 - n // 2, n // 2
```

1. Tant qu'il reste des électrons à répartir, on remplit la sous-couche courante et on passe à la couche suivante.

```
def souscouches(Z):
    C = []
    n, l = 1, 0
    while Z > 0:  # Tant qu'il reste des électrons à répartir
        if l == 0:  #
            C.append([])
        e = min(2 * (2 * l + 1), Z)  # e = nb d'électron sur la sous-couche
        C[n - 1].append(e)
        Z = Z - e
        n, l = couche_suivante(n, l)
    return C
```

2. On parcourt la liste de listes avec deux `for`.

```
def to_string(C):
    X = "spdfgh"  # Les lettres correspondant aux sous-couches.
    s = ""
    for n in range(len(C)):
        for l in range(len(C[n])):
            s += " " + str(n + 1) + X[l] + str(C[n][l])
    return s[1:]  # On supprime l'espace initial.
```

3. On passe une fois dans le `while` pour chaque sous-couche. La complexité est donc en $\mathcal{O}(s)$ avec s le nombre de sous-couches. Étant donné que chaque couche n contient au plus n sous-couches, et qu'il y a n_Z couches, on a $s \leqslant n_Z^2$ d'où le résultat voulu.

4. Au lieu de chercher un majorant de n_Z en fonction de Z, cherchons un minorant de Z en fonction de n_Z.

On remarque que si une couche n est complètement remplie, alors elle contient un nombre d'électrons de :

$$\sum_{l=0}^{n-1} 2(2\ell+1) \geqslant 2\sum_{l=0}^{n-1}(\ell+1) = 2\sum_{l=1}^{n}\ell = 2\frac{n(n+1)}{2} \geqslant n^2$$

De plus, toutes les couches jusqu'à $\lceil \frac{n_Z}{2} \rceil$ sont remplies, donc $Z \geqslant \sum_{n=1}^{\lceil n_Z/2 \rceil} n^2 \geqslant n_Z^3/6$ d'où le résultat recherché.

On en déduit que $n_Z = \mathcal{O}(Z^{1/3})$ et que $n_Z^2 = \mathcal{O}(Z^{2/3})$ d'où une complexité en $\mathcal{O}(Z^{2/3})$.

Correction d'un TP

Corrigé TP 3.0

0. Il suffit de parcourir la liste $C[i]$ de gauche à droite : on renvoie `False` si on rencontre j_1 avant j_2 et `True` sinon.

```
def prefere(C, i, j1, j2):
    # est-ce que i préfère j1 à j2?
    n = len(C)
    for j in range(n):
        if C[i][j] == j1:
            return False
        elif C[i][j] == j2:
            return True
```

Dans le pire des cas, les entiers j_1 et j_2 sont les derniers éléments de $C[i]$ et le temps de calcul est de l'ordre de n.

1. On initialise la matrice R, puis on parcourt chaque liste $C[i]$: l'élément $C[i][j]$ a le rang $n-1-j$ dans l'ordre des vœux de la i-ème personne.

```
def convertir_matrice(C):
    n = len(C)
    R = [[0 for i in range(n)] for j in range(n)]
    for i in range(n):
        for j in range(n):
            R[i][C[i][j]] = n - j - 1
    return R
```

2. Il suffit de comparer les rangs de j_1 et de j_2, ce qui se fait en temps constant :

```
def preference(R, i, j1, j2):
    # est-ce que i préfère j1 à j2?
    return R[i][j1] < R[i][j2]
```

3. On crée une liste vide L et on teste tous les couples (i, j) en ajoutant à L les instabilités détectées :

```
def liste_instabilités(CF, CH, sigma):
    n = len(CF)
    # on convertit les listes de choix en matrices de rangs
    RF = convertir_matrice(CF)
    RH = convertir_matrice(CH)
    L = []  # L va contenir la liste des instabilité
    for i in range(n):
        for j in range(n):
            if préférence(RF, i, sigma[j], sigma[i]) and préférence(RH, sigma[j], i, j):
                # on a détecté une instabilité
                L.append((i, j))
    return L
```

On montre ainsi qu'il y a une instabilité pour σ_0 et que $(0, 1, 2)$ est une proposition stable pour CF_0 et CH_0 :

```
>>> CF0 = [[0, 2, 1], [2, 0, 1], [0, 2, 1]]
>>> CH0 = [[0, 1, 2], [1, 0, 2], [0, 2, 1]]
>>> liste_instabilités(CF0, CH0, (1, 0, 2))
[(0, 1)]
>>> liste_instabilités(CF0, CH0, (0, 1, 2))
[]
```

4. On reprend la même méthode, mais en arrêtant le calcul dès que l'on détecte une instabilité :

```
def est_stable(RF, RH, sigma):
    n = len(RF)
    for i in range(n):
        for j in range(n):
            if préférence(RF, i, sigma[j], sigma[i]) and préférence(RH, sigma[j], i, j):
                # le calcul est terminé car on a détecté une instabilité
                return False
    return True    # tous les couples ont été étudiés et il n'y a pas d'instabilité
```

5. On teste les permutations en arrêtant une nouvelle fois le calcul dès que l'on a trouvé une proposition stable :

```
import itertools


def proposition_stable_naïve(CF, CH):
    n = len(CF)
    RF = convertir_matrice(CF)
    RH = convertir_matrice(CH)
    for sigma in itertools.permutations([i for i in range(n)]):
        if est_stable(RF, RH, sigma):
            return sigma  # on a trouvé une proposition stable
    return (n,)*n  # aucune proposition n'est stable
```

Cette fonction, appliquée à CF_1 et CH_1, nous donne la proposition stable $(0, 3, 1, 2)$.

6. Au début du calcul, $P = [0, 1, 2, 3]$; on extrait 3 de P et on fiance F_3 et H_3 (c'est le préféré de F_3). On extrait ensuite 2 de P et on fiance F_2 et H_1. On extrait 1 de P, mais F_1 préfère H_3 qui est déjà fiancé à F_3. Comme H_1 préfère F_1 à F_3, on fiance F_1 à H_3 et on remet 3 dans la pile. Nous avons donc à cet instant du calcul :

$$P = [0, 3],\ \sigma = [4, 3, 1, 4] \text{ et } CF = [[1, 2, 0, 3], [2, 1, 0], [0, 2, 3], [1, 2, 0]]$$

On dépile alors 3 et on fiance F_3 et H_0 ; on dépile ensuite 0 : le préféré de F_0 est H_3 qui préfère rester avec son actuelle fiancée F_1 ; F_0 essaie alors son second choix : H_0 la préfère à sa fiancée F_3, donc on fiance F_0 et H_0 et on empile 3, ce qui nous donne :

$$P = [3],\ \sigma = [0, 3, 1, 4] \text{ et } CF = [[1, 2], [2, 1, 0], [0, 2, 3], [1, 2, 0]]$$

On dépile 3, mais son préféré est H_0 qui préfère sa fiancée F_0 à F_3. Le suivant de sa liste est H_2 : comme il n'est pas fiancé, on fiance F_3 et H_2 et le calcul s'arrête avec la proposition stable $[0, 3, 1, 2]$.

7.

```
import numpy.random as rd


def choix(n):
    # construit une liste aléatoire de choix pour n personnes
    C = [[j for j in range(n)] for i in range(n)]
    for i in range(n):
```

```
        # on mélange aléatoirement la liste C[i]
        rd.shuffle(C[i])
    return C
```

8.

```
1  def proposition_stable(CF, CH):
2      n = len(CF)
3      # pour ne pas détruire la donnée CF
4      CF0 = [[CF[i][j] for j in range(n)] for i in range(n)]
5      RH = convertir_matrice(CH)
6      P = [i for i in range(n)]
7      # au début, personne n'est fiancé
8      sigma, tau = [n for i in range(n)], [n for i in range(n)]
9      while L != []:
10         i = P.pop(rd.randint(len(P)))  # i est le numéro aléatoire d'une femme non fiancée
11         j = CF0[i].pop()   # on regarde le numéro de l'homme qu'elle préfère
12         k = tau[j]  # k est le numéro de la fiancée de H_j
13         if k == n:  # H_j n'est pas fiancé :
14             sigma[i] = j    # on le fiance à F_i
15             tau[j] = i
16         elif preference(RH, j, i, k):  # H_j est fiancé mais préfère F_i à sa fiancée actuelle
17             sigma[i] = j   # on fiance F_i à H_j
18             tau[j] = i
19             sigma[k] = n
20             P.append(k)  # on remet F_k dans la pile
21         else:
22             # H_j ne souhaite pas changer de fiancée : on remet F_i dans la pile
23             P.append(i)
24     # toutes les femmes ont été fiancées: on renvoie une proposition stable.
25     return sigma
```

9. Voici les arguments qui prouvent que l'algorithme fonctionne en temps quadratique :

a) une fois qu'un homme est fiancé, il le reste jusqu'à la fin du calcul et on ne modifie sa fiancée que pour améliorer sa situation ;

b) la pile P contient exactement les indices i tels que F_i n'est pas fiancée ;

c) quand on supprime un des vœux j de la femme F_i, c'est ou bien que l'on va la fiancer avec H_j, ou bien que H_j est fiancé avec une femme qu'il préfère à F_i. Dans ce cas, jusqu'à la fin du calcul, H_j préfèrera sa fiancée à F_i ;

d) tous les indices j qui ont été supprimés d'une liste $CF0[i]$ correspondent à des hommes fiancés ; ainsi, si i est dans P, la liste $CF0[i]$ ne peut pas être vide (sinon, les n hommes seraient fiancés, et donc F_i le serait également), ce qui prouve qu'il n'y aura pas d'erreur au passage de la ligne 11 ;

e) à chaque instant du calcul, σ contient une proposition partielle stable ;

f) à chaque passage dans la boucle (lignes 9 à 23), on supprime un élément d'une des listes $CF0[i]$. Comme ces n listes contiennent chacune n éléments au début du calcul, on effectue au maximum n^2 tours de boucle (on ne peut pas supprimer plus de n^2 éléments), qui demande chacun un temps constant.

Ainsi, le programme termine en temps $\mathcal{O}(n^2)$ dans le pire des cas et, comme la pile est vide à la fin de l'appel, σ contient une proposition stable (toutes les femmes sont fiancées, donc la proposition partielle stable n'est plus partielle).

10. Quand on applique plusieurs fois l'algorithme aux même données, on obtient toujours la même solution stable, ce qui permet de conjecturer que la solution stable construite par notre algorithme ne dépend pas de l'ordre dans lequel les femmes sont choisies dans P.

11.

```
def test(n):
    CF, CH = choix(n), choix(n)
    sigma = proposition_stable(CF, CH)
    RF, RH = convertir_matrice(CF), convertir_matrice(CH)
    f, g = 0, 0
    for i in range(n):          # pour chaque couple
        f += RF[i][sigma[i]]    # on ajoute à f le rang de l'époux
        g += RH[sigma[i]][i]    # on ajoute à g le rang de l'épouse
    return f / n, g / n
```

```
>>> test(100)
(3.52, 19.36)
>>> test(1000)
(7.125, 125.155)
```

Dans cette situation probabiliste « uniforme », il semble donc que les femmes soient grandement favorisées. Le lecteur pourra affiner cette étude en imaginant une fonction `choix` qui simule des listes CF et CH plus réalistes.

12. On choisit, pour chaque i, la longueur aléatoire de la liste $C[i]$ (comprise au sens large entre $p/2$ et p), puis on remplit $C[i]$ avec des éléments extraits de $L = [0, 1, \ldots, p-1]$; on utilise pour cela l'expression `L.pop(rd.randint(len(L)))` qui renvoie un élément quelconque de L et le supprime de L :

```
def choix_bis(n, p):
    C = [[] for i in range(n)]
    for i in range(n):
        L = [j for j in range(p)]
        for k in range(rd.randint(p // 2, p+1)):
            C[i].append(L.pop(rd.randint(len(L))))
    return C
```

13. Si C est la liste des choix des femmes et p le nombre d'hommes, on donne le rang p à tous les hommes qui n'ont pas été classés (on ajoute donc le paramètre p en argument de la fonction) :

```
def convertir_matrice_bis(C, p):
    # C = liste des choix des femmes, p = nombre d'hommes
    n = len(C)  # n = nombre de femmes
    R = [[p for j in range(p)] for i in range(n)]
    # si une case R[i][j] n'est pas modifiée, c'est que F_i n'a pas classé H_j
    for i in range(n):
        for j in range(len(C[i])):
            R[i][C[i][j]] = len(C[i]) - j - 1
    return R
```

La fonction `preference` demande de traiter quelques cas supplémentaires :

```
def preference_bis(R, i, j1, j2):
    # R = RH
    # est-ce que H_i préfère F_j1 à F_j2 ?
    p = len(R)  # p = nombre d'hommes
    n = len(R[0])  # n = nombre de femmes
    if j1 == n:  # un homme ne préfère jamais redevenir célibataire
        return False
    elif j2 == n:  # si l'homme est célibataire
        # il préfère se fiancer à condition que la femme proposée soit dans sa
        # liste de voeux
        return R[i][j1] != n
    else:
        return R[i][j1] < R[i][j2]  # on compare les rangs
```

14.

```
def proposition_stable_bis(CF, CH):
    n, p = len(CF), len(CH)
    # pour ne pas détruire la donnée CF
    CF0 = [[CF[i][j] for j in range(len(CF[i]))] for i in range(n)]
    RH = convertir_matrice_bis(CH, n)
    L = [i for i in range(n)]
    sigma, tau = [p for i in range(n)], [n for i in range(p)]
    while L != []:  # il reste au moins une femme non éliminée et non fiancée
        i = L.pop()
        if CF0[i] != []:  # si cette femme n'a pas encore essayé tous les hommes de sa liste
            j = CF0[i].pop()  # on prend l'homme qu'elle préfère
            k = tau[j]  # qui est fiancé à F_k (si k=n, il n'est pas fiancé)
            if preference_bis(RH, j, i, k):
                sigma[i] = j  # on le fiance à F_i
                tau[j] = i
                if k != n:  # H_j était fiancé:
                    sigma[k] = p  # F_k n'est plus fiancée
                    L.append(k)  # et la remet dans la pile
            else:  # troisième cas : H_j refuse F_i
                    L.append(i)  # qui retourne dans la liste L
    # on construit les listes ordonnées des femmes et des hommes fiancés
    # F_i est fiancée si sigma[i] est différent de p
    # H_i est fiancé si tau[i] est différent de n
    return sigma, [i for i in range(n) if sigma[i] != p], [i for i in range(p) if tau[i] != n]
```

CHAPITRE 4

Calcul numérique (une dimension)

L'essentiel du cours

■ 0 Module numpy

Le module **numpy** regroupe les fonctions, constantes et méthodes premettant de réaliser des programmes d'ingénierie numérique (calculs numériques, travail sur les images, le son, les nuages de points). D'autres modules importants comme `matplotlib` et `scipy` utilisent `numpy`.

Définition

Le module **numpy** est dédié au calcul numérique. Il introduit en particulier le type **numpy.ndarray** (où **array** = tableau), ainsi que les fonctions mathématiques classiques.

Les utilisateurs de l'interface graphique Spyder doivent faire attention : certains modules dont `numpy` et `matplotlib` sont chargés automatiquement au lancement par Spyder. Il est néanmoins indispensable de laisser les commandes d'importation de ces modules au début du programme pour qu'il soit exécuté correctement dans n'importe quel environnement de programmation.

Définition

Le type **numpy.ndarray** représente un tableau multidimensionnel dont les données ont toutes le même type (par défaut `np.float64`). Ces tableaux sont optimisés pour les calculs vectoriels et matriciels comme les vecteurs et matrices en Scilab® utilisés en SII.

Le type `np.float64` correspond à un flottant binaire sur 64 bits (Un `binary64`, cf. chapitre 2, partie 2 p. 63). Il existe d'autres types possibles, par exemple `np.uint8` (un entier non signé sur 8 bits, utile pour le traitement d'images) ou `np.int32` (un entier signé sur 32 bits). Le type `np.object` permet de mettre n'importe quel objet Python dans le tableau.

Définition

Le **calcul vectoriel** consiste à réaliser des opérations vectorielles ou matricielles plutôt que des boucles (`for`, `while`) classiquement utilisées pour les listes.

Ces opérations ont certes été programmées avec des boucles for, mais dans des langages de programmation plus rapides : en C pour `numpy`, en C et en Fortran pour `scipy`.

Le module `numpy` propose différentes fonctions qui permettent de créer rapidement des tableaux initialisés à des valeurs maîtrisées.

```
import numpy as np

z = np.zeros(4)
# renvoie un tableau contenant quatre 0 array([ 0., 0., 0., 0.])
x = np.linspace(0, 2*np.pi, 1001)
# renvoie un tableau unidimensionnel de 1001 valeurs allant de 0 à 2π, pas = 2π/1000
a = np.arange(3, 13, 2)
# renvoie le tableau array([ 3,  5,  7,  9, 11])
l = np.array([1, 5, 42])
# convertit la liste [1, 5, 42] en tableau
```

Pour se rendre compte de la différence de vitesse d'exécution exécuter les deux codes suivants :

```
import numpy as np
a = np.array([])
for i in range(1000000):
    np.append(a, i)
```

```
import numpy as np
a = np.arange(1000000)
```

D'une manière générale, les `ndarray` et les fonctions `numpy` permettent d'accélérer les calculs, et sont très utiles lorsqu'on traite un gros volume de données (images, données acquises pendant une expérience de physique, etc.).
Les calculs vectoriels se font ensuite de manière assez naturelle en ce qui concerne les opérateurs simples `+`, `-`, `*`, `/` et `**` qui renvoient les résultats **terme à terme**. Les fonctions mathématiques usuelles de `numpy` fonctionnent de la même manière (`cos`, `sqrt`, etc.), c'est-à-dire terme à terme.

Toutes les fonctions ne peuvent pas s'appliquer à des `ndarray` (voir exercice 4.2). Pour appliquer quand même une telle fonction `f` à un tableau `T`, il existe plusieurs solutions :

- Les listes par compréhension `X = [f(t) for t in T]`
- La fonction `vectorize`.

```
f_vectorisee = vectorize(f)
X = f_vectorisee(T)
```

Les fonctions `cos`, `sin` etc. du module `math` ont le même effet que celles du module `numpy` sur des flottants mais *ne sont pas vectorisées*. Par exemple

```
T = np.linspace(-np.pi, np.pi 201)

X = math.cos(T)  # ne fonctionne pas
X = np.cos(T)  # pas de problème !
```

Comme pour les listes, il est possible de faire du slicing [1] sur les `ndarray` et `len` renvoie la longueur.

■ 1 Tracés graphiques (module `matplotlib.pyplot`)

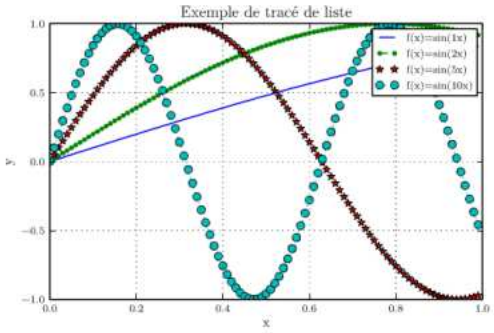
Le langage Python permet de tracer des courbes via le module `matplotlib` et plus particulièrement de son sous-module `pyplot` que l'on peut importer à l'aide de la commande

```
import matplotlib.pyplot as plt
```

Ce sous-module et en particulier sa fonction principale `plot` utilisent une syntaxe proche de celle utilisée par les logiciels de calcul numérique courant MATLAB® et Scilab® utilisés en SII.

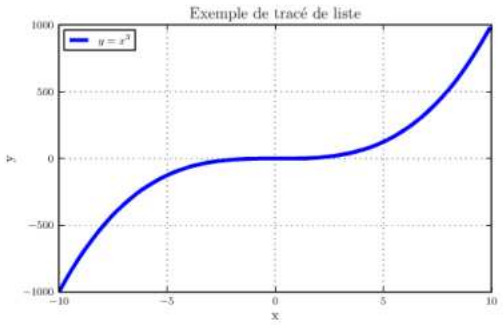
Un petit exemple en utilisant des listes pour décrire les styles de lignes et de marqueurs ainsi que la légende. On peut également faire la même chose pour les couleurs par exemple.

```
import matplotlib.pyplot as plt
import numpy as np
style_ligne = ['solid', 'dashed', 'dashdot', 'dotted']
style_marker = [' ', '.', '*', 'o']
k = [1, 2, 5, 10]
for i in range(len(k)):
    x = []
    y = []
    for j in range(100):
        x.append(j / 100)
        y.append(np.sin(k[i] * x[j]))
    plt.plot(x, y, linestyle=style_ligne[i],
             marker=style_marker[i], label='f(x)=sin('
             + str(k[i]) + 'x)')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Exemple de tracé de liste')
plt.legend()
plt.grid()
plt.show()
```



Il est possible de tracer des figures à partir de listes de valeurs numériques de type `float` :

```
import matplotlib.pyplot as plt
x = []
y = []
for i in range(-100, 101):
    x.append(i / 10)
    y.append((i / 10)**3)
plt.plot(x, y, label='$y=x^3$', linewidth=3)
plt.xlabel('x')
plt.ylabel('y')
plt.title('Exemple de tracé de liste')
plt.legend(loc=2)
plt.grid()
plt.show()
```

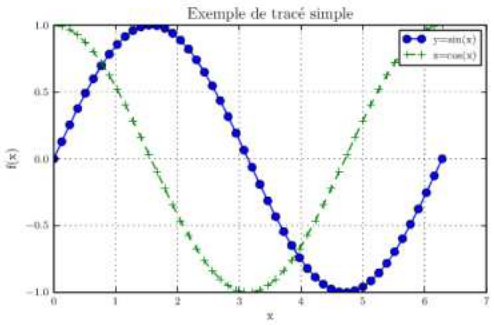


En réalité, la fonction `plt.plot` convertit le cas échéant ses deux premières entrées en tableau `array`.

On utilise très souvent la fonction `np.linspace` pour générer des subdivisions régulières.

1. Le slicing sur les `ndarray` fonctionne comme pour les listes, à une exception près. L'instruction `B=A[0:10:2]` fait une copie si `A` est une liste mais pas si `A` est un `ndarray`. Autrement dit, dans le cas où `A` est un `ndarray` mais pas dans le cas où `A` est une liste, modifier un coefficient de `A` modifie aussi `B` et *vice versa*.

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2 * np.pi, 51)
y = np.sin(x)
z = np.cos(x)
plt.plot(x, y, '-o', label='y=sin(x)')
plt.plot(x, z, '--+', label='z=cos(x)')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Exemple de tracé simple')
plt.legend()
plt.grid()
plt.savefig('trace_simple.png')
```



Nous n'avons ici que présenté des tracés de courbes élémentaires, mais `matplotlib` est très riche en possibilités graphiques (tracé d'histogrammes, courbes polaires, diagrammes log-log, cartographies, etc.). Il est impossible de traiter tous les cas ici et nous vous encourageons à consulter la documentation de `matplotlib` en fonction de vos besoins. D'autres utilisations sont notamment traitées dans le chapitre 5.

■ 2 Résolution numérique d'équations numériques

Nous présentons ici deux méthodes numériques permettant de résoudre des équations du type $f(x) = 0$: la méthode de la dichotomie et la méthode de Newton.
Le principe est le même : pour approximer le x_0 tel que $f(x_0) = 0$ on itère un certain procédé jusqu'à ce qu'une condition, appelée **critère de convergence**, soit satisfaite. Il existe plusieurs critères possibles dont les principaux sont résumés dans le tableau suivant (où r est le résultat de la méthode).

$f(r) = 0$	$\lvert x_0 - r\rvert \leqslant \varepsilon$	$\lvert f(r)\rvert < \varepsilon$	N **itérations.**
Précision infinie.	Valeur à ε près.	Erreur sur les ordonnées.	Temps de calcul borné.

Le premier critère, $f(r) = 0$, est utopique. Non seulement le calcul pourrait ne jamais terminer à cause des erreurs d'arrondis sur les flottants (on pourrait ne jamais tomber « pile » sur la solution) mais, en plus, pour certaines fonctions f, il n'existe même pas de nombre flottant r qui convienne. Par exemple[1] $f(t) = t^2 - 2$, il n'existe aucun nombre flottant r pour lequel le résultat numérique $f(r)$ soit exactement égal à 0 car $\sqrt{2}$ n'est pas exactement représentable en flottant.

Pour des exemples de problèmes d'arrondis, voir les exercices 2.4, 2.5 et 2.9.

Le second critère, $\lvert x_0 - r\rvert \leqslant \varepsilon$, correspond souvent à ce qu'on cherche : une approximation de x_0 avec une précision donnée. Il n'est pas toujours facile à satisfaire, car on ne peut pas calculer $\lvert x_0 - r\rvert$.
Le troisième critère, $\lvert f(r)\rvert < \varepsilon$, est facile à vérifier, mais peut mener à des valeurs de r éloignées de x_0. Avant de l'utiliser, il est préférable d'étudier la fonction f pour choisir judicieusement ε. On remarque que si f' est minorée par 1 au voisinage de x_0, alors ce critère entraine qu'on a aussi $\lvert x_0 - r\rvert \leqslant \varepsilon$.

1. Considérons `t1=2**0.5` et `t0` le flottant « juste avant » `t1`. Alors `f(t1)` renvoie `4.440892098500626e-16`, et `f(t0)` renvoie `-4.440892098500626e-16`.

Le dernier critère ne donne aucune garantie sur la précision du résultat, mais garantit que le temps de calcul ne sera pas trop long. Il est utile lorsque le temps est plus important que la précision.

Il est possible, et même conseillé, de combiner plusieurs critères d'arrêt, notamment un critère de précision avec un critère sur le nombre d'itérations.

Méthode de la dichotomie

Définition

La **dichotomie** est une méthode itérative simple permettant de déterminer une approximation d'une racine (ou zéro) sur un intervalle $[a, b]$, avec une précision ε d'une fonction continue et monotone f telle que $f(a)$ et $f(b)$ sont de signes opposés. Elle consiste comparer le signe de l'image $f\left(\dfrac{a+b}{2}\right)$ du milieu de l'intervalle $[a, b]$ avec le signe de $f(a)$ et $f(b)$ pour réduire l'intervalle de recherche de manière itérative. La figure 0 illustre cette méthode.

Pour déterminer x_0, racine de la fonction f avec une précision ε, strictement monotone sur l'intervalle $[a, b]$, on procède comme suit.

(0) On détermine le milieu de l'intervalle $m = \dfrac{a+b}{2}$;
(1) On compare le signe de $f(m)$ avec celui de $f(a)$ et $f(b)$ pour déterminer dans quel intervalle $[a, m]$ ou $[m, b]$ se trouve la racine x_0 ;
(2) On affecte à a (resp. b) la valeur de m si la racine se trouve entre m et b (resp. a).
(3) On itère tant que $|a - b| > \varepsilon$, la précision définie initialement et on renvoie m.

Terminaison de la dichotomie

Si le calcul sur les réels était exact, l'intervalle $[a, b]$ verrait sa longueur divisée par deux à chaque itération, donc le calcul terminerait toujours en $K = \left\lceil \log_2\left(\dfrac{b-a}{\varepsilon}\right)\right\rceil$ étapes.
Pour éviter que les erreurs de calculs sur les flottants nous mènent dans une boucle infinie, on peut, au lieu d'itérer tant que $|a - b| > \varepsilon$, itérer K fois.

Pour qu'il y ait existence d'une solution, il suffit que la fonction f soit continue et que les images respectives des deux bornes par la fonction soient de signes opposés. De plus, pour qu'il y ait unicité de la solution, il est suffisant que la fonction f soit strictement monotone sur l'intervalle $[a, b]$.

Correction de la dichotomie

Les itérations préservent l'invariant « $x_0 \in [a, b]$ ». Ainsi, à la fin des itérations, on a toujours $x_0 \in [a, b]$. On a de plus $|a - b[< eps$, donc $|m - x_0| < \varepsilon/2$. L'algorithme renvoie bien une approximation de x_0 avec une précision meilleure que ε.

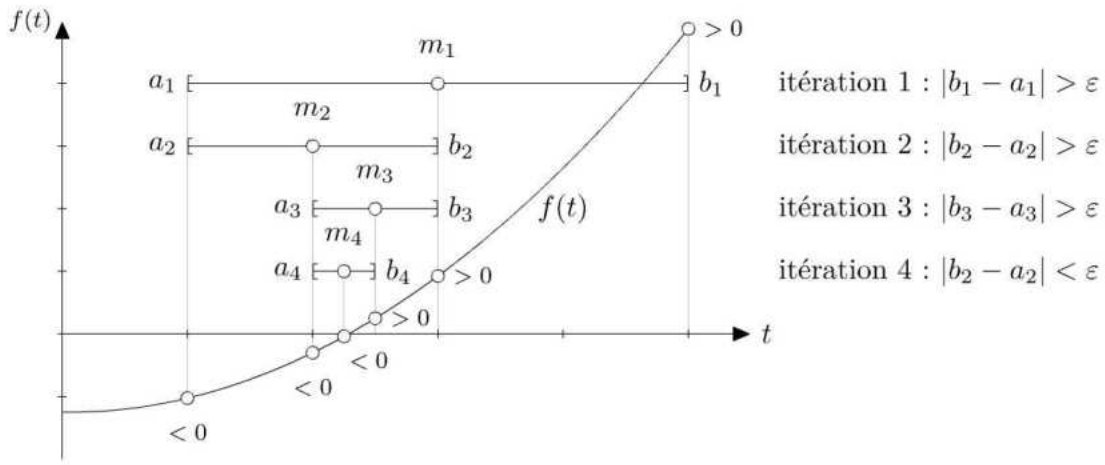
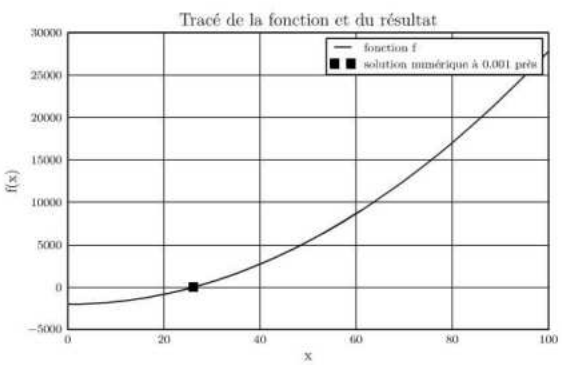


FIGURE 0. Algorithme de dichotomie

Exemple d'application

Déterminer la racine de la fonction $f : x \longmapsto f(x) = 3x^2 - 2x - 2000$ sur l'intervalle $[0, 100]$ avec une précision $\varepsilon = 10^{-3}$.

```
import numpy as np

def f(x):
    return 3 * x**2 - 2 * x - 2000

def dichotomie(f, eps, a, b, nbitermax=1000):
    ''' méthode de dichotomie
    précision eps, intervalle a, b
    renvoie racine, nombre itération
    si f(a) * f(b) > 0, renvoie 0.0, -1
    '''
    if f(a) * f(b) > 0:
        return 0., -1
    nbiter = 1
    m = (a + b) / 2
    while abs(a - b) > eps and nbiter < nbitermax:
        m = (a + b) / 2
        if f(a) * f(m) < 0:
            b = m
        else:
            a = m
        nbiter += 1
    return m, nbiter

# Recherche du zéro de la fonction f à 10**-3 près sur [0,100]
racine, iteration = dichotomie(f, 1e-3, 0, 100)
```



Le résultat numérique est $x_{num} = 26,1554718$, le résultat théorique[1] est $x_{th} = 26,1553738$ et $f(x_{num}) = 0,015$. On obtient bien la précision sur la valeur de x à 10^{-3} près. Notez que $f(x)$ n'est pas égal à 0 avec cette même précision.

1. Arrondie au septième chiffre après la virgule.

Méthode de Newton

Définition

La **méthode de Newton** est une méthode itérative de recherche d'un zéro x_0 d'une fonction dérivable f. On définit une suite d'approximations t_n comme suit :

- On fixe arbitrairement une valeur initiale t_0.
- Étant donné t_n, on construit la tangente à f en t_n. Cette tangente coupe l'axe des abscisses en t_{n+1} (figure 1)

On calcule les t_n jusqu'à ce que le critère de convergence soit atteint.

Proposition

La relation de récurrence satisfaite par t_n est :

$$t_{n+1} = t_n - \frac{f(t_n)}{f'(t_n)}$$

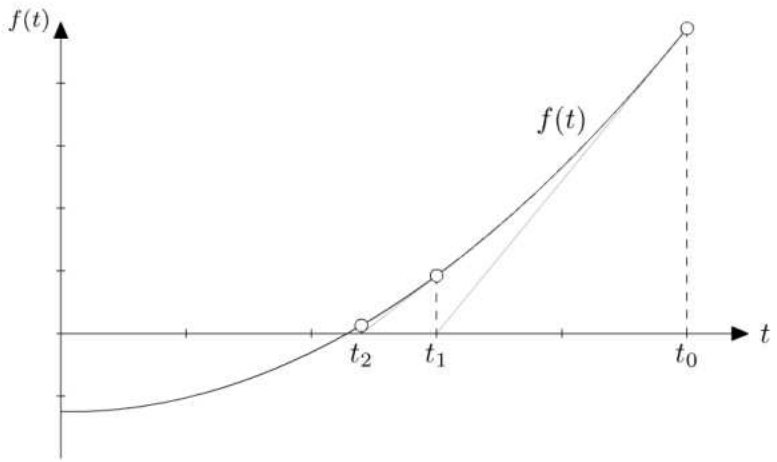


FIGURE 1. Algorithme de la méthode de Newton

La méthode de Newton est souvent plus rapide que la dichotomie mais peut ne pas aboutir pour certaines fonctions, notamment si le point de départ est mal choisi. Tester, par exemple, sur la fonction f_3 du TP 4.1 avec 1 ou 2 comme point de départ.
Pour certaines fonctions, nous savons déterminer des points de départ adéquats. Par exemple, dans le cas particulier où la fonction f est deux fois dérivable et où f' et f'' sont de signes constants sans s'annuler :

- Si f' et f'' sont de même signe, alors tout point de départ plus grand que la racine convient.
- Si f' et f'' sont de signes opposés, alors tout point de départ plus petit que la racine convient.

Pour la méthode de Newton, voir les exercices 4.9 et 4.10 ainsi que le TP 4.1.

Le (sous-)module `scipy.optimize` comporte les fonctions `bisect` (dichotomie) et `newton` (méthode de Newton).

■ 3 Résolution numérique d'équations différentielles

Définition

Si f est une application définie sur $\mathbb{R} \times \mathbb{R}^p$ à valeurs dans $\mathbb{R}^p$ (avec $p \in \mathbb{N}^*$), considérons :

- l'équation différentielle (E) : $x'(t) = f(t, x(t))$, parfois notée (E) : $x' = f(t, x)$ d'inconnue $x : \mathbb{R} \to \mathbb{R}^p$ avec $p \in \mathbb{N}^*$;
- la condition initiale $x(t_0) = x_0$ avec $t_0 \in \mathbb{R}$ et $x_0 \in \mathbb{R}^p$ deux paramètres.

La donnée de ces deux éléments constitue un **problème de Cauchy**.

En résumé, un problème de Cauchy est la donnée d'une équation différentielle avec une condition initiale. Par exemple, $x' = 1 + x^2, x(0) = 1$ est un problème de Cauchy admettant la fonction tangente pour solution.
Les mathématiciens démontrent, sous de bonnes hypothèses, l'existence et l'unicité de la solution φ d'un problème de Cauchy, mais il est en général impossible d'en calculer une expression, et même souvent difficile de décrire son domaine de définition.

Méthode d'Euler (ordre 1)

Si nous supposons que T est un réel non nul tel que $t_0 + T$ appartienne à ce domaine de définition, la **méthode d'Euler** permet d'approximer φ sur l'intervalle $J = [t_0, t_0 + T]$ (ou $J = [t_0 + T, t_0]$ si $T < 0$) par le biais d'un *schéma numérique* élémentaire :

- on fixe un entier $n \geqslant 1$ et on subdivise J à pas constant $p = \dfrac{T}{n}$, en posant $t_i = t_0 + i\,p$ pour tout $i \in \{0, 1, \ldots, n\}$;
- un développement limité donne $\varphi(t_1) \simeq \varphi(t_0) + (t_1 - t_0)\varphi'(t_0) = x_0 + pf(t_0, x_0)$, ce qui permet d'approximer $\varphi(t_1)$ par $x_1 = x_0 + pf(t_0, x_0)$;
- on définit selon le même schéma les valeurs $x_2, \ldots, x_n$:

$$\forall i \in \{0, 1, \ldots, n-1\},\ x_{i+1} = x_i + pf(t_i, x_i).$$

Cette construction est illustrée par la figure 2.
On peut espérer, sous de bonnes hypothèses, que les x_i soient des approximations correctes des $\varphi(t_i)$ quand n est suffisamment grand.

On dit que l'on a une **méthode explicite** car on dispose d'une expression explicite du calcul de y_{i+1} en fonction des calculs précédents. Il existe d'autres schémas, dits implicites, où l'expression de y_{i+1} est obtenue en résolvant une équation.

La précision de la méthode d'Euler est très dépendante du pas de temps choisi. Plus le pas de temps est petit, et donc plus le nombre de points calculés est grand, plus le calcul sera précis au début, mais plus le temps de calcul sera long. Il faut trouver un compromis en fonction de la situation.

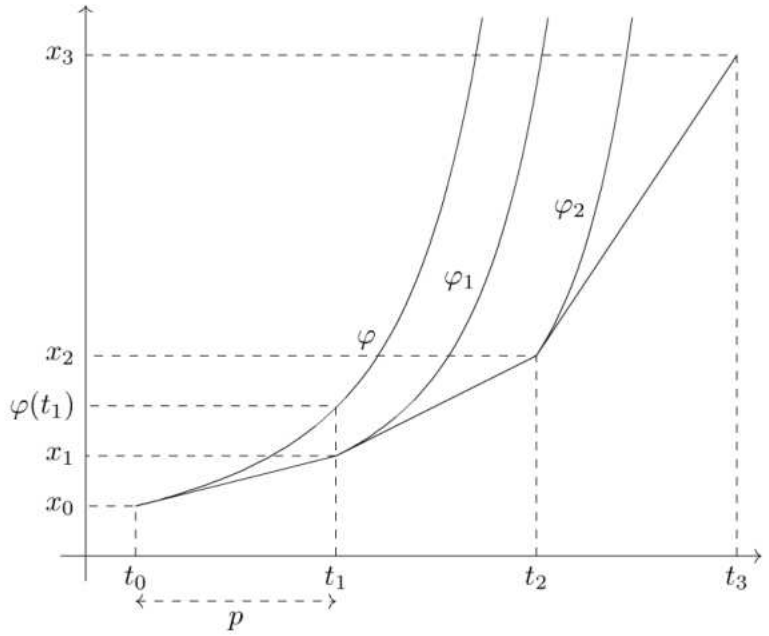


FIGURE 2. Méthode d'Euler pour $p = 1$.
φ_i est la solution à l'équation différentielle telle que $\varphi_i(t_i) = x_i$.

La bibliothèque `scipy.integrate` fournit la fonction `odeint` qui permet d'intégrer directement les équations différentielles et qui utilise des algorithmes plus rapides et plus précis que la méthode d'Euler présentée ici.

Méthode d'Euler (ordre 2)

Considérons l'équation différentielle d'ordre 2 (E) : $x'' = f(t, x, x')$ où f est une fonction définie sur $\mathbb{R} \times \mathbb{R}^p \times \mathbb{R}^p$ à valeurs dans $\mathbb{R}^p$ (avec $p \in \mathbb{N}^*$) ; l'inconnue x est donc une fonction deux fois dérivable définie sur un intervalle I et à dans $\mathbb{R}^p$. En posant $y = x'$, cette équation différentielle d'ordre 2 se ramène au système différentiel d'ordre 1 :

$$(E_1) : \begin{cases} x' &= y \\ y' &= f(t, x, y) \end{cases}$$

En imposant les conditions initiales $x(t_0) = x_0$ et $x'(t_0) = x'_0$, où $t_0 \in \mathbb{R}$ et $x_0, x'_0 \in \mathbb{R}^p$, nous nous ramenons donc à un problème de Cauchy pour l'équation (E_1), et il est ainsi possible d'approximer la solution cherchée en appliquant la méthode d'Euler à (E_1).

Les méthodes à maîtriser

Méthode 4.0 : Résoudre une équation numérique

Pour résoudre une équation numérique, on la met d'abord sous la forme $f(x) = 0$.
On choisit ensuite une méthode de résolution : dichotomie ou méthode de Newton.
Ce choix est souvent guidé par les contraintes numériques du problème :

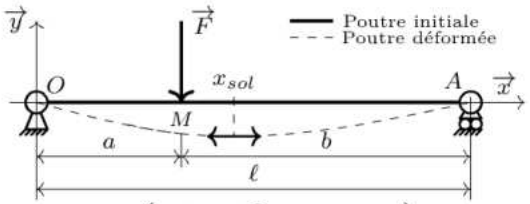
- pour mettre en œuvre la dichotomie, il faut au minimum connaître un intervalle $[a, b]$ pour lequel $f(a)$ et $f(b)$ sont de signes opposés ;
- pour mettre en œuvre la méthode de Newton, la dérivée doit être connue.
- si la vitesse est un critère important (résolution d'un grand nombre d'équations dépendant d'un paramètre par exemple), la méthode de Newton est à privilégier si possible ;
- la méthode de Newton peut ne pas aboutir (lorsque la dérivée est nulle ou très faible, lorsque des oscillations apparaissent entre autre cas). Lorsque la sécurité est importante la méthode de la dichotomie est à privilégier.

On choisit également un critère de convergence. Ce choix est souvent guidé en pratique par des paramètres physiques du problème. Par exemple si certains coefficients apparaissant dans l'équation sont expérimentaux et connus à une précision relative de 10^{-4}, il est inutile de résoudre l'équation avec une précision de 10^{-16}.
Enfin, on met en œuvre la méthode numérique choisie. Si le problème à résoudre est de nature concrète, on valide les résultats obtenus.

Exemple d'application

Résoudre l'équation de la déformée d'une poutre en flexion

On veut déterminer le lieu de la déformée maximale de la poutre ci-contre sous la charge $\overrightarrow{F}$. Une étude de RdM permet de montrer que cette solution est sur l'intervalle $[a, \ell]$ si $a < \dfrac{\ell}{2}$. On cherche donc la racine de $y'(x) = 0$ sur cet intervalle, avec $y(x)$ le déplacement suivant $\overrightarrow{y}$ de la poutre à l'abscisse x. Soit pour une poutre d'un mètre chargé à un tiers de sa longueur, l'équation à résoudre est :



$$y'(x) = \frac{-F}{EI_{Gz}}\left(\frac{a}{\ell}\frac{(x-\ell)^2}{2} - \frac{ab}{6}\frac{2a+b}{a+b}\right), x \in [a, \ell]$$

$$\frac{(x-1)^2}{6} - \frac{4}{81} = 0$$

On peut montrer que $y'(a) < 0$, $y'(\ell) > 0$ et que la fonction est monotone sur $[a, \ell]$.
On peut donc choisir d'effectuer une dichotomie avec un critère de convergence sur l'abscisse de $\varepsilon = 10^{-3}$, soit une erreur admissible d'un millimètre (à défaut ici d'utiliser la fonction `sqrt`...)

```
>>> dichotomie(f, 1e-3, 1 / 3, 1)
0.455078125
```

```
def f(x):
    return (x - 1)**2 / 6 - 4 / 81

def dichotomie(f, eps, a, b, itermax=1000):
    nbiter = 1
    m = (a + b) / 2
    while abs(a - b) > eps and nbiter < itermax:
        m = (a + b) / 2
        if f(a) * f(m) < 0:
            b = m
        else:
            a = m
        nbiter += 1
    return m
```

La déformée maximale a lieu en $x = 455$mm.

Méthode 4.1 : Tracer la courbe représentative d'une fonction f

Pour tracer la courbe représentative d'une fonction f, on :

(0) importe les modules nécessaires à ce travail :

```
import numpy as np
import matplotlib.pyplot as plt
```

(1) détermine l'intervalle $I = [a, b]$ sur lequel il est pertinent de tracer la courbe représentative de la fonction f

(2) discrétise l'intervalle I en le découpant en n sous-intervalles de même longueur. Cela peut par exemple se faire à l'aide de la fonction `linspace` du module `numpy` :

```
T = np.linspace(a,b,n+1)
```

(3) calcule la liste des images des points de `T` par la fonction f. Cela peut se faire à l'aide d'une construction de liste par compréhension :

```
Y = [f(t) for t in T]
```

ou encore par une opération vectorielle du module `numpy`.

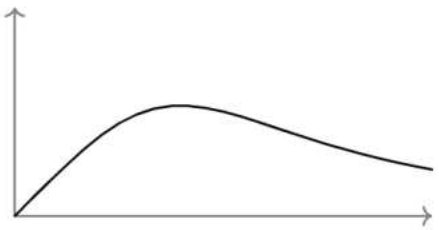
(4) on représente finalement la courbe :

```
plt.plot(T, Y)
plt.show()
```

Exemple d'application

Tracer sur $[0, 2]$ la courbe représentative de $f(x) = \dfrac{x}{1 + x^3}$.

```
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return x / (1 + x**3)

T = np.linspace(0, 2, 201)
Y = [f(t) for t in T]
plt.plot(T, Y)
plt.show()
```



Méthode 4.2 : Tracer une courbe paramétrée

Pour tracer une courbe paramétrée $(x(t), y(t))$, on :

(0) importe les modules nécessaires à ce travail :

```
import numpy as np
import matplotlib.pyplot as plt
```

(1) détermine l'intervalle $I = [a, b]$ du paramètre t sur lequel il est pertinent de tracer la courbe paramétrée. Lorsque les fonctions x et y sont toutes deux 2π-périodiques, l'intervalle $[0, 2\pi]$ convient notamment.

(2) discrétise l'intervalle I en le découpant en n sous-intervalles de même longueur. Cela peut par exemple se faire à l'aide de la fonction `linspace` du module `numpy` :

```
T = np.linspace(a, b, n+1)
```

(3) calcule la liste des images des points de `T` par les fonctions x et y. Cela peut se faire à l'aide d'une construction de liste par compréhension :

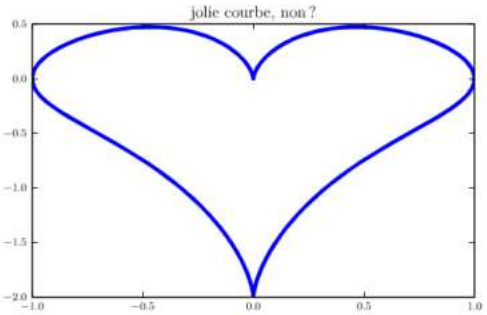
```
X = [x(t) for t in T]
```

```
Y = [y(t) for t in T]
```
ou encore par une opération vectorielle du module `numpy`.
(4) on représente finalement la courbe :
```
plt.plot(X, Y)
plt.show()
```

Exemple d'application

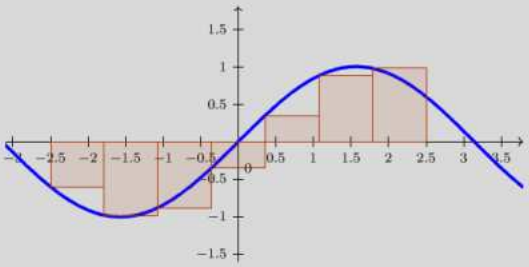
Tracer la courbe paramétrique définie par $\begin{cases} x &= \sin^3\theta \\ y &= \cos\theta - \cos^4\theta \end{cases}$ sur l'intervalle $\theta \in [-\pi, \pi]$.

```
import numpy as np
import matplotlib.pyplot as plt

T = np.linspace(-np.pi, np.pi, 501)

X = np.sin(T)**3
Y = np.cos(T) - np.cos(T)**4

plt.title('jolie courbe, non ?')
plt.plot(X, Y, linewidth=3)
plt.show()
```

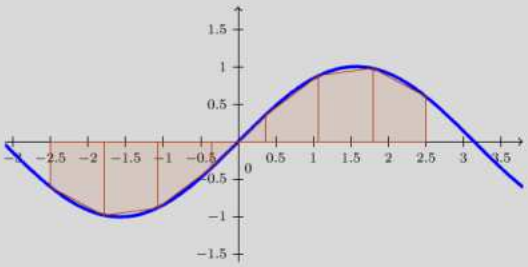


Méthode 4.3 : Intégration numérique (rectangles et trapèzes)

L'**intégration numérique** permet d'intégrer des fonctions dont on ne connaît pas l'expression exacte de l'intégrale, ainsi que les données issues d'acquisitions numériques.
De nombreuses méthodes d'intégration numérique existent.
Une méthode à pas constant consiste à subdiviser l'intervalle d'intégration $[a, b]$ en n écarts (= pas) de même longueur puis sur chaque petit intervalle $c \in [a_i, a_{i+1}]$ à effectuer un calcul approché de l'intégrale en remplaçant la fonction par une fonction beaucoup plus simple suivant un schéma bien défini.
Vous devez connaître les deux méthodes illustrées par les figures suivantes :



méthode des rectangles (à gauche)



méthode des trapèzes sur $[2.5, 2.5]$

- la **méthode des rectangles** pour laquelle on approxime la fonction f à intégrer par une fonction **constante** proche de f qui *interpole* f en un point $c \in [a_i, a_{i+1}]$, c peut valoir α (rectangle à gauche), β, $\dfrac{\alpha+\beta}{2}$ ou autre...

- la **méthode des trapèzes** pour laquelle on approxime la fonction f à intégrer par une fonction **affine** qui *interpole* f sur $[a_i, a_{i+1}]$ en a_i et a_{i+1}.

Voici un code Python implémentant ces deux méthodes :

```
def rectangle_gauche(f, a, b):
    '''
    méthode des rectangles à gauche
    renvoie l'integrale approchée de f entre a et b
    '''
    return f(a) * (b - a)

def trapeze(f, a, b):
    '''
    méthode des trapèzes
    '''
    return (f(a) + f(b)) * (b - a) / 2
```

```
def monint(f, a, b, n, methode):
    '''
    intègre entre a et b la fonction f
    en subdivisant [a, b] en n+1 points puis
    en appliquant la fonction methode(...) sur
    chaque subdivision [x_i, x_i+1]
    '''
    subdiv = np.linspace(a, b, n + 1)
    S = 0.
    for i in range(n):
        S += methode(f, subdiv[i], subdiv[i + 1])
    return S
```

Si le pas de temps est suffisamment petit, cette approximation est relativement précise. Elle est utilisée (en intégrant deux fois l'accélération mesurée par 3 accéléromètres) pour calculer la position, par exemple dans les sous-marins (qui n'ont pas toujours accès au signal GPS) et dans les fusées.

Le (sous-)module `scipy.integrate` contient
- la fonction `quad` qui permet une intégration numérique assez précise, utilisant des techniques qui ne sont pas détaillées dans cet ouvrage,
- la fonction `trapz` qui intègre avec la méthode des trapèzes.

Pour une étude des calculs approchés d'intégrales, voir le TP **4.0** p. 143.

Méthode 4.4 : Mettre en œuvre la méthode d'Euler (équation d'ordre 1)

Pour résoudre une équation différentielle d'ordre 1 à l'aide de la méthode d'Euler explicite, on :
- écrit l'équation sous la forme $x' = f(t, x)$;
- discrétise l'intervalle de résolution, en se posant éventuellement la question du nombre d'itérations à faire ;
- forme la relation de récurrence $x_{n+1} = x_n + hf(t_n, x_n)$;
- créé la liste des temps T
- initialise la liste X dans laquelle on va stocker les x_n
- écrit la boucle réalisant la relation de récurrence.

Exemple d'application

Résoudre sur [0,1] l'équation différentielle $x' + e^{-t}x = 0$ avec la condition initiale $x(0) = 1$.

Cette équation s'écrit $x' = -e^{-t}x$.

On choisit de résoudre cette équation en N pas et on discrétise donc l'intervalle de temps avec $t_n = \dfrac{n}{N}$.

La relation de récurrence s'écrit $x_{n+1} = x_n + \dfrac{1}{N}e^{-t_n}x_n$.

La résolution en Python s'écrit :

```
def resolution(N):
    T = [1 / N for i in range(N + 1)]
    X = [1]
    x, t = 1, 0
    for i in range(N):
        x += 1 / N * np.exp(t) * x
        t += 1 / N
        X.append(x)
    return T, X
```

Pour des applications de la méthode d'Euler à l'ordre 1, voir les exercices **4.11** à **4.17**.

Méthode 4.5 : Mettre en œuvre la méthode d'Euler (équation d'ordre 2)

Pour résoudre une équation différentielle d'ordre 2 à l'aide de la méthode d'Euler explicite, on :

- pose $y = x'$
- transforme l'équation différentielle $x'' = f(t, x, x')$ d'ordre 2 en un système différentiel d'ordre 1 sous la forme
$$\begin{cases} x' &= y \\ y' &= f(t, x, y) \end{cases}$$
- forme la relation de récurrence (vectorielle d'ordre 1) déduite de ce système différentiel par la méthode d'Euler explicite
- discrétise l'intervalle de résolution en se posant éventuellement la question du nombre d'itérations à faire.
- créé la liste des temps T
- initialise les listes X et Y
- écrit la boucle réalisant la relation de récurrence, en faisant attention à bien utiliser x_n et y_n pour calculer x_{n+1} et y_{n+1}

Exemple d'application

Résoudre sur [0,1] l'équation différentielle $x'' + \sin(tx) = 0$ avec les conditions initiales $x(0) = 1$ et $x'(0) = 0$.

Avec $y = x'$, cette équation est équivalente au système différentiel :
$$\begin{cases} x' &= y \\ y' &= -\sin(tx) \end{cases}$$
Ceci mène à la relation de récurrence :
$$\begin{cases} x_{n+1} &= x_n &+ hy_n \\ y_{n+1} &= y_n &- h\sin(t_n x_n) \end{cases}$$
qui peut s'écrire en Python :

```
def resolution2(N):
    T = [1 / N for i in range(N + 1)]
    X = [1]
    Y = [0]
    x, y, t = 1, 0, 0
    for i in range(N):
        t, x, y = t + 1 / N, x + y / N, y - np.sin(t * x) / N
        X.append(x)
        Y.append(y)
    return T, X, Y
```

Pour des applications de la méthode d'Euler à l'ordre 2, voir les exercices **4.18** à **4.21**.

Exercices

Tracé de courbes

Exercice 4.0 Autour de la moyenne

La fonction `binomial(n, p)` du module `numpy.random` simule une loi binomiale $\mathscr{B}(n,p)$.

0. Écrire une fonction `S(n, p)` qui simule une variable aléatoire $S_n = X/n$, où X suit la loi binomiale $\mathscr{B}(n,p)$.

1. En déduire une fonction `affiche(n,p)` qui affiche sur une même figure les courbes polygonales reliant les points (k, S_k), $\left(k, p - \sqrt{\frac{\ln k}{k}}\right)$ et $\left(k, p + \sqrt{\frac{\ln k}{k}}\right)$. Que remarque-t-on ?

Exercice 4.1

0. Donner les coordonnées dans un repère orthonormé des sommets A_k d'un polygone régulier à n côtés.

1. Écrire une fonction `polygone_regulier(n)` qui effectue le tracé d'un polygone régulier à n côtés.

2. Que se passe-t-il lorsque n devient grand ? On pourra essayer la fonction précédente avec $n = 1000$.

Exercice 4.2

On considère la fonction $f(t) = \displaystyle\sum_{k=0}^{+\infty} \frac{t^k}{(2k)!} = \begin{cases} \cos(\sqrt{|t|}) & \text{si } t < 0 \\ \mathrm{ch}(\sqrt{t}) & \text{si } t \geqslant 0. \end{cases}$ où ch est la fonction cosinus hyperbolique.

0. Définir la fonction `f` en Python à l'aide des fonctions cosinus et cosinus hyperbolique disponibles dans la bibliothèque `numpy`.

1. Définir la fonction `g` en Python, qui approxime f par l'expression $f(t) \approx g(t) = \displaystyle\sum_{k=0}^{20} \frac{t^k}{(2k)!}$.

La fonction `factorial` de la bibliothèque `scipy.misc` peut être utile.

2. Tracer la fonction `f` sur l'intervalle $[-200, 3]$, puis tracer `g` sur le même graphique et comparer.

Dichotomie

Exercice 4.3

0. Écrire une fonction `dicho(a, b, f, err)` qui, étant donné un intervalle réel `[a,b]`, une fonction `f` croissante de `[a,b]` dans $\mathbb{R}$ et s'annulant sur l'intervalle `[a,b]`, renvoie une approximation du zéro de `f` avec une erreur d'au plus `err`.

1. Utiliser la fonction `dicho` pour calculer une approximation de $\sqrt{2}$ à 10^{-2} près [1].
2. Que se passe-t-il si la fonction `f` n'est pas croissante mais continue et est telle que `f(a)<0<f(b)` ?
3. Réécrire la fonction `dicho` en ne supposant plus `f` croissante, mais seulement `f` monotone (croissante ou décroissante).

Exercice 4.4

Nous souhaitons estimer, dans une population donnée (des drosophiles, des électeurs, ou autre), la proportion p d'individus ayant une certaine caractéristique. Pour ce faire, nous réalisons un sondage : n individus sont tirés aléatoirement (uniformément, avec remise) dans la population totale, et nous comptons le nombre X d'individus ayant le caractère recherché. Ainsi X est une variable aléatoire suivant une loi binomiale de paramètres n, p.
Connaissant p, nous savons déterminer un intervalle de fluctuation $I_{\alpha,p,n}$, c'est-à-dire un intervalle tel que $\mathbb{P}(X \notin I_{\alpha,p,n}) \leqslant \alpha$. Nous utilisons ici l'intervalle donné par la fonction `binom.interval` de la bibliothèque `scipy.stats`.
Dans le but d'estimer p, nous utilisons un intervalle aléatoire $C_{\alpha,X,n}$, appelé intervalle de confiance, tel que $\forall p \in [0,1], \mathbb{P}(p \notin C_{\alpha,X,n}) \leqslant \alpha$. L'intervalle suivant convient :

$$C_{\alpha,X,n} = \{p \in [0,1] \mid X \in I_{\alpha,p,n}\}.$$

L'idée de $C_{\alpha,X,n}$ est la suivante : on a X (le résultat du tirage) et α. On suppose que X est dans $I_{\alpha,p,n}$ (hypothèse raisonnable car cet évènement arrive avec probabilité $1-\alpha$ au moins), et on en déduit les p possibles.

0. Écrire une fonction `PlusGrandP` qui, étant donné un risque d'erreur α, une réalisation de la variable aléatoire X et le nombre de sondés n, renvoie le plus grand p tel que $X \in I_{\alpha,p,n}$ (à 10^{-6} près). On remarque que le p recherché appartient à l'intervalle $[X/n, 1]$.
1. La fonction précédente renvoie la borne inférieure de l'intervalle de confiance. Écrire une fonction `confiance(alpha, X, n)` qui renvoie les bornes de l'intervalle de confiance à 10^{-6} près.

En 2002, J.M. Le Pen surprend de nombreux journalistes en se qualifiant au second tour avec 16,86% des voix. Peu avant les élections, plusieurs sondages [**Gr 11**] ont donné les résultats suivants :

Sondage	Taille de l'échantillon (n)	Intention de vote ($\approx X/n$)
IPSOS/Le Figaro (17-18/04)	989	14%
IFOP/L'Express (12-13/04)	1006	10,5%
BVA/Paris-Match (04-06/04)	963	12%

2. Parmi les sondages précédents, dans quels cas $p = 16.86\%$ est-il dans l'intervalle de confiance au risque d'erreur 5% ? Autrement dit, pour quels sondages l'évènement $p \in C_{5\%,X,n}$ s'est-il réalisé ? Même question pour $\alpha = 10\%$ et $\alpha = 1\%$.
3. Tracer sur un même graphique la borne supérieure et la borne inférieure de l'intervalle de confiance en fonction de α, pour $X = 140$ et $n = 1000$.
4. Pour chaque sondage, calculer par dichotomie, à 10^{-4} près, le plus grand α tel que l'évènement $p \in C_{\alpha,X,n}$ s'est réalisé.

1. En n'utilisant que les opérations arithmétiques de base `+`, `-`, `*`, `/`, `//` et `%`, bien évidemment et sans utiliser `**0.5`

Intégration numérique

Exercice 4.5

Nous souhaitons reprogrammer la fonction logarithme népérien à partir des opérations arithmétiques de base `+`, `-`, `*`, `/`, `//` et `%`.

Pour cela, nous utilisons la relation $\ln(x) = \displaystyle\int_1^x \frac{1}{t}\mathrm{d}t$.

0. Programmer une fonction `ln` qui calcule le logarithme népérien par la méthode des rectangles avec 1000 rectangles.

1. Tracer sur un même graphique votre fonction `ln` et la fonction `log` de la bibliothèque `numpy` et comparer.

En pratique, le logarithme en base 2 est déjà programmé dans le processeur (instruction FYL2X ou FYL2XP1 dans les processeurs Intel), et on l'utilise pour calculer le logarithme dans d'autres bases (`ln` ou `log10`).

Exercice 4.6 Fonction Gamma (d'après un oral de Centrale 2015)

On pose $m(x,t) = t^{x-1}e^{-t}$ puis $\varphi_{\alpha,\beta}(x) = \displaystyle\int_\alpha^\beta m(x,t)\mathrm{d}t$. Enfin, on pose $\Gamma(x) = \displaystyle\int_0^{+\infty} m(x,t)\mathrm{d}t$.

0. Écrire une fonction `phi` qui prend en argument α, β, x et n et calcule $\varphi_{\alpha,\beta}(x)$ avec la méthode des rectangles et n rectangles.

1. Représenter Γ sur un intervalle raisonnable.

Exercice 4.7 Oral Centrale 2016

On considère la fonction $f(x) = \displaystyle\int_0^\pi \cos(x \times \cos(t))\mathrm{d}t$

La fonction `quad` de la bibliothèque `scipy.integrate` permet d'intégrer une fonction. Elle prend en argument la fonction à intégrer et les deux bornes. Elle renvoie la valeur de l'intégrale avec une approximation de l'erreur.

0. Définir la fonction f en Python à l'aide de `quad`.

1. Tracer la fonction sur $[0, 10]$.

Exercice 4.8 D'après Mines 2015

Toutes les deux millisecondes, on mesure en ampères le courant électrique I dans un circuit. Les mesures sont stockées dans la liste `mesure`.

L'intensité moyenne est définie comme suit : $I_{\text{moy}} = \dfrac{1}{t_{\text{final}}} \displaystyle\int_0^{t_{\text{final}}} I(t)\mathrm{d}t$

0. Écrire une fonction `Imoy` prenant en entrée une liste de mesures et renvoyant l'intensité moyenne en ampères après l'avoir calculée par la méthode des trapèzes.

1. On suppose à présent que les mesures ne sont plus faites toutes les deux millisecondes mais à des intervalles de temps irréguliers. Écrire une fonction `Imoy` prenant en entrée une liste de mesures `mesure` ainsi qu'une liste de temps `temps` (les instants où ces mesures ont été prises) et renvoyant l'intensité moyenne en ampères après l'avoir calculée par la méthode des trapèzes.

Méthode de Newton

Exercice 4.9

On considère l'équation $f(x) = x^3 - 2x^2 + 1 = 0$

0. Implémenter la méthode de Newton avec comme critère d'arrêt $|f(x)| < \varepsilon$ et comme point de départ un paramètre x_0.

1. Résoudre l'équation précédente pour x_0 qui varie entre -1 et 3 par pas de 0.01 et représenter graphiquement la valeur de la solution trouvée en fonction de x_0.

Exercice 4.10 D'après un exercice d'oral de Centrale (PSI 2015)

Pour tout n de $\mathbb{N}$ on considère la fonction polynomiale $P_n(t) = \sum_{i=0}^{n} \frac{t^i}{i!}$ et on s'intéresse ici aux racines de ce polynôme.

0. Donner à l'écran des représentations graphiques de P_n sur des intervalles adaptés pour n dans $\{2, 3, 4, 5, 6, 7\}$. Que constate-t-on quant aux racines réelles de P_n suivant n ?

1. Mettre en œuvre la méthode de Newton (ou méthode de la tangente) pour la recherche d'une valeur approchée décimale d'une solution réelle de l'équation $P_n(t) = 0$, et déterminer ainsi les éventuelles racines réelles de cette équation pour n dans $\{2, 3, 4, 5, 6, 7\}$.

Extrait de la documentation donnée par le concours

La classe `Polynomial` du module `numpy.polynomial` permet de travailler avec des polynômes.

```
from numpy.polynomial import Polynomial
```

2. En utilisant la méthode `.root()` de la classe `Polynomial`, vérifier les résultats précédents.

3. Représenter à l'écran toutes les racines complexes de P_n dans les cas où $n = 3$, $n = 5$, $n = 8$ et $n = 15$.

Méthode d'Euler (ordre 1)

Exercice 4.11

Considérons l'équation différentielle $y' = 1 + y^2$ avec la condition initiale $y(0) = 0$.

0. Résoudre par la méthode d'Euler cette équation sur $[0, 1]$ avec un pas de 0.05 et tracer la fonction solution.

1. Tracer la solution sur l'intervalle $[-1.5, 1.5]$.

2. Tracer la fonction tangente sur le même graphique et comparer.

3. Que se passe-t-il lorsqu'on essaye de tracer la solution sur $[-2, 2]$?

Exercice 4.12

Nous considérons ici une équation différentielle scalaire $x' = f(t, x)$, i.e. une équation différentielle dont la fonction inconnue x est à valeurs réelles.

0. Écrire une fonction `Euler(f, t0, x0, T, n)` utilisant la méthode d'Euler pour tracer une approximation du graphe de la solution du problème $\big(x' = f(t, x),\ x(t_0) = x_0\big)$ sur $[t_0 - T, t_0 + T]$.

1. Modifier cette fonction en `Euler_Bis(f, t0, L, T, n)` qui prend en argument une liste $L = [x_0^1, x_0^2, \ldots, x_0^q]$ de valeurs initiales, au lieu de l'unique valeur x_0, et qui renvoie dans la même fenêtre graphique les graphes des q solutions correspondant à ces différentes valeurs initiales sur l'intervalle $[t_0 - T, t_0 + T]$.

Tester votre fonction avec l'équation $x' = t\sin(t + x)$, $t_0 = 0$, $T = 4$ et $n = 100$, pour une famille de valeurs initiales comprises entre -7 et 4.

Exercice 4.13 Modèle de Lotka-Voltera

De nombreuses modélisations font intervenir des systèmes différentiels autonomes. C'est le cas par exemple du modèle de Lotka-Voltera qui modélise l'évolution de deux populations, l'une constituée de proies et l'autre de prédateurs. Nous partons du modèle malthusien élémentaire : en l'absence de prédateurs, la fonction x représentant le nombre de proies vérifie l'équation différentielle $x' = ax$, où a est une constante égale à la différence entre le taux de natalité et le taux de mortalité des proies (avec $a > 0$). De même, en l'absence de proies, le nombre y de prédateurs vérifie l'équation différentielle $y' = -by$, où b est également une constante positive. L'interaction des deux populations se fait en introduisant dans ces deux équations un terme proportionnel à la fois à x et à y, qui rend compte de la probabilité de rencontre d'une proie et d'un prédateur. Ces rencontres étant évidemment favorables aux prédateurs, nous obtenons un système différentiel de la forme :

$$\begin{cases} x' = ax - cxy \\ y' = -by + dxy \end{cases}$$

où a, b, c, d sont des réels strictement positifs. Ce système est qualifié d'autonome car le temps t n'apparaît pas explicitement dans l'équation différentielle. Ainsi, l'instant initial t_0 ne jouera pas de rôle particulier dans l'étude des solutions, sinon de translater les solutions dans le temps. Nous choisirons donc $t_0 = 0$.

Dans tout l'exercice, f et g sont deux fonctions définies sur $\mathbb{R}^2$ et à valeurs réelles et nous noterons (E) le système différentiel autonome $\begin{cases} x' = f(x, y) \\ y' = g(x, y) \end{cases}$ où les inconnues x et y sont deux fonctions (dérivables) de la variable t. Les fonctions demandées seront testées sur l'équation de Lotka-Voltera de paramètres $a = 0.2$, $b = 0.3$, $c = 0.1$ et $d = 0.15$, pour différentes conditions initiales positives.

0. Écrire une fonction `Calcul_Euler(f, g, x0, y0, T, p)` qui renvoie les listes $Lt = [t_0, \ldots, t_n]$, $Lx = [x_0, \ldots, x_n]$ et $Ly = [y_0, \ldots, y_n]$ obtenues en appliquant la méthode d'Euler à l'équation (E) sur l'intervalle $[0, T]$, avec le pas p (nous supposerons que T est un multiple de p) et les conditions initiales $x(0) = x_0$ et $y(0) = y_0$.

Écrire une fonction `Graphe(f, g, x0, y0, T, p)` qui applique la méthode d'Euler à l'équation (E) sur l'intervalle $[0, T]$, avec le pas p et les conditions initiales $x(0) = x_0$ et $y(0) = y_0$, et ouvre une fenêtre graphique contenant les graphes des fonctions x et y sur $[0, T]$.

1. Écrire une fonction `Phase(f, g, x0, y0, T, p)` qui applique la méthode d'Euler à l'équation (E) sur l'intervalle $[0, T]$, avec le pas p et les conditions initiales $x(0) = x_0$ et $y(0) = y_0$, et ouvre une fenêtre graphique contenant le **portrait de phase**, i.e. le support de l'arc paramétré $t \longmapsto (x(t), y(t))$, toujours pour t décrivant $[0, T]$.
2. Écrire une fonction `Graphe_Phase(f, g, x0, y0, T, p)` qui ouvre une fenêtre graphique contenant côte à côte les deux graphiques renvoyés par les fonctions précédentes.
3. Si (x, y) est une solution exacte à valeurs strictement positives de ce système, on peut montrer que la fonction $a \ln y + b \ln x - cy - dx$ est constante, puis que les fonctions x et y sont périodiques de même période. Commenter les tracés obtenus au regard de ces résultats théoriques.
4. Un élève a proposé la fonction :

```
def Graphe_Phase(f, g, x0, y0, T, p):
    Lt = [0]
    Lx = [x0]
    Ly = [y0]
    for i in range(int(T / pas)):
        Lt.append(Tt[-1] + p)
        Lx.append(Lx[-1] + p * f(Lx[-1], Ly[-1]))
        Ly.append(Ly[-1] + p * g(Lx[-1], Ly[-1]))
    plt.figure()
    plt.subplot(1, 2, 1)
    plt.plot(Lt, Lx, color='b')
    plt.plot(Lt, Ly, color='r')
    plt.subplot(1, 2, 2)
    plt.plot(Lx, Ly, color='g')
    plt.show()
```

Que penser de la réponse de cet élève ? Tester sa fonction et commenter les résultats obtenus pour $T = 500$.

Exercice 4.14 Inspiré de Centrale Physique-Chimie, MP 2013

L'addition d'eau à de l'oxyde d'éthylène (noté ici O) provoque la formation de glycol (noté ici E) selon la réaction suivante : $O + H_2O \to E$.
Une réaction concurrente produit du diéthylèneglycol (noté ici D) : $O + E \to D$.
Notons ξ_1 et ξ_2 les avancements volumiques respectifs de chacune de ces deux réactions et notons $[X]$ la concentration du composé X.

0. Exprimer les concentrations $[O]$, $[E]$ et $[D]$ en fonction de ξ_1 et ξ_2, sachant que les concentrations initiales de H_2O et de O sont de $c_0 = 1.00\ \text{mol} \cdot \text{L}^{-1}$.
Indication. On remarquera que $[H_2O] = c_0 - \xi_1$.

Les chimistes nous donnent les équations différentielles suivantes, où $k_1 = 1\text{ua}$ et $k_2 = 5\text{ua}$ (ua signifie « unité arbitraire ») sont les constantes de réaction des deux réactions :

$$\begin{cases} \mathrm{d}\xi_1/\mathrm{d}t &= k_1[O] \cdot [H_2O] \\ \mathrm{d}\xi_2/\mathrm{d}t &= k_2[O] \cdot [E] \end{cases}$$

1. À l'aide de la méthode d'Euler, tracer entre 0 et 10 unités de temps, l'évolution des concentrations des différents composés intervenant dans les deux réactions.

Vous pouvez comparer les résultats obtenus aux courbes dessinées dans le sujet de Centrale (disponible sur le site du concours).

Exercice 4.15 Attracteur étrange de Lorenz

On considère la solution φ_{x_0,y_0,z_0} de l'équation de Lorenz :

$$\begin{cases} x' = 10(y-x) \\ y' = 28x - y - xz \\ z' = xy - \frac{8}{3}z \end{cases}$$

qui prend la valeur (x_0, y_0, z_0) à l'instant initial $t_0 = 0$. Pour tout réel $p > 0$, on note $\varphi_{x_0,y_0,z_0,p}$ l'approximation de φ_{x_0,y_0,z_0} obtenue en appliquant la méthode d'Euler avec le pas p.

0. Écrire une fonction `Lorenz(x0, y0, z0, T, p)` qui trace la trajectoire de l'arc $t \longmapsto \varphi_{x_0,y_0,z_0,p}(t)$ sur l'intervalle $[0, T]$ (dans tout l'exercice, on supposera que T est toujours un multiple du pas choisi). Une fois calculées les listes $Lx = [x_0, x_1, \ldots, x_n]$, $Ly = [y_0, y_1, \ldots, y_n]$ et $Lz = [z_0, z_1, \ldots, z_n]$, on utilisera la fonction `Axe3D` du module `mpl_toolkits.mplot3d`.

1. Que penser des trajectoires obtenues pour les conditions initiales $(1, 1, 1)$ et $(1.001, 1, 1)$, avec $T = 100$ et $p = 0.001$?

Écrire une fonction `Chaos_Conditions_Initiales(x0, y0, z0, X0, Y0, Z0, T, p)` qui ouvre trois fenêtres graphiques, contenant respectivement les graphes de $\varphi_{x_0,y_0,z_0,p}$, de $\varphi_{X_0,Y_0,Z_0,p}$ et de $\varphi_{X_0,Y_0,Z_0,p} - \varphi_{x_0,y_0,z_0,p}$ sur l'intervalle $[0, T]$.

Interpréter les résultats observés pour $(x_0, y_0, z_0) = (1, 1, 1)$ et $(X_0, Y_0, Z_0) = (1.001, 1, 1)$.

2. Que penser des trajectoires obtenues pour la condition initiale $(1, 1, 1)$, toujours avec $T = 100$ mais avec $p_1 = 0.001$ et $p_2 = 0.0001$?

Écrire une fonction `Chaos_Pas(x0, y0, z0, T, p)` qui ouvre une fenêtre graphique contenant le graphe de $\varphi_{x_0,y_0,z_0,p} - \varphi_{x_0,y_0,z_0,p/10}$ sur l'intervalle $[0, T]$. Interpréter les résultats observés pour $(x_0, y_0, z_0) = (1, 1, 1)$ et $p = 0.01$.

Exercice 4.16 Vitesse de convergence de la méthode d'Euler

Le problème de Cauchy $x' = t + x$, $x(0) = 1$ a une unique solution : la fonction $\varphi : t \longmapsto e^t - 1 - t$. Pour $n \in \mathbb{N}^*$ et $T > 0$, notons $\Phi(n, T)$ l'approximation de $\varphi(T)$ obtenue en appliquant la méthode d'Euler à ce problème de Cauchy sur l'intervalle $[0, T]$ avec le pas $p = T/n$.

0. Écrire une fonction `Epsilon(n, T)` qui, appliquée à (n, T), renvoie la valeur de l'erreur $\varepsilon(n, T) = |\Phi(n, T) - \varphi(T)|$.

1. En fixant quelques valeurs de T, estimer l'ordre de grandeur (en fonction de n) de $\varepsilon(n, T)$.

2. De même, en fixant la valeur du pas p, estimer l'ordre de grandeur (en fonction de T) de l'erreur $\varepsilon(n, T)$.

Exercice 4.17 Améliorations de la méthode d'Euler

La méthode d'Euler est basée sur l'approximation :

$$(1) \qquad \varphi(t_1) \simeq \varphi(t_0) + (t_1 - t_0)m_0$$

où $m_0 = f(t_0, x_0)$ est la dérivée de φ en t_0. Pour tenter d'améliorer la méthode d'Euler, on peut avoir l'idée de remplacer m_0 par un meilleur coefficient directeur, qui prendra en compte les variations de φ'. Cela peut se faire de deux façons élémentaires :

- *Méthode de type « point milieu ».*
  Un calcul élémentaire montre que l'approximation :
  $$\varphi(t_1) \simeq \varphi(t_0) + (t_1 - t_0)\varphi'\left(\frac{t_0 + t_1}{2}\right)$$
  est bien meilleure (quand φ est assez régulière) que (1). En posant $t_{1/2} = \frac{1}{2}(t_0 + t_1)$, nous savons que $\varphi'(t_{1/2}) = f\left(t_{1/2}, \varphi(t_{1/2})\right)$, mais on ne connaît pas $\varphi(t_{1/2})$: on va donc utiliser la méthode d'Euler pour approximer cette valeur, en écrivant
  $$\varphi(t_{1/2}) \simeq x_0 + \frac{p}{2} f(t_0, x_0) = x_{1/2}.$$
  On obtient ensuite l'approximation :
  $$\varphi(t_1) \simeq x_0 + p f(t_{1/2}, x_{1/2}) = x_0 + p\, f\left(t_0 + \frac{p}{2}, x_0 + \frac{p}{2} f(t_0, x_0)\right)$$
  Cette méthode conduit au schéma numérique :
  $$\forall i \in \{0, 1, \ldots, n-1\},\ x_{i+1} = x_i + p\, f\left(t_i + \frac{p}{2}, x_i + \frac{p}{2} f(t_i, x_i)\right)$$
- *Méthode de type « trapèze ».*
  Un calcul tout aussi élémentaire montre que l'approximation :
  $$\varphi(t_1) \simeq \varphi(t_0) + (t_1 - t_0)\,\frac{\varphi'(t_0) + \varphi'(t_1)}{2}$$
  est également bien meilleure que (1) (toujours quand φ est assez régulière). Nous connaissons $\varphi'(t_0) = f(t_0, x_0) = m_0$ et nous allons une nouvelle fois approximer $\varphi'(t_1)$ grâce à la méthode d'Euler :
  $$\varphi'(t_1) = f(t_1, \varphi(t_1)) \simeq f(t_1, X_1)$$
  avec $X_1 = x_0 + p m_0$. Cette méthode conduit au schéma numérique :
  $$\forall i \in \{0, 1, \ldots, n-1\},\ \begin{cases} m_i = f(x_i, t_i) \\ X_{i+1} = x_i + \delta\, f(x_i, t_i) \\ m'_i = f(t_{i+1}, X_{i+1}) \\ x_{i+1} = x_i + p\,\dfrac{m_i + m'_i}{2} \end{cases}$$

0. Écrire le code des fonctions `E(f, t0, x0, T, n)`, `PM(f, t0, x0, T, n)` et `TR(f, t0, x0, T, n)` qui appliquent les méthodes d'Euler, du point milieu et des trapèzes (avec le pas T/n) et renvoient une valeur approchée de la valeur en $t_0 + T$ du problème de Cauchy $(x' = f(t, x),\ x(t_0) = x_0)$.

1. Tester ces fonctions avec le problème de Cauchy $\left(x' = tx^2,\ x(0) = 1\right)$, dont la solution est la fonction $\varphi :\ t \longmapsto \dfrac{2}{2 - t^2}$, définie sur $]-\sqrt{2}, \sqrt{2}[$. Comparer ces méthodes entre elles.

2. Appliquer les deux nouvelles méthodes pour tracer le portrait de phase de l'équation de Lotka-Voltera étudiée à l'exercice **4.13** p. 136. Commenter les résultats obtenus.

Méthode d'Euler (ordre 2)

Exercice 4.18 Exercice d'oral de Centrale (2015)

Considérons l'équation différentielle $(E) : (1-x)^3 y''(x) = y(x)$. On note f l'unique solution de (E) sur l'intervalle $]-\infty, 1[$ vérifiant les conditions initiales $f(0) = 0$ et $f'(0) = 1$. En utilisant la méthode d'Euler, tracer une approximation du graphe de f sur $[0, 0.9]$.

Exercice 4.19

L'application $\varphi : t \longmapsto \sin t$ est la solution de problème de Cauchy $\big(x'' = -x,\ x(0) = 0,\ x'(0) = 1\big)$. Pour $n \in \mathbb{N}^*$ et $T > 0$, notons $\Phi(n, T)$ l'approximation obtenue en appliquant la méthode d'Euler à ce problème sur l'intervalle $[0, T]$, pour le pas $p = T/n$.

0. Écrire une fonction qui, appliquée à (n, T), renvoie la valeur $\Phi(n, T)$. En fixant quelques valeurs de T, estimer l'ordre de grandeur (en fonction de n) de l'erreur $|\Phi(n, T) - \varphi(T)|$.

1. Pour améliorer la méthode d'Euler, on peut penser à utiliser la meilleure approximation :

$$\varphi(t_1) \simeq \varphi(t_0) + p\,\varphi'(t_0) + \frac{p^2}{2}\varphi''(t_0) = x_0 + p\,x'_0 + \frac{p^2}{2} f(t_0, x_0, x'_0)$$

ce qui conduit au schéma numérique :

$$\forall i \in \{0, 1, \ldots, n-1\},\ \begin{cases} x_{i+1} = x_i + p\,x'_i + p^2/2\, f(t_i, x_i, x'_i) \\ \\ x'_{i+1} = x'_i + p\, f(t_i, x_i, x'_i) \end{cases}$$

Écrire une fonction qui, appliquée à (n, T), renvoie la valeur $\Phi_1(n, T)$ qui approxime $\varphi(T)$ par le biais de cette méthode. En fixant différentes valeurs de T, comparer les erreurs $|\Phi(n, T) - \varphi(T)|$ et $|\Phi_1(n, T) - \varphi(T)|$. Commenter le résultat obtenu.

Exercice 4.20

Nous considérons ici une équation différentielle scalaire $x'' = f(t, x, x')$, i.e. une équation différentielle dont la fonction inconnue x est à valeurs réelles.

0. Écrire une fonction `Euler(f, t0, x0, xprime0, T, P)` qui applique la méthode d'Euler au problème de Cauchy $\big(x'' = f(t, x, x'),\ x(t_0) = x_0,\ x'(t_0) = x'_0\big)$ sur $[t_0, t_0 + T]$ avec le pas p (on supposera que T est un multiple de p), puis ouvre deux fenêtres graphiques, l'une contenant le tracé du graphe de la fonction φ sur $[0, T]$, l'autre contenant le portait de phase, i.e. le support de l'arc paramétré $t \longmapsto (\varphi(t), \varphi'(t))$.

1. Nous souhaitons maintenant travailler sur une famille de conditions initiales $\big((x_{0,i}, x'_{0,i})\big)_{1 \leqslant i \leqslant q}$, qui sera représentée par la liste $L = [[x_{0,1}, x'_{0,1}], \ldots, [x_{0,q}, x'_{0,q}]]$.

Écrire une fonction `EulerPhase(f, t0, L, T, p)` qui applique la méthode d'Euler pour chacune des q conditions initiales et renvoie le tracé des q solutions approchées dans l'espace des phases.

2. Tester les fonctions précédentes sur les équations classiques :

a. Oscillateur harmonique : $x'' + x = 0$;

b. Oscillateur de Van der Pol : $x'' + (x^2 - 1)x' + x = 0$;

c. Pendule pesant : $x'' + \sin x = 0$;

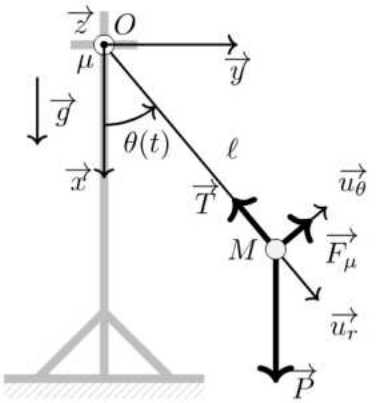
d. Pendule pesant amorti : $x'' + \frac{1}{5}\, x' + \sin x = 0$;

3. Il arrive qu'une solution φ d'un problème de Cauchy soit définie sur un intervalle $I =]\alpha, \beta[$ avec α fini (resp. β fini), et que le point $(\varphi(t), \varphi'(t))$ tende vers l'infini quand t tend vers α (resp. vers β). C'est ce qui se produit dans les cas **b.** et **d.** précédents. Nous allons modifier la fonction `EulerPhase` :

- au lieu d'étudier la trajectoire dans l'espace des phases pour $t \in [t_0, t_0 + T]$, nous allons l'étudier pour $t \in [t_0 - T, t_0 + T]$ (autrement-dit, nous souhaitons étudier ce qui s'est passé avant et ce qui va se passer après l'instant initial t_0) ;
- pour éviter les calculs aberrants, nous prenons en paramètre une fenêtre graphique, i.e. un rectangle $[a, b] \times [c, d]$, et nous arrêtons le calcul des (x_i, x_i') dès qu'on sort de cette fenêtre imposée.

Écrire une fonction `EulerPhaseBox(f, t0, L, T, p, a, b, c, d)` qui fait ce travail et la tester sur les équations **b** et **d**.

Exercice 4.21 Le pendule simple



Le pendule simple est un système constitué d'une masse considérée ponctuelle liée à un bâti fixe par une barre rigide sans masse de longueur ℓ qui tourne autour de l'axe $(O, \overrightarrow{z})$. On assimile les frottements dans la liaison à des frottements visqueux de coefficient μ tel qu'il existe un couple de frottement $\overrightarrow{C_\mu} = -\mu\dot{\theta}\overrightarrow{z}$ équivalent à une force de frottement $\overrightarrow{F_\mu} = -\dfrac{\mu}{\ell}\dot{\theta}\overrightarrow{u_\theta}$.
L'équation différentielle qui régit le mouvement du pendule est donc la suivante :

$$m\ell^2\ddot{\theta} + \mu\dot{\theta} + mg\ell\sin\theta = 0$$

Si cette équation se résout simplement au voisinage des petits angles ($\sin\theta \underset{\theta\to 0}{\sim} \theta$), elle n'est pas linéaire pour les grands angles.

0. Cas des petits angles

Au voisinage des petits angles l'équation s'écrit :

$$\ddot{\theta} + 2\xi\omega_0\dot{\theta} + \omega_0^2\theta = 0$$

avec $\omega_0 = \sqrt{\dfrac{g}{\ell}}$ et $\xi = \sqrt{\dfrac{\mu^2}{4m^2\ell^3 g}}$.
On admet que la solution de cette équation au voisinage des petits angles pour un amortissement μ faible (régime pseudo-périodique amorti) est :

$$\theta(t) = \theta_0 \mathrm{e}^{-\xi\omega_0 t}\cos(\omega_1 t)$$

avec $\omega_1 = \omega_0\sqrt{1-\xi^2}$, $\xi < 1$.

(0) Tracer sur une même figure la réponse théorique $\theta = f(t)$ pour $\theta_0 = 10°$, $\omega_0 = 10\,\mathrm{rad\,s^{-1}}$ et $\xi = 0.5$ ainsi que la solution numérique obtenue par méthode d'Euler explicite avec un pas de temps de $0.01s$.

(1) Créer une liste `theta_0 = [1, 5, 10, 20]` de valeurs de θ_0. À l'aide d'une boucle `for` tracer sur 4 graphiques différents (un par valeur de θ_0) l'évolution de l'angle $\theta(t)$ pour la solution théorique et la solution numérique.
On constate que la méthode d'Euler explicite ne converge pas pour cet exemple.

1. Cas des grands angles

On revient à l'équation non linéaire sous la forme :

$$\ddot{\theta} + 2\xi\omega_0\dot{\theta} + \omega_0^2 \sin\theta = 0$$

avec $\omega_0 = \sqrt{\dfrac{g}{r}}$ et $\xi = \dfrac{\mu}{2Mr^{\frac{3}{2}}g^{\frac{1}{2}}}$.

On ne connaît pas la solution exacte de cette équation, la résolution numérique nous permet donc d'avoir une approximation de l'évolution du pendule.

(0) Créer une liste `theta_0 = [10, 45, 90, 135, 180]` de valeurs de θ_0.
À l'aide d'une boucle `for` tracer sur un même graphique l'évolution de l'angle $\theta(t)$ pour les différentes valeurs angulaires initiales à l'aide de la méthode d'Euler.
On constate que le résultat de la méthode d'Euler n'est pas bon pour le cas du pendule. Afin d'améliorer la méthode d'Euler, on propose une variante qui consiste à utiliser le schéma suivant :

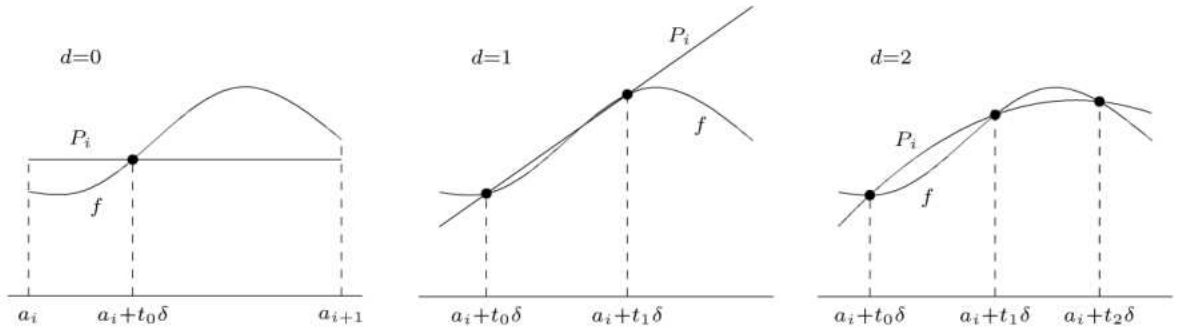
$$\begin{cases} \dot{x}_{i+1} &= \dot{x}_i + \Delta t\, \ddot{x}_i \\ x_{i+1} &= x_i + \Delta t\, \dot{x}_{i+1} \end{cases}$$

(1) Créer et tester sur le pendule la fonction `Euler_2_asym` qui résout une équation différentielle d'ordre 2 par la méthode d'Euler dite *asymétrique*.

Travaux pratiques

TP 4.0 – Points de Gauss (intégration numérique)

Les méthodes usuelles de calcul approché de l'intégrale $I = \int_a^b f(t)\,\mathrm{d}t$ consistent, après avoir fixé un entier $n \geqslant 1$, à subdiviser l'intervalle $[a, b]$ avec un pas constant $\delta = (b-a)/n$, en posant $a_i = a + i\delta$ pour $0 \leqslant i \leqslant n$, puis à approcher l'intégrale sur chaque petit intervalle $[a_i, a_{i+1}]$ en interpolant la fonction f par un polynôme P_i de petit degré d.
Dans ce TP, nous utiliserons 1, 2 ou 3 points d'interpolation, et d sera donc égal à 0, 1 ou 2, comme illustré dans les trois schémas suivants :



Quand $d = 0$, nous retrouvons la méthode des rectangles « pointés à gauche » avec $t_0 = 0$, celle des rectangles « pointés à droite » avec $t_0 = 1$ et la méthode du « point milieu » avec $t_0 = 1/2$. La méthode des trapèzes est obtenue pour $d = 1$, $t_0 = 0$ et $t_0 = 1$. Enfin, la **méthode de Simpson** correspond à $d = 2$, $t_0 = 0$, $t_1 = 1/2$ et $t_2 = 1$.
Pour $d \in \{0, 1, 2\}$ et $(t_i)_{0\leqslant i\leqslant d}$ éléments distincts de $[0, 1]$, nous noterons $I_{t_0,\ldots,t_d}(f, a, b, n)$ l'approximation de $I = \int_a^b f(t)\,\mathrm{d}t$ obtenue avec la méthode précédente. Nous avons par exemple, quand $d = 0$ et $0 \leqslant t_0 \leqslant 1$, $I_{t_0}(f, a, b, n) = \delta \sum_{i=0}^{n-1} f(a + (i + t_0)\delta)$, toujours avec $\delta = \frac{b-a}{n}$.

0. Écrire la fonction `monint0(f, a, b, n, t0)` qui renvoie $I_{t_0}(f, a, b, n)$.

1. Pour chaque $n \in \{100, 500, 1000, 5000\}$, tracer les graphes des applications

$$t_0 \longmapsto I_{t_0}(f_0, 0, 1, n) - I$$

pour $\boxed{f_0 : t \longmapsto \dfrac{1}{1+t}}$ (on calculera la valeur exacte de l'intégrale I).

Quelle valeur de t_0, notée t_0^{opt}, semble être optimale ?
Vérifier cette conjecture avec d'autres intégrales dont la valeur exacte est connue.

2. Estimer l'ordre de grandeur de l'erreur $|I_{t_0}(f, 0, 1, n) - I|$ quand n tend vers l'infini, selon que $t_0 = t_0^{opt}$ ou que $t_0 \neq t_0^{opt}$.

Nous supposons maintenant que $d = 1$. Si t_0 et t_1 sont deux réels distincts de $[0, 1]$, le polynôme P_i est l'unique polynôme de degré au plus 1 qui coïncide avec f en $a_i + t_0\delta$ et en $a_i + t_1\delta$. Nous allons donc approximer $\int_{a_i}^{a_{i+1}} f(t)\,\mathrm{d}t$ par $\int_{a_i}^{a_{i+1}} P_i(t)\,\mathrm{d}t$, et nous admettrons la relation :

$$\int_{a_i}^{a_{i+1}} P_i(t)\,\mathrm{d}t = \frac{2t_1 - 1}{2(t_1 - t_0)} f(a_i + t_0\delta) + \frac{2t_0 - 1}{2(t_0 - t_1)} f(a_i + t_1\delta).$$

Pour chercher les meilleures valeurs de t_0 et de t_1, nous pouvons nous contenter de faire varier t_0 dans $[0, 1/2]$ (par symétrie, les points (t_0, t_1) et $(1-t_0, 1-t_1)$ ont la même efficacité).

3. Écrire la fonction `monint1(f, a, b, n, t0, t1)` qui renvoie $I_{t_0,t_1}(f, a, b, n)$.

4. Pour $t_0 \in \{0, 0.1, 0.2, 0.3, 0.4, 0.5\}$, tracer le graphe de l'application $t_1 \longmapsto I_{t_0,t_1}(f_0, 0, 1, 10^3) - I$. Vous devez remarquer expérimentalement que ces fonctions sont presque affines.

5. Écrire une fonction `t_opt(t0)` qui approxime la valeur de t_1 optimale, c'est-à-dire la valeur $t_1^{opt} \in [0, 1]$ en laquelle la fonction $t_1 \longmapsto \left|I_{t_0,t_1}(f_0, 0, 1, 10^3) - I\right|$ atteint son minimum. Vérifier l'existence d'un réel $u \in]0, 1/2[$ tel que $t_1^{opt} = 1$ dès que $u \leqslant t_0 \leqslant 1/2$.

6. Vérifier que l'application $t_0 \longmapsto (1-2t_0)t_1^{opt}$ est également presque affine sur l'intervalle $[0, u]$ et donner des valeurs numériques α, β telles que $t_1^{opt} \simeq \frac{\alpha+\beta t_0}{1-2t_0}$ pour $t_0 \in [0, u]$. Nous fixerons alors u vérifiant $\frac{\alpha+\beta u}{1-2u} = 1$, puisque u est la valeur à partir de laquelle $t_1^{opt} = 1$. Vérifier que les valeurs u, α, β ne varient pratiquement pas quand on modifie la fonction f et l'entier n (en gardant n assez grand). Conjecturer les valeurs exactes de ces paramètres.

Comme le choix $t_1 = 1$ se ramène, par symétrie, au cas $t_0 = 0$, nous pouvons maintenant restreindre l'étude à $t_0 \in [0, u]$ et $t_1 = \frac{\alpha+\beta t_0}{1-2t_0}$.

7. Tracer le graphe de l'application $t_0 \longmapsto I_{t_0,t_1}(f_0, 0, 1, 10^3) - I$ pour $t_0 \in [0, u]$ et en déduire l'existence d'une valeur optimale de t_0, noté t_0^{opt}. Donner une valeur approchée de t_0^{opt} et du réel t_1^{opt} qui lui est associé, et vérifier que ces valeurs ne changent pratiquement pas quand on change de fonction f_0. En remarquant que $t_1^{opt} \simeq 1 - t_0^{opt}$, en déduire que les valeurs optimales de t_0 et t_1 sont les racines du polynôme $X^2 - X + 1/6$, soit $t_0 = \frac{1}{2} - \frac{\sqrt{3}}{6}$ et $t_1 = \frac{1}{2} + \frac{\sqrt{3}}{6}$.

8. On choisit maintenant pour t_0 et t_1 ces deux valeurs optimales. Calculer expérimentalement l'ordre de grandeur de $|I_{t_0,t_1}(f, 0, 1, n) - I|$ quand n tend vers l'infini. Comparer avec la méthode des trapèzes qui correspond au choix $(t_0, t_1) = (0, 1)$.

Nous supposons pour terminer que $d = 2$ et que t_0, t_1, t_2 sont trois réels distincts de $[0, 1]$. P_i est maintenant le polynôme de degré au plus 2 qui coïncide avec f en les points $a_i + t_0\delta$, $a_i + t_1\delta$ et $a_i + t_2\delta$. Nous admettrons la relation :

$$\int_{a_i}^{a_{i+1}} P_i(t)\,\mathrm{d}t = A_0 f(a_i + t_0\delta) + A_1 f(a_i + t_1\delta) + A_2 f(a_i + t_2\delta)$$

avec $A_0 = \dfrac{6t_1t_2 - 3t_1 - 3t_2 + 2}{6(t_0 - t_1)(t_0 - t_2)}$, $A_1 = \dfrac{6t_0t_2 - 3t_0 - 3t_2 + 2}{6(t_1 - t_0)(t_1 - t_2)}$ et $A_2 = \dfrac{6t_0t_1 - 3t_0 - 3t_1 + 2}{6(t_2 - t_0)(t_2 - t_1)}$.

9. Écrire la fonction `monint2(f, a, b, n, t0, t1, t2)` qui renvoie $I_{t_0,t_1,t_2}(f, a, b, n)$.

10. Par analogie avec ce qui précède, nous admettrons que le triplet optimal (t_0, t_1, t_2) est à chercher sous la forme $(t_0, 1/2, 1 - t_0)$, où $0 \leqslant t_0 < 1/2$. En utilisant une nouvelle fois la fonction f_0, proposer une valeur approchée α de la valeur optimale de t_0.
Calculer le polynôme $(X - \alpha)(X - 1 + \alpha)$ et en déduire qu'il est raisonnable de conjecturer que les valeurs optimales t_0, t_1, t_2 sont les racines du polynôme $(X - 1/2)(X^2 - X + 1/10)$, soit $t_0 = \frac{1}{2} - \frac{1}{10}\sqrt{15}$, $t_1 = \frac{1}{2}$ et $t_2 = \frac{1}{2} - \frac{1}{10}\sqrt{15}$.

11. On choisit maintenant pour t_0, t_1 et t_2 ces trois valeurs optimales. Calculer expérimentalement l'ordre de grandeur de $|I_{t_0,t_1,t_2}(f, 0, 1, n) - I|$ quand n tend vers l'infini.
Comparer avec la méthode de Simpson, qui correspond au choix $(t_0, t_1, t_2) = (0, 1/2, 1)$.

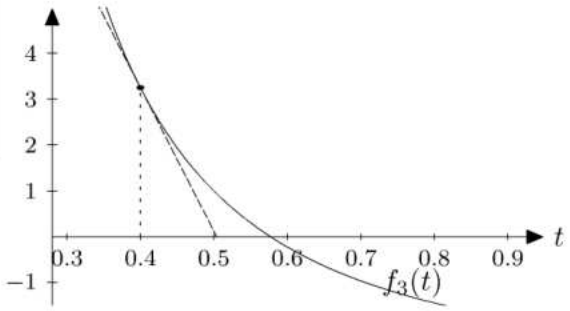
TP 4.1 – Quake III

Le jeu vidéo Quake III simule un environnement en 3 dimensions. Il a souvent besoin de calculer des vecteurs unitaires, c'est-à-dire des expressions de la forme $\frac{\vec{u}}{||\vec{u}||}$. Pour ce faire, il est important de calculer rapidement la fonction $x \mapsto \frac{1}{\sqrt{x}}$ (aussi appelée *racine carrée inverse*). Nous nous intéressons ici à la méthode utilisée dans le jeu Quake III pour calculer cette fonction.

Étape de la méthode de Newton

Soit $f_x(t) = \frac{1}{t^2} - x$. Une première itération de la méthode de Newton consiste, à partir d'une valeur a, à calculer l'abscisse de l'intersection entre la tangente à f_x en a et l'axe des abscisses.

Le dessin ci-contre illustre une étape de la méthode de Newton à partir de $a = 0.4$ pour la fonction f_3. La tangente en a coupe l'axe des abscisses en 0.504.



0. Déterminer le zéro[1] de f_x puis calculer une expression de f'_x.

1. Écrire une fonction `EtapeNewton(x, a)` qui renvoie l'abscisse de l'intersection entre la tangente à f_x en `a` et l'axe des abscisses. La tester pour `a = 0.4` et `x = 3`.

Méthode de Newton

La méthode de Newton consiste à recommencer le calcul à partir de la dernière valeur obtenue un certain nombre de fois. Dans les cas favorables, la convergence est très rapide.

2. Écrire une fonction `Newton(x, a, n)` qui calcule une approximation de $\frac{1}{\sqrt{x}}$ en itérant n fois la méthode de Newton en partant de `a` sur la fonction f_x.

3. Tracer sur l'intervalle $[0.1, 10]$, sur un même graphe la fonction $x \mapsto \frac{1}{\sqrt{x}}$ calculée avec la méthode de Newton et la même fonction calculée avec `**0.5`.

4. Considérons l'approximation $\log_2(1 + x) \approx x$ pour $x \in [0, 1]$. Pour quelles valeurs de x cette approximation est-elle exacte ? Tracer sur un même graphique $\log_2(1 + x)$ (calculé avec la bibliothèque `numpy`) et son approximation.

5. Tracer sur un même graphique $x \mapsto \log_2(1 + x)$ et $x \mapsto x + \sigma$ avec plusieurs valeurs pour la constante σ. Déterminer « au jugé », en observant les courbes précédemment tracées, une constante σ tel que $x + \sigma$ soit une approximation correcte de $\log_2(1 + x)$ pour $x \in [0, 1]$. Aucun calcul n'est demandé dans cette question.

1. C'est-à-dire l'antécédent de 0.

Définition

La norme IEEE 754 - 2008 [IEE] définit les `binary32`, c'est-à-dire les nombres à virgule flottante, en base 2, codés sur 32 bits. Parmi ces nombres, la norme définit les **nombres normaux**.
Un nombre x **normal** est un nombre qui peut être mis sous la forme $x = (-1)^{s_x} \times (1+m_x)2^{e_x}$ avec :

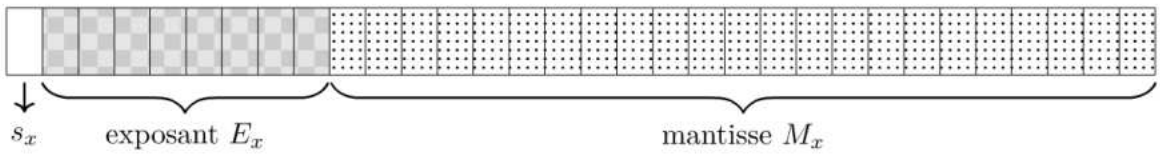
- $s_x \in \{0,1\}$,
- $e_x \in [\![-126;127]\!]$, un entier.
- $m_x \in [0,1[$, un réel pouvant s'écrire avec 23 chiffres (en base 2) après la virgule.

Représentation en machine

Un nombre **normal** est représenté en machine par un triplet d'entiers non-signés :

- $S_x = s_x \in \{0,1\}$,
- $E_x = e_x + B \in [\![1;254]\!]$ avec $B = 127$ (B est appelé biais, et vaut $B = 2^{8-1} - 1 = 127$).
- $M_x = m_x \times L \in [\![0; L-1]\!]$ avec $L = 2^{23}$.

Dans l'ordinateur, le flottant x est représenté comme suit :



Ici, nous ne considèrerons que des flottants strictement positifs, donc tels que s_x est nul.

Définition

L'anglicisme **cast** désigne la conversion d'une valeur d'un type à un autre. Cette conversion peut se faire en essayant de préserver le sens (par exemple en transformant l'entier `2` en `2.0`) ou en préservant la représentation binaire. Ici nous utiliserons le second cas.

Dans la suite, le résultat du cast du flottant x en entier est noté I_x.

Transformer l'entier 32 bits `2` en un flottant 32 bits en préservant la représentation binaire mène à une valeur très éloignée de `2` : on obtient `2.802596928649634e-45`.

Les fonctions `pack` et `unpack` de la bibliothèque `struct` permettent de manipuler les représentations binaires des entiers et des flottants.

6. Écrire une fonction `f2I(x)` qui convertit un flottant 32 bits `x` en un entier de 32 bits en préservant sa représentation binaire.
7. Écrire une fonction `I2f(n)` qui convertit un entier de 32 bits `n` en un flottant 32 bits en préservant sa représentation binaire.
8. Exprimer I_x en fonction de e_x, de m_x, de B et de L.
9. Exprimer $\log_2(x)$ en fonction de e_x et de m_x, puis, simplifier cette expression en utilisant l'approximation $\log_2(1+t) \approx t + \sigma$.
10. En déduire une approximation de $\log_2(x)$ en fonction de I_x, de L, de B et de σ.

11. Écrire une fonction `log2` en utilisant l'approximation précédente. La tracer sur le même graphique que la fonction `log2` de `numpy` sur l'intervalle $]0.1, 10[$.

12. Etant donné x strictement positif et $y = \frac{1}{\sqrt{x}}$, exprimer $\log_2(y)$ en fonction de $\log_2(x)$, puis, en utilisant l'approximation de la question **10**, exprimer I_y en fonction de I_x, de B, de L et de σ.

13. En déduire une fonction `QuickRSqrt(x)` qui, étant donné un flottant `x`, calcule une approximation de $\frac{1}{\sqrt{x}}$. Tracer sur un même graphique les graphes de `QuickRSqrt` et de la fonction racine carrée inverse calculée avec `**0.5`, pour un paramètre décrivant l'intervalle $[0.1, 10]$.

Voici le code en langage C (légèrement épuré) utilisé dans Quake III pour calculer la racine carrée inverse (le code est dans le fichier `code/game/q_math.c` disponible sur GitHub).

```
float Q_rsqrt( float number )
{
        long i;
        float x2, y;
        const float threehalfs = 1.5F;

        x2 = number * 0.5F;
        y  = number;
        i  = * ( long * ) &y;                       // evil floating point bit level hacking
        i  = 0x5f3759df - ( i >> 1 );               // ***censuré***
        y  = * ( float * ) &i;
        y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
//        y  = y * ( threehalfs - ( x2 * y * y ) );  // 2nd iteration, this can be removed
        return y;
}
```

14. Combien vaut `0x5f3759df` ? Donner la valeur en base 10.

En C, lorsqu'une nouvelle variable est créée, son type est précisé (`long` pour entier, `float` pour flottant, `const float` pour un flottant constant). Les constantes flottantes finissent par un `F`, l'opération `i>>1` divise `i` par 2, l'opération `* ( long * ) &` convertit en entier en conservant la représentation binaire, l'opération `* ( float * ) &` convertit en flottant.

15. Traduire cette fonction en Python.

16. Tracer sur un même graphe cette fonction racine carrée inverse et une fonction racine carrée inverse utilisant `**0.5`.

17. Quelle ligne de la fonction de Quake III correspond à la méthode de Newton ?

18. Pourquoi la seconde itération a-t-elle été mise en commentaire ?

19. Combien vaut σ dans la fonction de Quake III ?

TP 4.2 – Modèles compartimentaux en épidémiologie

Les modèles épidémiologiques sont des modèles mathématiques de la propagation de maladies infectieuses. Outre l'étude de l'évolution de maladies, ils permettent de prévoir les conséquences pour la population d'actions publiques telles que la vaccination [**KZVH00**, **LM02**], la mise en quarantaine [**GRD+04**, **WD16**] ou des mesures de dépistages [**HLS03**].
Les modèles compartimentaux sont des modèles déterministes où la population est divisée en plusieurs catégories selon leurs caractéristiques et leur état par rapport à la maladie.
Nous distinguerons notamment les compartiments suivants dans ce TP :

- le compartiment S (*Susceptible*) des individus sains et susceptibles d'être infectés ;

- le compartiment I (*Infectious*) des individus infectés et contagieux.

Nous nous intéresserons à une population de taille constante, ce qui revient à négliger les naissances et les morts (par d'autres causes que la maladie infectieuse étudiée) et est une hypothèse raisonnable pour une étude sur un intervalle de temps court. Les différents compartiments sont exprimés en proportions de population, ainsi, dans un modèle à deux compartiments S et I on a à tout instant $S(t) + I(t) = 1$.
Les individus passent, lors de la simulation, d'un état à l'autre à l'aide de règles de transitions qui sont décrites par des équations différentielles. L'objectif de ce TP est d'une part de présenter quelques modèles compartimentaux, et d'autre part d'étudier les effets d'actions publiques dans des cas simples.
Dans ce TP nous importons la bibliothèque `numpy` avec l'instruction `import numpy as np`.

0. Écrire une fonction `Euler(f, X0, tf, n)` qui applique la méthode d'Euler au problème de Cauchy $\left(X' = f(X,t),\ X(0) = X_0\right)$ sur $[0, tf]$ avec le pas $\mathrm{d}t = tf/n$, avec X un vecteur de $\mathbb{R}^k$, où k est le nombre de compartiments étudiés, codé sous la forme de `ndarray`. Cette fonction renverra un tableau de temps `T` et un tableau `Y` à deux dimensions tels que `Y[k]` contient la valeurs de $X(t)$ pour t valant `T[k]`.
Par exemple, `Euler(lambda x, t : x*t, np.array([1]), 0.1, 3)` doit renvoyer :

```
(array([0.,  0.03333333,  0.06666667,  0.1]),
 array([[1.], [1.], [1.00111111], [1.0033358]]))
```

Certaines infections, comme le rhume, ne permettent pas de développer une immunité à long terme. Seuls deux états interviennent dans le modèle : les compartiments S et I. Le modèle est alors appelé SIS (les individus sains sont susceptibles de devenir infectés et redeviennent sains après leur maladie), et est décrit par les équations différentielles ci-contre. Les variables β et γ sont globales.

$$\begin{cases} S' = -\beta SI + \gamma I \\ I' = \beta SI - \gamma I \end{cases}$$

Maladie X

Nous considérons pour tester nos premiers modèles une maladie fictive, la maladie X. Elle a pour paramètres $\gamma = 0.01$ jour^{-1} et $\beta = 0.03$ jour^{-1}. On étudiera les épidémies de X sur une période $tf = 500$ jours avec n assez grand (par exemple 1000).

1. Quelle est la signification biologique des paramètres β et γ ?

2. Écrire une fonction `SIS(X, t)` qui prend en argument `X` un tableau `numpy` (`ndarray`) représentant $[S(t), I(t)]$ et qui renvoie sous forme d'un `ndarray` $[S'(t), I'(t)]$ pour le système différentiel du modèle SIS.
L'expression `SIS(np.array([0, 1]), 0)` doit renvoyer `array([0.01, -0.01])`. On remarque qu'ici, l'argument `t` n'a pas d'influence sur le résultat renvoyé par la fonction `SIS`.

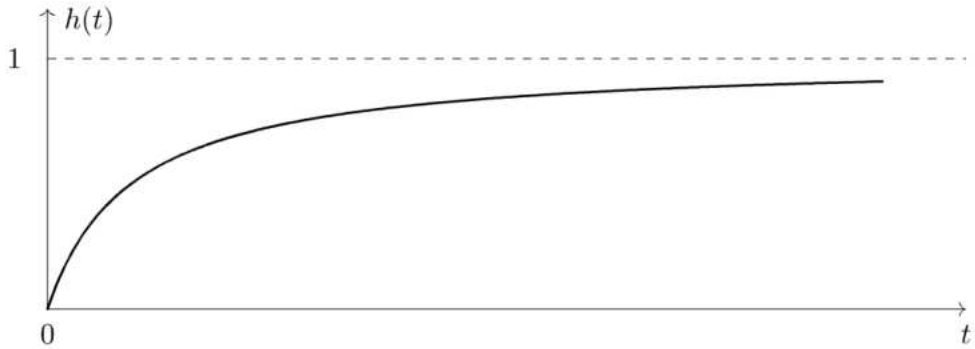
3. Tracer sur le même graphe $S(t)$ et $I(t)$ avec les conditions initiales suivantes :

- S(0)=0 et I(0)=1 ;
- S(0)=0.2 et I(0)=0.8 ;
- S(0)=0.5 et I(0)=0.5 ;
- S(0)=0.99 et I(0)=0.01.

Lorsque plusieurs courbes sont présentes sur un même graphique, il peut être intéressant d'utiliser la fonction `legend` de la bibliothèque `matplotlib.pyplot` pour les identifier. Dans le cas présent, on pourra utiliser `legend(['S', 'I'])` ou plus simplement `legend("SI")`.

4. Que constate-t-on lorsque t devient grand ?

Pour certaines maladies, comme la gastro-entérite, des mesures simples d'hygiène permettent de réduire le taux de transmission de la maladie. Nous supposons maintenant qu'une campagne de sensibilisation contre la maladie X est lancée à $t = 0$. La proportion $h(t)$ de la population sensibilisée aux mesures d'hygiène vaut initialement 0 (personne n'est au courant des mesures à prendre) et tend vers 1 à l'infini (toute la population a été informée). Nous modélisons h par une fonction de la forme $h(t) = at/(at + \tau)$ avec ici $a = 0.02$ et $\tau = 1$.



Nous supposerons ici que les personnes sensibilisées ont deux fois moins de chances de transmettre la maladie X, le terme βSI devient alors $\left(\beta(1-h(t))I + \frac{\beta}{2}h(t)I\right)S = \left(1 - \frac{h(t)}{2}\right)\beta SI$, ce qui donne le modèle SISh ci-contre.

$$\begin{cases} S' = -\left(1 - \frac{h(t)}{2}\right)\beta SI + \gamma I \\ I' = \left(1 - \frac{h(t)}{2}\right)\beta SI - \gamma I \end{cases}$$

5. Écrire une fonction `SISh(X, t)` qui prend en argument `X` un tableau `ndarray` représentant $[S(t), I(t)]$ et qui renvoie $[S'(t), I'(t)]$ sous forme d'un `ndarray` pour le système différentiel du modèle SISh.

6. Tracer, sur trois graphiques différents, avec $S(0) = 0.6$ et $I(0) = 0.4$, les courbes de $S(t)$ et de $I(t)$ pour : le modèle SISh, le modèle SIS et le modèle SIS avec β divisé par deux. Comparer.

D'autres infections permettent de développer une immunité ou tuent certains patients. Les individus guéris (et donc immunisés) ou décédés sont ainsi regroupés dans un nouveau compartiment R (pour *recovered* ou *removed*). Le modèle est appelé SIR. Les équations différentielles régissant la population sont indiquées ci-contre.

$$\begin{cases} S' = -\beta SI \\ I' = \beta SI - \gamma I \\ R' = \gamma I \end{cases}$$

7. Écrire une fonction `SIR(X, t)` qui prend en argument `X` un tableau `ndarray` représentant $[S(t), I(t), R(t)]$ et qui renvoie $[S'(t), I'(t), R'(t)]$ sous la forme d'un tableau `ndarray` pour le système différentiel du modèle SIR.

Notre modèle est bien adapté à l'épidémie de peste d'Eyam (1665-1666) : le village était isolé (la population totale, morts compris, était constante). Étant donné qu'un nombre négligeable d'individus survécurent à la peste, nous utilisons le compartiment R pour représenter uniquement les morts. En outre, nous négligeons les naissances et les décès liés à d'autres causes que la maladie.

Les valeurs numériques de la peste de ce TP proviennent toutes de l'article [**Rag82**] (qui traite de la seconde moitié de l'épidémie).

Peste d'Eyam

La peste est décrite par les paramètres [b] suivants : $\gamma = 2.78$ mois^{-1}, $\beta = \gamma \times \frac{261}{159} \approx 4.56$ mois^{-1}.
L'unité de temps est le mois de 31 jours.
Les conditions initiales d'Eyam sont : $S(0) = \frac{254}{261}$, $I(0) = \frac{7}{261}$ et $R(0) = 0$.

b. Notre γ correspond au b de l'article, et notre β à $a \times N$ dans l'article.

8. Réaliser les tracés des courbes représentatives de $S(t)$, $I(t)$ et $R(t)$ sur 5 mois (i.e., $tf = 5$).

Le tableau suivant donne, à différentes dates, le nombre d'indivus susceptibles [2] d'être infectés, le nombre d'infectés, reconstitués à partir de données historiques [3], et, pour des raisons de commodités, le temps t du modèle.

Date	Susceptibles	Infectés	Temps dans le modèle (t)
18 juin	254	7	0
3/4 juillet	235	14.5	0.5
19 juillet	201	22	1
3/4 août	153.5	29	1.5
19 août	121	20	2
3/4 septembre	108	8	2.5
19 septembre	97	8	3
4/5 octobre	Inconnu	Inconnu	3.5
20 octobre	83	0	4

9. Ajouter sur le graphique précédent les points correspondants au tableau, et comparer le modèle aux données historiques. Pour se simplifier la vie, on peut utiliser `float("nan")` pour représenter les valeurs inconnues.

Dans les villages avoisinants et même à Londres, la peste a, en moyenne, tué une proportion moins grande d'habitants. On subodore que la maladie se propage plus ou moins facilement selon les lieux, l'hygiène, etc. Autrement dit, d'un village à l'autre, le paramètre β peut changer.

10. Déterminer la valeur de β à 10^{-2} près correspondant à un taux de mortalité de 50%, tous les autres paramètres restant constants.

La modélisation de la peste d'Eyam est encore un sujet de recherche. Des travaux récents [**WD16**] proposent une modélisation plus précise (sur toute la durée de l'épidémie) et des données historiques plus fiables.

2. Il suffit de diviser par 261 (la taille de la population étudiée) pour obtenir $S(t)$.
3. Les .5 proviennent de calculs de moyennes car les données historiques ne donnent pas toujours le nombre d'individus dans chaque compartiment aux dates désirées

Les questions 11 à 16 sont inspirées du sujet de Mines-Ponts 2016. Ces questions sont intéressantes à programmer, mais utilisent une méthode non-standard (un système à retard) au lieu de simplement ajouter un compartiment E (cf. l'exemple du SRAS).

De nombreuses maladies possèdent une phase d'incubation pendant laquelle l'individu est porteur de la maladie mais ne possède pas de symptômes et n'est pas contagieux. On peut prendre en compte cette phase d'incubation à l'aide du système à retard ci-contre, où τ est le temps d'incubation. On suppose alors que S, I et R sont constants sur $[-\tau, 0]$. On suppose que τ est un multiple entier de $\mathrm{d}t$, et donc qu'il existe un entier p tel que $\tau = p \times \mathrm{d}t$; ainsi, p est le nombre de pas de retard.

$$\begin{cases} S'(t) = -\beta S(t) I(t-\tau) \\ I'(t) = \beta S(t) I(t-\tau) - \gamma I(t) \\ R'(t) = \gamma I(t) \end{cases}$$

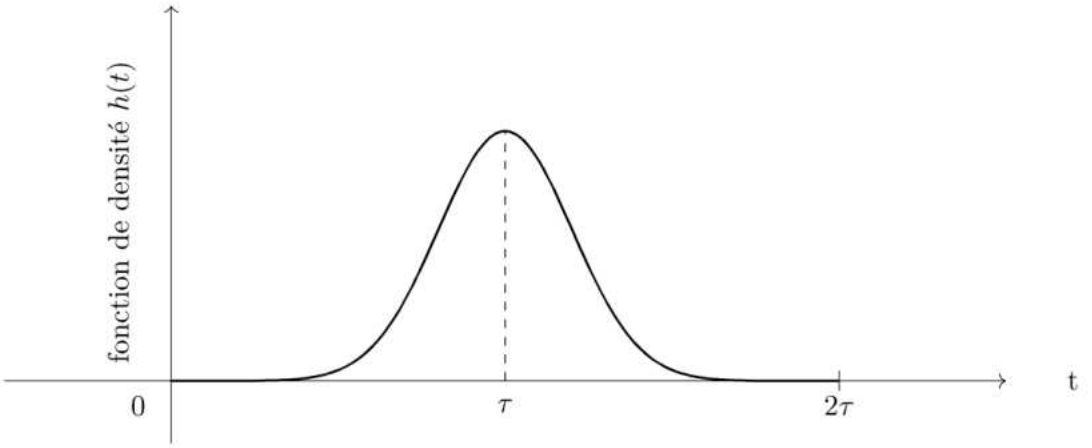
11. Écrire une fonction `SIRRetard(X, XRetard, t)` qui prend en argument `X` et `XRetard` deux tableaux `numpy` (`ndarray`) correspondant respectivement à $[S(t), I(t), R(t)]$ et à $[S(t-\tau), I(t-\tau), R(t-\tau)]$ et qui renvoie un `ndarray` correspondant à $[S'(t), I'(t), R'(t)]$ pour le système différentiel correspondant au modèle SIR avec retard.

12. Écrire une fonction `EulerRetard(f, X0, p, tf=500, n=1000)`, sur le modèle de la fonction `Euler` précédemment définie, qui adapte la méthode d'Euler pour résoudre le problème $\big(X' = f(X(t), X(t-\tau), t),\ \forall t \in [-\tau, 0], X(t) = X_0\big)$ sur $[0, tf]$.

13. Tracer les courbes correspondant à ce modèle et à la peste d'Eyam avec $\tau = 0.18$ mois (soit environ les 5.6 jours mentionnés dans [**WD16**]).

14. La proportion de morts change-t-elle en tenant compte de la phase d'incubation ? Tracer la courbe du taux de mortalité en fonction du temps d'incubation. On fera varier le temps d'incubation entre 0 et 5 mois (les autres paramètres étant constants), et on prendra tf suffisamment grand pour être sûr que l'épidémie est terminée.

On constate par ailleurs que le temps d'incubation n'est pas nécessairement le même pour tous les individus. On peut modéliser cette diversité à l'aide d'une fonction positive d'intégrale unitaire (dite de densité) $h : [0, 2\tau] \longmapsto \mathbb{R}_+$ telle que représentée ci-dessous.



On obtient alors le système intégro-différentiel suivant.

$$\begin{cases} S'(t) = -\beta S(t) \displaystyle\int_0^{2\tau} I(t-s)h(s)ds \\ I'(t) = \beta S(t) \displaystyle\int_0^{2\tau} I(t-s)h(s)ds - \gamma I(t) \\ R'(t) = \gamma I(t) \end{cases}$$

On suppose alors que S, I et R sont constants sur $[-2\tau, 0]$. Pour j un entier compris entre 0 et n on pose $t_j = j \times dt$. Pour un pas dt de temps donné, on peut calculer numériquement l'intégrale à l'instant t_i $(0 \leqslant t_i \leqslant n)$ à l'aide de la méthode des rectangles à gauche en utilisant l'approximation :

$$\int_0^{2\tau} I(t_i - s)h(s)ds \approx dt \times \sum_{j=0}^{2p-1} I(t_i - t_j)h(t_j)$$

15. On considère la fonction de densité donnée, sur $[0, 2\tau]$, par la formule suivante :

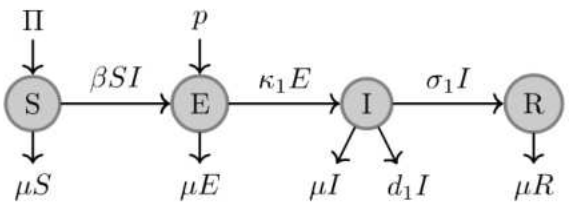
$$h(t) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(t-\tau)^2}{2\sigma^2}\right),$$

On pose $h(t) = 0$ en dehors de cet intervalle (l'intégrale de h vaut presque 1). Programmer cette fonction en prenant $\tau = 0.18$ mois et $\sigma = 0.04$ mois.

16. Écrire une fonction `SimuleIntegroDiff(h, X0, tf, p, n)` où `X0` est un `ndarray` correspondant aux conditions initiales, `tf` est le temps final, `n` le nombre de pas d'intégration et `p` est le nombre de pas de retard.

17. Tester cette fonction avec la peste d'Eyam.

Nous allons maintenant étudier les effets d'une politique publique : la quarantaine, dans le cadre de l'épidémie de SRAS de 2003. Nos données proviennent de l'article [**GRD+04**]. L'épidémie a eu lieu dans une population non-isolée, pour traiter plus facilement ce cas, nous décidons que les compartiments seront exprimés en nombre d'individus et non plus en proportion de la population totale. La population totale sera notée N. Cette population s'accroît de Π habitants[1] sains par jour, et de p habitants exposés[2] au SRAS par jour. De plus, la population est soumise à un taux de mortalité naturelle μ ; tous les compartiments perdrons la même proportion μ d'habitants.
Le SRAS a une période d'incubation. Pour le modéliser, nous utilisons non pas un système à retard comme dans le sujet Mines-Ponts, mais un compartiment E (*exposed* ou *asymptomatic*) représentant les individus en train d'incuber la maladie. Le schéma suivant résume les transferts entre compartiments (d_1 est le taux de mortalité du SRAS, R représente les immunisés et pas les morts).



Des mesures de quarantaine ont été prises par les gouvernements concernés. Pour le modéliser, nous introduisons deux nouveaux compartiments : le compartiment Q (*quarantined*) des individus

1. Accroissement par les flux de population (migrations) et par la natalité.
2. Ils proviennent d'une autre zone géographique touchée par le SRAS.

en phase d'incubation qui ont été mis en quarantaine et le compartiment J (*isolated*) des malades en quarantaine.
Notre modèle se résume par le système différentiel suivant :

$$\begin{cases} S' = \Pi - \beta S \times \frac{I+\varepsilon J}{N} - \mu S \\ E' = p + \beta S \times \frac{I+\varepsilon J}{N} - (\gamma_1 + \kappa_1 + \mu)E \\ Q' = \gamma_1 E - (\kappa_2 + \mu)Q \\ I' = \kappa_1 E - (\gamma_2 + d_1 + \sigma_1 + \mu)I \\ J' = \gamma_2 I + \kappa_2 Q - (\sigma_2 + d_2 + \mu)J \\ R' = \sigma_1 I + \sigma_2 J - \mu R \end{cases}$$

Les malades en quarantaine transmettent moins facilement la maladie (leur taux de transmission est de $\varepsilon\beta$ au lieu de β pour les malades non soignés), ont un taux de mortalité (d_2) plus bas grâce aux soins médicaux dont ils bénéficient, et, pour la même raison ont un taux de guérison σ_2 plus élevé que les autres malades.
Au modèle, nous ajoutons une catégorie D destinée à compter les morts, et vérifiant $D' = d_1 I + d_2 J$.
De plus, on pose $N = S + E + Q + I + J + R$.

SRAS de Toronto

À GTA (*Greater Toronto Area*), les paramètres du SRAS furent les suivants : $\beta = 0.2\ \text{j}^{-1}$; $\mu = 3.4 \times 10^{-5}\ \text{j}^{-1}$; $\kappa_1 = 0.1\ \text{j}^{-1}$; $\kappa_2 = 0.125\ \text{j}^{-1}$; $\sigma_1 = 0.0337\ \text{j}^{-1}$; $\sigma_2 = 0.0386\ \text{j}^{-1}$; $d_1 = 0.0079\ \text{j}^{-1}$; $d_2 = 0.0068\ \text{j}^{-1}$; $p = 0.06$ individus/j ; $\Pi = 136$ individus/j.
Initialement, seuls 3 compartiments sont non-vides :
$S_0 = 4$ millions d'individus ; $E_0 = 6$ individus ; $I_0 = 1$ individu.

Le paramètre ε dépend des mesures d'hygiène prises pour éviter la contamination dans les hôpitaux (masques, chambres à pression). Nous approximons ε par une fonction qui vaut 0.36 jusqu'au 56e jour (le 20 avril) puis qui vaut ensuite zéro. De même, γ_1 et γ_2 dépendent de la politique de quarantaine. Au début, il n'y a pas de mesure de quarantaine (ils valent zéro), puis, nous considérons que le 35e jour (le 30 mars) ils passent respectivement à $0.1\ \text{j}^{-1}$ et $0.5\ \text{j}^{-1}$.

18. Programmer trois fonctions `epsilon(t)`, `gamma1(t)` et `gamma2(t)` renvoyant les valeurs respectives de ε, γ_1 et γ_2 en fonction du temps.

19. Programmer une fonction `SRAS` pour ce modèle fonctionnant sur le même principe que les fonctions `SIS` et `SIR`.

20. Résoudre avec Euler ce système, puis tracer sur un même graphique la courbe des morts théorique et les points représentants les morts réelles données dans le tableau ci-après.

Date	17 mars	24 mars	31 mars	4 avril	14 avril	29 avril	26 mai	9 juin
$t =$	22	29	36	40	50	65	92	106
Morts	2	3	4	7	13	20	26	32

21. Selon ce modèle, combien y aurait-il eu de morts le 9 juin sans la quarantaine ?

22. Tracer la courbe des morts au 9 juin en fonction de γ_2.

TP 4.3 – Modélisation d'un tas de sable

Nous étudions dans ce TP l'évolution mécanique d'un tas de sable. Le tas est constitué de grains, modélisés par des sphères de même rayon R.
Un grain est représenté par une liste de six éléments `[xi, yi, vxi, vyi, Fxi, Fyi]` où :

- `xi` et `yi` désignent la position du centre du grain,
- `vxi` et `vyi` désignent les composantes de la vitesse du grain,
- `Fxi` et `Fyi` désignent les composantes de la force d'interaction.

Le tas de sable est représenté par une liste `tas`, qui est une liste de grains.
Initialement, les vitesses de chaque grain et les forces d'interactions sont nulles. Les grains ont pour rayon $R = 1$ et sont répartis sur n niveaux. Le i-ème niveau comporte $n - i$ grains. Avec $i = 0 \dots n-1$ et $j = 0 \dots n-i-1$, la position de chaque grain est $(R(i+2j), R(1+\sqrt{3}i))$.

0. Écrire une fonction `initialise(n)` qui renvoie le tas initial, celui-ci étant une liste de grains.

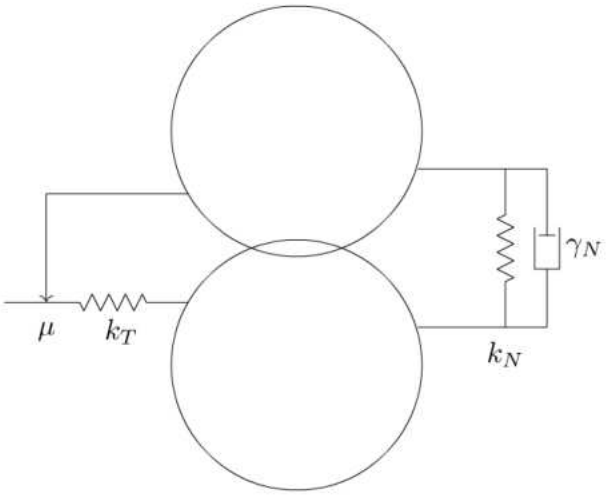
1. Écrire une suite d'instructions qui permet d'effectuer la représentation graphique du tas de grains ainsi constitué. On travaillera avec $n = 5$.

Pour modéliser les interactions entre les particules, nous allons utiliser la méthode des solides déformables « sphères molles » : le calcul des forces et des moments sera réalisé en considérant que les disques sont indéformables mais peuvent s'interpénétrer légèrement avec δ_N et δ_T le déplacement normal et tangentiel à partir du contact.
Ainsi, pour modéliser les forces normales, on utilise un modèle de ressort de raideur k_N associé à une dissipation visqueuse de coefficient γ_N permettant de reproduire une collision inélastique. Pour les forces tangentielles, un ressort de raideur k_T couplé à un patin de limite du glissement μ permet de modéliser la force de friction.
La table 0 recense différents types de sables testés par des laboratoires.

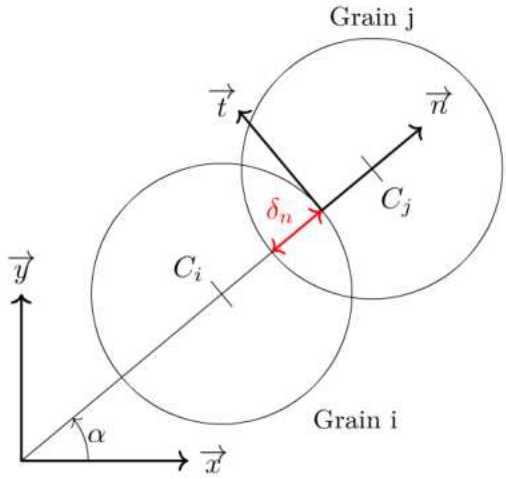
Labo	Geom	Mat	Densité $(g.cm^{-3})$	R (mm)	k_N $(N.m^{-1})$	γ_n $(N.s.m^{-1})$	k_T $(N.m^{-1})$	μ
MSC	poly	verre	$2,1$	–	3.10^7	20	2.10^6	$0,5$
LPMDI	poly	acier	6	–	5.10^6	1	4.10^5	$0,7$
LPGP	sphère	métal	$4,7$	4	10^6	10	3.10^4	$0,2$
LMGC	sphère	verre	$1,56$	1	2.10^7	5	10^7	$0,3$
PMMH	quartz	verre	$1,46$	2	7.10^8	$0,5$	2.10^8	$0,4$

TABLE 0. Base de données partielle des types de grains de sable et leurs paramètres mécaniques

Le sable que l'on considérera par la suite est celui testé par le LMGC, on relèvera les valeurs utiles dans le tableau 0.



Modélisation



Paramétrage

2. À partir du paramétrage de la figure, écrire une fonction `Trigo(Xi, Xj, Yi, Yj)` qui renvoie les valeurs de $\cos(\alpha)$ et $\sin(\alpha)$) avec $\alpha = (\vec{x}, \vec{n})$ en fonction des coordonnés `(Xi, Yi)` et `(Xj, Yj)` pour deux grains i et j.

Nous nous intéressons à la résolution du Principe Fondamental de la Dynamique en résultante selon $\vec{x}$ et $\vec{y}$ pour chaque grain i :

$$m_i \frac{\mathrm{d}^2 x_i}{\mathrm{d}t^2} = F_{ix} \quad \text{et} \quad m_i \frac{\mathrm{d}^2 y_i}{\mathrm{d}t^2} = F_{iy} - m_i g$$

Le modèle proposé donne les relations suivantes entre les efforts et les déplacements :

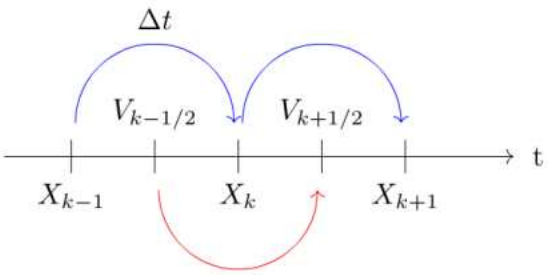
$$F_{jin} = k_N \delta_{jiN} + \gamma_N \frac{d\delta_{jiN}}{dt} \quad \text{et} \quad \begin{cases} F_{jit} = k_T \delta_{jiT} & \text{si } |k_T \delta_{jiT}| \leqslant |\mu F_{jiN}| \\ F_{jit} = \mu F_{jin} \mathrm{sgn}\left(\dfrac{d\delta_{jiT}}{dt}\right) & \text{sinon} \end{cases}$$

Avec $\delta_{jiN} = \overrightarrow{C_i C_j} \cdot \vec{n} - 2R$ et $\delta_{jiT} = \overrightarrow{C_i C_j} \cdot \vec{t}$ à l'instant du contact.

3. Créer la fonction `calcul_Fnt(i, j)` qui renvoie `(Fjin, Fjit)` correspondant respectivement aux composantes algébriques normales (direction $\vec{n}$) et tangentielles (direction $\vec{t}$) de la force exercée par la particule j sur la particule i.
Si les deux particules ne sont pas en contact alors la fonction renvoie (0, 0).

4. En déduire une fonction `somme_int()` effectuant la somme des interactions `Fix` et `Fiy` (selon $\vec{x}$ et $\vec{y}$) sur chaque grain i, exercées par tous les autres grains j. On utilisera la fonction `calcul_Fnt(i, j)`

Le schéma d'intégration utilisé dans la suite sera celui de l'algorithme de Verlet « saute-mouton ».
On calcule les positions des grains aux temps t = 0 ; Δt ; $2\Delta t$ ... où Δt est le pas de temps.
Les vitesses sont calculées et mémorisées pour des temps intermédiaires, t = $\Delta t/2$; $3\Delta t/2$...
On utilise les vitesses intermédiaires pour déterminer les positions aux instants $k\Delta t$.



On note :

- x_k la position du grain au temps $t = k\Delta t$;
- $v_{k+1/2}$ la vitesse du grain au temps $t = \left(k + \frac{1}{2}\right)\Delta t$;
- a_k l'accélération du grain au temps $t = k\Delta t$.

5. Donner une formule de calcul de $v_{k+1/2}$ en fonction de $v_{k-1/2}$, de a_k et de Δt ainsi qu'une formule de calcul de x_{k+1} en fonction de x_k, de $v_{k+1/2}$ et de Δt.

6. En utilisant le schéma d'intégration ainsi défini, écrire une fonction `Verlet(T, N)` où `T` est le temps final de la simulation, `N` est le nombre de pas de simulations, qui renvoie la liste des positions de chaque grain (codée par une liste de tuples de longueur 2) à chaque pas de temps. On utilisera `somme_int()` définie précédemment. On prendra aussi en compte la pesanteur telle que définie dans l'équation de la dynamique avec une même masse $m_i = M$ pour tous les grains.

7. Quelle est l'influence du pas de temps sur la qualité de l'approximation et sur le temps de calcul ?

Un problème d'oscillation apparaît dans le modèle précédent. Pour le résoudre, on propose d'introduire un amortissement supplémentaire e, utilisant un paramètre $f > 0$ supplémentaire.
L'équation de la dynamique devient :

$$m_i \frac{\mathrm{d}^2 x_i}{\mathrm{d}t^2} = F_{ix} - \frac{1}{f}\frac{\mathrm{d}x_i}{\mathrm{d}t} \quad \text{et} \quad m_i \frac{\mathrm{d}^2 y_i}{\mathrm{d}t^2} = F_{iy} - m_i g - \frac{1}{f}\frac{\mathrm{d}y_i}{\mathrm{d}t}$$

8. Pourquoi cette équation est-elle problématique par rapport au schéma d'intégration mis en place ?

9. En approchant v_k par $\dfrac{v_{k-1/2} + v_{k+1/2}}{2}$, montrer que l'on a :

$$\left(\frac{1 + \Delta t}{2fm}\right) v_{k+1/2} = \left(\frac{1 - \Delta t}{2fm}\right) v_{k-1/2} + \frac{F_k}{m}\Delta t$$

10. En déduire une relation de récurrence permettant de déterminer les positions successives des grains puis écrire une fonction `VerletAmortissement(T, N)` où `T` est le temps final de la simulation, `N` est le nombre de pas de simulations, qui renvoie la liste des positions de chaque grain (codée par une liste de tuples de longueur 2) à chaque pas de temps.

Corrections des exercices

Corrigé exo 4.0

0. Immédiat.

```
def S(n, p):
    return np.random.binomial(n, p) / n
```

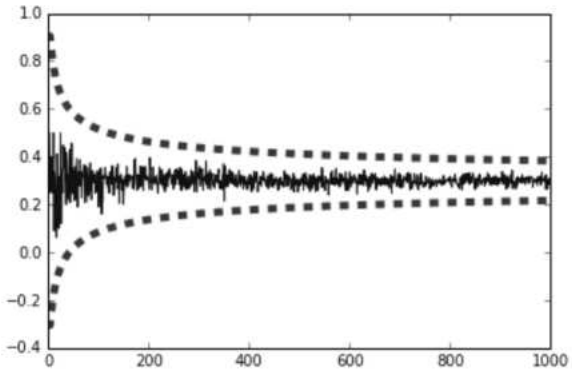
1. On affiche ce qui est demandé.

```
def affiche(n, p):
    X, Y, Lm, LM = [], [], [], []
    X = np.array(range(2, n + 1))
    Y = np.array([S(k, p) for k in range(2, n + 1)])
    Lm = np.array([p - np.sqrt(np.log(k) / k) for k in range(2, n + 1)])
    LM = np.array([p + np.sqrt(np.log(k) / k) for k in range(2, n + 1)])

    plt.plot(X, Y, 'b')
    plt.plot(X, Lm, 'r--', lw=5)
    plt.plot(X, LM, 'r--', lw=5)
    plt.show()
```

Le test

```
affiche(1000, .3)
```

donne



On peut constater que S_n reste compris dans $\left[p - \sqrt{\frac{\ln k}{k}}, p + \sqrt{\frac{\ln k}{k}}\right]$ (en tout cas avec une forte probabilité).

Corrigé exo 4.1

0. On peut choisir les points du plan complexe ayant pour affixe les racines énièmes de l'unité : $z_k = \mathrm{e}^{2i\pi k/n}$. En projetant on trouve les coordonnées (x_k, y_k) de A_k pour $k \in \{0, 1, \ldots, n-1\}$:

$$x_k = \cos\left(2\pi\frac{k}{n}\right) \qquad y_k = \sin\left(2\pi\frac{k}{n}\right)$$

1. On trace le polygone comme une ligne brisée avec `plot`. On prend garde à rajouter le point A_0 à la fin des listes pour que le polygone tracé soit fermé.

```
def polygone_regulier(n):
    L1 = [np.cos(2 * k * np.pi / n) for k in range(n)] + [1]
    L2 = [np.sin(2 * k * np.pi / n) for k in range(n)] + [0}
    plt.plot(L1, L2)
```

2. Lorsque n devient grand, on obtient visuellement un cercle. Ce n'est pas étonnant dans la mesure où pour tracer le cercle trigonométrique, défini par la courbe paramétrée :

$$\begin{cases} x(t) = \cos(2\pi t) \\ \\ y(t) = \sin(2\pi t) \end{cases}$$

on discrétise l'intervalle $[0, 1]$ en $n + 1$ points (avec n grand), ce qui revient à effectuer le tracé précédent.

Corrigé exo 4.2

0. La fonction cosinus hyperbolique est appelée `cosh` dans `numpy`.

2. La seule difficulté est que qu'on n'a pas le droit d'écrire `f(np.linspace(-200, 3))` à cause du `if`. On peut alors utiliser `vectorize` ou une liste par compréhension.

Corrigé exo 4.3

0.

```
def dicho(a, b, f, p):
    while b - a > p:
        m = (a + b) / 2
        if f(m) > 0:
            b = m
        else:
            a = m
    return m
```

1. On entre la commande suivante dans la console `dicho(0,10, lambda x: x**2-2, 0.001)`

2. La fonction calcule une approximation d'un des zéros de `f`.

3. On modifie la condition du `if` à la ligne `4` en `f(b)*f(m) > 0`

Corrigé exo 4.4

0. On procède par dichotomie sur l'intervalle $[X/n, 1]$. Si X est plus grand que la borne inférieure de $I_{\alpha,m,n}$ alors $X \in I_{\alpha,p,n}$ et donc m est plus petit que le plus grand p ayant X dans son intervalle de fluctuation.

```
import scipy.stats as st


def PlusGrandP(alpha, X, n):
    a, b = X / n, 1
    while b - a >= 10**-6:
        m = (a + b) / 2
        if X >= st.binom.interval(1 - alpha, n, m)[0]:
            a = m
        else:
```

```
            b = m
    return m
```

1. On procède de manière similaire pour la borne inférieure,

```
def PlusPetitP(alpha, X, n):
    a, b = 0, X / n
    while b - a >= 10**-6:
        m = (a + b) / 2
        if X >= st.binom.interval(1 - alpha, n, m)[1]:
            a = m
        else:
            b = m
    return m
```

ce qui permet d'écrire la fonction demandée.

```
def confiance(alpha, X, n):
    g = PlusPetitP(alpha, X, n)
    d = PlusGrandP(alpha, X, n)
    return g, d
```

2. Le seul cas où p est dans l'intervalle de confiance est le sondage IPSOS pour $\alpha = 1\%$.

3. On calcule les bornes supérieures et inférieures avec des listes par compréhension.

```
import numpy as np
import matplotlib.pyplot as plt

a = np.linspace(0.01, 0.5)
S = [PlusGrandP(t, 140, 1000) for t in a]
I = [PlusPetitP(t, 140, 1000) for t in a]
plt.plot(a, S)
plt.plot(a, I)
plt.show()
```

4. On cherche par dichotomie, le plus grand α tel que l'évènement $X \in I_{\alpha,p,n}$. Pour le sondage IPSOS, on trouve 1.4%.

Corrigé exo 4.5

0. On applique la méthode des rectangles à gauche. Il n'y a pas lieu de distinguer les cas $x > 1$ et $x < 1$.

```
def ln(x):
    dt = (x - 1) / 1000
    S = 0
    for k in range(1000):
        S += dt / (1 + k * dt)
    return S
```

1. On trace les deux courbes. Elles se confondent presque.

```
import numpy as np
import matplotlib.pyplot as plt
T = np.linspace(0.5, 20, 200)
plt.plot(T, ln(T))
plt.plot(T, np.log(T))
plt.show()
```

Corrigé exo 4.6

0. On commence par programmer la fonction m à intégrer.

```
def m(x, t):
    return t**(x - 1) * np.exp(-t)


def phi(alpha, beta, x, n):
    S = 0
    pas = (beta - alpha) / n
    for k in range(n):
        S += m(x, alpha + pas * k)
    return S * pas
```

1. On définit Γ puis on la trace.

```
def Gamma(x):
    return phi(0.02, 300, x, 3000)

import numpy as np
import matplotlib.pyplot as plt
X = np.linspace(0.1, 4.2)
plt.plot(X, Gamma(X))
plt.show()
```

Corrigé exo 4.7

0. On commence par définir la fonction à intégrer. Mais cette fonction a deux arguments alors que `quad` a besoin d'une fonction à un seul argument. On définit la fonction dont on a besoin avec `lambda`.

```
import numpy as np
import scipy.integrate as integr


def u(x, t):
    return np.cos(x * np.cos(t))


def f(x):
    return integr.quad(lambda t: u(x, t), 0, np.pi)[0]
```

Alternativement, on pourrait écrire :

```
def f(x):
    def g(t):
        return u(x, t)
    return integr.quad(g, 0, np.pi)[0]
```

1. Il reste à tracer la fonction. On ne peut pas directement appliquer `f` à un tableau, donc on définit `Y` avec une liste par compréhension.

```
X = np.linspace(0, 10)
Y = [f(x) for x in X]
plt.plot(X, Y, label="A")
```

```
plt.legend()
plt.show()
```

Corrigé exo 4.8

0. On applique la méthode des trapèzes. La seconde version de la fonction utilise le fait que le pas est constant pour faire moins d'opérations arithmétiques.

```
def Imoy(mesure):
    pas = 0.002
    S = 0
    for k in range(len(mesure) - 1):
        S += (mesure[k] + mesure[k + 1]) * pas / 2
    return S
```

```
def Imoy(mesure):
    pas = 0.002
    S = 0
    for k in range(len(mesure)):
        S += mesure[k]
    return pas * (S - (mesure[0] + mesure[-1]) / 2)
```

1. On adapte la première version de la fonction précédente.

```
def Imoy(mesure, temps):
    S = 0
    for k in range(len(mesure) - 1):
        S += (mesure[k] + mesure[k + 1]) * (temps[k + 1] - temps[k]) / 2
    return S
```

Corrigé exo 4.9

```
import numpy as np
import matplotlib.pyplot as plt


def f(x):
    return x**3 - 2 * x**2 + 1


def fp(x):
    return 3 * x**2 - 4 * x


def cherche_racine(a, b, x):  # trouve le zéro d'une droite passant par (x,b) de pente a
    if np.abs(a) > 0.0000001:  # éviter les divisions par zéro!
        return - b / a + x
    else:
        return "erreur"


def Newton(y, yp, x0, eps):
    x = x0
    while np.abs(f(x)) >= eps:
        b = f(x)
        a = fp(x)
        x = cherche_racine(a, b, x)
        if x == "erreur":
            return None
    return x

racines = []
x0 = []
for k in range(-100, 300):
```

```
    x0.append(k / 100)
    racines.append(Newton(f, fp, k / 100, 10**-6))

plt.plot(x0, racines)
plt.show()
```

La fonction f a 3 racines : $x = 1$ et $x = \dfrac{-1 \pm \sqrt{5}}{2}$. La méthode de Newton trouve l'une des trois racines, sans que ce soit la plus proche de x_0 qui soit nécessairement obtenue.
On constate également que pour $x_0 = 0$ la méthode de Newton ne fonctionne pas car $f'(0) = 0$, la dérivée ne coupe jamais l'axe des abscisses.

Corrigé exo 4.10

0. On constate que si n est pair, le polynôme n'a pas de racine, et que s'il est impair, il en a une seule.

1. Après avoir défini en Python la fonction `P(n, t)` qui calcule $P_n(t)$, on programme Newton comme suit :

```
def newton(n, t, N):
    for k in range(N):
        t = t - P(n, t) / P(n - 1, t)
    return t
```

Il ne reste plus qu'à tester avec $n \in \{3, 5, 7\}$.

2. On peut définir P_5 en Python comme suit :

`P5 = Polynomial([1 / math.factorial(k) for k in range(6)])`.

On obtient les racines complexes de P_5 avec l'expression `P5.roots()`. Il ne reste qu'à faire la même chose avec P_3 et P_7. On pourrait faire un `for`, mais pour 3 valeurs de n, ce n'est pas indispensable.

3. Le script suivant permet de tracer les racines de P_5, on procède de même avec les autres polynômes.

```
import numpy as np
import matplotlib.pyplot as plt
R5 = P5.roots()
plt.plot(np.real(R5), np.imag(R5), "o")
```

Corrigé exo 4.11

0. On commence par importer les bibliothèques.

```
import matplotlib.pyplot as plt
import numpy as np
```

On programme classiquement la résolution par Euler. On peut le faire avec des listes ou avec des tableaux `numpy`.

```
def euler(fin, pas):
    n = int(fin / pas)
    T = [0]
    Y = [0]
    for k in range(n):
        T.append((k + 1) * pas)
        Y.append(Y[k] + pas * (1 + Y[k]**2))
    return T, Y
```

```
def euler(fin, pas):
    n = int(fin / pas)
    T = np.linspace(0, fin, n + 1)
    Y = np.zeros(n + 1)
    for k in range(n):
        Y[k + 1] = Y[k] + pas * (1 + Y[k]**2)
    return T, Y
```

Pour tracer la courbe demandée, il suffit d'écrire :

```
T, Y = euler(1, 0.05)
plt.plot(T, Y)
plt.show()
```

1. On remarque que la courbe de la solution est nécessairement symétrique par rapport à l'origine (la fonction solution est impaire). On utilise alors la fonction `euler` utilisant des tableaux pour écrire ce qui suit :

```
def eulersym(fin, pas):
    n = int(fin / pas)
    T0, Y0 = euler(fin, pas)
    T = np.linspace(-fin, fin, 2 * n + 1)
    Y = np.zeros(2 * n + 1)
    Y[n:] = Y0  # Copier Y0 dans la seconde moitiée de Y
    Y[:n + 1] = -Y0[::-1]  # Inverser l'ordre de Y0, le multiplier par -1.
    return T, Y
```

Il suffit alors, pour tracer la courbe, d'écrire :

```
T, Y = eulersym(1.5, 0.05)
plt.plot(T, Y)
plt.show()
```

2. On adapte le code de la question précédente :

```
T, Y = eulersym(1.5, 0.05)
plt.plot(T, Y)
plt.plot(T, np.tan(T))
plt.show()
```

3. La fonction semble définie sur $[-2, 2]$. On ne voit pas la singularité à cause de l'erreur d'approximation. La fonction approximée diverge vers $+\infty$ (on la voit dépasser 10^{25}).

Corrigé exo 4.12

0. Il suffit d'appliquer la méthode d'Euler sur chacun des intervalles $[t_0, t_0 + T]$ et $[t_0 - T, t_0]$, et de terminer en traçant les deux parties du graphes dans le même graphique. Les deux calculs sont identiques au signe près du pas, ce qui permet d'écrire :

```
import matplotlib.pyplot as plt


def Euler(f, t0, x0, T, n):
    plt.figure()
```

```
    for p in [T / n, -T / n]:
        t, x = t0, x0
        Lt, Lx = [t0], [x0]
        for i in range(n):
            t, x = t + p, x + p * f(t, x)
            Lt.append(t)
            Lx.append(x)
        plt.plot(Lt, Lx, color='black')
    plt.show()
```

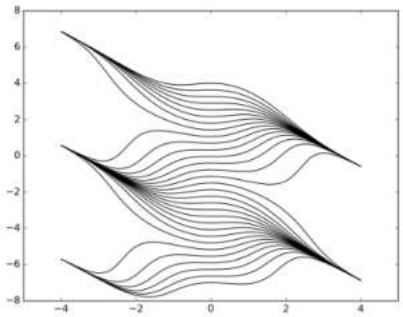
1. Il suffit d'ajouter une boucle au code précédent pour faire décrire à x_0 la liste L :

```
def Euler_Bis(f, t0, L, T, n):
    plt.figure()
    for x0 in L:
        for p in [T / n, -T / n]:
            t, x = t0, x0
            Lt, Lx = [t0], [x0]
            for i in range(n):
                t, x = t + p, x + p * f(t, x)
                Lt.append(t)
                Lx.append(x)
            plt.plot(Lt, Lx, color='black')
    plt.show()
```

On définit la fonction f et la liste L (contenant ici les 31 points d'une subdivision régulière de $[-7, 4]$) :

```
from math import sin


def f(t, x):
    return t * sin(t + x)
L = [(1 - i / 30) * (-7) + i / 30 * 4 for i in range(31)]
```

et l'on obtient le graphique ci-contre.



`Euler_Bis(f, 0, L, 4, 100)`

Corrigé exo 4.13

0. La méthode d'Euler consiste à partir de $t_0 = 0$, x_0 et y_0, puis à définir (avec $n = \frac{T}{p}$) :

$$\forall i \in \{0, \dots, n-1\}, \begin{cases} t_{i+1} = t_i + p \\ x_{i+1} = x_i + p\, f(x_i, y_i) \\ y_{i+1} = y_i + p\, g(x_i, y_i) \end{cases}$$

Nous initialisons donc trois listes $Lt = [0]$, $Lx = [x_0]$ et $Ly = [y_0]$ et les variables t, x, y permettent de stocker les différentes valeurs t_i, x_i et y_i, avant de les ajouter aux différentes listes. Cela donne :

```
def Calcul_Euler(f, g, x0, y0, T, p):
    Lt, Lx, Ly = [0], [x0], [y0]
    t, x, y = 0, x0, y0
```

```
    for i in range(int(T / p)):
        t, x, y = t + p, x + p * f(x, y), y + p * g(x, y)
        Lt.append(t)
        Lx.append(x)
        Ly.append(y)
    return Lt, Lx, Ly
```

Il est évidemment possible de se passer des variables t, x, y en utilisant les listes Lt, Lx et Ly : attention toutefois au fait qu'après avoir ajouté x_{i+1} à Lx, la valeur x_i n'est plus le dernier, mais l'avant-dernier élément de Lx :

```
def Calcul_Euler(f, g, x0, y0, T, p):
    Lt, Lx, Ly = [0], [x0], [y0]
    for i in range(int(T / p)):
        Lt.append(Lt[-1] + p)
        Lx.append(Lx[-1] + p * f(Lx[-1], Ly[-1])
        Ly.append(Ly[-1] + p * f(Lx[-2], Ly[-1]))
    return Lt, Lx, Ly
```

Après avoir défini les deux fonctions f et g :

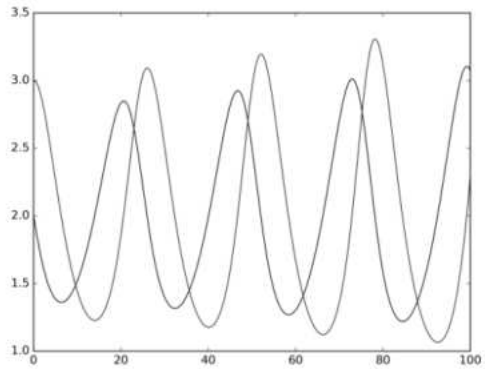
```
def f(x, y):
    return 0.2 * x - 0.1 * x * y

def g(x, y):
    return -0.3 * y + 0.15 * x * y
```

le lecteur pourra vérifier que les dernières valeurs des listes Lx et Ly retournées par l'appel `Calcul_Euler(f, g, 2, 3, 100, .01)` valent respectivement $2.7488\ldots$ et $2.3756\ldots$

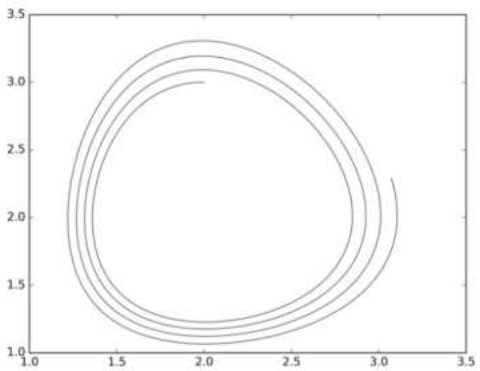
1. Une fois calculées les listes $(t_i)_{0\leqslant i\leqslant n}$, $(x_i)_{0\leqslant i\leqslant n}$ et $(y_i)_{0\leqslant i\leqslant n}$ à l'aide de `Calcul_Euler`, il suffit d'ouvrir une fenêtre graphique et de tracer (en les reliant) les points $(t_i, x_i)_{0\leqslant i\leqslant n}$ et $(t_i, y_i)_{0\leqslant i\leqslant n}$ ou bien les points $(x_i, y_i)_{0\leqslant i\leqslant n}$.

```
def Graphe(f, g, x0, y0, T, p):
    Lt, Lx, Ly = Calcul_Euler(f, g, x0, y0, T, p)
    plt.figure()
    plt.plot(Lt, Lx, color='b')
    plt.plot(Lt, Ly, color='r')
    plt.show()
```

```
def Phase(f, g, x0, y0, T, p):
    Lt, Lx, Ly = Calcul_Euler(f, g, x0, y0, T, p)
    plt.figure()
    plt.plot(Lx, Ly)
    plt.show()
```



`Graphe(f, g, 2, 3, 100, 0.1)`

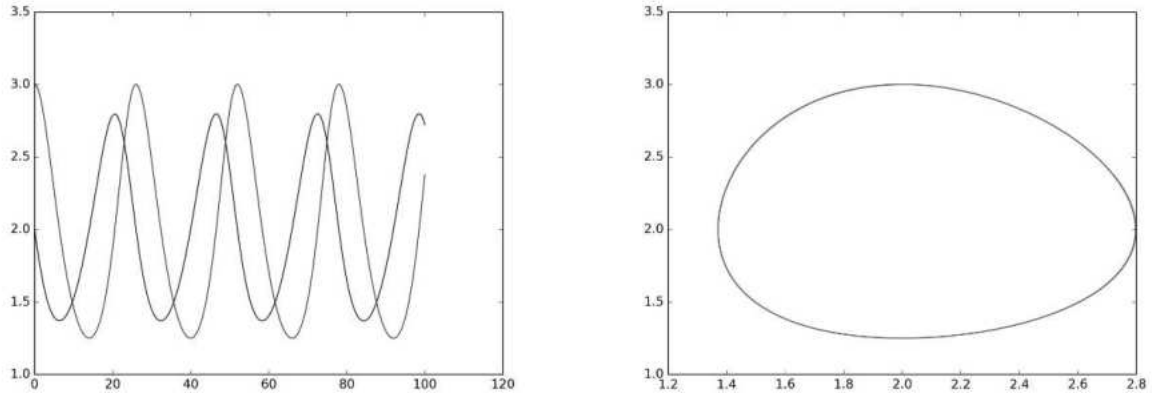


`Phase(f, g, 2, 3, 100, 0.1)`

2. Ici encore, on ouvre une fenêtre graphique et on utilise `subplot` pour créer deux sous-fenêtres alignées horizontalement :

```
def Graphe_Phase(f, g, x0, y0, T, p):
    Lt, Lx, Ly = Calcul_Euler(f, g, x0, y0, T, p)
    plt.figure()
    plt.subplot(1, 2, 1)
    plt.plot(Lt, Lx, color='b')
    plt.plot(Lt, Ly, color='r')
    plt.subplot(1, 2, 2)
    plt.plot(Lx, Ly, color='g')
    plt.show()
```

3. On part du points $(2,3)$ avec x et y décroissantes : le point (x,y) tourne donc dans le sens trigonométrique direct. Comme la vraie solution est périodique, la trajectoire devrait se refermer après le premier tour, mais celle obtenue avec la méthode d'Euler s'éloigne en spiralant de la trajectoire périodique que l'on cherche à approximer. Il est donc nécessaire de diminuer le pas pour obtenir un tracé acceptable ; on obtient une courbe qui semble fermée (pour la précision du tracé) en prenant un pas égal à 0.001 :

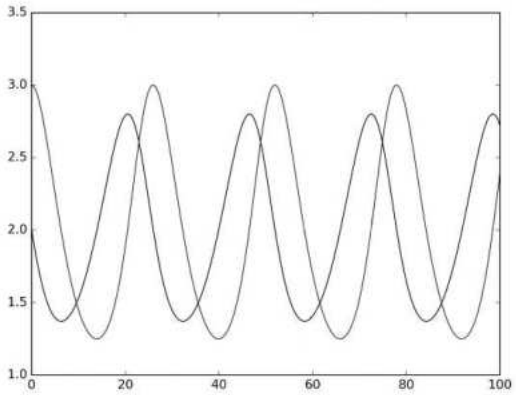


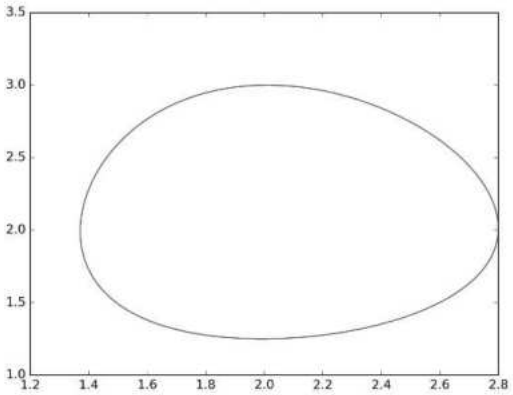
`Graphe_Phase(f, g, 2, 3, 100, 0.001)`

4. L'élève a commis l'erreur d'utiliser la nouvelle valeur x_{i+1} de x pour définir y_{i+1}, ce qui donne les relations :

$$\forall i \in \{0,\dots,n-1\},\ \begin{cases} t_{i+1} = t_i + p \\ x_{i+1} = x_i + p\, f(x_i, y_i) \\ y_{i+1} = y_i + p\, g(x_{i+1}, y_i) \end{cases}$$

Avec sa fonction, il obtient paradoxalement de meilleurs résultats qu'avec la méthode d'Euler (cette méthode est quelquefois appelée **méthode d'Euler asymétrique**). En effet, ses fonctions pour $p = 0.1$ semblent périodiques, alors que ce n'est pas du tout le cas avec celles obtenues avec la méthode d'Euler pour le même pas p :






```
Graphe_Phase_Eleve(f, g, 2, 3, 100, 0.1)
```

On peut démontrer que l'erreur de l'élève permet effectivement d'améliorer sensiblement la méthode d'Euler.

Corrigé exo 4.14

1. La fonction suivante permet de calculer l'évolution de chaque composé, et la liste des temps, en appliquant n étapes de la méthode d'Euler.

```
import numpy as np


def euler(tfinal, n, k1, k2):
    O, H2O, E, D, T = [1], [1], [0], [0], [0]
    dt = tfinal / n
    for k in range(n):
        dxi1 = k1 * O[-1] * H2O[-1] * dt
        dxi2 = k2 * O[-1] * E[-1] * dt
        O.append(O[-1] - dxi1 - dxi2)
        H2O.append(H2O[-1] - dxi1)
        E.append(E[-1] + dxi1 - dxi2)
        D.append(D[-1] + dxi2)
        T.append((k + 1) * dt)
    return (O, H2O, E, D), T
```

Il ne reste qu'à tracer les courbes avec le code suivant, puis à les comparer avec les courbes du sujet de Centrale.

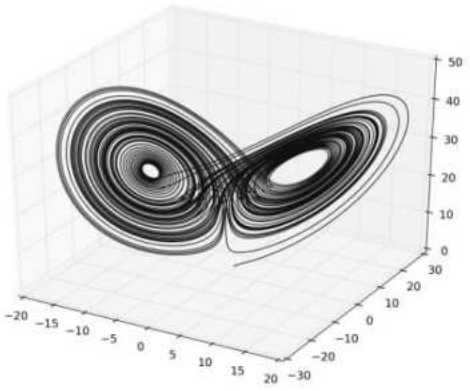
```
import matplotlib.pyplot as plt

V, T = euler(10, 1000, 1, 5)
for X in V:
    plt.plot(T, X)
```

Corrigé exo 4.15

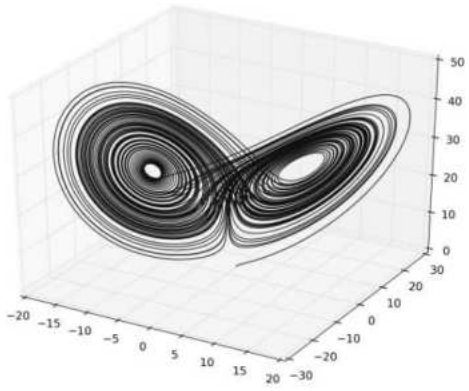
0. On obtient comme dans les exercices précédents :

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt


def Lorenz(x0, y0, z0, T, p):
    Lx, Ly, Lz = [x0], [y0], [z0]
    x, y, z = x0, y0, z0
    for i in range(int(T / pas)):
        x, y, z = x + p * 10 * (y - x), y + p * (28 *
                                                 x - y - x * z), z + p * (x * y - 8 / 3 * z)
        Lx.append(x)
        Ly.append(y)
        Lz.append(z)
    dessin = Axes3D(plt.figure())
    dessin.plot(Lx, Ly, Lz)
    plt.show()
```

1. On obtient les graphiques :



`Lorenz(1, 1, 1, 100, 0.001)`



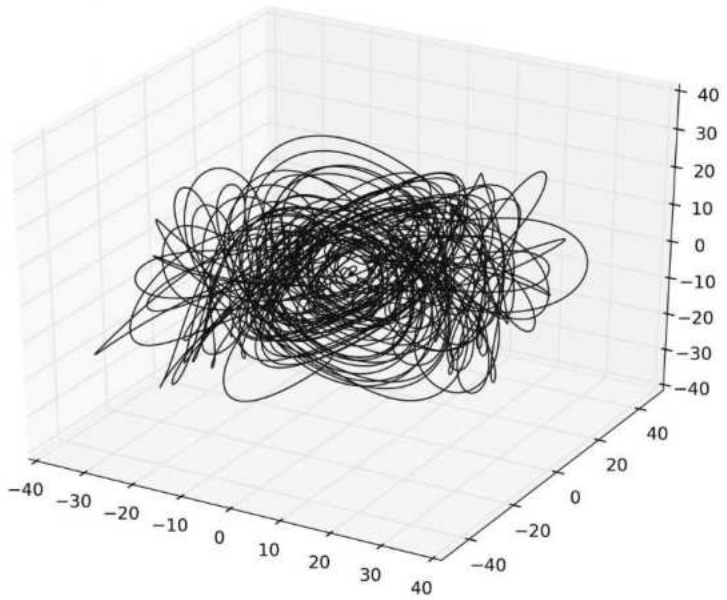
`Lorenz(1.0001, 1, 1, 100, 0.001)`

qui donne l'impression que les deux trajectoires sont pratiquement identiques. Pour calculer l'écart entre deux trajectoires, nous utilisons des listes qui stockent les coordonnées sur chaque trajectoires (listes Lx, Ly, Lz, LX, LY, et LZ) ainsi que les écarts entre ces coordonnées (listes Ex, Ey et Ez) :

```
def Chaos_Conditions_Initiales(x0, y0, z0, X0, Y0, Z0, T, pas):
    Lx, Ly, Lz = [x0], [y0], [z0]
    LX, LY, LZ = [X0], [Y0], [Z0]
    Ex, Ey, Ez = [X0 - x0], [Y0 - y0], [Z0 - z0]
    x, y, z, X, Y, Z = x0, y0, z0, X0, Y0, Z0
    for i in range(int(T / pas)):
        x, y, z = x + pas * 10 * \
            (y - x), y + pas * (28 * x - y - x * z), z + pas * (x * y - 8 / 3 * z)
        Lx.append(x)
        Ly.append(y)
        Lz.append(z)
        X, Y, Z = X + pas * 10 * \
            (Y - X), Y + pas * (28 * X - Y - X * Z), Z + pas * (X * Y - 8 / 3 * Z)
        LX.append(X)
        LY.append(Y)
        LZ.append(Z)
        Ex.append(X - x)
        Ey.append(Y - y)
        Ez.append(Z - z)
    dessin1 = Axes3D(plt.figure())
    dessin1.plot(Lx, Ly, Lz)
    plt.show()
```

```
dessin2 = Axes3D(plt.figure())
dessin2.plot(LX, LY, LZ)
plt.show()
dessin3 = Axes3D(plt.figure())
dessin3.plot(Ex, Ey, Ez)
plt.show()
```

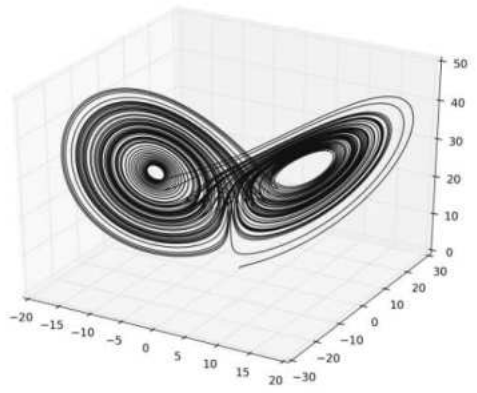
Cette fonction donne (seul le dernier graphique est reproduit) :



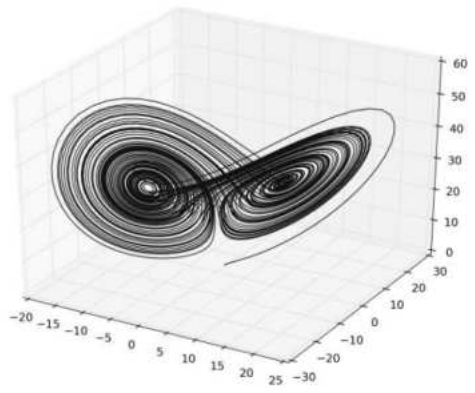
`Chaos_Conditions_Initiales(1, 1, 1, 1.0001, 1, 1, 100, 0.001)`

On voit donc que les deux solutions sont en fait très différentes et que leur différence est chaotique.

2. On observe exactement le même comportement si l'on modifie le pas de la méthode d'Euler :



`Lorenz(1, 1, 1, 100, 0.001)`

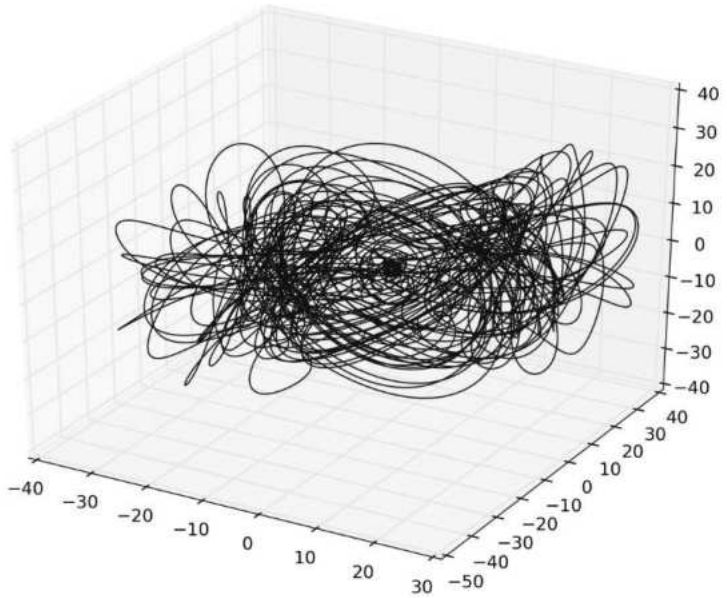


`Lorenz(1.0001, 1, 1, 100, 0.0001)`

Pour étudier l'écart entre ces deux trajectoires, il faut faire attention au fait que le temps varie 10 fois plus vite quand le pas est divisé par 10. Nous reprenons donc la méthode de la question précédente : x, y, z est la coordonnée (variable) pour le pas $p/10$ et X, Y, Z celle pour le pas p ; on modifie ces valeurs à l'aide d'une boucle de longueur $N = 10/p \times T$. En faisant varier i

entre 1 et N, nous modifions x, y et z à chaque tour de boucle et, quand i est un multiple de 10, nous modifions également X, Y et Z et nous ajoutons $X-x$, $Y-y$ et $Z-z$ aux listes des écarts.

```
def Chaos_Pas(x0, y0, z0, T, pas):
    x, y, z = x0, y0, z0
    X, Y, Z = x0, y0, z0
    Ex, Ey, Ez = [0], [0], [0]
    for i in range(1, int(T * 10 / pas) + 1):
        x, y, z = x + pas / 10 * 10 * \
            (y - x), y + pas / 10 * (28 * x - y - x * z), z + \
            pas / 10 * (x * y - 8 / 3 * z)
        if i % 10 == 0:
            X, Y, Z = X + pas * 10 * \
                (Y - X), Y + pas * (28 * X - Y - X * Z), Z + \
                pas * (X * Y - 8 / 3 * Z)
            Ex.append(X - x)
            Ey.append(Y - y)
            Ez.append(Z - z)
    dessin = Axes3D(plt.figure())
    dessin.plot(Ex, Ey, Ez)
    plt.show()
```



`Chaos_Pas(1, 1, 1, 100, 0.001)`

Corrigé exo 4.16

0. Pour appliquer la méthode d'Euler, nous utilisons deux variables `t` et `x` qui contiennent successivement les valeurs t_i et x_i. Après avoir initialisé ces variables, il suffit d'effectuer une boucle de longueur n, à la fin de laquelle x contiendra la valeur $x_n = \varphi(n, T)$:

```
from math import exp


def epsilon(n, T):
    t, x = 0, 0
    x = 0
    p = T / n
    for i in range(n):
        t, x = t + p, x + p * (x + t)
    return abs(x + 1 + T - exp(T))
```

Attention de bien utiliser l'ancienne valeur de t pour calculer la nouvelle valeur de x. Si l'on veut éviter l'affectation multiple de la ligne 6, il faut commencer par modifier x, puis t (et pas l'inverse).

1. Le script suivant affiche les valeurs de l'erreur pour $T = 2$ et $n \in \{10, 10^2, \ldots, 10^6\}$:

```
for i in range(1, 7):
    print(epsilon(10**i, 2))
```

L'observation des résultats de ce script laisse penser que l'erreur est de l'ordre de $1/n$, puisqu'elle est divisée par 10 quand n est multiplié par 10. Cette impression peut être confirmée en calculant $n\,\varepsilon(n, T)$ pour $T \in \{1, 2, 3\}$ et pour $n \in \{10, 10^3, 10^5, 10^7\}$:

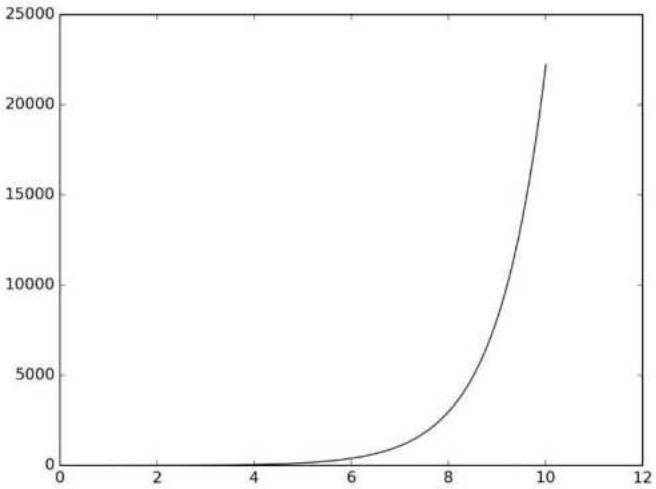
```
for T in [1, 2, 3]:
    print([n*epsilon(n, T) for n in [10,10**3,10**5,10**7]])
```

Les résultats permettent de conjecturer que pour T fixé, il existe une constante K telle que $\varepsilon(n, T)$ soit équivalent à K/n quand n tend vers l'infini.

2. Pour étudier cette erreur, nous allons fixer $N \in \mathbb{N}^*$ et calculer $\varepsilon(n, n\,p)$ pour n variant de 1 à N. Il serait maladroit d'utiliser la fonction `epsilon`, puisque le calcul de $\varepsilon(N, N\,p)$ contient déjà celui de tous les $\varphi(n, n\,p)$ pour n de 1 à N. Nous allons donc ici reprendre le code de la fonction `epsilon`, stocker toutes les erreurs et renvoyer le graphe formé par les points $(n\,p,\ \varepsilon(n, n\,p))$.

```
def erreur(p, N):
    Lt, Le = [0], [0]
    t, x = 0, 0
    for i in range(N + 1):
        t, x = t + p, x + p * x * t
        Le.append(abs(x + 1 + t - exp(t)))
        Lt.append(t)
    plt.figure()
    plt.plot(Lt, Le)
    plt.show()
```

Avec $p = 10^{-2}$ et $N = 1000$ (on travaille donc sur le segment $[0, 10]$), nous obtenons le graphique ci-contre.
L'erreur semble donc être exponentielle par rapport à T. On confirme ce caractère en traçant le logarithme népérien de l'erreur, grâce à la fonction définie ci-après.

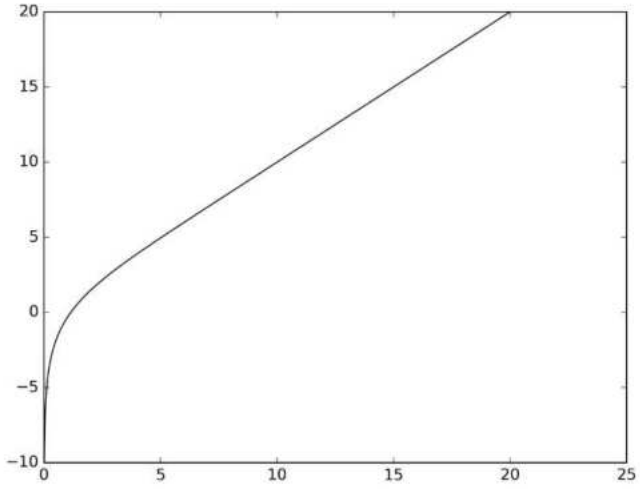



```
from math import log


def erreur_log(p, N):
    Lt, Le = [0], [0]
    t, x = 0, 0
    for i in range(N + 1):
        t, x = t + p, x + p * x * t
        Le.append(log(abs(x + 1 + t - exp(t))))
        Lt.append(t)
    plt.figure()
    plt.plot(Lt, Le)
    plt.show()
```

Nous obtenons, toujours pour $p = 10^{-2}$ et $N = 2000$, le graphique ci-contre.
Nous pouvons donc conjecturer que pour $p > 0$ fixé, il existe deux constantes a et b telles que

$$\ln(\varepsilon(n, p\,n)) = an + b + o(1),$$

i.e., qu'il existe deux constantes $\lambda > 1$ et $C > 0$ telles que $\varepsilon(n, p\,n)$ soit équivalent à $C\,\lambda^n$.



Corrigé exo 4.17

0. On obtient facilement :

```
def E(f, t0, x0, T, n):
    t, x = t0, x0
    p = T / n
    for i in range(n):
        t, x = t + p, x + p * f(t, x)
    return x


def PM(f, t0, x0, T, n):
    t, x = t0, x0
    p = T / n
    for i in range(n):
        t1, x1 = t + p / 2, x + p / 2 * f(t, x)
        t, x = t + p, x + p * f(t1, x1)
    return x


def TR(f, t0, x0, T, n):
    t, x = t0, x0
    p = T / n
    for i in range(n):
        t1, x1 = t + p, x + p * f(t, x)
        t, x = t + p, x + p / 2 * (f(t, x) + f(t1, x1))
    return x
```

1. On obtient les résultats suivants (ici $T = 1$ et $n \in \{10, 10^2, \ldots, 10^6\}$, mais les choses seraient similaires en choisissant une autre valeur de T dans $]-\sqrt{2}, \sqrt{2}[$) :

```
>>> def f(t,x):
    return t*x**2
>>> T = 1
>>> def phi(T):
    return 2/(2 - T**2)
>>> for i in range(1,7):
    print([E(f,0,1,T,10**i)-phi(T), PM(f,0,1,T,10**i)-phi(T), TR(f,0,1,T,10**i)-phi(T)])
[-0.2871474140956658, -0.02517014230930137, -0.01187430956438007]
[-0.038156185656135655, -0.0003153566362308346 5, -0.000122915032089832]
[-0.003955298695805354, -3.2200458439657353e-06, -1.2276630267926691e-06]
[-0.0003969965871071235, -3.226690892255135e-08, -1.2274513627730244e-08]
[-3.971440104200141e-05, -3.2402658334262924e-10, -1.240545444147756e-10]
[-3.971588933504577e-06, -4.759304061963121e-12, -2.7973179328455444e-12]
```

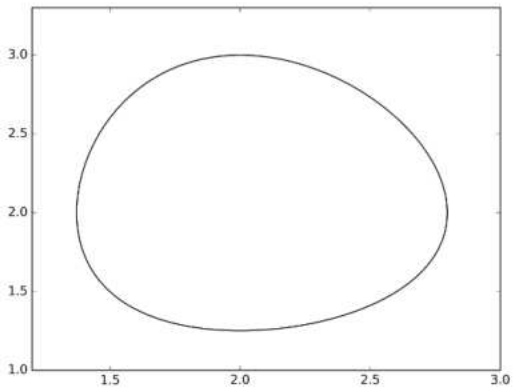
Les deux nouvelles méthodes semblent donc donner des erreurs de l'ordre de $1/n^2$ (l'erreur est divisée par 100 quand n est multiplié par 10), ce qui est bien meilleur que l'erreur en $1/n$ de la méthode d'Euler.

2. De la même manière que dans l'exercice **4.13**, nous obtenons les fonctions :

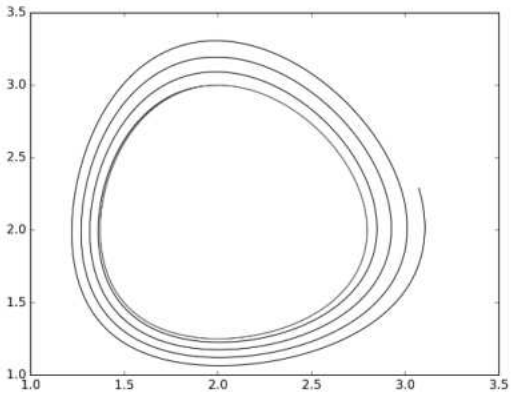
```
def G_milieu(f, g, x0, y0, T, p):
    Lx, Ly = [x0], [y0]
    t, x, y = 0, x0, y0
    for i in range(int(T / p)):
        x1, y1 = x + p / 2 * f(x, y), y + p / 2 * g(x, y)
        x, y = x + p * f(x1, y1), y + p * g(x1, y1)
        Lx.append(x)
        Ly.append(y)
    plt.figure()
    plt.plot(Lx, Ly, color='g')
    plt.show()
```

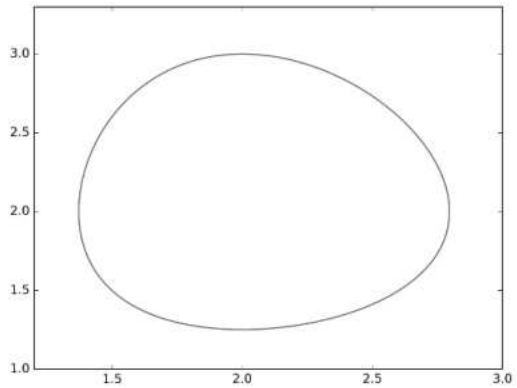
```
def G_trapeze(f, g, x0, y0, T, p):
    Lx, Ly = [x0], [y0]
    t, x, y = 0, x0, y0
    for i in range(int(T / p)):
        x1, y1 = x + p * f(x, y), y + p * g(x, y)
        x, y = x + p / 2 * (f(x, y) + f(x1, y1)), y + \
            p / 2 * (g(x, y) + g(x1, y1))
        Lx.append(x)
        Ly.append(y)
    plt.figure()
    plt.plot(Lx, Ly, color='g')
    plt.show()
```

Les instructions `G_milieu(f, g, 2, 3, 100, 0.1)` et `G_trapeze(f, g, 2, 3, 100, 0.1)` donnent le même graphique :



Ainsi, avec un pas égal à 0.1, ces deux méthodes donnent déjà une trajectoire qui semble se refermer, alors que la méthode d'Euler, avec les mêmes paramètres, donne une spirale très nette. Il faut toutefois être prudent, car le fait que la courbe se referme (à la précision près du tracé) ne prouve pas que le résultat obtenu est proche de la trajectoire réelle. Les deux graphiques qui suivent montrent le tracé de la solution approchée pour le pas 0.1 et de la trajectoire réelle, à gauche pour la méthode d'Euler et à droite pour la méthode de type trapèze :





Dans le cas général, ces deux méthodes avec le pas 0.1 ont à peu près la même efficacité que la méthode d'Euler avec le pas 0.01, et le cas particulier étudié ici est très flatteur (il faut un pas de 0.001 pour que la méthode d'Euler donne un résultat équivalent).

Corrigé exo 4.18

Pas besoin d'écrire une fonction.

```
import matplotlib.pyplot as plt
X = [0]
Y = [0]
Yp = [1]  # La dérivée de Y
pas = 0.9 / 1000  # Le pas de la méthode d'Euler
for k in range(1, 1000):
    X.append(k * pas)
    Y.append(Y[k - 1] + Yp[k - 1] * pas)
    Ys = Y[k - 1] / (1 - X[k - 1])**3  # La dérivée seconde
    Yp.append(Yp[k - 1] + Ys * pas)
plt.plot(X, Y)
plt.show()
```

Corrigé exo 4.19

0. Dans ce cas particulier, la méthode d'Euler de pas $p = T/n$, consiste à définir la famille $(x_i, x'_i)_{0 \leqslant i \leqslant n}$ par les relations :

$$x_0 = 0,\ x'_0 = 1 \text{ et } \forall i \in \{1, \dots, n\},\ \begin{cases} x_{i+1} = x_i + p\, x'_i \\ \\ x'_{i+1} = x'_i - p\, x_i \end{cases}$$

Nous utilisons ici deux variables x et xp, qui contiennent les différentes valeurs x_i et x'_i, que nous modifions à l'aide d'une boucle. À la fin du calcul, il reste à renvoyer la valeur x, qui est égale à x_n, c'est-à-dire à $\Phi(n, T)$:

```
def phi(n, T):
    x, xp = 0, 1
    p = T / n
    for i in range(n):
        x, xp = x + p * xp, xp - p * x
    return x
```

Ainsi, pour $T = 3$ (mais le lecteur pourra tester d'autres valeurs), nous pouvons calculer les erreurs $|\Phi(n, T) - \varphi(n)|$ pour $n \in \{10, 10^2, \dots, 10^6\}$:

```
>>> from math import sin
>>> T = 3
>>> for i in range(7):
    print(phi2(10**i,T) - sin(T))
2.8588799919401326
0.20519282194013302
0.007423886675805597
0.0006454182147336285
6.360743073452468e-05
```

```
6.351434282819701e-06
6.350503145291508e-07
```

L'erreur semble donc être de l'ordre de $1/n$.

1. On obtient facilement :

```
def phi1(n, T):
    x, xp = 0, 1
    p = T / n
    for i in range(n):
        x, xp = x + p * xp - p ^ 2 / 2 * x, xp - p * x
    return x
```

Cette méthode n'améliore que très peu la méthode d'Euler, comme on le voit en affichant les valeurs $\left|\dfrac{\Phi_1(n,T)-\varphi(T)}{\Phi(n,T)-\varphi(T)}\right|$ pour $T \in \{-2,-1,3,10\}$ et $n \in \{10, 10^2, \ldots, 10^6\}$ (chaque ligne correspond à une valeur de T et on n'affiche que deux décimales pour faciliter la lecture) :

```
>>> for T in [-2,-1,3,10]:
    print(["%.2f" %abs((phi1(10**i,T) - sin(T))/(phi(10**i,T) - sin(T))) for i in range(7)])
['1.00', '0.48', '0.50', '0.50', '0.50', '0.50', '0.50']
['1.00', '0.51', '0.50', '0.50', '0.50', '0.50', '0.50']
['1.00', '0.35', '0.48', '0.50', '0.50', '0.50', '0.50']
['1.00', '0.09', '0.47', '0.50', '0.50', '0.50', '0.50']
```

On peut conjecturer que le quotient des deux erreurs tend vers $1/2$ quand n tend vers l'infini, ce qui limite l'intérêt de la « méthode d'Euler améliorée » (voir l'exercice **4.17** qui expose deux méthodes qui accélèrent de façon élémentaire mais significative la méthode d'Euler).

Corrigé exo 4.20

0. En notant $n = T/p$, nous devons construire les listes $Lt = [t_0, \ldots, t_n]$, $Lx = [x_0, \ldots, x_n]$ et $Lx' = [x'_0, \ldots, x'_n]$ définies par les relations :

$$\forall i \in \{1, \ldots, n\}, \begin{cases} t_{i+1} = t_i + p \\ x_{i+1} = x_i + p\,x'_i \\ x'_{i+1} = x'_i + p\,f(t_i, x_i, x'_i) \end{cases}$$

Nous utilisons trois variables `t`, `x` et `xp` qui contiennent les valeurs successives des t_i, x_i et x'_i : une simple boucle permet de remplir les listes `Lt`, `Lx` et `Lxp`, après les avoir initialisées aux valeurs `[t0]`, `[x0]` et `[xprime0]`. Il reste ensuite à ouvrir les fenêtres graphiques et à tracer les courbes définies par les points (t_i, x_i) dans la première fenêtre et (x_i, x'_i) dans la seconde.

```
def Euler(f, t0, x0, xprime0, T, p):
    Lt, Lx, Lxp = [t0], [x0], [xprime0]
    t, x, xp = t0, x0, xprime0
    for i in range(int(T / p)):
        t, x, xp = t + p, x + p * xp, xp + p * f(t, x, xp)
        Lt.append(t)
        Lx.append(x)
        Lxp.append(xp)
    plt.figure()
    plt.plot(Lt, Lx, color='b')
```

```
    plt.show()
    plt.figure()
    plt.plot(Lx, Lxp, color='r')
    plt.show()
```

1. Pour cette deuxième fonction, après avoir ouvert une fenêtre graphique, nous créons, pour chaque condition initiale C de L, les listes Lx et $L'x$, puis nous traçons la trajectoire associée. Il ne reste pour finir qu'à afficher la fenêtre graphique.

```
def EulerPhase(f, t0, L, T, p):
    plt.figure()
    for C in L:
        Lx, Lxp = [C[0]], [C[1]]
        t, x, xp = t0, C[0], C[1]
        for i in range(int(T / p)):
            t, x, xp = t + p, x + p * xp, xp + p * f(t, x, xp)
            Lx.append(x)
            Lxp.append(xp)
        plt.plot(Lx, Lxp)
    plt.show()
```

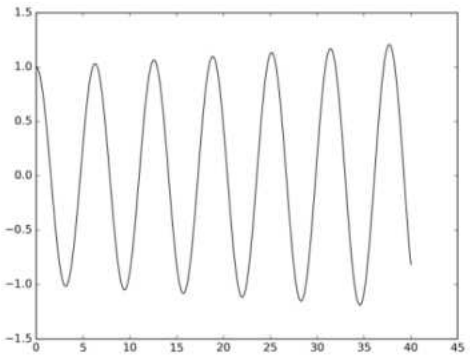
2. Nous commençons par définir les fonctions `fa`, `fb`, `fc` et `fd` correspondant aux quatre exemples étudiés (on suppose que la fonction `sin` a été préalablement chargée) :

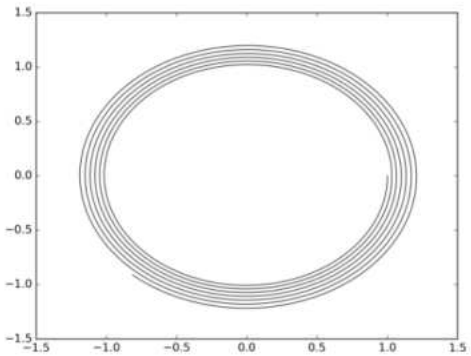
```
def fa(t, x, y):
    return -x


def fb(t, x, y):
    return -(x**2 - 1) * y - x
```

```
def fc(t, x, y):
    return -sin(x)

def fd(t, x, y):
    return -sin(x) - 1 / 5 * y
```

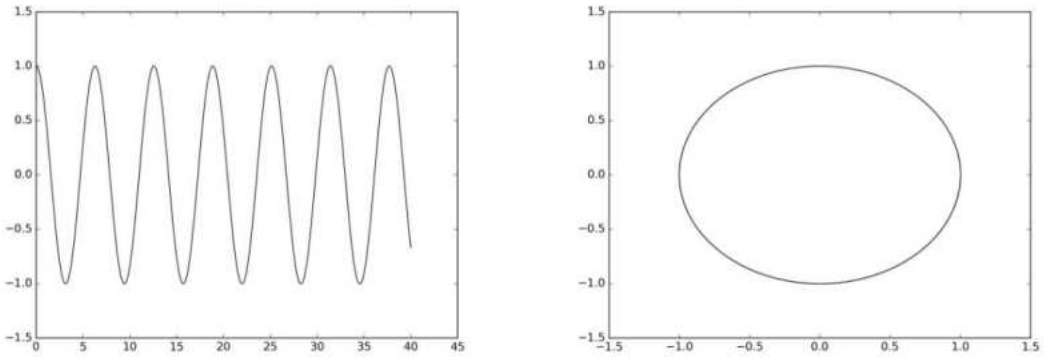
a. Les solutions de l'oscillateur harmonique sont périodiques ; la convergence lente de la méthode d'Euler nous oblige à choisir un pas assez petit :

```
Euler(fa, 0, 1, 0, 40, 0.01)
```



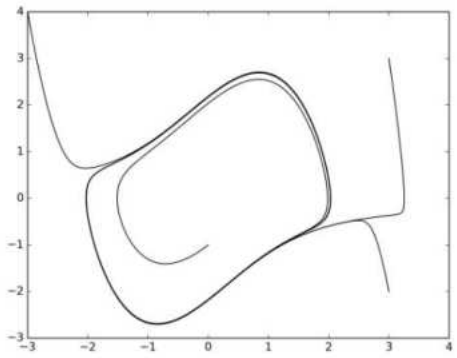



```
Euler(fa, 0, 1, 0, 40, 0.0001)
```



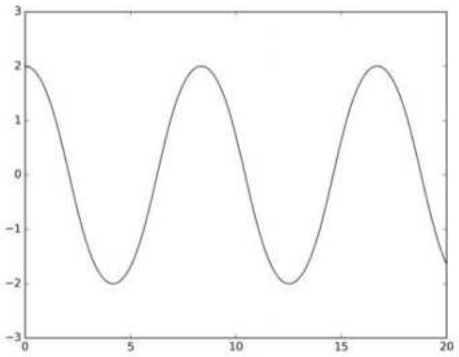
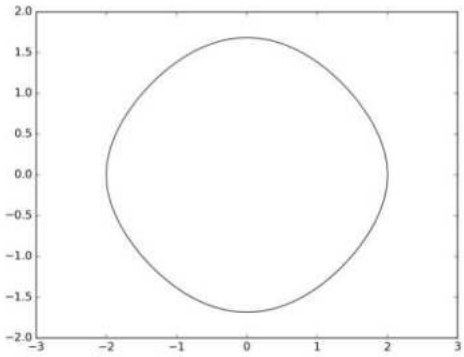
b. Ici, le pas 0.01 suffit pour obtenir un graphique correct : nous avons tracé ci-dessous quelques solutions particulières qui font apparaître une trajectoire asymptote fermée (qui correspond à une solution périodique). Les autres solutions convergent vers cette solution périodique en tournant dans le sens des aiguilles d'une montre :

```
EulerPhase(fb, 0, [[0, -1], [-3, 4], [3, 3], [3, -2]], 30, 0.01)
```



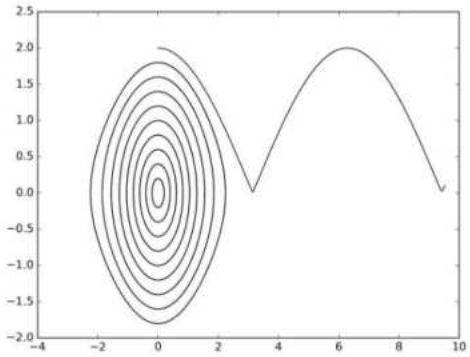
c. Pour le pendule pesant, on voit qu'avec la condition initiale $x(0) = 2$ et $x'(0) = 0$, le portrait de phase n'est pas du tout elliptique (on n'est évidemment pas dans le cas particulier des petits angles et l'approximation de $\sin x$ par x n'est pas acceptable) :

```
Euler(fc, 0, 2, 0, 20, 0.0001)
```

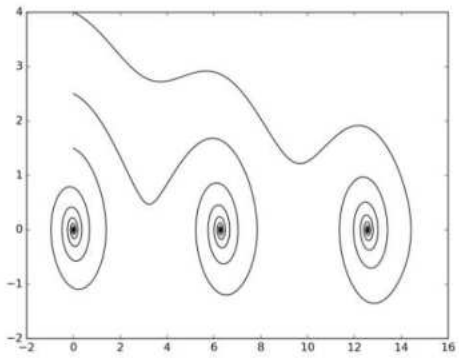
Nous pouvons ensuite tracer un ensemble de solutions, avec $x(0) = 0$ et $x'(0)$ décrivant $[0, 2]$ avec un pas de $1/10$:

```
EulerPhase(fc, 0, [[0, 2 * i / 10] for i in range(11)], 20, 0.0001)
```



d. Nous traçons ici trois trajectoires : en partant de $x(0) = 0$, on obtient selon la vitesse initiale un pendule qui fait 0, 1 ou 2 tours avant de rejoindre (en un temps infini) sa position d'équilibre stable :

```
EulerPhase(fd, 0, [[0, 1.5], [0, 2.5], [0, 4]], 50, 0.0001)
```

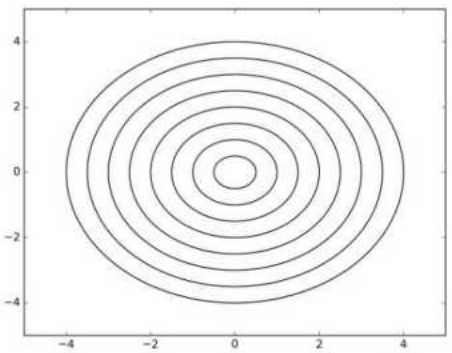


3. Pour chaque condition initiale, nous calculons les valeurs (t_i, x_i, x_i') successivement pour les pas $-p$ et p, en remplaçant la boucle `for` par une boucle `while` portant sur la condition $\big(t_0 - T \leqslant t_i \leqslant t_0 + T,\ a \leqslant x_i \leqslant b\ c \leqslant x_i' \leqslant d\big)$. Ici, nous initialisons les listes `Lx` et `Lxp` à la valeur `[]`, puisque nous ajoutons les valeurs courantes `x` et `xp` une fois testée la condition. Cela donne :

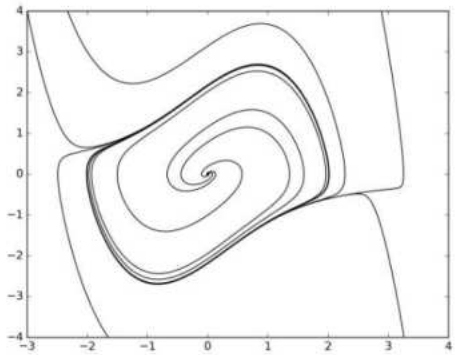
```
def EulerPhaseBox(f, t0, L, T, p, a, b, c, d):
    plt.figure()
    for C in L:
        for pas in [-p, p]:
            Lx, Lxp = [], []
            t, x, xp = t0, C[0], C[1]
            while t0 - T <= t <= t0 + T and a <= x <= b and c <= xp <= d:
                Lx.append(x)
                Lxp.append(xp)
                t, x, xp = t + pas, x + pas * xp, xp + pas * f(t, x, xp)
            plt.plot(Lx, Lxp, color='black')
    plt.show()
```

Pour obtenir des schémas intéressants, le plus efficace est de fixer une fenêtre $[a, b] \times [c, d]$ puis d'ajouter les conditions initiales « à la main », pour remplir l'espace vide. Nous obtenons ainsi les quatre graphiques ci-dessous, par le biais des instructions :

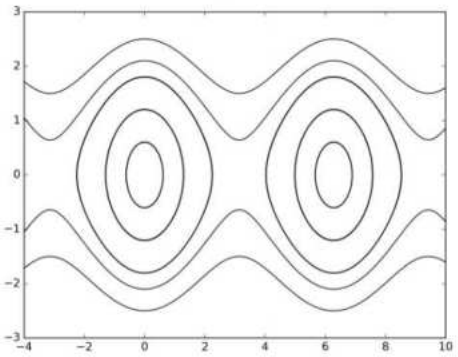
```
EulerPhaseBox(fa, 0, [[0, 5 * i / 10] for i in range(11)], 10, 0.001, -4.1, 4.1, -4.1, 4.1)
EulerPhaseBox(fb, 0, [[0, -1], [-3, 4], [3, 3], [3, -2], [-2, 3], [1, 1], [1, 1.5], [-2, -3]],
                     20, 0.001, -4, 4, -4, 4)
EulerPhaseBox(fc, 0, [[0, 0.6], [0, 1.2], [0, 1.8], [2 * pi, 0.6], [2 * pi, 1.2], [2 * pi, 1.8],
                     [-4, -2], [-4, -1], [-4, 0], [-4, 1], [-4, 2]], 20, 0.001, -4, 10, -3, 3)
EulerPhaseBox(fd, 0, [[1, 0], [-2, 0], [-3, 0], [-3.5, 0], [-4, 0], [4, 0], [3, 2], [3.5, 3],
                     [4, 4], [-2, -4], [0, -3.5], [8, 5]], 50, 0.0001, -5, 10, -5, 5)
```



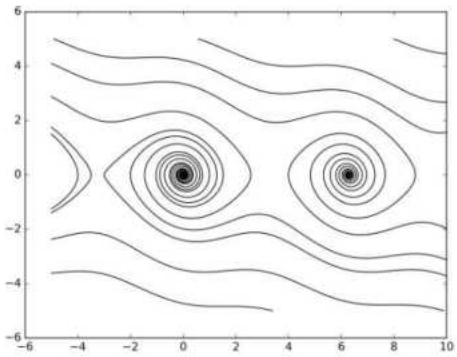
Oscillateur harmonique



Oscillateur de Van der Pol



Pendule pesant



Pendule pesant amorti

Corrigé exo 4.21

```
# Chargement des bibliothèques
import matplotlib.pyplot as plt
import numpy as np

# Définition des fonctions


def pendule_petits_angles(theta, thetap):
    return -omega0**2 * theta - 2 * xi * omega0 * thetap


def pendule(theta, thetap):
    return -omega0**2 * np.sin(theta) - 2 * xi * omega0 * thetap


def pendule_petits_angles_th(theta0, t):
    return (theta0 * np.exp(-xi * omega0 * t) * np.cos(omega0 * np.sqrt(1 - xi**2) * t)) * 180 / np.pi


def Euler_2_exp(f, x0, xp0, t):  # Intégration par la méthode d'Euler explicite
    x = [x0]
    xp = [xp0]
    xpp = []
    for i in range(1, len(t)):
        xpp.append(f(x[i - 1], xp[i - 1]))
        xp.append(xp[i - 1] + (t[i] - t[i - 1]) * xpp[i - 1])
        x.append(x[i - 1] + (t[i] - t[i - 1]) * xp[i - 1])
    for i in range(len(x)):
        x[i] = x[i] * 180 / np.pi
    return x


def Euler_2_asym(f, x0, xp0, t):  # Intégration par la méthode d'Euler asymétrique
    x = [x0]
    xp = [x0]
    xpp = []
    for i in range(1, len(time)):
        xpp.append(f(x[i - 1], xp[i - 1]))
        xp.append(xp[i - 1] + (t[i] - t[i - 1]) * xpp[i - 1])
        x.append((x[i - 1] + (t[i] - t[i - 1]) * xp[i]))
    for i in range(len(x)):
        x[i] = x[i] * 180 / np.pi
    return x
```

```
# Définition des paramètres
theta_0_deg = 0.1  # degré
theta_0 = theta_0_deg * 2 * np.pi / 360  # radians
thetap_0 = 0
tmax = 5  # s
pas = 0.01  # s
xi = 0.1  # amortissement
omega0 = 10  # rad.s^-1

# Définition de la liste des temps
time = np.linspace(0, tmax, tmax / pas + 1)

# Les tracés
# Tracé Q1-(1)
plt.figure(1)
theta_th = pendule_petits_angles_th(theta_0, time)
plt.plot(time, theta_th, '--', label='Solution théorique')
theta_eul_exp = Euler_2_exp(pendule_petits_angles, theta_0, thetap_0, time)
plt.plot(time, theta_eul_exp, label='Solution numérique explicite')
plt.title('Tracé des différentes solutions aux petits angles, theta_0 = ' +
          str(theta_0_deg) + 'degré')
plt.xlabel('temps en s')
plt.ylabel('Angle theta en degré')
plt.legend()
plt.show()

# Réponse Q1-(2)
theta_0_deg_list = [1, 5, 10, 20]

for i in range(len(theta_0_deg_list)):
    theta_0_temp = (theta_0_deg_list[i] * 2 * np.pi / 360)
    theta_th = pendule_petits_angles_th(theta_0_temp, time)
    theta_eul_exp = Euler_2_exp(
        pendule_petits_angles, theta_0_temp, thetap_0, time)
    plt.figure()
    plt.plot(time, theta_eul_exp, label="solution d'Euler")
    plt.plot(time, theta_th, label='solution théorique')
    plt.title('Tracé de la solution pour theta_0 = ' +
              str(theta_0_deg_list[i]) + 'degré')
    plt.xlabel('temps en s')
    plt.ylabel('Angle theta en degré')
    plt.legend()
plt.show()
```

Correction d'un TP

Corrigé TP 4.0

0.

```
def monint0(f, a, b, n, t0):
    delta = (b - a) / n
    S = 0
    for i in range(n):
        S += f(a + (i + t0) * delta)
    return delta * S
```

1. Nous commençons par définir trois fonctions f_0, f_1, f_2 et les valeurs I_0, I_1, I_2 de leurs intégrales sur $[0,1]$, ainsi qu'une fonction `tracer0(f, I, n)` qui tracera les graphes demandés :

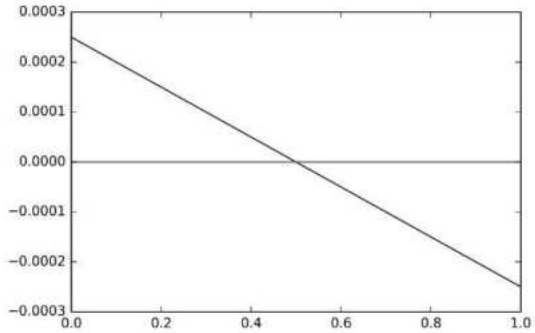
```
from math import *

def f0(t):
    return(1 / (1 + t))

def f1(t):
    return(t**6)

def f2(t):
    return sin(t)

I0, I1, I2 = log(2), 1 / 7, 1 - cos(1)
```

```
import matplotlib.pyplot as plt

def tracer0(f, I, n):
    Lt = []
    Le = []
    for i in range(101):
        Lt.append(i / 100)
        Le.append(monint0(f, 0, 1, n, i / 100) - I)
    plt.figure()
    plt.plot(Lt, Le)
    plt.plot([0, 1], [0, 0])
    plt.show()
```

L'instruction

```
for (f, I) in [(f0, I1), (f1,I1), (f2, I2)]:
    for n in [100, 500, 1000, 5000]:
        tracer0(f, I, n)
```

donne 12 graphes du type de celui représenté ci-contre, qui correspond à $f = f_0$ et $n = 1000$.



L'erreur semble s'annuler pour une valeur de t_0 voisine de $1/2$ (cela se vérifie en modifiant la fonction `tracer0` pour travailler au voisinage de $t_0 = 1/2$), et ceci pour chacune des trois fonctions testées (dès que n n'est pas trop petit).

2. Le cours nous apprend que, pour des fonctions de classe $\mathscr{C}^1$, la méthode des rectangles pointés à gauche (ou à droite) donne une erreur de l'ordre de $1/n$. Nous observons le même résultat pour $t_0 \neq 1/2$ (ici avec $t_0 = 0.3$) :

```
>>> for (f, I) in [(f0, I0), (f1, I1), (f2, I2)]:
        print([round(n * (monint0(f, 0, 1, n, 0.3) - I), 3)
               for n in [10, 20, 50, 100, 500, 1000, 5000]])
[0.098, 0.099, 0.1, 0.1, 0.1, 0.1, 0.1]
[-0.211, -0.206, -0.203, -0.201, -0.2, -0.2, -0.2]
[-0.167, -0.168, -0.168, -0.168, -0.168, -0.168, -0.168]
```

En revanche, avec $t_0 = 1/2$, l'erreur semble être, pour les trois exemples, de l'ordre de $1/n^2$. C'est effectivement le cas dès que la fonction f est de classe C^2.

```
>>> for (f, I) in [(f0, I0), (f1, I1), (f2, I2)]:
        print([round(n**2 * (monint0(f, 0, 1, n, 0.5) - I), 3)
               for n in [10, 20, 50, 100, 500, 1000, 5000]])
[-0.031, -0.031, -0.031, -0.031, -0.031, -0.031, -0.031]
[0.083, 0.083, 0.083, 0.083, 0.083, 0.083, 0.083]
[0.019, 0.019, 0.019, 0.019, 0.019, 0.019, 0.019]
```

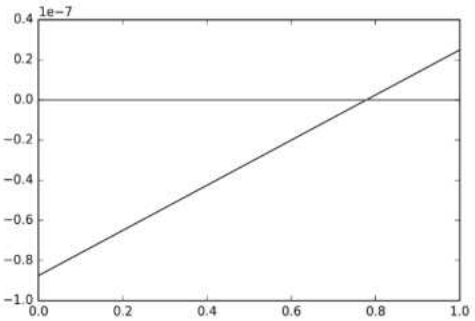
3. On obtient

```
def monint1(f, a, b, n, t0, t1):
    delta = (b - a) / n
    A0 = (t1 - 1 / 2) / (t1 - t0)
    A1 = (t0 - 1 / 2) / (t0 - t1)
    S = 0
    for i in range(n):
        S += A0 * f(a + (i + t0) * delta) + A1 * f(a + (i + t1) * delta)
    return delta * S
```

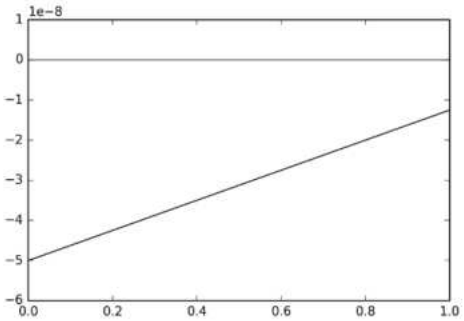
4. On reprend la même méthode qu'à la question **1.** :

```
def tracer1(f, I, t0, n):
    Lt = []
    Le = []
    for i in range(101):
        if i / 100 != t0:
            Lt.append(i / 100)
            Le.append(monint1(f, 0, 1, n, t0, i / 100) - I)
    plt.figure()
    plt.plot(Lt, Le)
    plt.plot([0, 1], [0, 0])
    plt.show()
```

Nous obtenons ainsi deux types de graphes, selon la valeur de t_0 :



`tracer1(f0, I0, 0.2, 1000)`



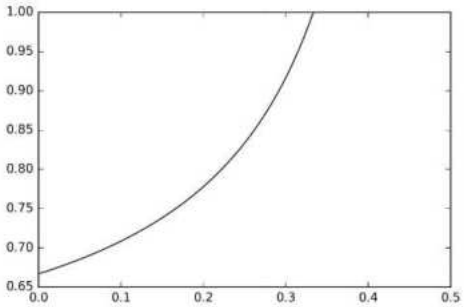
`tracer1(f0, I0, 0.4, 1000)`

5. Pour chaque courbe, le coefficient de corrélation est égal à 1 ou -1 à 10^{-7} près. Il est donc légitime de considérer que $P(t_1) = I_{t_0,t_1}(f, a, b, 10^3) - I$ est une fonction affine de la forme $P(t_1) = A + t_1 B$. Si cette fonction garde un signe constant sur $[0, 1]$, $t_1^{opt} = 1$. Sinon, t_1^{opt} sera la racine de P. En notant $a = P(0)$ et $b = P(1)$ (ceci impose d'éviter la valeur $t_0 = 0$), nous obtenons facilement :

```
def t1opt(t0, f, I, n):
    a = monint1(f, 0, 1, n, t0, 0) - I
    b = monint1(f, 0, 1, n, t0, 1) - I
    if a * b > 0:
        return 1  # l'erreur minimale est obtenue pour t0 = 1
    else:
        return a / (a - b)  # l'erreur est nulle pour cette unique valeur de [0,1]
```

On peut ensuite tracer, pour f, I, n donnés, le graphe de l'application $t_0 \longmapsto t_1^{opt}$:

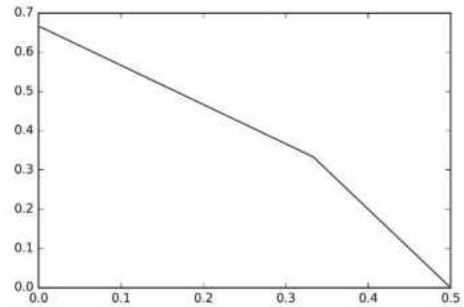
```
def tracer2(f, I, n):
    X = [i / 500 for i in range(1, 250)]
    Y = [t1opt(t0, f, I, n) for t0 in X]
    plt.figure()
    plt.plot(X, Y)
    plt.show()
```



`tracer2(f0, I0, 1000)`

6. On trace maintenant le graphe de l'application $t_0 \longmapsto (1-2t_0)t_1^{opt}$:

```
def tracer3(f, I, n):
    X = [i / 500 for i in range(1, 250)]
    Y = [(1 - 2 * t) * t1opt(t, f, I, n) for t in X]
    plt.figure()
    plt.plot(X, Y)
    plt.show()
```



`tracer3(f0, I0, 1000)`

On peut calculer α et β en utilisant par exemple les valeurs $t_0 = 0.1$ et $t_0 = 0.3$. On en déduit ensuite u en résolvant $\alpha + \beta u = 1 - 2u$:

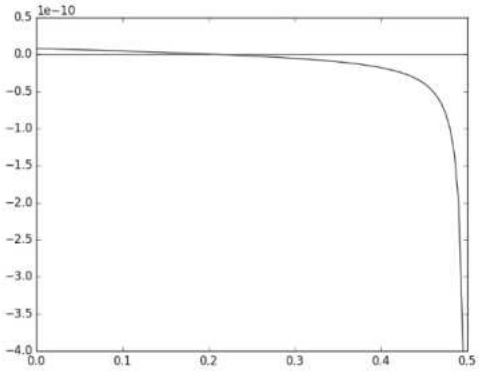
```
def alpha_beta_u(f, I, n):
    a = (1 - 2 * 0.1) * t1opt(.1, f, I, n)
    b = (1 - 2 * 0.3) * t1opt(.3, f, I, n)
    beta = (b - a) * 5
    alpha = a - beta * 0.1
    u = (1 - alpha) / (2 + beta)
    return(alpha, beta, u)
```

```
>>> for (f, I) in [(f0, I0), (f1, I1), (f2, I2)]:
        for n in [500, 1000]:
            print(alpha_beta_u(f, I, n))
(0.666925891..., -1.0005185..., 0.3332469112...)
(0.666796287..., -1.00025925..., 0.333290118...)
(0.666110673..., -0.998887999..., 0.33351845...)
(0.66638877..., -0.999444222..., 0.3334259097...)
(0.666463152..., -0.999592864..., 0.333401108...)
(0.666564944..., -0.999796529..., 0.3333672253...)
```

On peut raisonnablement conjecturer que $\alpha = \frac{2}{3}$, $\beta = -1$ et $u = \frac{1}{3}$.

7. Nous obtenons

```
def tracer4(f, I, n):
    Lt = []
    Le = []
    for i in range(100):
        t0 = i / 300
        # t0 décrit [0, 1/3[ par pas de 1/300
        t1 = (2 / 3 - t0) / (1 - 2 * t0)
        Lt.append(t0)
        Le.append(monint1(f, 0, 1, n, t0, t1) - I)
    plt.figure()
    plt.plot(Lt, Le)
    plt.plot([0, 1 / 3], [0, 0])
    plt.show()
```



`tracer4(f0, I0, 1000)`

L'erreur est strictement monotone et continue : nous pouvons approcher la valeur optimale de t_0 en utilisant la méthode de dichotomie. La fonction qui suit renvoie le couple (t_0^{opt}, t_1^{opt}) avec une précision de `eps` pour t_0^{opt}.

```
def dichotomie1(f, I, n, eps):
    x, y = 0, 0.33 # la racine est dans l'intervalle [x, y]
    if monint1(f, 0, 1, n, x, (2 / 3 - x) / (1 - 2 * x)) > I:
        s = 1 # s est le signe de l'erreur en x
    else:
        s = -1
    while(y - x > eps): # tant que l'intervalle est trop large
        t0 = (x + y) / 2
        t1 = (2 / 3 - t0) / (1 - 2 * t0)
        if (monint1(f, 0, 1, n, t0, t1) - I) * s > 0: # si l'erreur en t0 est du même signe qu'en x
            x = t0 # [x, y] est remplacé par [t0, y]
        else: # sinon
            y = t0 # [x, y] est remplacé par [x, t0]
    return x, (2 / 3 - x) / (1 - 2 * x)
```

Nous obtenons :

```
>>> for (f, I) in [(f0, I0), (f1, I1), (f2, I2)]:
        print(dichotomie1(f, I, 1000, 0.000001))
(0.21131813049316406, 0.7886683998399144)
(0.21135778427124022, 0.7887080572151738)
(0.21130491256713865, 0.7886551831357824)
```

Dès que n est suffisamment grand, on obtient des valeurs qui ne semblent pas dépendre de la fonction f et dont la somme est presque égale à 1. Il est donc naturel de conjecturer que la meilleure valeur t_0 est solution de l'équation $\dfrac{2/3 - t_0}{1 - 2t_0} = t_1 = 1 - t_0$, i.e. que t_0^{opt} et t_1^{opt} sont les deux racines du polynôme $L_2 = X^2 - X + 1/6$.

8. On remarque cette fois-ci que l'erreur semble être de l'ordre de $1/n^4$, puisque le produit de $I_{t_0,t_1}(f, 0, 1, n) - I$ par n^4 est à peu près constant (il ne faut pas choisir n trop grand, car la faible précision des calculs numériques ne permettrait plus de mesurer l'erreur « de méthode »).

```
>>> t0, t1 = 1/2-sqrt(3)/6, 1/2+sqrt(3)/6
>>> for (f, I) in [(f0, I0), (f1, I1), (f2, I2)]:
        print([round(n**4 * (monint1(f, 0, 1, n, t0, t1) - I),7)
            for n in [10, 20, 50, 100, 200]])
```

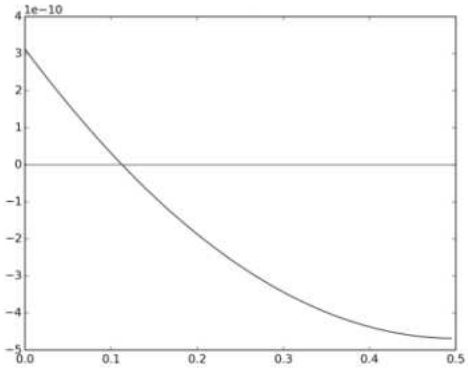
```
[-0.0012935, -0.0012999, -0.0013017, -0.001302, -0.0013019]
[-0.0277249, -0.0277646, -0.0277757, -0.0277772, -0.0277776]
[-0.0001064, -0.0001064, -0.0001064, -0.0001064, -0.000106]
```

9. Nous obtenons

```
def monint2(f, a, b, n, t0, t1, t2):
    delta = (b - a) / n
    A0 = (1 / 6) * (6 * t1 * t2 - 3 * t1 - 3 * t2 + 2) / ((t0 - t2) * (t0 - t1))
    A1 = (1 / 6) * (6 * t0 * t2 - 3 * t0 - 3 * t2 + 2) / ((t1 - t2) * (t1 - t0))
    A2 = (1 / 6) * (6 * t1 * t0 - 3 * t1 - 3 * t0 + 2) / ((t2 - t0) * (t2 - t1))
    S = 0
    for i in range(n):
        S += A0 * f(a + (i + t0) * delta) + A1 * f(a + (i + t1)
                                                   * delta) + A2 * f(a + (i + t2) * delta)
    return delta * S
```

10. Nous reprenons la même démarche qu'à la question **7.** :

```
def tracer5(f, I, n):
    Lt = []
    Le = []
    for i in range(100):
        t0 = i / 200
        # t0 décrit [0, 1/2[ par pas de 1/200
        Lt.append(t0)
        Le.append(monint2(f, 0, 1, n, t0, .5, 1-t0) - I)
    plt.figure()
    plt.plot(Lt, Le)
    plt.plot([0, 1 / 2], [0, 0])
    plt.show()
```



`tracer5(f0, I0, 100)`

```
def dichotomie2(f, I, n, eps):
    x, y = 0, 0.4
    if monint2(f, 0, 1, n, 0, .5, 1) > I:
        s = 1
    else:
        s = -1
    while(y - x > eps):
        t0 = (x + y) / 2
        if (monint2(f, 0, 1, n, t0, 0.5, 1 - t0) - I) * s > 0:
            x = t0
        else:
            y = t0
    return x
```

Nous obtenons :

```
>>> [dichotomie2(f, I, 100, 0.00000001) for (f, I) in [(f0, I0), (f1, I1), (f2, I2)]]
[0.1126992404460907, 0.11270157098770141, 0.11269734501838685]
>>> alpha = dichotomie2(f0, I0, 100, 0.00000001)
>>> alpha*(1-alpha)
0.09999812164896493
```

Ainsi, $(X-\alpha)(X-1+\alpha)$ est environ égal à $X^2-X+\frac{1}{10}$. Nous pouvons donc conjecturer que les valeurs optimales de t_0, t_1, t_2 sont les racines du polynôme $L_3 = (X-1/2)(X^2-X+1/10)$.

11. On remarque cette fois-ci que l'erreur semble être de l'ordre de $1/n^4$ pour la méthode de Simpson, alors qu'elle semble être de l'ordre de $1/n^6$ pour les valeurs optimales de t_0, t_1 et t_2 :

```
>>> for (f, I) in [(f0, I0), (f1, I1), (f2, I2)]: # Méthode de Newton
          print([round(n**4 * (monint1(f, 0, 1, n, t0, t1) - I),7)
              for n in [10, 20, 50, 100, 200]])
[0.0019411, 0.0019501, 0.0019526, 0.001953, 0.0019527]
[0.0415923, 0.0416481, 0.0416637, 0.0416659, 0.0416665]
[0.0001597, 0.0001596, 0.0001596, 0.0001596, 0.0001597]
>>> t0, t1 = 1 / 2 - sqrt(15) / 10, .5, 1 / 2 + sqrt(15) / 10 # Meilleurs points
>>> for (f, I) in [(f0, I0), (f1, I1), (f2, I2)]:
          print([round(n**6 * (monint1(f, 0, 1, n, t0, t1) - I), 10)
              for n in [5, 10, 15, 20, 30]])
[-5.54436e-05, -5.77626e-05, -5.82228e-05, -5.83853e-05, -5.84385e-05]
[-0.0003571429, -0.0003571428, -0.0003571433, -0.0003571419, -0.0003571497]
[2.283e-07, 2.28e-07, 2.27e-07, 2.238e-07, 2.033e-07]
```

On voit que l'analyse d'erreur est ici délicate car, pour $n \geqslant 50$, nous arrivons aux limites de la précision des calculs flottants. Il est possible de démontrer que l'erreur est effectivement en $\mathscr{O}(1/n^6)$ quand f est de classe $\mathscr{C}^6$ sur $I = [0, 1]$.
Plus généralement, pour tout $d \geqslant 0$, il existe un unique polynôme normalisé L_{d+1} de degré $d+1$, appelé **polynôme de Legendre**, tel que :

$$\forall P \in \mathbb{R}_d[X],\ \int_0^1 L_{d+1}(t)P(t)\,\mathrm{d}t = 0.$$

On montre que L_{d+1} possède $d+1$ racines réelles distinctes et strictement comprises entre 0 et 1, notées $(t_i)_{0\leqslant i\leqslant d}$. Pour toute fonction f de classe $\mathscr{C}^{2d+2}$ sur un segment $[a, b]$, $I_{t_0,\ldots,t_d}(f, a, b, n)$ approxime $\displaystyle\int_a^b f(t)\,\mathrm{d}t$ avec une erreur en $\mathscr{O}\left(\frac{1}{n^{2d+2}}\right)$.

CHAPITRE

5

Calcul numérique (deux dimensions)

L'essentiel du cours

■ 0 Résolution de systèmes d'équations linéaires

Nous supposons connu du lecteur le principe général de la résolution de systèmes d'équations linéaires à l'aide de l'algorithme **Fang cheng**[1] aussi connu sous le nom de **pivot de Gauss**. Au besoin, prière de se référer à votre cours de Mathématiques.

Description de l'algorithme

Le problème est de résoudre le système d'équations linéaires suivant :

$$\begin{cases} a_{0,0} & x_0 & + & a_{0,1} & x_1 & + & \cdots & + & a_{0,n-1} & x_{n-1} & = & y_0 \\ a_{1,0} & x_0 & + & a_{1,1} & x_1 & + & \cdots & + & a_{1,n-1} & x_{n-1} & = & y_1 \\ \vdots & & & & & & & & & & & \vdots \\ a_{n-1,0} & x_0 & + & a_{n-1,1} & x_1 & + & \cdots & + & a_{n-1,n-1} & x_{n-1} & = & y_{n-1} \end{cases}$$

que l'on peut écrire $AX = Y$ avec $A = (a_{i,j})_{0 \leqslant i,j \leqslant n-1}$, $X = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix}$ et $Y = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}$.

On suppose dans ce chapitre la matrice A inversible ; cette équation a donc une unique solution $X = A^{-1}Y$.

Nous allons transformer cette matrice A en la matrice identité à l'aide d'opérations élémentaires sur les lignes (échange, transvection, dilatation). Les opérations effectuées sur A seront également effectuées sur Y, ainsi, Y sera transformée en $A^{-1}Y$ et le système sera résolu. Si on veut calculer l'inverse de A, il suffit de prendre pour Y la matrice identité.

1. La plus ancienne référence à cet algorithme est dans *Les Neuf Chapitres sur l'art mathématique* (九章算術) dont on peut trouver une traduction chez Dunod [**KC05**].

Le principe de l'algorithme est de normaliser les colonnes une par une, c'est-à-dire de les transformer en colonnes avec un « un » sur la diagonale et des « zéros » sur les autres coefficients. Ainsi, après $j \in [\![0, n-2]\!]$ étapes, la matrice A est de la forme suivante :

$$M = \begin{pmatrix} 1 & 0 & 0 & a_{0,3} & a_{0,4} & \cdots \\ 0 & 1 & 0 & a_{1,3} & a_{1,4} & \cdots \\ 0 & 0 & \ddots & & & \\ 0 & 0 & 0 & a_{j,j} & a_{j,j+1} & \cdots \\ 0 & 0 & 0 & a_{j+1,j} & a_{j+1,j+1} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \\ 0 & 0 & 0 & a_{n-1,j} & a_{n-1,j+1} & \cdots \end{pmatrix}$$

Pour un j fixé, nous allons réaliser les opérations suivantes, qui n'interviendront que sur les lignes (on note L_i les lignes de A et Y_i celles de Y) :

- On trouve la ligne p de la matrice pour laquelle $|a_{p,j}|$ avec $j \leqslant p \leqslant n-1$ est maximal. L'élément $a_{p,j}$ servira de pivot. Il est choisi pour minimiser les erreurs de calcul flottantes.
- On échange les lignes p et j dans A et dans Y.
- On divise (dilatation) la j^e ligne de A et de J par $a_{p,j}$
- Pour $i \in [\![0, n-1]\!] \setminus \{j\}$ (i va de 0 à $n-1$ en sautant la valeur j), on effectue les opérations élémentaires $L_p \leftarrow L_p - \dfrac{a_{p,k}}{a_{k,k}} L_k$ et $Y_p \leftarrow Y_p - \dfrac{a_{p,k}}{a_{k,k}} Y_k$. Ces opérations sont appelées transvections.

Étude de la complexité temporelle de l'algorithme du pivot de Gauss partiel

Proposition

L'algorithme de Gauss a une complexité temporelle (dans le pire et dans le meilleur des cas) en $\mathcal{O}(n^3)$.

La boucle externe est répétée $n-2 = \mathcal{O}(n)$ fois. À chaque étape j, la recherche de pivot coûte $\mathcal{O}(n)$ opérations, tout comme les échanges de lignes et les dilatations. Chaque transvection coûte $\mathcal{O}(n)$ opérations et est répétée $\mathcal{O}(n)$ fois. Les transvections sont donc en $\mathcal{O}(n^3)$ et même en $\Theta(n^3)$. La complexité totale de cet algorithme est donc en $\mathcal{O}(n^3)$.

Code Python utilisant les listes de listes

```
def recherche_pivot(A, i):
    p = i  # p est l'indice du pivot
    for k in range(i + 1, len(A)):
        if abs(A[k][i]) > abs(A[p][i]):  # Si on trouve un meilleur pivot
            p = k
    return p


def echange_ligne(A, i, j):
    for k in range(len(A[0])):
        A[i][k], A[j][k] = A[j][k], A[i][k]


def transvection(A, i, j, mu):
    for k in range(len(A[0])):
        A[i][k] += mu * A[j][k]


def dilatation(A, i, mu):
```

```
    for k in range(len(A[0])):
        A[i][k] *= mu


def fang_sheng(A, Y):
    n = len(A)
    for j in range(n):
        p = recherche_pivot(A, j)
        echange_ligne(A, j, p)
        echange_ligne(Y, j, p)
        dilatation(Y, j, 1 / A[j][j])
                    # Y est modifié avant que A[j][i] ne soit modifié
        dilatation(A, j, 1 / A[j][j])
        for i in range(n):
            if i != j:
                transvection(Y, j, i, -A[j][i])
                transvection(A, j, i, -A[j][i])
```

■ 1 Numpy (tableau `array`)

Les tableaux (`array`) sont les objets de base du module `Numpy`. Ils sont composés de cellules toutes de **même type** (`dtype`, `np.float64` par défaut). Ils sont multi-dimensionnels (plusieurs indices possibles). On manipulera en général des vecteurs (unidimensionnel), des « matrices » (bidimensionnel) mais on peut aussi avoir des tableaux tri-dimensionnels, c'est le cas par exemple des images (chaque pixel `[i,j]` est composé d'une vecteur à trois coordonnées `[R, G, B]` de type `np.uint` c'est-à-dire un octet non signé – valeurs possibles de 0 à 255).
Ces tableaux prennent moins de place en mémoire et sont plus rapides d'accès que les listes Python. On les utilise donc pour les calculs numériques, en imagerie, en traitement du signal, etc.

On définit un tableau de zéros par la fonction `np.zeros` et on convertit des listes par `np.array`. On accède à la cellule d'indice `(i, j)` du tableau `a` par `a[i, j]` ou `a[i][j]`.

```
import numpy as np   # alias np plus court

a = np.zeros((3, 3))
b = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]],
             dtype=int)

a[0, 0] = 20
a[2][1] = -1
print(a)
print(a.shape, a.dtype, a.size, b.dtype, sep=", ")
```

```
[[ 20.   0.   0.]
 [  0.   0.   0.]
 [  0.  -1.   0.]]
(3, 3), float64, 9, int64
```

La méthode `shape` donne la taille (forme) du tableau, à ne pas confondre avec la méthode `size` qui fournit le nombre total de cellules élémentaires (moins utiles).

```
a = np.array([4, 7, 9])
b = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]],
             dtype=int)
print(a.size, a.shape)
print(b.size, b.shape)
```

```
3 (3,)
9 (3, 3)
```

On peut remplir un tableau de différentes façons, par exemple

```
def f(i, j):
    return 5 / (i + j + 1)

H1 = np.array([[f(i, j) for j in range(5)] for i in range(5)])
H2 = np.fromfunction(f, (5, 5))  # un peu moins de manipulations mémoire
```

On peut changer le type des cellules d'un tableau. Remarquer la méthode `copy` qui copie tout le tableau.

```
Hentier = H2.copy()
Hentier = Hentier.astype(int)
print(H2.dtype)
print(H2)
print(Hentier)
```

```
float64
[[ 5.          2.5         1.66666667  1.25        1.        ]
 [ 2.5         1.66666667  1.25        1.          0.83333333]
 [ 1.66666667  1.25        1.          0.83333333  0.71428571]
 [ 1.25        1.          0.83333333  0.71428571  0.625     ]
 [ 1.          0.83333333  0.71428571  0.625       0.55555556]]
[[5 2 1 1 1]
 [2 1 1 1 0]
 [1 1 1 0 0]
 [1 1 0 0 0]
 [1 0 0 0 0]]
```

Mise en garde Examinons la sortie du code suivant :

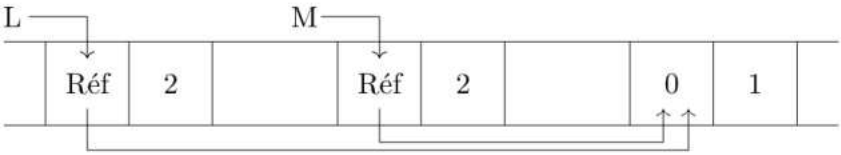
```
B = [[0] * 5] * 5
B[2][1] = 2017
print(B)
C = np.zeros((5, 5), dtype=int)
C[2, 1] = 2016
print(C)
```

```
[[0, 2016, 0, 0, 0],
 [0, 2016, 0, 0, 0],
 [0, 2016, 0, 0, 0],
 [0, 2016, 0, 0, 0],
 [0, 2016, 0, 0, 0]]
[[   0    0    0    0    0]
 [   0    0    0    0    0]
 [   0 2016    0    0    0]
 [   0    0    0    0    0]
 [   0    0    0    0    0]]
```

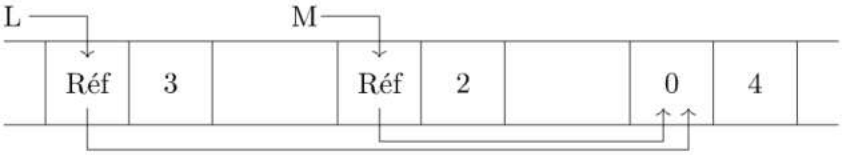
Cela s'explique en se souvenant que les listes Python sont des listes d'*étiquettes* sur des objets. Les listes de listes sont donc des objets assez complexes. On n'a pas ce problème avec les tableaux `arrray` de `numpy`.

`L=[[0,1],2]`
Création de L

`M=L[:]`
Le contenu de L est copié




```
L[1]=3
L[0][1]=4
print(M):[[0,4],2]
```



Pour résoudre le problème précédent avec des listes, il faut plutôt écrire :

```
B = [[0] * 5 for i in range(5)]
B[2][1] = 2017
print(B)
```

```
[[0, 0, 0, 0, 0],
[0, 0, 0, 0, 0],
[0, 2016, 0, 0, 0],
[0, 0, 0, 0, 0],
[0, 0, 0, 0, 0]]
```

Méthode 5.0 : Copier un tableau

On retiendra que pour copier un tableau `A`, on peut utiliser `A.copy()` mais que si on a affaire à une liste de liste `L`, il nous faut copier *en profondeur* la liste (descendre récursivement les étiquettes), on utilisera alors la fonction `deepcopy` du module copy.

```
A2 = A.copy    # A est un array

from copy import deepcopy
L2 = deepcopy(L)   # L est une liste de liste
```

Expressions algébriques

On peut utiliser les opérateurs `+` et `*` avec les tableaux `array`. Cela rend le code très lisible par exemple pour l'implémentation de la méthode d'Euler (équations différentielles).

```
A = np.random.randint(1, 10, (3, 3))  # coefficients de 1 à 9
B = np.eye(3, 3)  # matrice identité

print(A + 10 * B)
```

```
[[ 11.   7.   4.]
 [  5.  11.   5.]
 [  8.   2.  14.]]
```

Comparer le comportement des opérateurs avec les listes.

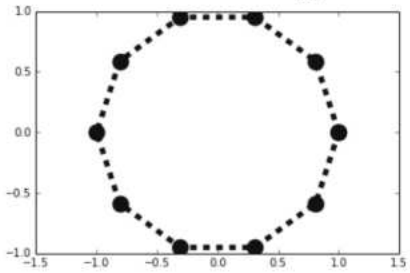
```
A, B = [-1, -2], [10, 20]

print(A + 5 * B)
```

```
[-1, -2, 10, 20, 10, 20, 10, 20, 10, 20, 10, 20]
```

Fonction vectorisée

Les fonctions mathématiques usuelles ont été vectorisées dans le module `numpy`

```
T = np.array([np.pi * i / 5 for i in range(11)])
# cos et sin ont été vectorisées
X, Y = np.cos(T), np.sin(T)
plt.axis('equal')
plt.plot(X, Y, 'o--', lw=5, markersize=15)
plt.show()
```



On peut vectoriser sa propre fonction avec la fonction `np.vectorize` (si elle n'utilise pas directement les fonctions usuelles).

```
def f(x):
    if x > 5:
        return x**2
    else:
        return -x

f = np.vectorize(f)
T = np.arange(1, 10)
f(T)
```

```
array([-1, -2, -3, -4, -5, 36, 49, 64, 81])
```

Utilisation du *slicing*

Très pratique, le *slicing* permet de gagner en lisibilité du code (et en rapidité d'exécution).

```
print(b)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

```
print(b[:, 1:3])
```

```
[[2 3]
 [5 6]
 [8 9]]
```

```
print(b[:2, 1:2])
```

```
[[2]
 [5]]
```

```
print(b[:2, 1])
```

```
[2 5]
```

```
b[:, 1] = 2017
print(b)
```

```
[[   1 2017    3]
 [   4 2017    6]
 [   7 2017    9]]
```

Remodelage

```
t = np.array([[1, 2, 3], [4, 5, 6]])
v = t.flatten()   # il aplatit en faisant une copie
print(t)
print(v)
```

```
[[1 2 3]
 [4 5 6]]
[1 2 3 4 5 6]
```

```
t2 = t.reshape((3, 2))
print(t2)
t[0, 0] = 2014
print(t2)  # attention, pas une copie
# t2 pointe sur la même vue !!
```

```
[[1 2]
 [3 4]
 [5 6]]
[[2014    2]
 [   3    4]
 [   5    6]]
```

```
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])
print(a)
print(b)
```

```
[[1 2]
 [3 4]]
[[5 6]
 [7 8]]
```

Concaténation

```
c = np.concatenate((a, b))  # axis=0 sous-entendu
print(c)
d = np.concatenate((a, b), axis=1)
a[0, 0] = 2017
print(d)
```

```
[[1 2]
 [3 4]
 [5 6]
 [7 8]]
[[1 2 5 6]
 [3 4 7 8]]
```

Indexation booléenne (*fancy indexing*)

Sans rentrer dans les détails subtils de la syntaxe de `numpy`, les manipulations suivantes sont très agréables notamment en traitement d'images.

```
C = np.random.randint(-9, 10, (5, 5))  # se souvenir de np.random
test = C > 0
print(test)
print("nombre de cellules concernées:", test.sum())
```

```
[[False  True False False False]
 [ True False False False False]
 [ True  True  True  True False]
 [False False  True  True False]
 [False  True  True False False]]
nombre de cellules concernées: 10
```

```
print(C)
C[C > 0] = 100
print(C)
```

```
[[-3  7 -5 -8 -9]
 [ 4 -2 -1 -9 -4]
 [ 1  2  2  5 -1]
 [-5 -3  9  2 -4]
 [-3  9  2 -7 -2]]
[[ -3 100  -5  -8  -9]
 [100  -2  -1  -9  -4]
 [100 100 100 100  -1]
 [ -5  -3 100 100  -4]
 [ -3 100 100  -7  -2]]
```

Les méthodes à maîtriser

Méthode 5.1 : numpy (résumé)

`np.array([1, 5, 7])`
`np.arange(deb, fin, incr)`
`np.linspace(deb, fin, nbpoints)`
	tableaux (`array`) de subdivision

`np.vectorize(f)`	permet d'utiliser la fonction scalaire `f` avec des `array`
`M = np.array([[1.5, 2.1],` `[3.7, 4.9]])`	matrice 2×2 (type `ndarray`)
`M = np.array([[1, 2], [3, 4]],` `dtype=int)`	matrice 2×2 (cellule de type int)
`dimi, dimj = M.shape`	renvoie la taille de la matrice (attention `.size`
`M.reshape(4)`	renvoie le nombre total de cellules
`M = np.zeros((a, b))`	création d'une matrice de zéros de taille `(a, b)`
`M[0, :]`	1^{re} ligne de `M`
`M[:, 1]`	2^{e} colonne de `M`
`N = M.copy()`	copie de `M` (si les coefficients sont des nombres)
`M = np.concatenate([L1, L2])`	concatène les lignes (attention une seule entrée)
`M = np.concatenate([C1, C2, C3],` `axis=1)`	concatène les colonnes

Méthode 5.2 : numpy et algèbre linéaire

`M * M`	le produit terme à terme
`np.dot(M, N)  # ou M.dot(N)`	le **vrai** produit matriciel $M\dot{N}$
`M.T`	la transposée de `M`
`M.sum()`	somme de tous les éléments de `M`
`v.T * w  # ou np.vdot(v, w)`	produit scalaire $(u\|v)$
`np.cross(v, w)`	produit vectoriel $u \wedge v$
`np.linalg.solve(A, V)`	résolution d'un système linéaire $AX = V$
`import numpy.linalg as lg`	
`lg.det(M)`	déterminant
`lg.inv(M)`	inverse de la matrice
`valeurspropres, P = lg.eig(M)`	valeurs propres et matrice de passage (diag.)
`valeurspropres = lg.eigvals(M)`	spectre de la matrice

Méthode 5.3 : numpy et probabilités

Les fonctions de `numpy.random` décrites dans le chapitre 1 section 8 (page 23) peuvent renvoyer des tableaux de valeurs aléatoires.

```
import numpy.random as rd
# Renvoie un tableau de 5 entiers compris entre 1 et 6 inclus.
rd.randint(1, 7, 5)
# Renvoie un tableau de 5 lignes et 2 colonnes de flottants aléatoires


# compris entre 0 et 1.
rd.random((5, 2))
rd.binomial(n, p, nbtirages)  # Renvoie nbtirages valeurs aléatoires.
```

Méthode 5.4 : lignes de niveaux de fonction à deux variables

Pour afficher des lignes de niveau de la surface $z = \sin(x)\sin(y)$, on peut éventuellement utiliser l'option `levels=[...]`.

```
import numpy as np
import matplotlib.pyplot as plt
X = np.linspace(-np.pi, np.pi, 101)
Y = np.linspace(-np.pi, np.pi, 101)
XX, YY = np.meshgrid(X, Y)
Z = np.sin(XX) * np.sin(YY)

plt.contour(XX, YY, Z)
plt.show()
```

À connaître, pour tracer la surface représentative d'une fonction, c'est-à-dire la surface d'équation $z = \sin(xy)$.

```
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")
ax.plot_surface(X, Y, Z)
plt.show()
```

Exercices

Exercice 5.0 Remplir un tableau

On se déplace sur une grille $(n+1)\times(n+1)$ en partant du coin en haut à gauche (indice `[0, 0]`).
On n'a le droit de se déplacer que d'une case en allant soit à **droite**, soit en **bas**.
Construire un tableau $(n+1)\times(n+1)$ à l'aide d'un algorithme qui calcule de proche en proche pour chaque case `[i, j]`, le nombre de chemins pour aller de la case `[0, 0]` à `[i, j]`.
Que remarque-t-on ?

Exercice 5.1 Suites récurrentes croisées

On considère trois suites réelles (x_n), (y_n), (z_n) vérifiant

$$x_0 = y_0 = \frac{1}{2} \text{ et } z_0 = 0$$

0. On suppose que

$$\begin{cases} 10x_{n+1} = 7x_n + 4y_n + 5z_n \\ 10y_{n+1} = 3x_n + z_n \\ 10z_{n+1} = 6y_n + 4z_n \end{cases}$$

À l'aide d'un programme en `Python`, calculer $(x_{2018}, y_{2018}, z_{2018})$ par un calcul matriciel.

1. Peut-on réduire le nombre de multiplications dans le calcul matriciel ?

2. On reprend le même problème (même condition initiale) mais avec le système

$$\begin{cases} x_{n+1} = 7x_n + 4y_n + 5z_n \\ y_{n+1} = 3x_n + z_n \\ z_n = 6y_n + 4z_n \end{cases}$$

Calculer $(x_{2018}, y_{2018}, z_{2018})$.

Exercice 5.2 Décomposition LU

On considère une matrice inversible $A = (a_{i,j}) \in \mathfrak{M}_n(\mathbb{R})$.
On effectue des transformations sur la matrice A.
On notera les matrices $A^{(0)} = A$, $A^{(1)}, \cdots, A^{(k)} = (a_{i,j}^{(k)}), \cdots, A^{(n-1)}$.

- Pour k variant de 1 à $n-1$,

  pour $i \in [\![k+1, n]\!]$, on pose $\ell_{i,k} = \dfrac{a_{i,k}^{(k-1)}}{a_{k,k}^{(k-1)}}$ (on suppose qu'à chaque étape $a_{k,k}^{(k-1)} \neq 0$) et on effectue les transformations sur la matrice $A^{(k-1)}$

  $$L_i \leftarrow L_i - \ell_{i,k} L_k$$

  pour obtenir la matrice $A^{(k)}$.
- On pose alors $U = A^{(n-1)}$ et $L = (\widetilde{\ell}_{i,j})$ avec

$$\widetilde{\ell}_{i,j} = \begin{cases} \ell_{i,j} \text{ si } i > j \\ 1 \text{ si } i = j \\ 0 \text{ sinon} \end{cases}$$

On démontre que $A = LU$ (on dit que l'on a effectué une décomposition LU de la matrice A).

0. Écrire une fonction `decompLU(A)` qui effectue cette décomposition et qui renvoie les matrices L et U.

1. Tester votre fonction sur la matrice $A = \begin{pmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{pmatrix}$

2. Comment résoudre simplement une équation du type $AX = Y$ avec $X, Y \in (\mathfrak{M}_{n,1}(\mathbb{R}))^2$, X inconnu, lorsqu'on connaît une décomposition LU de la matrice A ?

Exercice 5.3 Méthode de la puissance (algèbre linéaire)

On définit la matrice $A = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 1 \\ 3 & 1 & 0 \end{pmatrix}$.

0. Vérifier que cette matrice possède trois valeurs propres distinctes.
Déterminer une base de vecteurs propres approchés.

1. On pose $X = {}^t(1,1,1)$ et $U_n = \dfrac{1}{\|A^n X\|} A^n X$.
On montre que pour n assez grand, le vecteur U_n est proche de $\pm U$ où U est vecteur propre de la plus grande valeur propre en valeur absolue de U. Ce résultat est vrai pour la plupart des vecteurs X non nuls et pour toute matrice à valeurs propres de valeurs absolues toutes distinctes [1].
Écrire en Python, la fonction `vecteurpuiss(A, X, eps)` qui calcule une valeur approchée de U en calculant les valeurs successives de U_n et en s'arrêtant lorsque $\|U_n - U\| \leqslant \varepsilon$ ou $\|U_n + U\| \leqslant \varepsilon$ et renvoie le vecteur approché U_n.

2. Vérifier qu'on obtient bien un vecteur propre.

3. On pose $B = (I_3 - U\,{}^tU)\,A\,(I_3 - U\,{}^tU)$. Déterminer un vecteur propre de la matrice B par la méthode précédente. Que remarque-t-on ?

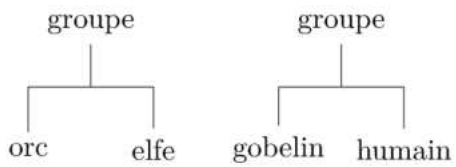
Exercice 5.4 Arbre philogénétique

$$D = \begin{pmatrix} 0 & 3 & 2 & 3 \\ 3 & 0 & 5 & 2 \\ 2 & 5 & 0 & 3 \\ 3 & 2 & 3 & 0 \end{pmatrix}$$

(0.A) Matrice des distances

$$\begin{pmatrix} & & & \\ 3 & & & \\ 2 & 5 & & \\ 3 & 2 & 3 & \end{pmatrix}$$

(0.B) Demi-matrice des distances



(0.C) Exemple d'arbre

Nous souhaitons retrouver l'arbre philogénétique d'une liste d'espèces, par exemple : `["gobelin", "orc", "humain", "elfe"]`.
Les distances entre les espèces sont supposées avoir été précalculées (e.g., selon la méthode décrite à l'exercice 3.9). Ces distances sont stockées dans une matrice D (voir par exemple la figure 0.A). $D[i,j]$ est la distance entre l'espèce i et l'espèce j (par exemple $D[0,3]$ est la distance entre les elfes et les gobelins).

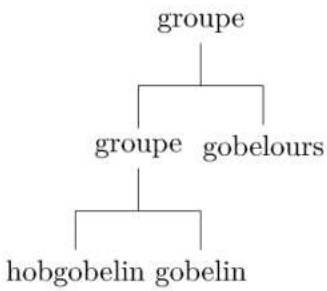
1. On peut prendre des hypothèses plus générales, mais avec ces hypothèses, un élève de MP/PSI/PC/PT peut être capable de le démontrer avec son cours de mathématiques...

D est symétrique de diagonale nulle, il est donc inutile de la stocker en entier, le triangle inférieur (hors diagonale) suffit (voir figure 0.B).
Nous représenterons cette demi-matrice par une liste de listes `D`, telle que `D[i][j]` n'est défini que si `j < i`. Par exemple, ici on a `D = [[], [3], [2, 5], [3, 2, 3]]`.
Nous représenterons un arbre philogénétique partiel par une liste. Par exemple, l'arbre partiel de la figure 0.C peut être représenté par `[['gobelin', 0, 5], ['orc', 1, 4], ['humain', 2, 5], ['elfe', 3, 4], ['groupe', 4, None], ['groupe', 5, None]]`.
L'arbre est représenté par une liste de groupes philogénétiques. Chaque groupe philogénétique est représenté par une liste de 3 éléments :

- son nom (une chaîne de caractères, c'est le nom de l'espèce ou `groupe` si c'est un regroupement d'espèces),
- sa position dans la liste,
- la position de son ancêtre dans la liste (ou `None` s'il n'en n'a pas).

On remarque que la représentation n'est pas unique, car l'ordre des groupes philogénétiques peut être changé.

0. Donner une représentation en Python de l'arbre suivant.



1. Écrire une fonction `nb_fils(A, i)` qui, étant donné une liste `A` représentant un arbre philogénétique, renvoie le nombre de descendants du groupe numéro `i` dans l'arbre.
2. Écrire une fonction `orphelins(A)` qui prend en entrée un arbre philogénétique partiel et renvoie la liste des indices des groupes sans ancêtres (dans l'ordre croissant). Sur le premier exemple, cette fonction renvoie `[4, 5]`.
3. Écrire une fonction `nouvel_ancetre(A, i, j)` qui, étant donné un arbre `A` représenté par une liste et deux numéros de groupes `i` et `j` supposés sans ancêtre dans l'arbre, ajoute un nouveau groupe (sans ancêtre) ayant pour descendants les groupes `i` et `j`.

On considère à présent que la distance entre deux groupes est le minimum des distances entre les membres des groupes. Ainsi, la distance entre le groupe « humain-gobelin » et le groupe « elfe-orc » est de 3 (le minimum entre 3 et 5). De même, la distance entre le groupe des humains et le groupe « elfe-orc » est de 3.

4. Écrire une fonction `distance(D, i, j)` qui étant donné deux groupes `i` et `j` et une demi-matrice des distances `D` renvoie une liste `L` telle que `L[k]` est la distance entre le groupe `k` et le groupe $\{i, j\}$. On remarque que `L[i]` et `L[j]` valent tous les deux zéros.
5. Écrire une fonction `plus_proche(A, D)` qui étant donné un arbre partiel `A` et une demi-matrice des distances `D` renvoie les deux groupes orphelins les plus proches [1].
6. Écrire une fonction `regroupement(A, D)` qui étant donné un arbre partiel `A` et une demi-matrice des distances `D` rajoute à la fin de la liste `A` un nouveau groupe (ancêtre des deux

1. On supposera qu'il existe au moins deux groupes orphelins dans l'arbre philogénétique, et, en cas d'égalité, on renverra arbitrairement un des couples les plus proches.

groupes orphelins les plus proches) et qui rajoute une ligne dans la matrice `D` représentant les distances entre ce nouveau groupe et les anciens groupes.

7. Écrire une fonction `arbre_philogenetique(L, D)` qui étant donné une liste d'espèces `L` et la demi-matrice des distances correspondante `D` calcule et renvoie l'arbre philogénétique `A` des espèces calculé selon la méthode suivante[1] :

- on prend en entrée la liste `L` des espèces ainsi que la demi-matrice des distances associée,
- on construit l'arbre partiel `A` avec toutes les espèces de `L` (qui sont orphelines au début),
- tant qu'il reste au moins deux groupes orphelins, on utilise la fonction `regroupement` pour créer un nouveau groupe,
- on renvoie `A`.

Exercice 5.5 Méthode du gradient conjugué

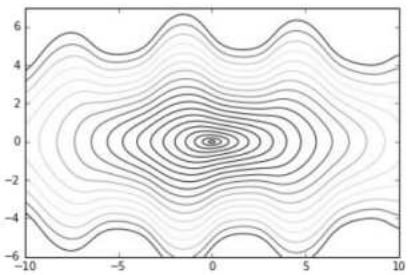
On définit la fonction f de la manière suivante :

$$f(x,y) = \frac{x^2}{2} + \frac{7}{2}y^2\,(1 + 0.4 \times \arctan(\sin(x)))\,.$$

Tracé des lignes de niveau

On rappelle ici la technique pour tracer des lignes de niveaux d'une fonction à deux variables. Voici comment tracer ces lignes niveaux pour $(x,y) \in [-10,10] \times [-6,7]$.

```
X = np.linspace(-10, 10, 1001)
Y = np.linspace(-6, 7, 1001)
XX, YY = np.meshgrid(X, Y)
ZZ = f(XX, YY)

plt.contour(XX, YY, ZZ, levels=[(k / 4)**3 for k in range(20)])
plt.show()
```



0. Définir un gradient unitaire (= de norme 1), `grad(f, x, y, h)` de la fonction `f` à deux variables au point `(x, y)` avec un déplacement de longueur `h` (petit).

1. Cette méthode s'appelle *classification hiérarchique ascendante.*

Méthode du gradient à pas constant / à pas optimal

Méthode du gradient à pas constant

1. On cherche à trouver le minimum de la fonction f.
   Pour cela, on propose avec un pas constant α de se déplacer à partir d'un point de coordonnées (x_0, y_0) de $-\alpha \times \text{grad}(f)(x_0, y_0)$. On « remonte » ainsi le gradient par une ligne polygonale où chaque segment a pour longueur α.
   Tracer une ligne polygonale de 30 points pour la fonction f avec $(x_0, y_0) = (3, 5)$, $\alpha = 2$ et $h = 10^{-3}$.

Méthode du gradient à pas optimal

On va adapter à chaque étape la longueur α.
Notons $(gx, gy) = \text{grad}(f)(x_0, y_0)$. On parcourt $\alpha \mapsto f(x - \alpha \times gx, y - \alpha \times gy)$ pour $\alpha > 0$ qui décroît au début et on prend pour α le premier minimum local rencontré. Puis on réitère le procédé à partir du nouveau point obtenu.

2. Écrire la fonction `meilleuralpha(f, x, y, gx, gy, eps)` qui récupère cette valeur en parcourant la fonction en discrétisant α d'un pas `eps`.
3. Tracer la ligne polygonale pour la même fonction f (même point de départ) en utilisant la méthode proposée. On prendra `eps = 1e-2`.
   On tracera sur la même figure les lignes de niveau de la fonction f et les deux lignes polygonales correspondant aux deux méthodes de recherche de minimum.

Exercice 5.6 Algorithme de Floyd-Steinberg

Il vous est conseillé d'avoir traité les questions 1 à 4 du T.P. **5.1** d'imagerie p. 208

Supposons que chaque pixel n'est plus codé par un triplet $(r, g, b) \in [\![0, 255]\!]^3$ mais seulement par $(r, g, b) \in \{0, 255\}^3$ c'est-à-dire que chaque pixel ne dispose que de 8 couleurs.

On définit un seuil s pour chaque pixel.
Si $c_{i,j} \geqslant s$ alors $\widetilde{c_{i,j}} = 255$ sinon $\widetilde{c_{i,j}} = 0$. Mais on se propose de redistribuer cette erreur de quantification aux pixels voisins qui n'ont pas encore été traités.
Au final, on obtiendra une image qui, bien qu'elle soit constituée de pixels à 8 couleurs, donnera l'apparence d'une image possédant une palette de couleurs plus importante.
L'algorithme de Floyd-Steinberg propose de redistribuer l'erreur de quantification [1] $e = c_{i,j} - \widetilde{c_{i,j}}$ aux voisins non traités suivant le schéma

	$\widetilde{c_{i,j}}$	$\_ + \frac{7}{16}e$
$\_ + \frac{3}{16}e$	$\_ + \frac{5}{16}e$	$\_ + \frac{1}{16}$

On part ainsi du pixel $(0, 1)$ en lisant de gauche à droite puis ligne par ligne.
Il y a bien sûr un problème au bord. On pourra par exemple rogner les premières et dernières colonnes et lignes...

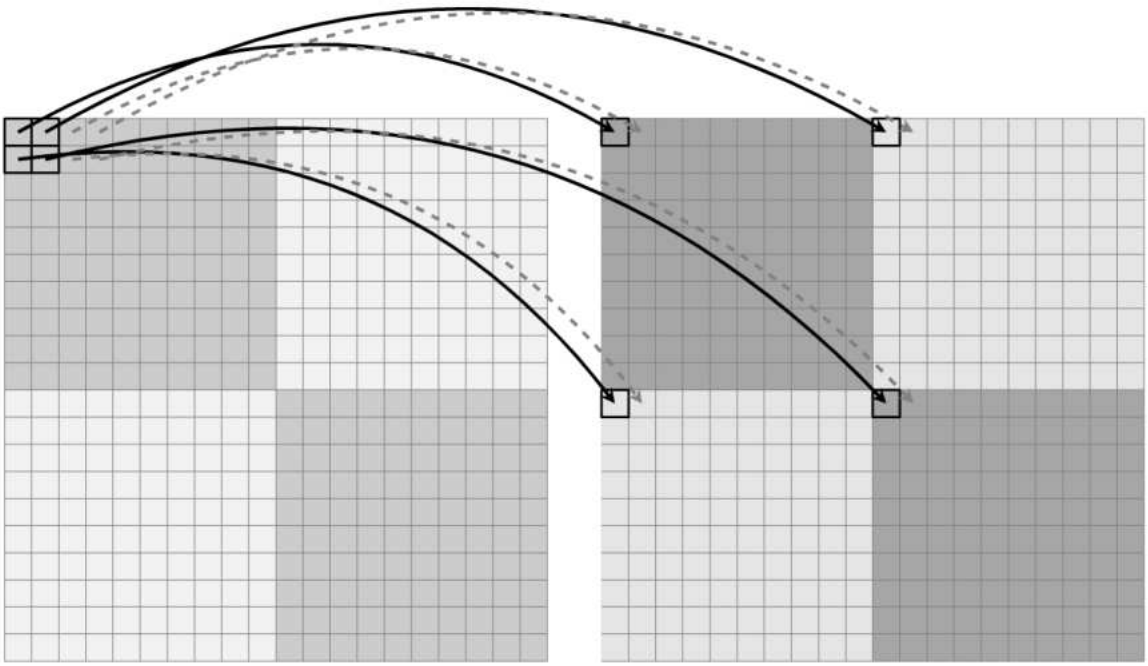
1. pour chaque tableau R, G ou B

Écrire une fonction `floyd(T, seuil)` qui renvoie à partir du tableau bidimensionnel `T` un tableau transformé suivant l'algorithme précédent et tester la fonction sur une image noir et blanc puis sur une image couleur.

Exercice 5.7 Photomaton

Il vous est conseillé d'avoir traité les questions 1 à 4 du T.P. **5.1** d'imagerie p. 208

La **transformation du photomaton** consiste à opérer la bijection suivante sur un tableau de pixels de taille $\alpha \times \beta$ où α et β sont des entiers pairs.



Ainsi si $\alpha = 2a$ et $\beta = 2b$, après transformation, un pixel de coordonnées (i, j) pour $i \leqslant a$ et $j \leqslant b$ est la copie de celui de coordonnées $(2i, 2j)$, un pixel de coordonnées $(i + a, j)$ pour $i \leqslant a$ et $j \leqslant b$ est la copie de celui de coordonnées $(2i + 1, 2j)$, etc.
Puisqu'on a une bijection sur un ensemble fini, cette transformation est périodique (sa période est l'ordre de cette bijection dans le groupe $\mathscr{S}_{\alpha\beta}$).
Écrire une fonction `photomaton(P)` qui effectue cette transformation sur le tableau `P` et tester la fonction sur l'image `'image256.png'` fournie sur le site.
La période est relativement importante... À vous de la deviner.

Étude mathématique : comment calculer la période ? [1]
La transformation correspond à la permutation dans $\mathscr{S}_{2n} = \mathscr{B}ij\,(\llbracket 0, 2n-1 \rrbracket)$

$$\sigma(i) = \begin{cases} \dfrac{i}{2} \text{ si } i \text{ est pair} \\ \dfrac{i-1}{2} + n \text{ si } i \text{ est impair.} \end{cases}$$

1. Plutôt destinée aux élèves de MPSI/MP

Remarquons que 0 et $2n-1$ sont des points fixes de σ.
Nous avons

$$\sigma^{-1}(i) = \begin{cases} 2i \text{ si } 0 \leqslant i \leqslant n-1 \\ 2i+1-2n \text{ si } n \leqslant i \leqslant 2n-1. \end{cases}$$

On peut se restreindre aux éléments de $[\![0, 2n-2]\!]$ puisque $2n-1$ est un point fixe.
Remarquons alors que dans $\mathbb{Z}/(2n-1)\mathbb{Z}$,

$$\sigma^{-1}(i) = 2i \text{ donc } \sigma^{-k}(i) = 2^k i.$$

En conséquence, l'ordre de la permutation σ (ou σ^{-1}) dans $\mathscr{S}_{\mathbb{Z}/(2n-1)\mathbb{Z}}$ est égal à l'ordre de 2 dans $\mathbb{Z}/(2n-1)\mathbb{Z}$ c'est-à-dire le plus petit entier $p \geqslant 1$ tel que $2n-1 \mid 2^p - 1$.
Pour la transformation photomaton d'une image de taille $2a \times 2b$, on effectue la transformation précédente sur $2a$ lignes et de façon indépendante sur $2b$ colonnes.
Posons $p = \mathscr{O}_{2a-1}(2)$ et $q = \mathscr{O}_{2b-1}(2)$, on sait que la transformation « photomaton » sera de période $p \vee q$.

Exercice 5.8 Stéganographie dans une image par réécriture du LSB

Dans cet exercice, on convertit une image en couleur en une image en niveaux de gris puis on cache un texte dans cette image.
L'œil est plus sensible à certaines couleurs qu'à d'autres. Le vert pur par exemple paraît plus clair que le bleu pur. Pour tenir compte de cette sensibilité lors de la transformation d'une image en couleurs en image en niveaux de gris, on ne prend généralement pas la moyenne arithmétique des intensités des couleurs fondamentales mais une moyenne pondérée. Une formule standard donnant le niveau de gris en fonction des trois composantes est :

$$\text{gris} = \lfloor\ 0.299 * \text{rouge} + 0.587 * \text{vert} + 0.114 * \text{bleu}\ \rfloor$$

0. Écrire une fonction `niveau_gris(M)` qui prend en entrée une matrice codant une image RGB stockée sous la forme d'un `numpy.ndarray` de dimensions `(n, p, 3)` et qui renvoie une matrice codant une image en niveaux de gris, codée sous la forme d'un `numpy.ndarray` de dimensions `(n, p)` selon la formule présentée ici.

La stéganographie par réécriture du LSB (*least significant bit*) consiste à cacher un texte dans une image (en niveaux de gris, donc une matrice de nombres entiers compris entre 0 et 255) de la manière suivante :

- Pour chaque pixel de l'image, le bit le moins significatif (donc correspondant à 2^0 dans l'écriture binaire du nombre) ne sera pas en réalité associé à une différence de niveau de gris dans l'image mais à un bit dans l'encodage du message caché ;
- les bits extraits de l'image (les moins significatifs, donc) seront regroupés 8 par 8 ; on lit la matrice ligne par ligne ;
- chaque groupement de bits sera converti en un entier non signé (compris entre 0 et 255) ;
- cet entier sera converti en un caractère Unicode à l'aide de la commande `chr` de Python ; par exemple `chr(97)` est évalué à `a` ;
- on admettra par convention que le texte s'arrête lorsque l'entier lu est 128. Celui-ci correspond au caractère `"\x80"` qui est un caractère de contrôle en Unicode.

Par exemple, supposons que la première ligne de l'image commence par 24 13 8 24 24 24 255 0
Les bits extraits sont 0100 0010 ; ceci correspond à 66 ; et, comme `chr(66)` est évalué à `B`, c'est ce caractère qui est caché dans ces pixels de l'image.

Notons que ces pixels auraient pu au départ être les mêmes ; ou 25 12 8 25 24 25 254 1 ou encore 24 13 9 25 25 24 255 0, etc.

1. Écrire une fonction `stegano(image, chaineaencoder)` qui prend en argument un `numpy.ndarray` correspondant à une image en niveaux de gris, et une chaîne de caractères `chaineaencoder` qui représente le texte que l'on souhaite cacher dans l'image. Cette fonction renverra une matrice correspondant à une image en niveaux de gris.
2. Écrire une fonction `unstegano(image)` qui prend en argument un `numpy.ndarray` correspondant à une image en niveaux de gris dans lequel est caché un message par le procédé décrit ci-dessus et qui renvoie la chaîne de caractères cachée dans l'image.
3. Écrire une fonction `decode(fichier)` qui prend en argument une chaîne de caractères `fichier` qui correspond au nom d'une image codée en niveaux de gris et contenant un texte caché et stocké dans le répertoire courant et qui renvoie le texte caché dans cette image. On trouvera un exemple sur la page dédiée à cet ouvrage sur le site de Dunod.

Travaux pratiques

TP 5.0 – Poule Renard Vipère

Le Poule-Renard-Vipère est un jeu populaire chez les jeunes enfants. Il en existe de nombreuses variantes. Nous allons modéliser ici le déroulement du jeu.
Le principe est qu'un grand nombre d'enfants se déplacent sur un terrain (que nous modéliserons par un damier). Chaque enfant est soit une Poule, soit un Renard, soit une Vipère. Chaque Poule doit attraper les Vipères, chaque Vipère doit attraper les Renards, chaque Renard doit attraper les Poules. Lorsqu'un enfant est attrapé, il change de catégorie, et devient du même type que celui qui l'a attrapé (une Poule attrapée par un Renard devient un Renard).
Nous commençons par introduire en Python quelques variables globales. Chacun des trois animaux est représenté par un entier (0, 1, ou 2).

```
POULE = 0
RENARD = 1
VIPERE = 2
NBANIMAUX = 3
```

Il est préférable, pour rendre le code plus lisible, d'utiliser des constantes globales comme `POULE` plutôt que d'utiliser directement les nombres 0, 1 et 2.

0. Écrire une fonction `grille(n)` qui renvoie un tableau `numpy` carré de `n` lignes et `n` colonnes contenant, dans chaque case, une liste vide. Par exemple, `grille(3)` doit renvoyer :

```
array([[[], [], []],
       [[], [], []],
       [[], [], []]], dtype=object)
```

Cette grille va modéliser le terrain de jeu (qui est découpé en cases), les listes représentent la liste des enfants (ce sont des listes de 0, 1 et 2).

1. Écrire une fonction `init(nombre, n)` qui crée un terrain de jeu de taille `n` puis la remplit au hasard[1] avec des Poules, des Renards et des Vipères (on met `nombre` enfants de chaque type).
Par exemple, `init(5, 3)` peut renvoyer :

```
array([[[2], [], [0, 2, 2]],
       [[], [0], [1]],
       [[1, 1, 2, 0], [1, 0, 1], [2, 0]]], dtype=object)
```

2. Écrire une fonction `compte(L)` qui étant donné une liste de 0, de 1 de 2 renvoie un tableau `numpy` `t` tel que `t[k]` donne le nombre d'occurrences de `k` dans `L`. Ainsi `t[RENARD]` renvoie le nombre de Renards présents dans la liste `L`.
Par exemple, si `A` est la matrice précédente, alors `compte(A[2,0])` renvoie `array([ 1., 2., 1.])`.

3. Écrire une fonction `compte_total(G)` qui prend en entrée un terrain de jeu `G` et qui renvoie, à l'instar de la question précédente, le nombre de Poules, de Renards et de Vipères dans un tableau. Ainsi, `compte_total(A)` renvoie `array([ 5., 5., 5.])`.

1. Avec une probabilité uniforme.

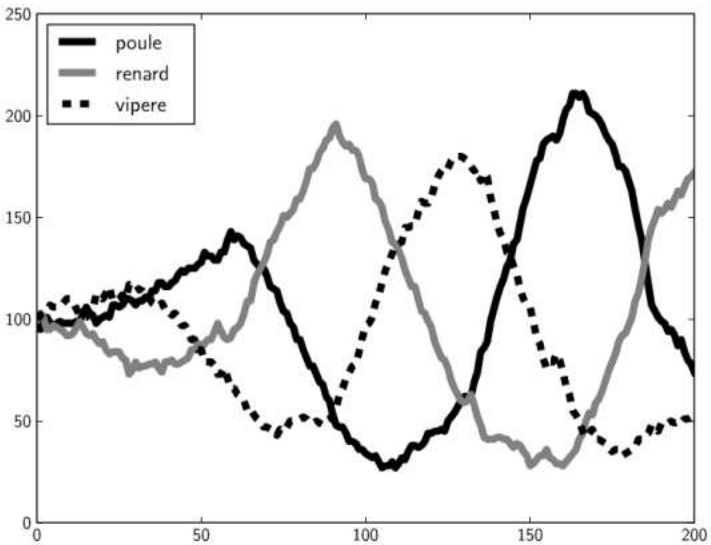
4. Écrire une fonction `voisins(n, i, j)` qui renvoie la liste des coordonnées des voisins (diagonale comprise, case comprise) de la case de coordonnées `(i, j)` dans un terrain de jeu de taille `n`. Ainsi `voisins(3, 0, 2)` renvoie `[(0, 1), (0, 2), (1, 1), (1, 2)]`.

On considère que la probabilité qu'une POULE soit attrapée est égale au nombre de RENARDS sur la case divisé par le nombre total d'enfants. On considère de même que la probabilité pour respectivement un RENARD ou une VIPÈRE d'être attrapé est égale au nombre de VIPÈRES ou au nombre de POULES divisé par le nombre total d'enfants. Ainsi, dans la case `A[2, 0]` il y a deux RENARDS, une POULE et une VIPÈRE. Chacun des RENARDS a 25% de chance d'être attrapé. La POULE a 50% de chance de se faire attraper et la VIPÈRE 25%. On remarque qu'il est possible que tout le monde se fasse attraper (et donc que tout le monde change de catégorie) avec une probabilité d'environ 0.8%.

5. Écrire une fonction `attrape(L)` qui prend en entrée une liste de POULES, RENARDS et VIPÈRES, et qui tire au sort quels enfants sont attrapés puis qui change leur catégorie en conséquence. La fonction `attrape` modifie `L` et renvoie `None`.
6. Écrire une fonction `attrape_total(G)` qui prend en entrée un terrain de jeu `G` et qui applique la fonction `attrape` sur chacune de ses cases.

On décompose le jeu en plusieurs étapes. À chaque étape, chaque enfant se déplace au hasard sur une case voisine [1] de la sienne. Ensuite, chaque enfant peut éventuellement être attrapé.

7. Écrire une fonction `Etape(G)` qui prend en entrée un terrain de jeu `G` et qui applique une étape du jeu. Cette fonction modifie `G` et renvoie `None`.
8. Écrire une fonction `evolution(G, N)` qui applique `N` fois la fonction évolution au terrain de jeu `G` et qui renvoie une liste de tableaux `L`. `L[k]` est le tableau du nombre de POULES, RENARDS et VIPÈRES à l'étape `k`.
9. Simuler une partie de POULE-RENARD-VIPÈRE impliquant 100 enfants de chaque catégorie et durant 200 étapes, puis tracer les résultats. On est censé obtenir un graphe qui ressemble à ça :



1. Avec une probabilité uniforme. Il est possible que l'enfant reste sur la même case, car à la question 4 on a inclus dans les voisins d'une case la case elle-même.

TP 5.1 – Imagerie (base)

Avec le module PIL/Pillow Si l'on a installé la librairie PIL/Pillow, on peut facilement récupérer sous forme d'un tableau `array` les données d'un fichier image.

```
# solution avec scipy.misc (qui utilise PIL)
import scipy.misc as scm
A = scm.imread('Brehatsol.png')

# solution directe avec PIL
from PIL import Image as im
photo = im.open('Brehatsol.png')
A = np.array(photo)
```

Examinons le tableau `A`

```
A.shape, A.dtype
```

```
((500, 375, 3), dtype('uint8'))
```

La tableau `A` est une matrice de points colorés appelés **pixels**. Chaque pixel est un triplet de nombres entre 0 et 255 : un nombre pour chaque couleur primaire **rouge, vert, bleu**. Un tel nombre est représentable sur 8 bits par un octet non signé (valeur de 0 à 255, `np.uint8` en `numpy`). Il y a donc $2^{24} = 16777216$ couleurs possibles. On utilise ici la synthèse *additive* des couleurs : le triplet (0, 0, 0) correspond à un pixel noir alors qu'un pixel blanc est donné par (255, 255, 255). Un pixel pur rouge est codé par (255, 0, 0).
Remarque Certaines images sont codés avec un quadruplet, la quatrième composante indiquant le niveau de transparence du pixel.
On peut visualiser l'image simplement avec l'instruction `imshow` du module `matplotlib.pyplot` (avec l'option `interpolation='nearest'` pour éviter un lissage sur l'image).

```
A[:250, :250] = (255, 0, 0)

plt.imshow(A, interpolation='nearest')
plt.axis('off')
plt.show()
```

Pour sauver la nouvelle image à partir du tableau `A`.

```
A = 255 - A
# solution avec scipy.misc (qui utilise PIL)
scm.imsave('Brehatsolmodif.png', A)

# solution directe avec PIL
monimage = im.fromarray(A)
monimage.save('Brehatsolmodif.png')
```

```
A = scm.imread('Brehatsolmodif.png')
plt.imshow(A, interpolation='nearest')
plt.axis('off')
plt.show()
```

Rustine : si l'on ne dispose pas du module PIL

Si on ne dispose pas de la librairie PIL, on peut imiter son comportement en passant par la fonction `imread` du module `matplotlib.pyplot` qui ne lit (sans `PIL`) que des fichiers image de type `.png` mais convertit les niveaux de rouge, vert, bleu nativement en octets non signés en flottant dans [0,1].

La lecture d'un fichier image au format `png` puis l'affichage se fait suivant la syntaxe

```
image = plt.imread('Brehatsol.png')
plt.imshow(image)
plt.show()
```

Pour définir le répertoire courant

```
import os
os.chdir(r'C:\monrepertoire')  # exemple
```

Si l'image est en couleurs, la fonction `imread` renvoie un `array` tridimensionnel (RGB) de flottants entre 0 et 1 ; si l'image est en noir est blanc, le tableau renvoyé est bidimensionnel toujours avec des flottants dans $[0, 1]$.

Voici deux fonctions que l'on va utiliser pour respectivement lire un fichier image et afficher une image

```
def litimage(nomf):
    # pour combler l'absence de PIL
    return np.uint8(plt.imread(nomf) * 255)
```

```
def affiche(imag, NB=False):
    if NB:
        plt.imshow(imag, cmap=plt.get_cmap('gray'),
                   vmin=0, vmax=255, interpolation='nearest')
    else:
        plt.imshow(imag, interpolation='nearest')
    plt.axis('off')
    plt.show()
```

On peut sauver son image avec la fonction `savefig` du module `matplotlib.pyplot`.

```
plt.savefig('mafigure.png')
```

En résumé, un tableau correspondant à une image en noir et blanc ou en couleurs peut être créé avec la fonction `zeros` de `numpy`

```
imageNB = np.zeros((a, b), dtype=np.uint8)
imagecouleur = np.zeros((a, b, 3), dtype=np.uint8)
```

La valeur 0 correspondant à l'absence de couleur (ou de blanc) et la valeur 255 la valeur maximale de lumière (rouge, verte ou bleu ou encore blanche).

On joue avec les couleurs

0. Ouvrir et afficher un fichier image couleur (de type `png`) au choix.
1. Lire la taille (`shape`) et le type (`dtype`) des cellules du tableau `array` de l'image.
2. Discrétiser les couleurs de manière à ce que les valeurs prises par R, G ou B soient des multiples de `16` et afficher le résultat.

3. Transformer chacune des couleurs `c` de l'image par `255-c` et afficher l'image obtenue.
4. Transformer le tableau de manière à afficher seulement la couleur rouge.

On convertit en noir et blanc

5. On convertit une image couleur (R, G, B) en une image en niveau de gris en prenant une moyenne pondérée des valeurs de R, G, B.
   Nous nous contenterons de prendre la moyenne classique (voir sur `wikipedia` les vraies pondérations employées, c'est assez compliqué...).
   Attention à convertir les octets en `int` pour des calculs qui dépassent la valeur 255...

Contraste

6. Reprendre la photo noir et blanc que vous avez construite. Chaque pixel correspond à un niveau de gris dans $[\![0, 255]\!]$. Regarder le résultat obtenu en transformant le niveau de gris c par c^2 ou $\sqrt{c}$ que l'on « renormalisera » sur $[\![0, 255]\!]$.

Seuil(s)

7. Les tableaux (`array`) `numpy` autorisent les affectations par *fancy indexing*

```
T[T < 100] = 0
T[T >= 150] = 255
```

   En modifiant le seuil (ici 100), regarder le résultat obtenu sur l'image en noir et blanc.
   Faire de même sur une image en couleurs.

Réduire l'image par $D \times D$

8. À partir de l'image en noir et blanc construite avec un effet de seuil, construire une image dont la longueur et la largeur est divisée par $D = 16$.

Filtre, convolution

9. Définir une fonction `convolution(A, C)` qui prend en entrée un tableau `numpy` (image avec des cellules de type `np.uint8`) $A = (a_{i,j})$, une matrice de convolution de taille impaire $N = 2k+1$, $C = (c_{i,j})$ et qui renvoie un tableau $B = A * C$, produit de convolution des deux tableaux,

$$b_{i,j} = \sum_{(I,J)\in[\![0,N-1]\!]^2} a_{i+I-k,j+J-k} c_{I,J}.$$

   pour i et j pas trop prêt du bord...
   Voici la structure du programme :

```
def convolution(A, C):
    hauteur, largeur = A.shape
    B = np.zeros((hauteur, largeur), dtype=np.uint8)
    N, P = C.shape
    assert(N == P and N % 2 == 1)  # par sécurité
    k = (N - 1) // 2  # N = 2k+1
    pass  # a vous de jouer
    return B
```

   Tester la fonction convolution sur une image de votre choix avec les matrices suivantes

$$C_1 = \frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \quad C_2 = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{pmatrix} \quad C_3 = \begin{pmatrix} -2 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 2 \end{pmatrix}$$

effet de flou — effet de contours — effet de relief

Détection des contours/bords

10. Une technique simple de détection de bords d'un objet dans une image consiste à calculer la valeur du changement au point `C[i,j]` par

$$\texttt{V[i,j]}=\sqrt{(\texttt{C[i}-1,\texttt{j]}-\texttt{C[i}+1,\texttt{j]})^2+(\texttt{C[i,j}-1\texttt{]}-\texttt{C[i,j}+1\texttt{]})^2}$$

On crée alors une image noir et blanc dont les pixels sont blancs (255) si V[i,j]<s et noir (0) si V[i,j]⩾s où `s` est une valeur de seuil à définir.

Écrire une fonction `bords(T, seuil)` qui renvoie une tableau image de bords à partir du tableau `T` et du `seuil`. Tester cette fonction sur l'image `rochers.png`.

On pourra éventuellement s'écarter de plus d'une unité du point (i, j) pour améliorer la recherche du bord.

Histoire de réviser la notion de récursivité, remplir alors une partie blanche en gris sur l'image obtenue précédemment.

Fast Fourier Transform (boîte noire)

La transformée de Fourier bidimensionnelle permet de passer d'un tableau $(f(x,y))_{(x,y)\in[\![0,a-1]\!]\times[\![0,b-1]\!]}$ à un tableau de fréquences $(F(u,v))$ suivant les formules suivantes

$$F(u,v)=\sum_{(x,y)\in[\![0,a-1]\!]\times[\![0,b-1]\!]} f(x,y)e^{-2i\pi\left(\frac{ux}{a}+\frac{uy}{b}\right)}$$

$$f(x,y)=\frac{1}{ab}\sum_{(u,v)\in[\![0,a-1]\!]\times[\![0,b-1]\!]} F(u,v)e^{2i\pi\left(\frac{ux}{a}+\frac{uy}{b}\right)}$$

Il existe un algorithme de transformée de Fourier rapide (FFT) qui permet d'effectuer $\mathcal{O}(ab\ln(ab))$ multiplications au lieu de $\mathcal{O}(a^2b^2)$ (en pratique on fait une transformée rapide 1D de Fourier ligne par ligne puis une transformée rapide 1D de la transformée ligne par ligne).

Pour étudier cet algorithme, voir le T.P. **7.4** p. 307.

On va utiliser la transformée de Fourier rapide bidimensionnelle pour transformer les tableaux R, G, ou B en tableaux de fréquence de même taille. On considérera les fonctions suivantes

```
def tofreq(img):
    # fft convertit l'image en tableau de fréquences centre en (0,0)
    f = np.fft.fft2(img)
    # on recentre le... centre
    return np.fft.fftshift(f)

def fromfreq(f):
    f_ishift = np.fft.ifftshift(f) # on décentre le centre
    # on inverse fourier
    im = np.fft.ifft2(f_ishift)
    # on a des flottants complexes, on récupère le module
    im = np.abs(im)
    return np.uint8(im) # on convertit
```

11. À partir d'une image de votre choix, créer un filtre-bas, c'est-à-dire que vous ne gardez que les fréquences proches de l'origine (centre du tableau) et vous reconstruisez une image avec seulement ses basses fréquences.

TP 5.2 – Le solitaire de Schelling

Thomas Crombie Schelling est un économiste américain ayant obtenu le « prix de la Banque de Suède en sciences économiques en mémoire d'Alfred Nobel », parfois abrégé en « prix Nobel d'économie ». L'un de ses travaux porte sur la ségrégation urbaine (Chapitre 4 de [Sch06]). Il a imaginé un modèle très simple, appelé « solitaire de Schelling » dont nous présentons ici une variante.

On suppose que la population d'une ville est séparée en deux groupes totalement distincts, que nous nommerons noirs et blancs. La ville est modélisée par un damier, chaque case est occupée par un habitant (qui est soit noir, soit blanc) ou est vide. Nous appelons **voisins** d'un individu tous les individus situées sur les cases adjacentes (diagonales comprises) ; ainsi, chaque individu a au plus 8 voisins. Chaque habitant est supposé avoir une tolérance limitée vis-à-vis des personnes de l'autre couleur, et devient **mécontent** lorsque les voisins de l'autre couleur représentent deux tiers ou plus de l'ensemble de ses voisins. Par exemple, un habitant ayant cinq voisins est mécontent s'il n'a pas au moins deux voisins de sa couleur.

Ce modèle va évoluer en plusieurs tours. À chaque tour, un individu mécontent tiré au hasard déménage. Il se déplace sur un emplacement vide tiré au hasard (il peut alors devenir content ou rester mécontent). On itère ce procédé jusqu'à obtenir une ville stable, c'est-à-dire une ville sans mécontent. Enfin, on observe si il y a, dans la ville finale, des « ghettos » de couleur.

Modèle informatique

Informatiquement, le modèle est constitué de 4 éléments :

- Une matrice carrée `M` qui représente la ville. C'est un `ndarray` à deux dimensions. Les uns de la matrices représentent les blancs, les deux les noirs, et les zéros les emplacements vides.
- Un entier `n`, qui est la taille de la matrice `M` (son nombre de colonnes et de lignes).
- Une liste `LV` des cases vides de `M`, c'est-à-dire des cases de `M` contenant un zéro.
- Une liste `LM` des cases contenant des habitants mécontents.

Une « case » est un couple d'entier `i,j` où `i` et `j` sont deux entiers compris entre `0` inclus et `n` exclu qui représentent respectivement le numéro de ligne et le numéro de colonne de la case.

Dans toute la suite, `M`, `n`, `LV` et `LM` auront le sens décrit ci-dessus sans que ce soit à chaque fois rappelé. Ces différentes valeurs doivent rester « cohérentes » entre elles, c'est pour cela que tout au long du programme, on veille à préserver les invariants suivants :

Invariants

(0) Il n'y a pas de doublons (i.e., pas deux fois la même valeur) dans `LV`, ni dans `LM`.
(1) Une case `i,j` est dans `LV` si et seulement si `M[i, j] = 0`.
(2) Une case est dans `LM` si et seulement si elle contient un individu mécontent.

Pour éviter des effets liés aux bords de la matrice, et pour simplifier la programmation en évitant à avoir à considérer différents cas (selon que la case considérée est ou non sur un bord de la matrice), nous utilisons le modèle torique.

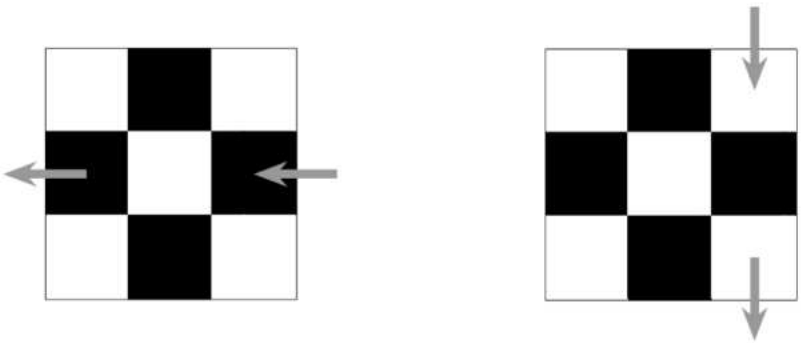


FIGURE 1. Lorsqu'on sort de la matrice par la gauche, on se retrouve à droite. Lorsqu'on sort de la matrice par le bas, on se retrouve en haut.

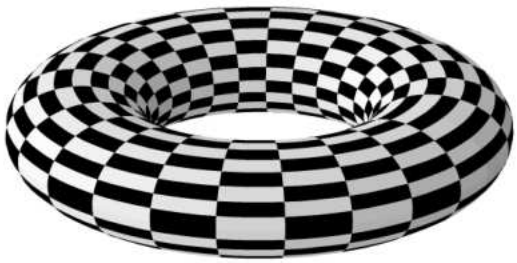


FIGURE 2. Un damier sur un tore.

Définition

Dans le **modèle torique**, si on se déplace d'une case « hors » de la matrice, on se retrouve de l'autre côté, comme illustré sur la figure 1. Ce modèle est dit torique, car il revient à considérer que notre damier a été dessiné sur un tore, voir figure 2.

Dans le modèle torique, chaque case est voisine de huit autres cases, y compris les cases au bord du damier. Attention toutefois, un habitant peut avoir moins de huits voisins, car des cases peuvent être inocupées.

Nous utilisons les bibliothèques suivantes.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as col
import random
```

0. Écrire une fonction `rand_couple(n)` qui étant donné un entier `n`, tire au hasard (uniformément) un couple d'entiers chacun compris entre 0 inclus et `n` exclu.

1. Écrire une fonction `rand_pop(L)` qui, étant donné une liste `L`, retire au hasard à `L` un élément et renvoie sa valeur. La méthode `L.pop(i)` peut être ici utile.
2. Écrire une fonction `quadrillage(n)` qui étant donné un entier `n`, renvoie une matrice contenant alternativement des uns et des deux comme il y a alternance de blancs et noirs sur un vrai damier. Comme nous n'utilisons dans ce TP que 3 valeurs (0, 1 et 2), la fonction renvoie un `ndarray` contenant des entiers 8 bits non signés (utiliser l'argument optionnel `dtype = np.uint8`).

L'échelle des couleurs (gris, blanc, noir) est définie comme suit :

```
couleurs = col.ListedColormap(['grey', 'white', 'black'])
```

Pour afficher une matrice `M`, on utilise la commande suivante :

```
plt.imshow(M, interpolation='Nearest', cmap=couleurs, vmin=0)
```

3. Écrire une fonction `voisinage(M, n, i, j)` qui renvoie une liste (ou un triplet) de valeurs : le nombre de 0, le nombre de 1 et le nombre de 2 dans les 8 cases voisines de la case `i, j`.
4. Écrire une fonction `pascontent(M, n, i, j)` qui renvoie `True` si la case `i, j` contient un individu mécontent et `False` sinon.

Pour initialiser une matrice, nous utiliserons le procédé suivant :

- Nous créons une matrice avec la fonction `quadrillage`.
- Nous tirons au hasard `nbvides` cases et les vidons (i.e. remplaçons leur contenu par des zéros).
- Nous remplissons au hasard `nbnouv` cases vides par des 1 ou des 2 (on tire au hasard avec une probabibilité de 50%/50%).

On suppose bien évidemment que `nbvides` est strictement plus grand que `nbnouv`. À la fin de l'initialisation, la matrice contient exactement `nbvides - nbnouv` cases vides (pas une de plus, pas une de moins).

5. Écrire une fonction `init(n, nbvide, nbnouv)` qui initialise une matrice. Cette fonction doit renvoyer `M`, `LV` et `LM`.
6. Écrire une fonction `nouveaux_pas_contents(M, n, LM, i, j)` qui met à jour `LM` après que `M[i, j]` a été modifié. On veillera à rétablir les invariants (2) et (0), c'est-à-dire éviter les doublons dans `LV` et dans `LM` et faire en sorte que `LM` contienne exactement les individus mécontents. On veillera également au fait que la complexité ne dépende pas de `n`, en évitant tout parcours de matrice inutile.
7. Écrire une fonction `deplace(M, n, LV, LM)` qui tire au hasard un mécontent et le déplace au hasard dans une place vide, tout en maintenant les invariants.
8. Écrire une fonction `evolution(M, n, LV, LM, limit=1000)` qui fait évoluer le modèle jusqu'à ce que `LM` soit vide ou qu'on atteigne un nombre d'itérations égal à `limit`.
9. Simuler quelques villes avec les paramètres suivants : `n = 10`, `nbvides = 20`, `nbnouv = 5`. Après ces simulations, qu'observe-t-on ?
10. Simuler quelques villes avec les paramètres suivants : `n = 100`, `nbvides = 2000`, `nbnouv = 500`. Observe-t-on des ghettos, c'est-à-dire des zones habitées par une seule couleur ?

Si vous n'avez pas fait attention à la complexité, le cas `n = 100` pourrait poser problème.

TP 5.3 – Son et fichier `.wav`

On utilisera les modules suivants

```
import scipy.io.wavfile as w
import numpy as np
import matplotlib.pyplot as plt
```

Jouer un fichier son

On appelle directement un programme qui lit des fichiers média (encore faut-il connaitre son nom…). On pourra télécharger sur la page dédiée à cet ouvrage le fichier `musique.wav`.

```
import os

os.system('vlc musique.wav')   # ou vlc.exe
```

On peut essayer d'utiliser PySide

```
from PySide.QtGui import QSound

print('son disponible :', QSound.isAvailable())

monrep = r"/home/marc/ECHANGE/IPT/IPTpreparationCRS/pyTD/-00-sons/"
QSound.play(monrep + "musique.wav")
```

Lire un fichier son `.wav`

0. Voici comment procéder

```
son = open('musique.wav', 'br')
L = w.read(son)
son.close()
```

La fonction `w.read` renvoie un couple (`fréquence`,`T`) où `T` est un tableau `numpy` de type `array`.
La fréquence est exprimée en hertz ; pour un CD, la norme est de 44 100 Hz.
Le nombre de colonnes du tableau `T` donne le nombre de canaux, c'est-à-dire le nombre de pistes (1 = mono, 2 = stéréo, 6 = 5.1, etc.)
Chaque ligne contient les valeurs (traduit en tension électrique quand on fait vibrer un haut-parleur) de chacune des pistes. Les valeurs possibles dépendent de la résolution, pour un CD, la norme est celle d'un entier signé 16 bits donc 65 536 valeurs de -32768 à 32767. On peut récupérer cette résolution avec `T.dtype`.
Préciser les différentes caractéristiques du morceau contenu dans le fichier `musique.wav`.
Quelle est la durée du morceau ?

Créer trois notes

1. On peut créer un ficher de type `wav` de la manière suivante

```
def creefichier(nomfich, freq, X):
    '''
    X est un tableau (n, nb de pistes) de type np.int16 par exemple
    '''
    monson = open(nomfich, 'bw')
    w.write(monson, freq, X)
    monson.close()
```

Créer un fichier `mestroisnotes.wav` qui joue les trois notes *do* (hauteur medium, en dessous du *la* de référence), *mi*, *sol* durant 2 secondes chacunes, à la fréquence 44100 Hz.
On utilisera des sinusoïdes pures et le *la* de référence sera pris à la fréquence 440 Hz.
On construira la gamme sachant que

- passer d'une octave à celle plus aigue revient à *doubler* la fréquence,
- les fréquences des *douze* demi-tons d'une octave suivent une progression géométrique.

2. Factoriser le code en définissant la fonction `creenote(vol, freq, freqnote, duree)` qui renvoie un tableau `array` de type `np.int16` correspondant au son de la note de fréquence `freqnote` pour une durée `duree` en secondes, à la fréquence d'échantillonnage `freq`, et au volume `vol` (amplitude maximale de la sinusoïde).

3. On va ajouter quelques harmoniques à notre note pour lui donner un *timbre* un peu plus intéressant (en réalité la notion de timbre est une notion difficile dont la décomposition harmonique continue n'est qu'une des composantes, il y a aussi l'attaque de la note, la modulation de la fréquence, etc.).
Écrire la fonction `creenotetimbre(vol, freq, freqnote, duree, timbre=[])` qui reprend la fonction précédente mais ajoute les harmoniques supérieures (= les notes aux octaves supérieures) avec un volume suivant la liste `timbre`.
Par exemple si `timbre = [.5, .4, .3]`, on jouera la note de fréquence `freqnote` superposée avec la note de fréquence double mais avec un volume de 50% de la note de base, superposée avec la note de fréquence $\times 4$ avec un volume de 40% puis celle de fréquence $\times 8$ avec un volume de 30%. On s'arrangera pour que la somme des volumes obtenus vale l'entrée initiale `vol`.
Puis reprendre l'exemple précédent avec le timbre

```
timbre = [(1 + (-1)**n) * .15 * .8**n for n in range(1, 19)]
```

Lecture de notes

4. On propose d'utiliser un dictionnaire, c'est-à-dire un ensemble de couples clé/valeur pour stocker les fréquences des notes d'un morceau que l'on veut jouer.
On peut voir un dictionnaire comme une liste Python où l'indice a été remplacé par une clé qui, pour nous, sera une chaîne de caractères.
Voici un exemple de manipulations élémentaires.

```
mondico = {}   # création d'un dico vide

mondico['maclé'] = 5
mondico['autre'] = -1

print(mondico)
```

```
{'maclé': 5, 'autre': -1}
```

```
for cle in mondico:
    print(mondico[cle])

print(list(mondico))
```

```
5
-1
['maclé', 'autre']
```

• Créer un dictionnaire `dico` contenant
les clés `'c', 'c#', 'd', 'eb', 'e', 'f', 'f#', 'g', 'g#', 'a', 'bb', 'b'`
de la gamme classique, c = *do*, bien sûr a = *la* et # = *dièze*, eb = *la bémol*, bb = *si bémol*
(c d e f g a b = *do re mi fa sol la si*)
et ayant pour valeurs les fréquences correspondantes pour un *la* à 440 Hz (`dico['a'] = 440`).

```
Lgammechromatique = ['c', 'c#', 'd', 'eb',
                     'e', 'f', 'f#', 'g', 'g#', 'a', 'bb', 'b']
```

5. Ajouter ensuite dans `dico` l'octave inférieure (les notes étant notées c0, c#0, etc.) et l'octave supérieure (les notes étant notées c2 c#2, etc.)
On ajoutera également le code `pause` pour un silence (prendre une fréquence de note égale à 0).
`dico['pause'] = 0`

6. Créer un fichier `gammechromatique.wav` jouant la gamme chromatique de l'octave classique avec des notes de durée 0,3 secondes.

Extrait d'une fugue ?

7. Nous proposons de jouer un petit extrait d'une fugue très connue.
Les notes sont codées en utilisant les notations de la liste `Lgammechromatique` et rajoutant une octave inférieure (`0`) ou une octave supérieure (`2`). L'extrait est découpé en trois parties. On prendra un tempo égal à 100, c'est-à-dire que 100 noires sont nécessaires pour une minute. La première partie est jouée par la main gauche en double croche (= 1/4 de la durée d'une noire) et la main droite ne joue rien. La deuxième partie est jouée à la double croche pour la main droite et à la croche pour la main gauche. Enfin, la troisième et dernière partie est jouée à la croche pour la main droite et à la double croche pour la main gauche.
On pourra télécharger le code suivant sur la page dédiée à cet ouvrage sur le site de Dunod :

```
Part2maindroite = '''pause d2 c2 d2 bb d2 a d2
g d2 f# d2 g d2 a d2 bb d2 d d2 e d2 f# d2
g d2 f# d2 g d2 a d2'''

Part3maindroite = '''bb d2 bb d2
eb2 g eb2 g c2 a c2 a
d2 f d2 f bb g bb g
c#2 e c#2 e a f a f
g c# g c# f d f d
e bb0 e bb0'''

Part1maingauche = '''pause a g a f a e a
d a c# a d a e a f a a0 a b0 a c# a
d a c# a d a e a'''

Part2maingauche = '''f f# g c
bb0 a0 bb0 c d f#0 g0 a0
bb0 a0 bb0 f#0'''
```

```
Part3maingauche = '''g0 g g0 g d g d g
c eb c eb c eb c eb c f c f c f c f
bb0 d bb0 d bb0 d bb0 d bb0 e bb0 e bb0 e bb0 e
a0 c# a0 c# a0 c# a0 c# f0 d f0 d f0 d f0 d
e0 bb0 e0 bb0 e0 bb0 e0 bb0 d0 a0 d0 a0 d0 a0 d0 a0
e0 g0 e0 g0 e0 g0 e0 g0'''
```

Écrire la fonction `listenote(L)` qui découpe les chaînes de caractères précédentes et renvoie une liste de notes.

Indication Utiliser les méthodes `.strip()` et `.split()` des chaînes de caractères.

8. Écrire le fichier son `maindroite.wav` qui joue la main droite (et bien sûr écouter le résultat) puis faire de même pour la main gauche.

9. Enfin, écrire la fichier son `extraitfuguebach.wav` qui joue les deux mains en même temps (avec un effet stéréo).

TP 5.4 – Diffusion thermique

Ce TP est consacré à la résolution approchée de l'équation de la diffusion thermique.

Résolution en une dimension par la méthode des éléments finis ; schéma explicite

On considérera dans cette partie l'évolution de la température en fonction du temps pour un fil conducteur unidimensionnel. La température de ce fil sera donc une fonction de deux variables $T(z,t)$.

Le fil conducteur électrique est initialement dans son régime stationnaire. Il est brutalement exposé à un refroidissement à 300 K à ses bornes.

On va s'intéresser dans un premier temps à l'équation de la diffusion thermique en une dimension d'espace et sans source (D est appelé la diffusivité thermique du fil conducteur) :

$$\frac{\partial T}{\partial t} = D\,\frac{\partial^2 T}{\partial x^2} \qquad (E_0)$$

on considérera la condition initiale suivante :

$$\forall x,\ T(x,0) = 300 + 100x/L \qquad (C_1)$$

et la condition aux limites suivante :

$$\forall t > 0,\ T(0,t) = T(L,t) = 300 \qquad (C_2)$$

La méthode des différences finies est une technique courante de recherche de solutions approchées d'équations aux dérivées partielles qui consiste à résoudre un système de relations (schéma numérique) liant les valeurs des fonctions inconnues en certains points suffisamment proches les uns des autres.

Ainsi, on va chercher à discrétiser notre problème. Nous ferons l'hypothèse qui est celle utilisée pour la méthode d'Euler de résolution d'une équation différentielle, à savoir que la dérivée d'une fonction en z peut être approchée par l'accroissement de cette fonction entre z et $z + \Delta z$, où Δz désigne notre pas de discrétisation (pour la variable z).

Les valeurs de la température à l'instant initial seront stockées dans une liste T_j^0 de longueur N. Il y aura donc $N-1$ pas de discrétisation en espace. Les valeurs de la température à l'instant $t = \Delta t$ seront stockées dans une liste T_j^1, qui aura également comme longueur N. De même, les valeurs de la température à l'instant $t = n\Delta t$ seront stockées dans une liste T_j^n.

0. Écrire une fonction `PasEspace(L, N)` qui renvoie le pas d'espace Δx en fonction de la longueur L sur laquelle on résout le problème et du nombre N de points souhaités.

Le pas de temps Δt ne peut pas être choisi indépendamment du pas d'espace, pour des questions de stabilité de schéma numérique. On supposera donc que l'on dispose d'une variable globale `Delta_t` qui stocke la valeur de Δt. Aux fins d'applications numériques, la valeur de `Delta_t` sera précisée dans les questions où l'on en aura besoin.

1. Montrer que l'on a dans le cadre de l'approximation des différences finies :

$$\frac{\partial T}{\partial t} \approx \frac{T_j^{n+1} - T_j^n}{\Delta t}$$

et que :

$$\frac{\partial^2 T}{\partial x^2} \approx \frac{T_{j+1}^n - 2T_j^n + T_{j-1}^n}{\Delta x^2}$$

2. En déduire la relation de récurrence suivante pour $j \neq 0$ et $j \neq N-1$:

$$T_j^{n+1} = T_j^n + c\,(T_{j-1}^n - 2\,T_j^n + T_{j+1}^n)\,, \qquad \text{avec} \quad c = \frac{\Delta t\, D}{\Delta x^2}$$

3. Comment s'écrit la relation de récurrence précédente lorsque $j = 0$ et lorsque $j = N-1$?

4. Écrire une fonction `initialisation(L, N)` qui renvoie la liste comprenant les valeurs de T_j^0 respectant la condition initiale (C_1) de l'équation aux dérivées partielles (E_0).

5. Écrire une fonction `transition(T, c, N, Tbord=300)` qui prend en argument c et une liste T, contenant les valeurs de T^n pour un n quelconque et renvoie la liste constituée des éléments de T^{n+1}.

6. Écrire une suite de commandes qui permet d'avoir une représentation graphique du profil en température toutes les 6 secondes.
On prendra $\Delta T = 0,01s$; $N = 101$; $L = 0,5m$ et $D = 1,2 \times 10^{-4}\ m^2.s^{-1}$.
La température sera calculée jusqu'à un temps de 10 minutes.

Résolution en deux dimensions

On s'intéresse désormais au problème de la diffusion thermique sans source avec deux dimensions d'espace. Celle-ci a pour équation :

$$\frac{\partial T}{\partial t} = D\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) \qquad (E_2)$$

7. En s'inspirant de la résolution de ce problème en une dimension, montrer que le schéma suivant permet de résoudre numériquement cette équation :

$$T_{i,j}^{n+1} = T_{i,j}^n + \Delta t\, D\left[\frac{T_{i-1,j}^n - 2\,T_{i,j}^n + T_{i+1,j}^n}{\Delta x^2} + \frac{T_{i,j-1}^n - 2\,T_{i,j}^n + T_{i,j+1}^n}{\Delta y^2}\right]$$

Nous nous intéressons désormais au problème de la diffusion thermique autour d'une source rectangulaire de chaleur après son extinction. On résout ainsi le problème précédent sur le domaine d'espace carré $[-0,2;0,2] \times [-0,2;0,2]$.
Nous prendrons une discrétisation de 101 pas selon l'axe des abscisses et de 101 pas selon l'axe des ordonnées. La température à un instant sera ainsi codée dans une matrice 101×101.
On suppose à l'instant initial que la température vaut 400 K si $(x, y) \in [-0,1;0,1] \times [-0,12;0,12]$ et vaut 300 K sinon.

8. Écrire une fonction `Initial2d()` qui renvoie la matrice contenant les températures initiales. On commencera par chercher à quelle condition sur les lignes et les colonnes les coefficients de cette matrice valent 400 et à quelle condition ils valent 300.

9. Écrire une fonction `transition(T, D, Delta_t, Delta_x, Delta_y)` qui renvoie la température après une itération du schéma numérique. Nous supposerons que nous avons une température de 300 K à la frontière de la matrice.

10. Écrire alors une fonction `température(n)` qui renvoie la température après n secondes. On prendra `D = 0.00012` et `Delta_t = 0.01` (on pourra observer que les solutions sont instables numériquement si on choisit `Delta_t = 0.1` dans la question suivante).

11. Nous désirons faire une représentation en couleurs de la solution obtenue après un temps donné. Pour cela, commencer par créer des **tableaux numpy** `XX` et `YY` des abscisses et ordonnées, équiréparties en 101 points entre -0,2 et 0,2.

12. On donne le code suivant :

```
T = np.array(temperature(50))
plt.figure()
p = plt.pcolormesh(XX, YY, T,shading='flat')
plt.colorbar(p)
plt.axis('image')
plt.show()}
```

Le tester et l'adapter pour avoir une représentation graphique toutes les secondes pendant une période de 10 secondes dans une nouvelle figure.

Résolution implicite en une dimension

Le schéma explicite (2) ne converge que si le pas de temps Δt est suffisamment faible par rapport au pas d'espace Δx. Ceci est caractérisé en informatique par le nombre de Courant (et la condition de Courant-Friedrichs-Lewy).

On définit $C_o = \dfrac{v\Delta t}{\Delta x}$ avec :

- v : vitesse dans la direction x
- Δt : intervalle temporel
- Δx : intervalle dimensionnel

Pratiquement, si C_0 est inférieur à un seuil, on observe une instabilité de calcul, erreur d'approximation dans des calculs numériques, grandissant rapidement au fur et à mesure des calculs. Si la dimension de la grille est inférieure à la distance parcourue dans l'intervalle de pas de temps par l'onde la plus rapide que permet l'équation, l'erreur grandit et envahit la solution physique.

Si l'on souhaite effectuer un calcul pour un temps physique long, beaucoup d'itérations seront nécessaires et le temps de calcul sera très long. C'est pourquoi on préfère d'autres types de schémas appelés schémas implicites.

Dans cette partie, la dérivée partielle seconde par rapport à x de la température apparaissant dans l'équation (1) est évaluée au point d'abscisse x_i et à l'instant k+1 :

$$\frac{\partial^2 T}{\partial x^2}(x,t) \approx \frac{\partial^2 T}{\partial x^2}(x_i, t_{k+1})$$

et la dérivée partielle par rapport à t est évaluée au point d'abscisse x_i et à l'instant k :

$$\frac{\partial T}{\partial t}(x,t) \approx \frac{\partial T}{\partial t}(x_i, t_k)$$

13. Montrer que l'équation obtenue de la diffusion thermique peut être mise sous la forme

$$T_i^k = -cT_{i-1}^{k+1} + (1+2c)T_i^{k+1} - cT_{i+1}^{k+1}\,, \qquad \text{avec} \quad c = \frac{\Delta t\, D}{\Delta x^2}.$$

Cette équation est appelée schéma implicite car la température à l'instant t_k est exprimée en fonction de la température à l'instant ultérieur.

14. Mettre ce schéma sous forme matricielle.

15. Effectuer l'inversion du système obtenu à chaque itération en utilisant la fonction `solve` du sous-module `numpy.linalg` puis réaliser une représentation graphique avec les mêmes données que précédemment.

Corrections des exercices

Corrigé exo 5.0

On remarque que le tableau n'a que des uns sur la première ligne et la première colonne et qu'on peut alors remplir le tableau grâce à la relation

```
[ligne, col] = tab[ligne-1, col] + tab[ligne, col-1]
```

ce qui donne

```
def nbchemin(n):
    # on place des 1 sur les bords supérieurs et gauches du tableau
    tab = np.zeros((n + 1, n + 1), dtype=int)
    # initialisation élégante
    tab[:, 0] = 1
    tab[0, :] = 1
    for ligne in range(1, n + 1):
        for col in range(1, n + 1):
            tab[ligne, col] = tab[ligne - 1, col] + tab[ligne, col - 1]
    return tab
```

```
T = nbchemin(8)
print(T)
print(T[-1, -1])
```

```
[[   1    1    1    1    1    1    1    1    1]
 [   1    2    3    4    5    6    7    8    9]
 [   1    3    6   10   15   21   28   36   45]
 [   1    4   10   20   35   56   84  120  165]
 [   1    5   15   35   70  126  210  330  495]
 [   1    6   21   56  126  252  462  792 1287]
 [   1    7   28   84  210  462  924 1716 3003]
 [   1    8   36  120  330  792 1716 3432 6435]
 [   1    9   45  165  495 1287 3003 6435 12870]]
12870
```

On remarque que sur les diagonales figurent les différentes combinaisons de Pascal, ce qui n'est pas du tout un hasard !

Corrigé exo 5.1

0. En notant A la matrice du système, on calcule $A^n V$.

```
def puiss(A, n):
    p = A.shape[0]
    B = np.eye(p)
    for i in range(n):
        B = B.dot(A)
    return B

A = np.array([[7, 4, 5],
              [3, 0, 1],
              [0, 6, 4]])
V = np.array([.5, .5, 0])
```

```
A2 = puiss(1 / 10 * A, 2018)
res = A2.dot(V)

print(res)  # OK
```

```
[ 0.6  0.2  0.2]
```

1. Pour calculer A^n, nous avons utilisé $n-1$ multiplications matricielles. On peut utiliser l'exponentiation rapide pour avoir un coût en $O(\log n)$.

```
# mieux pour n grand l'exponentiation rapide
def puissrap(A, n):
    if n == 0:
        return np.eye(A.shape[0])
    elif n % 2 == 0:
        B = puissrap(A, n // 2)
        return B.dot(B)
    else:
        B = puissrap(A, n // 2)
        return B.dot(B).dot(A)
```

2. Avec le deuxième système, les coefficients de la matrices deviennent trop grands.

```
A2 = puiss(A, 2018)
res = A2.dot(V)
print(res)  # ça ne marche pas... Les coefficients explosent
```

```
[ nan  nan  nan]
```

Il vaut mieux calculer les vecteurs consécutivement.

```
# Deuxième essai
def puiss2(A, V, n):
    V1 = A.dot(V)
    for i in range(n):
        V1 = A.dot(V)
    return V1

res = puiss2(A, V, 2018)
print(res)
```

```
[ 5.5  1.5  3. ]
```

Corrigé exo 5.2

0.

```
import numpy as np

def decompLU(A0):
    n = A0.shape[0]

    L = np.eye(n)
    A = A0.copy()
```

```
    for k in range(n):
        for i in range(k + 1, n):
            a = A[i, k] / A[k, k]
            L[i, k] = a
            A[i, :] = A[i, :] - a * A[k, :]

    return L, A
```

1. Mise en pratique

```
A = np.array([[2, -1, 0],
              [-1, 2, -1],
              [0, -1, 2]], dtype=float)

L, U = decompLU(A)
print('L =', L)
print('U =', U)
```

```
L = [[ 1.   0.   0. ]
     [ 2.   1.   0. ]
     [ 1.   0.5  1. ]]
U = [[ 1.   2.   1. ]
     [ 0.  -2.  -1. ]
     [ 0.   0.   0.5]]
L = [[ 1.          0.          0.        ]
     [-0.5         1.          0.        ]
     [ 0.         -0.66666667  1.        ]]
U = [[ 2.         -1.          0.        ]
     [ 0.          1.5        -1.        ]
     [ 0.          0.          1.33333333]]
```

```
print(A - L.dot(U))  # petit test
```

```
[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]]
```

C'est bon.

2. On calcule X en résolvant $LX_0 = Y$ puis $UX = X_0$ qui sont deux systèmes triangulaires.

Corrigé exo 5.3

0. On définit la matrice `A` et on liste des vecteurs propres.

```
A = np.array([[1, 2, 3],
              [2, 4, 1],
              [3, 1, 0]])

import numpy.linalg as lg

vp, P = lg.eig(A)
print(max(vp), vp)
```

```
5.97344343324 [ 5.97344343 -2.58959802  1.61615459]
```

La matrice est diagonalisable avec des valeurs propres toutes distinctes.

```
print(lg.inv(P).dot(A).dot(P))
```

```
[[  5.97344343e+00   1.99840144e-15  -2.22044605e-16]
 [ -2.55351296e-15  -2.58959802e+00  -1.55431223e-15]
 [  4.99600361e-16  -2.22044605e-15   1.61615459e+00]]
```

Une base de vecteurs propres approchés est donnée par les colonnes de la matrice `P`.

```
P
```

```
array([[-0.53771649, -0.66787703,  0.51458843],
       [-0.74499437,  0.09059996, -0.66088957],
       [-0.39477127,  0.73873671,  0.54628172]])
```

1. On définit la fonction `normalise` qui normalise un vecteur puis la fonction `vecteurpuiss`.

```
def normcar(X):
    return X.dot(X)


def normalise(X):
    return 1 / np.sqrt(normcar(X)) * X


def vecteurpuiss(A, X, eps):
    Y = normalise(A.dot(X))
    while normcar(Y - X) > eps**2 and normcar(Y + X) > eps**2:
        X = Y
        Y = normalise(A.dot(Y))
    return Y
```

2. On teste la fonction :

```
X = np.array([1, 1, 1])
Y = vecteurpuiss(A, X, 1e-9)
print(Y)
```

```
[ 0.53771649  0.74499437  0.39477127]
```

Apparemment, il s'agit bien d'un vecteur propre de la plus grande valeur propre.

```
Z = A.dot(Y)
print(Z / Y)
```

```
[ 5.97344344  5.97344343  5.97344343]
```

3. On suit la définition de l'énoncé.

```
I = np.eye(3)

YL = Y.reshape(1, 3)
YC = Y.reshape((3, 1))
M = YC.dot(YL)

B = (I - M).dot(A).dot(I - M)
```

On a récupéré un vecteur propre de la deuxième plus grande valeur propre (en valeur absolue).

```
X = np.array([1, 1, 1])
Y = vecteurpuiss(B, X, 1e-9)
print(Y)
Z = B.dot(Y)
print(Z / Y)
```

```
[-0.66787703  0.09059996  0.73873671]
[-2.58959802 -2.58959803 -2.58959802]
```

Corrigé exo 5.4

0.

```
[['gobelin', 0, 3], ['hobgobelin', 1, 3], ['gobelours', 2, 4],
 ['groupe', 3, 4], ['groupe', 4, None]]
```

1. Il suffit de parcourir la liste et de compter le nombre des descendants de `i`.

```
def nb_fils(A, i):
    nb = 0
    for E in A:
        if E[2] == i:
            nb += 1
    return nb
```

2. On cherche dans la liste tous les groupes dont la troisième composante est `None`.

```
def orphelins(A):
    L = []
    for k in range(len(A)):
        if A[k][2] == None:
            L.append(k)
    return L
```

3. On crée le nouvel ancêtre, et on modifie l'ancêtre des groupes numéros `i` et `j`. On ne vérifie pas que `i` et `j` sont des groupes orphelins car on a supposé qu'ils le sont.

```
def nouvel_ancetre(A, i, j):
    k = len(A)
    A[i][2] = k
    A[j][2] = k
    A.append(["groupe", k, None])
```

4. On commence par introduire une fonction `dist` lisant dans la demi-matrice la distance entre deux groupes.

```
def dist(D, i, k):
    if i<k : return D[k][i]
    elif i==k : return 0
    else : return D[i][k]
```

```
def distance(D, i, j):
    L = []
    for k in range(len(D)):
        L.append(min(dist(D, i, k), dist(D, j, k)))
    return L
```

5. On teste tous les couples possibles.

```
def plus_proche(A, D):
    L = orphelins(A)
    i0, j0 = L[1], L[0]
    for i in range(len(L)):
        for j in range(i):
            if D[L[i]][L[j]] < D[i0][j0]:
                i0, j0 = i, j
    return i0, j0
```

6.

```
def regroupement(A, D):
    i, j = plus_proche(A, D)
    L = distance(D, i, j)
    nouvel_ancetre(A, i, j)
    D.append(L)
```

7. Tant qu'il reste au moins deux orphelins, on regroupe deux orphelins.

```
def arbre_philogenetique(L, D):
    A = [[L[k], k, None] for k in range(len(L))]
    X = orphelins(A)
    while len(X) != 1:
        regroupement(A, D)
        X = orphelins(A)
    return A
```

Corrigé exo 5.5

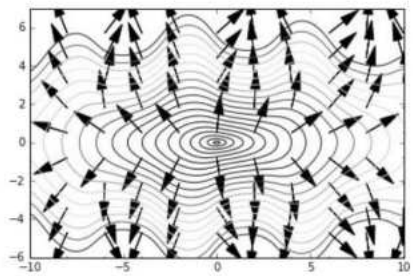
0. On obtient facilement :

```
def f(x, y):
    return 1 / 2 * x**2 + 7 / 2 * y**2 * (1 + .4 * np.arctan(np.sin(x)))
```

```
def grad(f, x, y, h):
    """ gradient unitaire """
    gx = (f(x + h, y) - f(x - h, y)) / (2 * h)
    gy = (f(x, y + h) - f(x, y - h)) / (2 * h)

    nn = np.sqrt(gx**2 + gy**2)
    return gx / nn, gy / nn
```

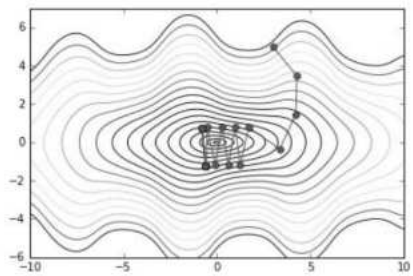
```
Xg = np.linspace(-10, 10, 11)
Yg = np.linspace(-6, 7, 11)

h = 1e-3

ax = plt.axes()
```

```
for x in Xg:
    for y in Yg:
        a, b = grad(f, x, y, h)
        ax.arrow(x, y, a, b, head_width=0.5, head_length=1, fc='k', ec='k')

plt.contour(XX, YY, ZZ, levels=[(k / 4)**3 for k in range(20)])
plt.show()
```



1.

```
h = 1e-3

alpha = 2  # pas constant
x0, y0 = 3, 5
Lx = [x0]
Ly = [y0]

# à pas constant alpha
x, y = x0, y0
for i in range(29):
    gx, gy = grad(f, x, y, h)
    x = x - alpha * gx
    y = y - alpha * gy
    Lx.append(x)
    Ly.append(y)
plt.plot(Lx, Ly, 'go-', label='constant')
plt.contour(XX, YY, ZZ, levels=[(k / 4)**3 for k in range(20)])
plt.show()
```

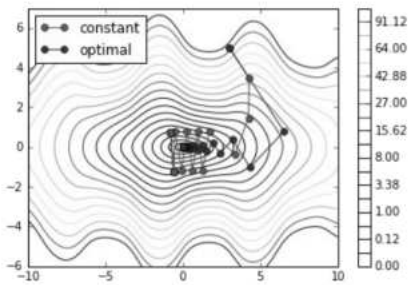


2.

```
# à pas optimal

def meilleuralpha(f, x, y, gx, gy, eps, cptmax=50000):
    alpha = eps
    F = f(x, y)
    fin = False
    cpt = 1
    while not fin and cpt < cptmax:
        Fnext = f(x - alpha * gx, y - alpha * gy)
        if F < Fnext:
            fin = True
        else:
            F = Fnext

        alpha += eps
        cpt += 1
    if cpt == cptmax:
        print("nombre maximum d'itérations dépassé")
    return alpha

Lxopt = [x0]
Lyopt = [y0]
x, y = x0, y0
eps = 0.01

for i in range(29):
    gx, gy = grad(f, x, y, h)
    alpha = meilleuralpha(f, x, y, gx, gy, eps)
    x = x - alpha * gx
    y = y - alpha * gy
    Lxopt.append(x)
    Lyopt.append(y)
```

3.

```
plt.plot(Lx, Ly, 'go-', label='constant')
plt.plot(Lxopt, Lyopt, 'ro-', label='optimal')

# tracé des lignes de niveau

X = np.linspace(-10, 10, 1001)
Y = np.linspace(-6, 7, 1001)
XX, YY = np.meshgrid(X, Y)
ZZ = f(XX, YY)

plt.contour(XX, YY, ZZ, levels=[(k / 4)**3 for k in range(20)])

plt.legend()
plt.colorbar()
plt.show()
```



Corrigé exo 5.6

```
'''
algorithme de Floyd-Steinberg
8 couleurs...
'''

# *       x 7/16
# 3/16 5/16 1/16


def convoct(a):
    '''
    a est un int, on renvoie un np.uint8
    '''
    if a < 0:
        return np.uint8(0)
    if a > 255:
        return np.uint8(255)

    return np.uint8(a)


def floyd(T, seuil):
    R = T.copy()
    a, b = R.shape
    for i in range(a - 1):
        for j in range(1, b - 1):
            if R[i, j] >= seuil:
                erreur = R[i, j] - 255
                R[i, j] = 255
            else:
                erreur = R[i, j]
                R[i, j] = 0
            R[i, j + 1] = convoct(R[i, j + 1] + erreur * 7 // 16)
            R[i + 1, j - 1] = convoct(R[i + 1, j - 1] + erreur * 3 // 16)
            R[i + 1, j] = convoct(R[i + 1, j] + erreur * 5 // 16)
            R[i + 1, j + 1] = convoct(R[i + 1, j + 1] + erreur * 1 // 16)

    return R[:a - 1, 1:b - 1]
```

Corrigé exo 5.7

Voici le code de la fonction `photomaton`.

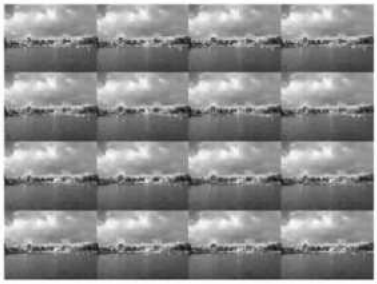
```
def photomaton(P):
    a, b, n = P.shape

    aa = a // 2
    bb = b // 2
    a = aa * 2
    b = bb * 2

    T = np.zeros((a, b, 3), dtype=np.uint8)

    for i in range(aa):
        for j in range(bb):
            T[i, j] = P[2*i, 2*j]
            T[i + aa, j] = P[2*i+1, 2*j]
```

```
            T[i, j + bb] = P[2*i, 2*j+1]
            T[i + aa, j + bb] = P[2*i+1, 2*j+1]
    return T
```

Voici un exemple

```
P1 = litimage('image256.png')
T1 = P1.copy()
affiche(T1)
```

```
T1 = photomaton(T1)
affiche(T1)
```



Corrigé exo 5.8

0. On définit :

```
def niveau_gris(M):
    n, p, q = np.shape(M)
    P = np.zeros((n, p))
    for i in range(n):
        for j in range(p):
            P[i][j] = int(0.299 * M[i][j][0] + 0.587 *
                          M[i][j][1] + 0.114 * M[i][j][2])
    return P
```

1.

```
def stegano(image, chaineaencoder):
    U = [ord(x) for x in chaineaencoder] + [128]
    code = ''
    for n in U:
        codage = ''
        for i in range(8):
            codage += str(n // (2**(7 - i)))
            n %= 2**(7 - i)
        code += codage
    newim = np.zeros(np.shape(im))
    k = 0
    for i in range(np.shape[0]):
        for j in range(np.shape[1]):
            if k < len(code):
                newim[i][j] = 2 * im[i][j] // 2 + int(code[k])
```

```
                k += 1
            else:
                newim[i][j] = im[i][j]
    return newim
```

2.

```
def unstegano(image):
    T = []
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            T.append(image[i][j] % 2)
    ch = ""
    for i in range(len(T) // 8):
        s = 0
        for j in range(8):
            s += T[8 * i + j] * 2**(7 - j)
        ch += chr(s)
    return ch.split(chr(128))[0]
```

3.

```
from PIL import Image


def decode(fichier):
    f = Image.open(fichier)
    im = np.array(f)
    return unstegano(im)
```

Corrections des TP

Corrigé TP 5.2

0. On peut utiliser `randrange(n)` ou `randint(0, n-1)`.

```
def rand_couple(n):
    i = random.randrange(n)
    j = random.randrange(n)
    return i, j
```

1.

```
def rand_pop(L):
    return L.pop(random.randrange(len(L)))
```

2. Le contenu d'une case varie selon que `i` et `j` sont de même parité ou de parités différentes. Il suffit donc de s'intéresser à la parité de `i+j`.

```
def quadrillage(n):
    M = np.zeros((n, n), dtype=np.uint8)
    for j in range(n):
        for i in range(n):
                M[i, j] = 1+(i+j)%2
    return M
```

3. Comme nous sommes dans un modèle torique, chaque case est adjacente à 8 autres cases. Nous n'avons donc pas besoin de faire des cas particuliers selon que la case est au milieu ou au bord de la matrice ; à la place, nous utilisons le modulo (`%`).
`V` est une liste de trois compteurs. `V[0]` compte le nombre de zéros, `V[1]` le nombre de uns, et `V[2]` le nombre de deux.
Pour éviter d'avoir un `if`, on compte comme voisin la case elle-même (et on corrige à la fin).

```
def voisinage(M, n, i, j):
    V = np.zeros(3, dtype=np.uint8)
    for i2 in range(i - 1, i + 2):
        for j2 in range(j - 1, j + 2):
            V[M[i2 % n, j2 % n]] += 1 # On ajoute 1 au compteur de la bonne couleur
    V[M[i, j]] -= 1 # On retire la case (i,j) qui a été comptée en trop
    return V
```

4. Pour qu'un individu soit mécontent, il faut que le nombre des individus de sa couleur (`v[M[i, j]]`) représente un tiers ou moins de ses voisins. Pour obtenir le nombre de voisins, on fait la somme des voisins des deux couleurs.
Bien évidemment, on n'oublie pas de tester que la case est non vide (il n'y a pas de mécontent si la case est vide).

```
def pas_content(M, n, i, j):
    v = voisinage(M, n, i, j)
    return M[i, j]!=0 and v[M[i, j]] <= (v[1]+v[2])//3
```

5. La matrice est initialisée en trois temps. D'abord on fait appel à `quadrillage`.

Ensuite, on tire au hasard des positions dans la matrice (en utilisant `rand_couples`), et on vide la case si elle n'a pas déjà été vidée. Si la case a déjà été vidée, on recommence.
Enfin, on choisit au hasard dans la liste des positions vides des cases qu'on remplit au hasard avec `randint`. La fonction `randint` est piégeuse, cf. l'avertissement page 23.

```
def init_matrice(n, nbvides, nbnouv):
    M = quadrillage(n)
    v = 0
    LV = []
    while v < nbvides:
        i, j = rand_couple(n)
        if M[i, j] != 0:
            M[i, j] = 0
            v += 1
            LV.append((i, j))
    for k in range(nbnouv):
        i, j = rand_pop(LV)
        M[i, j] = random.randint(1, 2)
    LM = []
    for i in range(n):
        for j in range(n):
            if pas_content(M, n, i, j):
                LM.append((i,j))
    return M, LV, LM
```

6. Pour mettre à jour la liste des mécontents, il faut considérer deux cas : les contents devenus mécontents et les mécontents devenus contents.

```
def nouveaux_pas_contents(M, n, LM, i, j):
    for i0 in range(i - 1, i + 2):
        for j0 in range(j - 1, j + 2):
            i0 = i0 % n
            j0 = j0 % n
            if pas_content(M, n, i0, j0) and (i0, j0) not in LM:
                LM.append((i0, j0))
            elif pas_content(M, n, i0, j0) == False and (i0, j0) in LM:
                LM.remove((i0,j0))
```

7. Comme deux cases de la matrices sont modifiées, deux appels à `nouveaux_pas_contents` sont effectués.

```
def deplace(M, n, LV, LM):
    i1, j1 = rand_pop(LM)
    i2, j2 = rand_pop(LV)
    if M[i2, j2] != 0:
        print(M[i2, j2])
    M[i2, j2] = M[i1, j1]
    M[i1, j1] = 0
    LV.append((i1, j1))
    nouveaux_pas_contents(M, n, LM, i2, j2)
    nouveaux_pas_contents(M, n, LM, i1, j1)
```

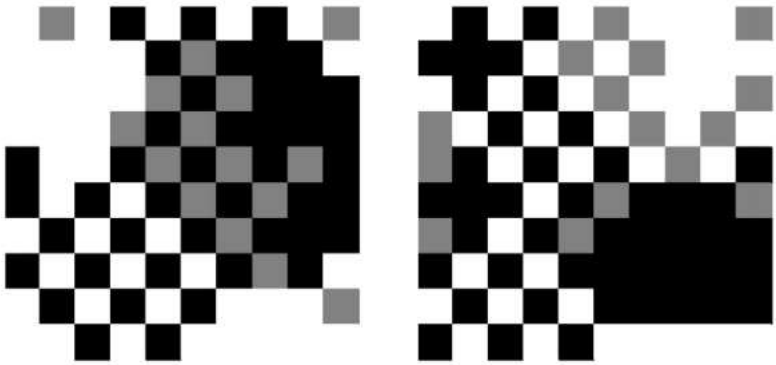
8. On fait évoluer la matrice jusqu'à ce qu'on obtienne une ville stable (sans mécontents) ou qu'on atteigne le nombre maximal d'itérations [1].

```
def evolution(M, n, LV, LM, limit=1000):
    k = 0
    while(LM != [] and k < limit):
        deplace(M, n, LV, LM)
```

1. Cette technique combinant un critère de convergence à une limite au nombre d'itérations est classique.

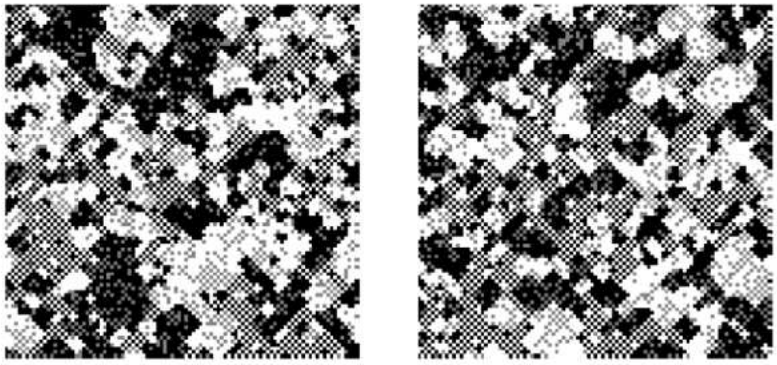
```
            k += 1
        print (k)
        return k != limit
```

9. En utilisant les paramètres de l'énoncé, on obtient des résultats de ce type.



On observe la formation de ghettos blancs (des zones blanches/grises) et des ghettos noirs (des zones noires/grises) mais aussi des parties de la ville qui sont restées à l'état initial (en damier).

10. On observe la même chose avec des villes plus grandes.



Corrigé TP 5.3

0. On obtient :

```
freq, T = L
print('Fréquence (hertz):', freq)
n, nbcanaux = T.shape
typ = T.dtype
print('Durée (en seconde): {:.2f}'.format(n / freq))
if nbcanaux == 2:
    print('stéréo')
elif nbcanaux == 1:
    print('mono')
else:
    print('multicanaux, nombre =', nbcanaux)
print('résolution:', typ)

duree = n / freq

print('durée en secondes: {:.2f}'.format(duree))
```

```
Fréquence (hertz): 44100
Durée (en seconde): 49.01
stéréo
résolution: int16
durée en secondes: 49.01
```

1.

```
t = np.linspace(0, 2, 2 * freq + 1)    # sur 2 secondes
t = t[:-1]

freqson = 440  # la
la = freqson  # la à 440 Hz
demiton = 2**(1 / 12)
sol = la / demiton**2
do = la / demiton**9
doa = 2 * do
mi = la / demiton**5


X = 32700 * np.cos(t * 2 * np.pi * do)
X1 = np.array(X, dtype=np.int16)

X = 32700 * np.cos(t * 2 * np.pi * mi)
X2 = np.array(X, dtype=np.int16)

X = 32700 * np.cos(t * 2 * np.pi * sol)
X3 = np.array(X, dtype=np.int16)

X = np.concatenate((X1, X2, X3))

creefichier('mestroisnotes.wav', freq, X)
```

2.

```
def creenote(vol, freq, freqnote, duree):
    t = np.arange(0, duree, 1 / freq)    # sur duree secondes

    X = np.int16(vol * np.cos(t * 2 * np.pi * freqnote))

    return X

Lnote = [do, mi, sol, doa, sol, mi, do]

duree = .5
vol = 32700 / 8

X = np.array([], dtype=np.int16)

for freqnote in Lnote:
    Y = creenote(vol, freq, freqnote, duree)
    X = np.concatenate((X, Y))


creefichier('domisol.wav', freq, X)

#os.system('vlc testson.wav')
```

3.

```
def creenotetimbre(vol, freq, freqnote, duree, timbre=[]):
    t = np.arange(0, duree, 1 / freq)    # sur duree secondes

    N = 1 + sum(timbre)
    vol = vol / N
    X = creenote(vol, freq, freqnote, duree)
    for amp in timbre:
        freqnote = freqnote * 2
        X = X + creenote(vol * amp, freq, freqnote, duree)

    return X
```

```
X = np.array([], dtype=np.int16)

timbre = [(1 + (-1)**n) * .15 * .8**n for n in range(1, 19)]

for freqnote in Lnote:
    Y = creenotetimbre(vol, freq, freqnote, duree, timbre=timbre)
    X = np.concatenate((X, Y))

creefichier('domisoltimbre.wav', freq, X)
```

4.

```
freqson = 440  # la
la = freqson  # la à 440 Hz
demiton = 2**(1 / 12)
do = la / demiton**9

dico = {}
note = do
for l in Lgammechromatique:
    dico[l] = note
    note = note * demiton
```

5.

```
for d in Lgammechromatique:
    dico[d + '2'] = dico[d] * 2

for d in Lgammechromatique:
    dico[d + '0'] = dico[d] / 2

dico['pause'] = 0
```

6.

```
# Gamme chromatique

duree = .3

X = np.array([], dtype=np.int16)
for note in Lgammechromatique:
    Y = creenotetimbre(vol, freq, dico[note], duree, timbre=timbre)
    X = np.concatenate((X, Y))

creefichier('gammechromatique.wav', freq, X)

#os.system('vlc gammechromatique.wav')
```

7.

```
def listenote(L):
    L = L.split('\n')
    Note = []
    for t in L:
        Note.extend(t.strip().split(' '))
    return Note
```

```
Note2maindroite = listenote(Part2maindroite)
Note3maindroite = listenote(Part3maindroite)

Note1maingauche = listenote(Part1maingauche)
Note2maingauche = listenote(Part2maingauche)
```

```
Note3maingauche = listenote(Part3maingauche)

# pas de main droite pour la première partie
Note1maindroite = ['pause'] * len(Note1maingauche)
```

8. Pour la main droite

```
# main droite

dblecroche = .15
duree = dblecroche

X = np.array([], dtype=np.int16)
for note in Note2maindroite:
    Y = creenotetimbre(vol, freq, dico[note], duree, timbre=timbre)
    X = np.concatenate((X, Y))

duree *= 2    # on passe aux croches
for note in Note3maindroite:
    Y = creenotetimbre(vol, freq, dico[note], duree, timbre=timbre)
    X = np.concatenate((X, Y))

creefichier('maindroite.wav', freq, X)

#os.system('vlc maindroite.wav')
```

et pour la main gauche

```
# main gauche

duree = dblecroche

X2 = np.array([], dtype=np.int16)
for note in Note1maingauche:
    Y = creenotetimbre(vol, freq, dico[note], duree, timbre=timbre)
    X2 = np.concatenate((X2, Y))

duree *= 2
for note in Note2maingauche:
    Y = creenotetimbre(vol, freq, dico[note], duree, timbre=timbre)
    X2 = np.concatenate((X2, Y))

duree /= 2
for note in Note3maingauche:
    Y = creenotetimbre(vol, freq, dico[note], duree, timbre=timbre)
    X2 = np.concatenate((X2, Y))

creefichier('maingauche.wav', freq, X2)

#os.system('vlc maingauche.wav')
```

9. On réutilise les listes `X` et `X2` précédentes.

```
Xp = np.array([], dtype=np.int16)
duree = dblecroche

for note in Note1maindroite:
    Y = creenotetimbre(vol, freq, dico[note], duree, timbre=timbre)
    Xp = np.concatenate((Xp, Y))

X = np.concatenate((Xp, X))

XX = np.vstack((X, X2))
XX = XX.T
```

```
creefichier('extraitfuguebach.wav', freq, XX)

os.system('vlc extraitfuguebach.wav')
```

Corrigé TP 5.4

0.

```
def PasEspace(L, N):
    return L / (N - 1)
```

1. C'est immédiat pour la dérivée partielle d'ordre 1, pour l'ordre 2, on peut par exemple écrire

$$T(x \pm h, t) = T(x,t) \pm h\frac{\partial T}{\partial x} + \frac{h^2}{2}\frac{\partial^2 T}{\partial x^2} + \underset{h\to 0}{o}\left(h^2\right)$$

D'où

$$\frac{\partial^2 T}{\partial x^2}(x,t) = \frac{T(x+h,t) + T(x-h,t) - 2T(x,t)}{h^2} + \underset{h\to 0}{o}(1)$$

2. On remplace les dérivées partielles par leur approximation discrétisée.

3. Les conditions aux limites imposent $\forall n, T_0^n = T_{N-1}^n = 0$.

4.

```
import numpy as np


def initialisation(L, N):
    X = np.linspace(0, L, N)
    return [300 + 100 * x / L for x in X]
```

5.

```
def transition(T, c, N, Tbord=300):
    U = [0 for i in range(N)]
    U[0] = Tbord
    U[N - 1] = Tbord
    for j in range(1, N - 1):
        U[j] = T[j] + c * (T[j - 1] - 2 * T[j] + T[j + 1])
    return U
```

6.

```
import matplotlib.pyplot as plt

Delta_t = 0.01
N = 101
L = 0.5
D = 1.2 * 10**(-4)
pas = PasEspace(L, N)
c = Delta_t * D / pas**2

plt.figure("diffusion thermique")

T = initialisation(L, N)
Z = [k * pas for k in range(N)]
Nt = int(10 * 60 / Delta_t)
sixmin = int(6 / Delta_t)

for i in range(Nt):
    if i % sixmin == 0:
```

```
        plt.plot(Z, T)
    T = transition(T, c)
plt.show()
```

7. Il suffit encore de remplacer les dérivées partielles par leurs approximations.

8.

```
def Initial2d():
    M = [[300. for i in range(101)] for j in range(101)]
    for i in range(25, 76):  # 0.1 / pas puis + 0.2 / pas
        for j in range(20, 81):  # (0.2 - 0.12) / pas, pas = 0.004
            M[i][j] = 400
    return M
```

9.

```
def transition(T, D, Delta_t, Delta_x, Delta_y):
    M = [[300. for i in range(101)] for j in range(101)]
    for i in range(1, 100):
        for j in range(1, 100):
            M[i][j] = T[i][j] + \
                Delta_t * D * ((T[i + 1][j] - 2 * T[i][j] +
                                T[i - 1][j]) / Delta_x**2 +
                               (T[i][j + 1] - 2 * T[i][j] + T[i][j - 1]) / Delta_y**2)
    return M
```

10.

```
def temperature(n):
    D, Delta_t, Delta_x, Delta_y = 0.00012, 0.01, 0.004, 0.004
    T = Initial2d()
    for t in range(n):
        T = transition(T, D, Delta_t, Delta_x, Delta_y)
    return T
```

11.

```
XX = np.linspace(-0.2, 0.2, 101)
YY = np.linspace(-0.2, 0.2, 101)
```

12.

```
# animation
D, Delta_t, Delta_x, Delta_y = 0.00012, 0.01, 0.004, 0.004

def tempsuiv(T):
    return transition(T, D, Delta_t, Delta_x, Delta_y)


XX = np.linspace(-0.2, 0.2, 101)
YY = np.linspace(-0.2, 0.2, 101)
T = Initial2d()

plt.figure()

Nt = int(1 / Delta_t)
for i in range(10 * Nt):
    T = np.array(tempsuiv(T))
    if i % Nt == 0:
        plt.clf()
        plt.pcolormesh(XX, YY, T, shading='flat')
        #plt.imshow(T, interpolation='nearest')
```

```
        plt.axis('image')
        plt.colorbar()
        plt.pause(.001)

plt.pause(100)
```

13.

$$\frac{T_j^{n+1} - T_j^n}{\Delta t} \approx \frac{T_{j-1}^{\mathbf{n+1}} - 2\,T_j^{\mathbf{n+1}} + T_{j+1}^{\mathbf{n+1}}}{(\Delta x)^2}$$

d'où pour $j \in [\![1, N-2]\!]$,

$$T_j^n = -cT_{j-1}^{n+1} + (1+2c)\,T_j^{n+1} - cT_{j+1}^{n+1}$$

14. ce qui s'écrit avec le bord,

$$T^n + cV^n = \widetilde{M}T^{n+1}$$

avec $\widetilde{M} \in \mathfrak{M}_{N-2}(\mathbb{R})$ et $V \in \mathfrak{M}_{N-2,1}(\mathbb{R})$,

$$M = \begin{pmatrix} 1+2c & -c & & & (0) \\ -c & 1+2c & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & -c \\ (0) & & & -c & 1+2c \end{pmatrix}, V = \begin{pmatrix} T_0^n \\ 0 \\ \vdots \\ 0 \\ T_{N-1}^n \end{pmatrix}, T^n = \begin{pmatrix} T_1^n \\ T_2^n \\ \vdots \\ T_{N-2}^n \\ T_{N-2}^n \end{pmatrix}$$

15.

```
Nx = 101   # nombre de points x
Nt = 5001   # nombre points t
dx = 1/(Nx-1)  # pas de x
dt = .1/(Nt-1)  # pas de t

L = 0.5
D = 1.2*10**(-4)

X = np.linspace(0, L, Nx)
T = np.zeros((Nt, Nx))

T[:, :] = 300
T[0, :] = 300 + 100 * X / L

c = dt / dx**2

M = np.zeros((Nx-2, Nx-2))
for i in range(Nx-3):
    M[i, i+1] = -c
    M[i+1, i] = -c
```

```
for i in range(Nx-2):
    M[i, i] = 1 + 2*c

V = np.zeros(Nx-2)
V[0] = 300
V[-1] = 300
# T[n, 0] = 300 et T[n, Nx-1] = 300

for n in range(Nt-1):
    T[n+1, 1:Nx-1] = np.linalg.solve(M,
                          T[n, 1:Nx-1] + c*V)
# on passe 400 étapes de temps
    if n % 400 == 399:
        plt.plot(X, T[n, :], lw=2)

plt.plot(X, T[0, :], 'k--', lw=5)

plt.xlabel('x', fontsize=26)
plt.ylabel('T', fontsize=26, rotation=0)
plt.show()
```

CHAPITRE 6

Bases de données

L'essentiel du cours

■ 0 Représentation de bases de données

Lorsque les données à représenter d'un problème deviennent complexes et volumineuses, les structures de données étudiées précédemment (listes et listes de listes principalement) deviennent insuffisantes.
Une solution alternative est de regrouper les données dans une ou plusieurs tables liées entre elles. Ce type de structure s'appelle **modèle relationnel**.

Définition

Un **domaine** est un ensemble de valeurs que peut prendre une donnée (par exemple des entiers, des chaînes de caractères, comme la notion de **type** en Python).
Un **attribut** (ou **champ**) (une colonne en SQL) est un couple `nom:domaine`.

On appelle **schéma relationnel**, un ensemble ordonné d'attributs de la forme $S=(A_1,...,A_n)$ où les A_i sont des attributs deux à deux distincts.
Exemple : `S=(nom: str, prenom: str, age: int)`.

Une **relation** (une **table** SQL mais sans doublons de lignes) associée à un schéma relationnel $S=(A_1,...,A_n)$ est un ensemble fini de n-uplets de $\text{dom}(A_1) \times ... \times \text{dom}(A_n)$
On note R(S) la relation R pour dire qu'elle est associée au schéma relationnel S.

Les éléments de R sont appelés les **valeurs** ou les **enregistrements** de la relation. Leur nombre (fini) est appelé le cardinal de R et est noté # R.
Exemple : `R(S)={('Dupont', 'Albert', 45),('Gaborit', 'René', 37)}`.

Considérons l'exemple du stockage des informations et de la gestion d'un forum Internet. On peut distinguer plusieurs entités (les utilisateurs, les messages...) qui seront représentées chacune par une relation. Ainsi, les utilisateurs seront représentés par une relation, dont le schéma relationnel pourrait être en simplifiant à l'extrême :

```
utilisateurs ((Pseudonyme,str),(Date_inscription,date),(Mode_de_Passe,str),(email,str))
```

On souhaite également garder en mémoire l'ensemble des messages écrits sur le forum. Un message est évidemment rédigé par un seul utilisateur. Cet utilisateur va apparaître dans tout enregistrement dans la nouvelle table, aussi il est intéressant de pouvoir identifier les utilisateurs de manière aisée et efficace : la notion de clé primaire vient répondre à cette problématique.

Définition

Soit R(S) une relation pour un schéma relationnel S et K $\subset$ S.
On note que $t_1(K)$ désigne le uplet partiel construit à partir de $t_1 \in$ R ne contenant que les éléments des domaines contenus dans K.
On dit que K est une **clé** pour R si pour toutes valeurs $t_1, t_2 \in$ R telles que $t_1(K) = t_2(K)$ alors $t_1 = t_2$ (unicité de l'enregistrement).
Lorsque K est réduit à un seul attribut, on dit que l'on a une **clé primaire**.

Une clé primaire est un attribut qui permet de **caractériser** un enregistrement. En termes plus intuitifs, une clé permet d'identifier de manière unique un enregistrement.

Si on admet qu'un pseudonyme donné ne peut être utilisé que par un seul utilisateur, cet attribut peut jouer le rôle de clé primaire. De même, si on impose qu'un seul compte peut être ouvert à l'aide d'une adresse email, cet attribut peut également jouer le rôle de clé primaire.
Dans la table `messages`, il n'y a pas d'attribut qui peut jouer le rôle de clé primaire naturelle. Il y a néanmoins une solution simple et qui est utilisée pour toutes les relations dans la plupart des bases de données : il s'agit de prendre une clé primaire artificielle, qui est un nombre (souvent appelé **id**) et qui est auto-incrémenté par le gestionnaire de base de données.
Si on ne souhaite pas imposer de contraintes sur l'usage de pseudonymes et d'emails, cette solution peut également être adoptée dans la première table.

Les schémas relationnels deviennent alors :

```
utilisateurs
(id_usr,int),(Pseudonyme,str),(Date_inscription,date),(Mode_de_Passe,str),(email,str))
messages
((id_msg,int),(Date,date),(Heure,heure),(id_u,int),(Contenu,str))
```

Définition

Un attribut dans une relation R'(S') qui sert à caractériser un enregistrement dans une relation R(S) et qui est une clé primaire de R est appelé **clé étrangère** dans la relation R'.

Ainsi id_user est une clé primaire dans la relation Utilisateurs et est une clé étrangère dans la table Messages.

Il existe des techniques spécifiques pour modéliser des bases de données. Celles-ci doivent obéir à certaines spécifications, décrites par les formes normales pour éviter les anomalies de lecture, la redondance des données et la contre-performance.
Sans entrer dans les détails des formes normales, citons quelques principes élémentaires :

- principe d'atomicité : aucune subdivision de l'information initiale n'apporte une information supplémentaire ou complémentaire ;
- constance dans le temps : ainsi, on ne crée pas un attribut âge mais un attribut date de naissance ;
- pas de lien fonctionnel dans la clé : un attribut non clé ne dépend pas d'une partie de la clé.

Pour modéliser par exemple les parcours des élèves d'un lycée en cours d'année, on créera trois tables, une table `élève` qui contiendra leurs informations fixes, une table `classes` qui contiendra

les noms et années des classes, et une table `eleveclasse` qui contiendra des clés dans les tables précédentes et qui permettra de déterminer par quelles classes sont passés tels élèves.

■ 1 Algèbre relationnelle

L'algèbre relationnelle donne un cadre mathématique rigoureux aux opérations que l'on peut effectuer sur une ou plusieurs relations.
Les opérations sur une seule table vont avoir comme rôle d'éliminer certaines lignes ou certaines colonnes, ainsi que de changer le nom des colonnes.

Définition

Soit R(S) une relation de schéma S et X ⊂ S. On appelle **projection** de R selon X la relation

$$\Pi_X(R) = \{e(X) \mid e \in R\}$$

où $e(X)$, comme précédemment, désigne le uplet partiel construit à partir de e ne contenant que les éléments des domaines contenus dans X.

La projection a comme effet d'éliminer certaines colonnes dans une table. Le schéma de $\Pi_X(R)$ est donc X. Une projection ne contient pas forcément autant de valeurs que la relation de départ : en effet, plusieurs valeurs peuvent être fusionnées.

Définition

Soit $R(S)$ une relation de schéma S. Soit E une **expression logique** sur S, c'est-à-dire une expression composée d'attributs de S et de valeurs dans le domaine de ces attributs, dont la valeur pour chaque enregistrement est soit vraie soit fausse.
On appelle **sélection** de R selon E la relation :

$$\sigma_E(R) = \{\text{e} \in \text{R} \mid \text{E est vraie pour e}\}$$

La sélection a comme effet d'éliminer certaines lignes dans une table. $\sigma_E(R)$ a le même schéma que R.

Si on souhaite éliminer à la fois des lignes et des colonnes d'une table, on compose une sélection par une projection. Le langage SQL permet de le faire en une seule commande.

Il est possible, souvent pour des raisons pratiques afin de lever une ambiguïté, de renommer un attribut d'une relation à l'aide d'un opérateur, dit de renommage. La relation obtenue après avoir effectué cette opération est alors identique à la relation de départ, mis à part le schéma qui a été changé pour présenter le nouveau nom :

Définition

Soit S=(A_1,...,A_n) un schéma, i ∈ $[\![1, n]\!]$ et B un attribut tel que dom(B)=dom(A_i).
On note $S_{A_i \leftarrow B}$ = (A_1,...,A_{i-1},B,A_{i+1},...,A_n) le schéma déduit de S en **renommant** A_i en B.

Les opérations peuvent se faire sur deux tables ayant un même schéma relationnel. Elles sont alors à rapprocher des opérations ensemblistes usuelles :

Définition

Soit $R_1(S)$ et $R_2(S)$ deux relations de même schéma S.

On appelle **union** de R_1 et de R_2 la relation de schéma S dont l'ensemble des valeurs est constitué des valeurs comprises dans R_1 ou dans R_2. Cette relation est notée $R_1 \cup R_2$.

On appelle **intersection** de R_1 et de R_2 la relation de schéma S dont l'ensemble des valeurs est constitué des valeurs comprises dans R_1 et dans R_2. Cette relation est notée $R_1 \cap R_2$.

On appelle **différence** de R_1 et de R_2 la relation de schéma S dont l'ensemble des valeurs est constitué des valeurs comprises dans R_1 mais pas dans R_2. Cette relation est notée $R_1 \setminus R_2$.

Définition

Soit R(S) et R'(S') deux relations de schémas disjoints, leur produit cartésien noté R × R' est :
($(v_1,...,v_n,v'_1,...,v'_m)$ | $(v_1,...,v_n) \in R$ et $(v'_1,...,v'_m) \in R'$)
Son schéma est $S \uplus S' = (A_1,...,A_n,B_1,...,B_m)$ où $S = (A_1,...,A_n)$ et $S' = (B_1,...,B_m)$.

R × R' contient donc l'ensemble des possibilités d'association entre une valeur de R et une valeur de R'. La notation S ⊎ S' rappelle qu'il ne s'agit pas seulement de prendre l'union des schémas mais aussi de s'assurer qu'ils soient disjoints. Notons qu'il est toujours possible de s'y ramener par l'intermédiaire d'un renommage des attributs présents dans les deux schémas.
On a : # (R × R') = # R × # R'

Définition

Soit R(S) et R'(S') deux relations de schémas respectifs S et S' disjoints (pour simplifier).
Soit K une expression logique sur S et S'.
La **jointure symétrique** selon K de R et R' notée $R \bowtie_K R'$ est la relation de schéma S ∪ S' définie par

$$R \bowtie_K R' = \sigma_K(R \times R')$$

Cette construction permet de construire les tables « de travail » à partir des tables de référence reliant les clés étrangères aux clés primaires entre tables. Par exemple, si l'on veut construire la table des messages avec toutes les informations de l'utilisateur sur une même ligne, on utilisera la jointure

$$\texttt{utilisateurs} \bowtie_{\text{utilisateurs.id\_usr=messages.id\_u}} \texttt{messages}$$

Les avantages sont :

- on évite les doublons (si par exemple il fallait corriger l'orthographe d'un utilisateur),
- on utilise une taille réduite de mémoire de stockage.

Un peu plus exotique et moins utile :

Définition

La **division cartésienne** R ÷ R' est la plus grande relation, vis-à-vis de l'inclusion, telle qu'il existe une relation R" vérifiant :

$[(R \div R') \times R'] \cup R'' = R$ (ou $[R' \times (R \div R')] \cup R'' = R$) et
$[(R \div R') \times R'] \cap R'' = \varnothing$ (ou $[R' \times (R \div R')] \cap R'' = \varnothing$)

Considérons un exemple :

Plat

Type Viande	Type Accompagnement
Steak	Frites
Steak	Salade
Poulet	Frites
Poulet	Légumes
Magret	Frites
Magret	Légumes

Viande

Type Viande
Steak
Poulet
Magret

La division cartésienne de Plat par Viande a comme schéma relationnel Type Accompagnement. Elle est constituée d'un seul enregistrement : Frites. En effet l'ensemble des associations de viandes avec Frites existe dans la table Plat, mais ce n'est pas le cas pour Salade ou Légumes.

Il est enfin utile de pouvoir effectuer des **opérations** sur certaines colonnes d'une relation. C'est possible à l'aide des opérateurs d'**agrégation** qui portent sur toute une table, ou sur des groupes constitués à partir d'une table suivant des critères logiques.

- les fonctions **min** et **max** qui retournent respectivement le minimum et le maximum des valeurs d'une colonne (contenant des valeurs numériques)
- la fonction **somme** qui permet de sommer les valeurs d'une colonne (contenant des valeurs numériques),
- la fonction **moyenne** qui permet de calculer la moyenne des valeurs d'une colonne (contenant des valeurs numériques),
- la **fonction de comptage** qui retourne le nombre de valeurs d'une colonne.

■ 2 Requêtes et langage SQL

Définition

Le langage **SQL** permet d'interroger les bases de données en traduisant les opération de l'algèbre relationnelle par des mots-clés simples. C'est un langage normalisé, mais chaque SGBD agrémente ce langage d'autres mots-clés non définis dans la norme.

Le langage SQL est insensible à la casse, néanmoins il est coutumier d'écrire les mots-clés en majuscule, ce que nous ferons dans cet ouvrage. Il en est de même pour les noms de tables, de colonnes, de valeurs, etc.

Pour illustrer cette partie de chapitre, nous utiliserons une base de données fictive d'un vendeur de véhicules d'occasion. Cette base de données est constituée de quatre tables détaillées ci-dessous et organisée de manière à éviter les redondances d'information selon les règles énoncées ci-dessous.

vehicules

id	desig	idmod	couleur	puiss	annee	km	prix	idvend	idach
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	

modeles

id	nom	idmarque
⋮	⋮	⋮

marques

id	nom
⋮	⋮

clients

id	nom	prenom	tel	email
⋮	⋮	⋮	⋮	⋮

Définition

L'instruction de base pour l'interrogation d'une base de données en SQL est constituée du mot-clé `SELECT` suivi du mot-clé `FROM`. Il s'agit de l'opération de **projection** sur les colonnes définies après `SELECT` sur la table définies après `FROM`.

```
SELECT nom, prenom
FROM client;
/* Cette instruction renvoie la liste de tous clients de la table clients
en donnant les informations stockées dans les colonnes nom et prenom */
```

On peut placer le caractère `*` après le mot-clé `SELECT` afin de projeter toutes les colonnes des tables indiquées après `FROM`.

```
SELECT *
FROM client;
/* Renvoie toutes les colonnes de la table clients. */
```

L'utilisation du caractère `*` est déconseillée car elle est gourmande en espace de stockage. Il est préférable de ne projeter que les colonnes utiles.

En SQL, une requête doit en principe se terminer par un point-virgule « ; »

On peut utiliser le mot-clé `DISTINCT` immédiatement après `SELECT` pour n'afficher que les valeurs différentes du résultat de la projection.

```
SELECT DISTINCT annee
FROM vehicules;
/* Renvoie la liste des annees de fabrication des vehicules présents en n'affichant qu'une fois chaque annee */
```

L'opérateur `SELECT` en SQL représente l'opérateur de projection Π de l'algèbre relationnelle. C'est l'opérateur `WHERE` qui réalise la sélection σ.

Définition

La réalisation de la **sélection** se fait avec le mot-clé `WHERE` après lequel on place une expression logique permettant de sélectionner les lignes à garder.

```
SELECT desig
FROM vehicules
WHERE annee > (YEAR(CURDATE()) - 5) AND km < 50000 ;
/* Cette instruction renvoie la liste de tous les véhicules de la table vehicules ayant moins de 5 ans
et affichant moins de 50 000 km au compteur, en donnant uniquement leur désignation. */
```

Ici nous avons utilisé les fonctions `YEAR` et `CURDATE` qui font partie des nombreuses fonctions SQL que nous ne pouvons détailler dans ce livre. L'avantage de cette syntaxe est de ne pas avoir à modifier la requête tous les ans.

Une liste non exhaustive des expressions logiques utilisables en SQL est donnée dans le tableau suivant :

Opérateurs	désignation
<, <=, >=, <>, =	comparaison classique
AND, OR, NOT	opérateurs booléens
IN	est dans une liste de valeurs définies
LIKE	permet de rechercher les valeurs contenant une chaîne de caractères

Voici un exemple d'utilisation de quelques-uns de ces opérateurs.

```
SELECT desig
FROM vehicules
WHERE (couleur LIKE "rouge%") -- sélectionne les différentes variantes de rouge
AND (prix < 5000) -- sélectionne les véhicules coûtant moins de 5000 euros
```

Définition

Si l'on veut récupérer les informations venant de plusieurs tables, il faut réaliser un **produit cartésien**, celui-ci est obtenu en insérant une virgule « , » entre les tables listées après le mot-clé `FROM`.

Définition

On peut donner un **alias** aux différentes tables pour simplifier la notation en utilisant le mot-clé `AS` (facultatif) suivi du nom d'alias. Cela est utile lorsqu'on fait appel à plusieurs tables avec des noms de colonnes identiques. Il est alors indispensable de préciser dans le `SELECT` la table dans laquelle se trouve la colonne en question.

```
SELECT v.desig, mo.nom, ma.nom -- la table avant la colonne à l'aide de l'alias
FROM vehicules AS v, modeles AS mo, marques AS ma -- définition des alias ;
/* Les virgules entre les noms de table réalisent le produit cartésien entre ces différentes tables. */
```

Pour la définition de l'alias, le mot-clé `AS` est facultatif : `table AS alias` est équivalent à `table alias`.

Définition

La **jointure** est équivalente à la sélection du produit cartésien de deux tables suivant une colonne donnée. Elle est obtenue par la séquence suivante :
`table1 JOIN table2 ON table1.col1 = table2.col2`
On sélectionne donc uniquement les occurrences identiques de `table1.col1` et `table1.col1`.

```
SELECT v.desig, mo.nom, ma.nom
FROM vehicules AS v
JOIN modeles AS mo ON v.idmod = mo.id -- jointure entre modele et vehicules
JOIN marques AS ma ON mo.idmarque = ma.id -- jointure entre modele et marque;
/* On récupère la liste des vehicules disponibles avec le nom du modèle et de la marque. */
```

Définition

Pour clarifier le résultat, le **renommage** est utile. Il consiste à donner un nouveau nom aux colonnes du résultat de la requête. On utilise le mot-clé `AS` dans le `SELECT`.

```
SELECT v.desig AS "nom du vehicule", mo.nom AS "modèle", ma.nom AS "marque"
FROM vehicules AS v
JOIN modeles AS mo ON v.idmod = mo.id -- jointure entre modele et vehicules
JOIN marques AS ma ON mo.idmarque = ma.id -- jointure entre modele et marque;
/* Les attributs de sortie ont à présent les noms "nom du vehicule", "modele" et "marque". */
```

Définition

Les **opérateurs d'agrégation** min, max, somme, moyenne et comptage sont respectivement utilisables avec les mots-clés `MIN`, `MAX`, `SUM`, `AVG` et `COUNT`. On les place après le `SELECT`.

```
SELECT COUNT(*)
FROM modeles;
/* Renvoie le nombre de tuples de la table modeles,
donc le nombre de modèles de véhicules différents */
```

Définition

Les opérateurs d'agrégation sont surtout intéressants lorsqu'ils sont utilisés avec la commande `GROUP BY`. Cette dernière construit des groupes qui sont ensuite traduits par une ligne en sortie construite à partir du ou des opérateurs d'agrégation. On la place toujours après la clause `WHERE`.

```
SELECT couleur, COUNT(*)
FROM Vehicule
GROUP BY couleur;
/* Renvoie un tableau dont la première colonne est la couleur et la seconde le nombre de véhicules correspondant. */
```

Définition

On peut spécifier de n'afficher que les lignes qui répondent à un critère résultant d'un calcul d'agrégation avec la clause `HAVING` placée après le `GROUP BY`. Elle a une action similaire au `WHERE` mais filtre en fonction des opérations d'agrégation.

```
SELECT couleur, COUNT(*)
FROM Vehicule
GROUP BY couleur
HAVING COUNT(*) <= 10;
/* Renvoie uniquement les couleurs qui correspondent à moins de 10 véhicules */
```

Définition

Une **sous-requête**, requête imbriquée ou requête en cascade consiste à exécuter une requête à l'intérieur d'une autre requête. On l'utilise souvent à l'intérieur d'une clause WHERE ou de HAVING pour remplacer une ou plusieurs constantes.

```
SELECT nom
FROM modele
WHERE id = (
    SELECT idmod
    FROM vehicules
    ORDER BY km DESC
    LIMIT 1);
/* Renvoie le nom du véhicule ayant le plus fort kilométrage */
```

On notera l'utilisation de `ORDER BY` qui permet de trier les résultats, ici en ordre décroissant (`DESC`) de la colonne km.

`LIMIT n` permet de ne renvoyer que les `n` premières valeurs.

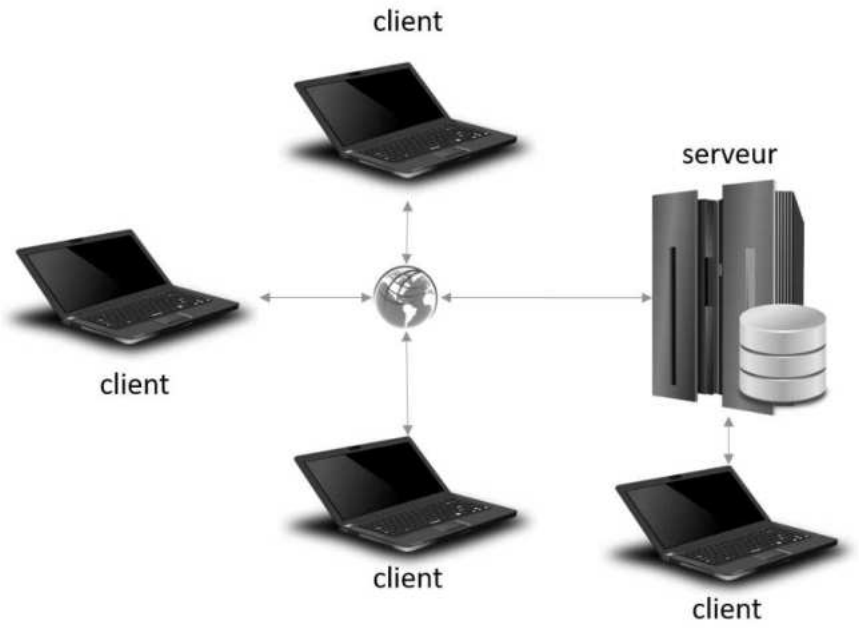
```
SELECT *
FROM modele
WHERE id IN (
    SELECT idmod
    FROM vehicules
    WHERE annee > 2010);
/* Renvoie les lignes complètes de la table modele en ne sélectionnant
que les modèles des véhicules immatriculés après 2010 */
```

■ 3 Architecture matérielle

Définition

L'architecture **client-serveur** est la base de l'organisation des bases de données. Le **serveur** contient la base de données et le **client** envoie la requête au serveur.

On a généralement plusieurs (beaucoup) clients qui accèdent à un même serveur via un réseau (Internet, intranet, etc.)



Définition

Une architecture plus élaborée est l'**architecture 3-tiers** (tiers signifie niveaux), aussi appelée architecture 3 couches. Entre le client et le serveur s'intercale une couche qui permet d'alléger la couche client qui se limite à l'aspect visuel. On obtient donc :

- la couche **présentation** qui se limite à l'interface graphique ;
- la couche **métier**. C'est elle qui exécute les requêtes SQL à partir des informations que l'utilisateur a fournies dans la couche présentation ;
- la couche **accès aux données** qui contient le SGBD et les données.

Le module `sqlite3` de Python correspond à la deuxième couche, la couche métier (Voir TP 6.1 p. 263).

Les méthodes à maîtriser

Méthode 6.0 : Écrire une requête SQL simple (une seule table utile)

- Vérifier que les informations utiles sont stockées dans la même table ;
- Lister les colonnes à afficher (projection, `SELECT`) ;
- Déterminer le critère de sélection (`WHERE`) ;
- Choisir la manière d'afficher les résultats (`AS`, `ORDER BY`, etc.)

Exemple d'application

Écrire une requête permettant de donner la désignation, le kilométrage et le prix des véhicules à vendre ayant moins de 2 ans, en les triant par prix croissant

vehicules

id	desig	idmod	couleur	puiss	annee	km	prix	idvend	idach
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	

La table `vehicules` contient toutes ces informations. On a besoin d'afficher les colonnes *designation*, *prix* et *kilometrage* que l'on renomme pour plus de cohérence :

```
SELECT designation AS "Désignation", prix, kilometrage AS "Kilométrage"
FROM vehicules
WHERE annee > (YEAR(CURDATE()) - 2)
ORDER BY kilometrage
```

Méthode 6.1 : Écrire une requête SQL faisant intervenir plusieurs tables

- Lister les tables contenant les informations nécessaires ;
- Lister les colonnes à afficher (projection, `SELECT`) ;
- Identifier les clés étrangères permettant de faire le lien entre les tables pour réaliser la jointure entre les tables.

Exemple d'application

Écrire une requête permettant de donner le nombre de véhicules en vente de chaque marque

marques

id	nom
⋮	⋮

modeles

id	nom	idmarque
⋮	⋮	⋮

vehicules

id	desig	idmod	couleur	puiss	annee	km	prix	idvend	idach
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	

Les tables `vehicules`, `modeles` et `marques` sont nécessaires car il n'y a pas de relation directe entre les tables `vehicules` et `marques`. La clé étrangère `idmodele` dans la table `vehicules` permet de désigner le modèle, la clé *idmarque* de la table `modele` permet de désigner la marque.

```sql
SELECT mar.nom AS "Marque", COUNT(*)
FROM vehicules AS v
JOIN modeles AS mod ON v.idmodele = mod.id
JOIN marque AS mar ON mod.idmarque = mar.id
GROUP BY mar.nom
```

Méthode 6.2 : Écrire une requête SQL complète

- Lister les colonnes à renvoyer et celles qu'il faut renommer ;
  - `SELECT [DISTINCT]` attributs `[AS` nouveaunom `]`, fonctions d'agrégation
- Lister les tables dans lesquelles se trouvent ces données ;
  - `FROM` liste de tables
- identifier les colonnes qui permettent de réaliser la jointure entre les tables (clés étrangères)
  - `[JOIN` table `ON` conditionjointure `]`
- Identifier la condition de sélection ;
  - `WHERE` conditions
- Identifier les éventuelles règles de regroupement avec les conditions d'affichage ;
  - `[GROUP BY` liste d'attributs `HAVING` conditions`]`
- Identifier l'éventuel ordre de tri du résultat de la requête
  - `[ORDER BY` liste d'attributs [`ASC` ou `DESC`]`]`

Les éléments entre `[]` sont optionnels.
L'ordre des mots-clés de la requête SQL n'est pas imposé par la norme mais souvent par le SGBD. Il est toutefois recommandé d'utiliser l'ordre présenté ci-dessus.

Exemple d'application

Écrire une requête permettant de donner le kilométrage moyen des véhicules de chaque marque en vente seulement s'il est supérieur à 50 000 km et trier le résultat par ordre alphabétique de marque

marques

id	nom
⋮	⋮

modeles

id	nom	idmarque
⋮	⋮	⋮

vehicules

id	desig	idmod	couleur	puiss	annee	km	prix	idvend	idach
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	

```sql
SELECT DISTINCT mar.nom AS "Marque", AVG(km) AS "kilométrage moyen"
FROM vehicules AS v
JOIN modeles AS mod ON v.idmodele = mod.id
JOIN marque AS mar ON mod.idmarque = mar.id
GROUP BY mar.nom
HAVING AVG(km) > 50000
ORDER BY mar.nom ASC
```

Vrai/Faux sur le cours

0. On considère la base de données contenant les informations (simplifiées au-delà du raisonnable !) relatives aux opérations bancaires réalisées par une entreprise s'adressant à des particuliers. Le schéma relationnel des tables est fourni : ☐ **Vrai** ☐ **Faux**

Table Clients (id_client, nom, prenom)
Table Comptes (id_client, numero_compte)
Table Transaction (numero_compte, date_heure, montant)

Un attribut peut jouer le rôle de clé primaire dans la table Transaction

1. id_client est une clé primaire dans la table Clients ☐ **Vrai** ☐ **Faux**

2. id_client est une clé primaire dans la table Comptes ☐ **Vrai** ☐ **Faux**

3. L'opération suivante en algèbre relationnelle : ☐ **Vrai** ☐ **Faux**

$$\Pi_{\text{numero\_compte}}(\sigma_{\text{Transaction}}(\text{montant}>7800))$$

donne les numéros de compte qui ont effectué des transactions strictement supérieures à 7800 €.

4. La syntaxe suivante permet d'afficher l'ensemble des noms et prénoms des clients : ☐ **Vrai** ☐ **Faux**

```
FROM Clients IMPORT nom, prenom;
```

5. Une double jointure est nécessaire si l'on souhaite récupérer les noms des clients ayant effectué des transactions le 01/01/2017 à minuit. ☐ **Vrai** ☐ **Faux**

6. Une jointure est nécessaire pour trouver le montant total des transactions d'un compte donné. ☐ **Vrai** ☐ **Faux**

Vrai/Faux sur le cours – corrigé

0. Non, aucun attribut de cette table ne caractérise une transaction de manière unique. Un même compte aura évidemment plusieurs transactions, des comptes différents peuvent effectuer une transaction au même moment et le montant d'une transaction ne la caractérise bien évidemment pas. ☐ **Vrai** ☑ **Faux**

1. Cet attribut permet effectivement de caractériser de manière unique un client. ☑ **Vrai** ☐ **Faux**

2. Non, un client peut posséder plusieurs comptes, il sera donc impossible de caractériser un enregistrement de manière unique à l'aide de ce seul attribut dans cette table. En revanche, id_client est une clé étrangère dans cette table, car c'est la clé primaire de la table Clients. ☐ **Vrai** ☑ **Faux**

3. Il y a une erreur dans la notation. Cette opération s'écrit : ☐ **Vrai** ☑ **Faux**

$$\Pi_{\text{numero\_compte}}(\sigma_{\text{montant}>7800}(\text{Transaction}))$$

4. Non, il ne faut pas confondre la syntaxe de l'opérateur « SELECT » en SQL avec celle de l'importation de modules Python. La bonne syntaxe est : ☐ **Vrai** ☑ **Faux**

```
SELECT nom, prenom
FROM Clients;
```

5. Il faut en effet accéder à des données dans les trois tables de la base : l'attribut date_heure dans la table Transactions est nécessaire, ce qui permet de récupérer les numéros de comptes correspondants. Ces numéros de comptes permettent grâce à une jointure avec la table Comptes de récupérer les id_clients correspondants. Une nouvelle jointure est nécessaire avec la table Clients pour récupérer les noms et prénoms des dits clients. ☑ **Vrai** ☐ **Faux**

6. Si l'on dispose du numéro de compte, une opération d'agrégation suffit pour connaître le montant total des transactions d'un compte : ☐ **Vrai** ☑ **Faux**

```
SELECT SUM(montant)
FROM Clients
WHERE numero_compte=...;
```

Exercices

Exercice 6.0

Certains prénoms sont purement masculins, ainsi Lionel et Jean-Pierre ne sont attribués qu'à des garçons. D'autres sont purement féminins, comme Angélique et Delphine, qui ne sont attribués qu'à des filles. Enfin, d'autres prénoms sont donnés, avec la même orthographe, à des garçons et à des filles, comme Andréa, Alix et Dominique. Ces prénoms sont dits *épicènes*.
Le taux de féminité d'un prénom P est la proportion de filles appelées P à la naissance sur le nombre total de bébés prénommés P. Par exemple, le taux de féminité f d'Alix, donné à 8217 filles françaises et 2360 garçons est donné par la formule

$$f = \frac{8217}{2360 + 8217} \approx 0.777 = 77.7\%.$$

Ce taux de féminité est utilisé dans le cadre d'études sociologiques pour « deviner » le sexe quand seul le prénom est connu. Par exemple, il fut utilisé par la sociologue Valérie Carrasco, dans une étude [**Car07**] pour le ministère de la justice publiée en octobre 2007, pour attribuer un genre aux PACS : féminin (deux femmes) ou masculin ou mixte.
Dans cet exercice, nous disposons d'une base de données contenant une table `baseprenoms` ayant la forme suivante :

`baseprenoms`

prenom	nombre	sexe	annee	departement
Manon	19.0	F	1983	Bouches-du-Rhône
Zakaria	24.0	M	2006	Hauts-de-Seine
Andrea	23.0	F	2001	Gironde
Andrea	30.0	M	2004	Alpes-Maritimes
⋮	⋮	⋮	⋮	⋮

Cette table indique pour chaque année, chaque département, chaque prénom et chaque sexe, le nombre de bébés nés avec ce prénom. Ainsi, la troisième ligne signifie qu'en 2001, en Gironde, 23 bébés filles furent prénommées « Andréa ».

Dans cet exercice, chaque question demande d'écrire une requête, et est suivie de quelques lignes retournées par la requête. Lorsque le résultat d'une requête est sauvegardé sous un nom, il peut être utilisé dans une autre requête comme n'importe quelle table.

0. Écrire une requête donnant la table des prénoms féminins et le nombre de filles nées avec ce prénom. Écrire une requête donnant la table des prénoms masculins ainsi que le nombre de garçons nés avec ce prénom.

prenom	nombreF
Alix	8217.0
Charlie	162.0
Julie	171878.0
⋮	⋮

prenom	nombreM
Alix	2360.0
Charlie	2909.0
Jean-Claude	124137.0
⋮	⋮

On supposera par la suite que les résultats de ces deux requêtes sont sauvegardés sous les noms respectifs `feminin` et `masculin`.

1. Écrire une requête donnant la table des prénoms épicènes avec le nombre de filles et le nombre de garçons nés avec ce prénom ainsi que le taux de féminité.

prenom	nombreF	nombreM	TauxF
Alix	8217.0	2360.0	0.777
Charlie	162.0	2909.0	0.053
Dominique	157761.0	219359.0	0.418
⋮	⋮	⋮	⋮

   On supposera par la suite que cette requête est sauvegardée sous le nom `epicene`.
2. Écrire une requête renvoyant la table des prénoms exclusivement féminins. Écrire une requête renvoyant la table des prénoms exclusivement masculins.

prenom
Angélique
Delphine
⋮

prenom
Lionel
Jean-Pierre
⋮

   On supposera par la suite que les résultats de ces deux requêtes sont sauvegardés sous les noms respectifs `prenomfeminin` et `prenommasculin`.
3. Écrire une requête renvoyant la liste des prénoms avec leur taux de féminité.

prenom	tauxF
Alix	0.777
Angelique	1.0
Lionel	0.0
⋮	⋮

Exercice 6.1

Une société de vente d'aimants dispose d'une base de données comprenant deux tables. La première table s'appelle `standards` et a les colonnes suivantes :

- `standard` : le standard de qualité, selon la norme chinoise (Y), américaine (C) ou selon une autre norme.
- `rem_min` : la rémanence minimale, mesurée en Tesla. Plus la rémanence est importante, plus l'aimant est « fort ».
- `rem_max` : la rémanence maximale, mesurée en Tesla. La rémanence réelle de l'aimant est comprise entre `rem_min` et `rem_max`.
- `coerc_min` et `coerc_max`, les valeurs minimales et maximales de la cœrcion, en kiloampère par mètre, qui mesure la capacité d'un aimant à conserver son aimantation.
- `temp` : la température maximale en degrés sous laquelle l'aimant va fonctionner.

`standards`

standard	rem_min	rem_max	coerc_min	coerc_max	temp
Y35	0.40	0.41	175	195	250
N40	1.26	1.29	860	955	80
N38H	1.22	1.26	860	915	120
S24	0.96	1.00	730	770	250
⋮	⋮	⋮	⋮	⋮	⋮

Cette première table décrit les caractéristiques physiques associées à chacun des standards (Y35, N40,...).

La seconde table `aimants` a les colonnes :

- `ref` : la référence du produit dans le catalogue ;
- `standard` : le standard respecté par cet aimant ;
- `forme` : la forme géométrique de l'aimant ;
- `materiau` : le matériau dans lequel est fabriqué l'aimant.

`aimants`

ref	standard	forme	materiau
MagnBr02	Y35	brique	ferrite
MagnCy08	S24	cylindre	samarium-cobalt
MagnAn42	N40	anneaux	neodyme
⋮	⋮	⋮	⋮

Cette seconde table décrit les aimants vendus par cette société.

0. Quelle(s) colonne(s) peuve(nt) [1] servir de clef primaire pour la table `aimants` ?

1. Écrire une requête renvoyant une table des standards avec comme colonnes : le nom du standard, la rémanence moyenne (la moyenne entre la rémanence minimale et la rémanence maximale) et la température maximale de travail.

2. Écrire une requête renvoyant la table des aimants cylindriques ayant une rémanence d'au moins $0,50\,\mathrm{T}$ pouvant être utilisés dans un environnement de $100\,°\mathrm{C}$. La table aura trois colonnes : la référence, le standard et le matériau.

3. Écrire une requête renvoyant la liste des standards disponibles dans toutes les formes possibles [2].

Exercice 6.2 Chocolat

Une entreprise de fabrication de chocolats stocke ses recettes dans une base de données. Cette base contient une table `nomenclature` qui associe à chaque recette de chocolat une référence et un prix en euros (€) et d'une table `matieres_premieres` donnant pour chaque ingrédient le prix au kilogramme.

`nomenclature`

Nom	Reference	prix
Wasachoco	W08	1.50
Monster chocolate	Mons01	4.99
Chocolat à l'orange	Ora03	0.99
⋮	⋮	⋮

`matieres_premieres`

Ingredient	prix
Cacao Forastero	2.78
Cacao Criollo	6.30
Cacao Trinitario	5.50
⋮	⋮

Elle contient aussi une table `recette` donnant pour chaque référence et chaque ingrédient la quantité en grammes.

1. S'il y a plusieurs possibilités, ne donnez que la plus pertinente.
2. Les formes possibles sont celles qui apparaissent au moins une fois dans la table `aimants`.

recette

Reference	Ingredient	Quantite
W08	Wasabi	0.2
W08	Cacao Forastero	5
W08	Beurre de cacao	4.8
Mons01	Cacao Trinitario	8
⋮	⋮	⋮

Par exemple, pour préparer un Wasachoco il faut 0,2 g de Wasabi, 5 g de cacao Forastero [1] et 4,8 g de beurre de cacao. Et peut-être faut-il aussi d'autres ingrédients [2] listés plus loin dans la table.

0. Citer une clef primaire possible pour la table `recette`.

1. Écrire une requête donnant la liste des ingrédients du Wasachoco (on pourra directement utiliser sa référence W08).

2. Écrire une requête donnant la liste des ingrédients du ChocoPlusPlus (il faudra chercher sa référence dans la table `nomenclature`).

3. Écrire une requête qui donne la table des recettes avec le prix total des ingrédients de la recette. Par exemple, pour le Wasachoco, il faudra sommer le prix de 0,2 g de Wasabi au prix de 5 g de cacao Forastero, etc.
La requête doit renvoyer un résultat de cette forme.

Reference	Prix
W08	0.98
Mons01	3
⋮	⋮

4. Quelles économies ferait-on en remplaçant le *cacao Criollo* par un cacao moins cher, le *cacao Forastero* ? Adapter la requête précédente pour avoir cette fois le prix total des ingrédients de chaque recette dans le cas où ce changement a été fait.

Exercice 6.3

La bibliothèque francophone d'Arkham utilise une base de données pour gérer les emprunts.
La table `inscrits` contient cinq colonnes : Nom, Prenom, Naissance, DateCotisation, Numero.
Voici quelques lignes de la table :

inscrits

Nom	Prenom	Naissance	DateCotisation	Numero
LAMBERT	Bernard	1985-03-22	2013-08-19	7
RICE	Warren	1975-04-03	2014-09-21	11
VASSEUR	Jade	1995-05-17	2014-02-15	12
⋮	⋮	⋮	⋮	⋮

La table des livres indique le titre, l'auteur et la cote de classement de chaque livre.

1. C'est la variété la plus courante de cacao.
2. Exempli gratia, du sucre.

livres

Titre	Auteur	Cote
Unaussprechlichen Kulten	Friedrich Wilhelm von Junzt	FWJ01
Le Roi en Jaune	Robert William Chambers	RWC03
De Vermis Mysteriis	Inconnu	JD01
Le culte des goules	comte François-Honoré Balfour d'Erlette	FHBE01
⋮	⋮	⋮

Enfin, nous avons une table des prêts. Un prêt est une relation entre un inscrit et un livre.

prets

NumeroInscription	Cote	DatePret	Rendu
2	FHBE01	2008-10-01	Oui
2	FHBE01	2014-01-09	Oui
11	JD01	2011-11-05	Non
11	RWC03	2013-11-05	Oui
⋮	⋮	⋮	⋮

Dans la table des prêts, chaque client est représenté par son numéro et chaque livre par sa cote.

0. Écrire une requête renvoyant la liste des cotes des livres prêtés non rendus.
1. Écrire une requête renvoyant le nombre d'inscrits dans la bibliothèque.
2. Écrire une requête renvoyant le nombre d'inscrits dans la bibliothèque ayant déjà emprunté au moins un livre.
3. Écrire une requête renvoyant la liste des livres déjà empruntés par Roland Franklyn (Roland est le prénom, on suppose que Roland Franklyn n'a pas d'homonyme).

Exercice 6.4 Pizzeria

La société Golden Web Pizza[1] permet de se faire livrer chez soi une pizza parmi un catalogue immense. La société gère ses commandes via une base de données. Cette base contient comme tables :

- Une table `pizza` contenant les champs `nom_pizza` et `prix`.

pizza

nom_pizza	prix
Margherita	8€
Caviar_totale	70€
⋮	⋮

- Une table `client` contenant les champs `nom`, `prénom`, `adresse`, `numero_client`.

client

nom	prénom	adresse	numéro_client
BLANCHARD	Nathan	7 rue du mail, 75002 Paris	42
FABRE	Lucas	18 rue de l'Electricite, 21000 Dijon	47
⋮	⋮	⋮	⋮

1. Société fictive.

- Une table `commande` contenant les champs `numero_client`, `nom_pizza`, `prix`, `numero_commande`, `date`.

`commande`

numero_client	nom_pizza	prix	numero_commande	date
42	Caviar_totale	68€	1024	1999-08-19
47	Margherita	8€	16384	2014-12-12
47	Super_Champignons	13€	16384	2014-12-12
⋮	⋮	⋮	⋮	⋮

Les quelques lignes données en exemple montrent que Nathan Blanchard a commandé en 1999 une pizza « Caviar totale » qui coûtait à l'époque la modique somme de 68€. Lucas Fabre a commandé, le 12 décembre 2014, une Margherita et une « Super champignons » dans la même commande.

- Une table `ingredients` contenant les champs `ingredient` et `nom_pizza`.

`ingredients`

nom_pizza	ingredient
Margherita	purée de tomate
Margherita	préparation fromagère
Caviar_totale	caviar
Caviar_totale	foie gras
Caviar_totale	crème fraîche
⋮	⋮

La pizza Margherita contient au moins[1] deux ingrédients : de la purée de tomate et une préparation fromagère[2].

0. Il est possible d'avoir deux pizzas différentes dans la même commande, c'est le cas de la commande de Lucas Fabre (une Margherita et une Super champignons). La base de données permet-elle d'avoir deux fois la même pizza (par exemple deux Margherita) dans la même commande ?

1. Quel intérêt de mettre le prix dans la table des commandes vu qu'il est déjà dans la table des pizzas ?

2. Un client est allergique à la tomate. Les deux seuls ingrédients contenant de la tomate sont « purée de tomates » et « tomate ». Quelle requête permet d'avoir la table des pizzas (avec leur nom et leur prix) ne contenant pas ces ingrédients ? Un résultat de cette forme est attendu :

nom_pizza	prix
Caviar_totale	70€
Super_champignons	13€
⋮	⋮

En Sqlite (un dialecte de SQL), la date d'aujourd'hui moins un an peut être obtenue grâce à l'expression suivante : `date("now","-1 year")`.
On supposera dans cet exercice que vous utilisez Sqlite.

3. Écrire une requête renvoyant la table des pizzas qui n'ont pas été commandées depuis un an.

1. Nous ne voyons pas la table en entier, peut-être que d'autres lignes précisent d'autres ingrédients.
2. Un ersatz de fromage d'origine végétale.

Un résultat de la forme suivante est attendu :

nom_pizza	prix
Saumon_et_ananas	12€
Melon_cacao	15€
⋮	⋮

En Sqlite, l'expression `strftime("%Y",X)` extrait l'année de la date `X`.

4. Écrire une requête qui, pour chaque client, et chaque année, donne l'argent qu'il a rapporté à la société.
Un résultat de la forme suivante est attendu :

numero_client	annee	depense
47	2014	2851€
47	2013	5412€
⋮	⋮	

Travaux pratiques

TP 6.0 – Données boursières

Préparation du TP : création du fichier `bourse.sqlite`

Télécharger en csv depuis le site Yahoo Finance l'historique des données boursières d'Airbus groupe (symbole EADSY) et de Orange (symbole ORA.PA). Puis, les importer comme deux tables (`airbus` et `orange`) dans une base de données Sqlite `bourse.sqlite` (par exemple en utilisant le logiciel Dbeaver).
Attention à bien préciser le type des données de chaque colonne.

Les tables ont chacune les colonnes suivantes : Date, Open, High, Low, Close, Volume, Adj Close donnant la valeur de chaque action chaque jour (valeur à l'ouverture de la bourse, valeur minimale, maximale, valeur à la fermeture) et le volume de transations (le nombre d'actions échangées dans la journée).

Les lignes d'une table peuvent être ordonnées selon une colonne grâce à la clause `ORDER BY colonne`. Par défaut, les lignes sont classées par ordre croissant. L'ordre décroissant est obtenu avec le mot-clef `DESC` (qu'on ajoute à la fin de la clause).
En Sqlite (un dialecte de SQL), la date d'aujourd'hui plus un jour peut être obtenue grâce à l'expression suivante : `date("now","+1 day")`. De plus, le mois peut être obtenu à partir de la date `X` grâce à l'expression suivante `strftime("%Y-%m",X)`.

0. Calculer le volume total de toutes les transactions d'Orange.

1. À quelles dates n'y a-t-il eu aucune transaction pour Orange ou pour Airbus ? On s'attend à un résultat de la forme suivante :

Date	Open	High	Low	Close	Volume	Adj Close	entreprise
2016-12-26	14.29	14.29	14.29	14.29	0	14.29	Orange
2013-03-04	50,68	50,68	50,68	50,68	0	11,804404	Airbus
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

2. Trier les résultats précédents par date (la date la plus récente en premier).

3. Écrire une requête qui restreint la table `airbus` aux entrées datant de moins de trois ans.

4. Donner en une seule requête la date du premier et la date du dernier enregistrement d'Airbus.

5. Donner la première date pour laquelle acheter à l'ouverture une action Airbus et la vendre à la fermeture est intéressant (par intéressant, on veut dire qu'on gagne de l'argent).

6. Calculer, pour chaque mois, pour Orange, la valeur minimale et la valeur maximale de l'action.

7. Donner la liste des dates pour lesquelles on a les données d'Orange et pas celles d'Airbus.

8. Calculer la liste des dates pour lesquelles l'action d'Airbus coûte plus cher à l'ouverture que l'action d'Orange.

9. On appelle gain maximum l'argent gagné en achetant au plus bas et en revendant au plus haut. Créer une table, à 3 colonnes, donnant, pour chaque jour [1], le gain maximal d'Orange et celui d'Airbus.

1. On ne considérera que les jours où l'on dispose des données des deux entreprises.

10. Ajouter à la table précédente une colonne indiquant l'entreprise ayant fait le plus fort gain (en cas d'égalité, on met Airbus).

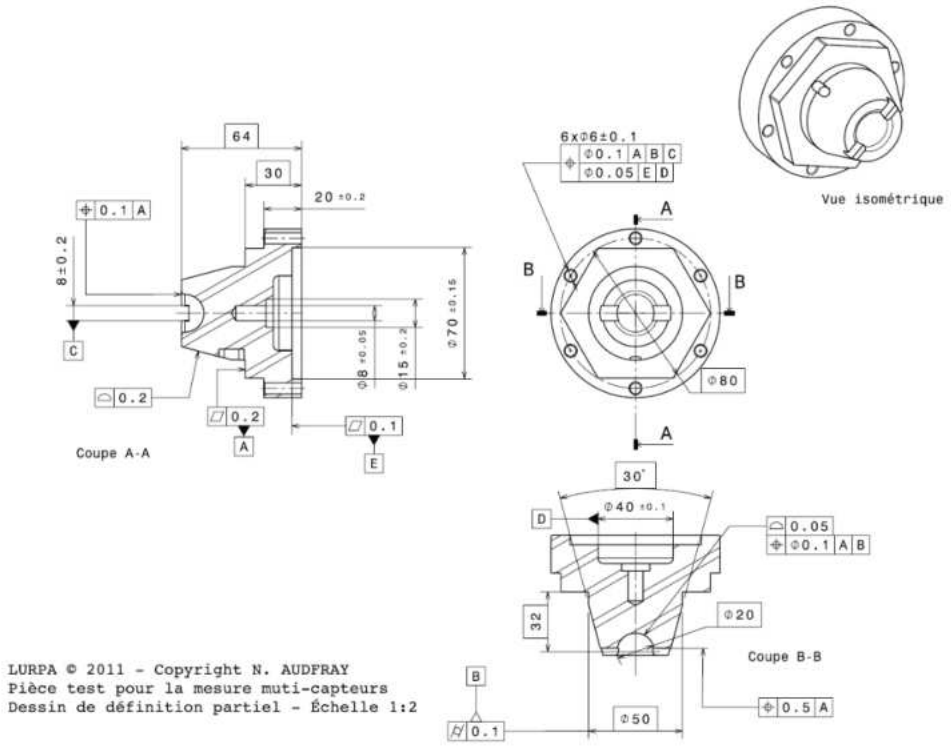
11. Je me demande combien je gagne si j'achète une action Airbus au jour n au plus bas et que je revends au plus cher au jour $n+1$ (c'est-à-dire le lendemain). Créer une table me donnant en fonction du jour n le gain réalisé.

12. Donner, pour chaque mois, pour Airbus :
- la valeur de l'action à l'ouverture du mois [1] et à la fermeture du mois ;
- la valeur maximale et minimale de l'action sur le mois.

13. Donner, pour chaque année et pour chaque entreprise, le bénéfice réalisé en achetant au minimum et revendant au maximum chaque jour. On ne donnera que les années où les données des deux entreprises existent dans la base de données.

TP 6.1 – Sélection de systèmes de numérisation en métrologie

L'entreprise Mécafab, spécialisée dans la production de pièces mécaniques, lance une nouvelle production de pièces et la vérification du cahier des charges est une partie fondamentale du processus. Une part importante de ce cahier des charges réside dans la vérification des dimensions de la pièce conformément au dessin de définition.

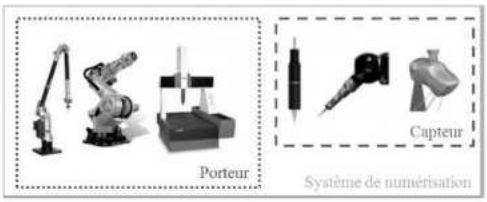


Le choix du système de numérisation est très important, il permet de déterminer le système le plus approprié pour réaliser une mesure. Le but est de choisir un système permettant de respecter les contraintes liées à la qualité des acquisitions (bruit de numérisation, justesse de mesure, densité du nuage de points, etc.), tout en minimisant le coût de numérisation (temps de numérisation).

1. Le premier jour du mois existant dans la base de données. On prendra garde au fait que certains jours, comme les dimanches, il n'y a pas d'ouverture de la bourse.

Le but de cette partie va donc être de sélectionner le système de numérisation le mieux adapté pour cette application, à partir d'une base de données.

Les systèmes de numérisation



Un système de numérisation est un ensemble composé de :

- un capteur (ou système d'acquisition, *sensor* en anglais) ;
- un porteur (ou système de déplacement, *device* en anglais).

C'est l'association capteur/porteur qui permet d'obtenir un nuage de points de coordonnées 3D exprimées dans un référentiel. Un capteur peut être utilisé sur différents porteurs et un porteur peut accueillir différents capteurs. Afin que le capteur soit orientable, on peut également trouver une interface entre le capteur et le porteur.
L'entreprise possède une base de données de ses différents systèmes de numérisation. Vous en trouverez une copie *digitizing_systems.db3* sur la page dédiée à cet ouvrage sur le site de Dunod.

0. Écrire le schéma relationnel entre les tables *sensor*, *device*, *interface* et *manufacturer*, la dernière correspondant au fabricant des différents matériels.

Les principes mis en jeu pour l'obtention des coordonnées 3D sont différents pour chaque technologie de capteur et de porteur et ne font pas l'objet de l'étude ici. Chaque type de capteur a ses avantages et ses inconvénients. En fonction de l'application (contrôle de pièce mécanique, rétroconception, sauvegarde d'œuvres d'art, etc.), chaque capteur aura plus ou moins d'avantages et d'inconvénients. Les principaux critères sont ceux liés à la qualité des données acquises, à la rapidité d'acquisition et à la densité de points obtenue.

Base de données de systèmes de numérisation

Une base de données des systèmes de numérisation a été mise en place pour que l'opérateur de contrôle puisse sélectionner le meilleur système de numérisation parmi ceux qu'il a à disposition pour réaliser ses mesures.
Cette base de données est structurée de la manière suivante :

- une partie « capteur » regroupant un certain nombre de tables contenant des informations intrinsèques aux capteurs (solution technique, technologie, catégorie) ;
- une partie « porteur » regroupant un certain nombre de tables contenant des informations intrinsèques aux porteurs (type de porteur, catégorie porteur) ;
- une table « données qualifiées » qui regroupe les informations liées à un couple capteur/porteur, c'est-à-dire à un système de numérisation, notamment les données liées à la qualité de numérisation.

1. À l'aide du logiciel SQLiteSpy, ouvrir la base de données *digitizing_systems* pour observer les différentes tables et données.

La requête suivante donne normalement le tableau ci-dessous :

```
SELECT *
FROM sensor
;
```

id	name	id_manuf	id_sol	resol.	distance	fov	trueness	repeat.
1	ZephyrKZ25	1	1	0,003	distance	50,0		0,003
2	LC60Dx	2	1	0,06	95,0	60,0	0,009	
3	TP2	3	7	0,000 2	0,0	0,0		0,000 35
4	LJ-G200	7	1	0,003	110,0	62,0		
5	G-scan RX2	8	1	0,1	150,0	110,0		
6	OptiNum-RE	9	2	0,3	450,0	210,0		
7	ATOS cs 2M	6	2	0,021	800,0	520,0		
8	CL2	5	4	0,0	11,0	0,0		

2. Quelle est *a priori* la clé primaire de cette table ? Peut-on en proposer une autre ?
3. Quelles sont les clés étrangères ?

Manipulation de la base de données

Nous souhaitons interroger la base pour extraire les systèmes de numérisation compatibles avec notre besoin. Nous proposons dans un premier temps la requête suivante :

```
SELECT sensor.name AS capteur, device.name AS porteur, inter.name AS interface
FROM sensor
JOIN qualified_system AS qual ON sensor.id=qual.id_sensor
JOIN device ON device.id=qual.id_device,
interface AS inter
WHERE qual.id_interface=inter.id
AND qual.trueness < 0.05;
```

4. Que signifie cette requête, qu'est-elle sensée retourner ? Exécuter cette requête pour valider. Vous devriez obtenir six lignes et trois colonnes.
5. À partir de l'exemple précédent, effectuer une requête SQL permettant de faire apparaître : le nom de tous les capteurs renseignés dans cette base de données, le nom du fabricant pour chaque capteur, la catégorie (contact, sans contact), la résolution, la distance d'acquisition.

On souhaite à présent renseigner dans cette table une nouvelle information au capteur « TP2 ».

6. À partir de requêtes SQL, donner les caractéristiques de ce capteur.
7. Grâce à la commande `UPDATE TABLE`, renseigner une résolution de $1\mu m$ pour ce capteur.

Choix du systèmes de numérisation

Dans les données qualifiées apparaissent deux grandeurs qualifiées qui dépendent du couple « système d'acquisition/système de déplacement » :

- Le bruit qualifié : il correspond aux erreurs de mesure de type aléatoire ;
- La justesse qualifiée : elle correspond aux erreurs de mesure systématiques.

Pour le choix du système de numérisation, le critère prépondérant va être le bruit de numérisation pour les spécifications de forme et d'orientation, et la justesse de mesure pour les spécifications de position ainsi que les vérifications dimensionnelles. On considère un système apte à réaliser une mesure si son incertitude de mesure U est supérieure à $U_{ad} = \frac{IT}{8}$, avec IT, l'intervalle de tolérance

de la spécification. La valeur de U_{ad} doit être supérieure à la valeur de bruit et la valeur de justesse renseignées pour le système.
Dans un premier temps, on détermine l'ensemble des systèmes aptes à réaliser les mesures, puis on sélectionne le plus rapide parmi les systèmes aptes.

8. Déterminer l'ensemble des systèmes de numérisation aptes à la vérification de la spécification géométrique ayant le plus petit intervalle de tolérance (on ne prendra pas en compte les spécifications de localisation des trous qui imposent l'utilisation des palpeurs classiques de par l'accessibilité des surfaces).

On considère à présent que l'incertitude de mesure est une moyenne pondérée entre le bruit et la justesse.

9. En considérant une pondération de 1 pour le bruit et la justesse dans le calcul de l'incertitude et en respectant toujours $U < U_{ad} = \frac{IT}{8}$, déterminer l'ensemble des systèmes aptes à la vérification des différentes spécifications.

10. Déterminer le meilleur système de numérisation, parmi les systèmes aptes, sur un critère de temps de numérisation (vitesse de numérisation maximale). On pourra ainsi classer les résultats avec le mot clé `ORDER BY`.

Utilisation de Python pour les bases de données

Dans le but de réaliser des calculs automatisés de vitesse de numérisation en fonction du capteur sélectionné, on souhaite utiliser Python pour récupérer les résultats des requêtes SQL.
Dans Python, l'intégration des requêtes SQL se fait comme le montre le code suivant. Il convient donc de se connecter à la base de données avant d'exécuter la requête.

```
import sqlite3
conn = sqlite3.connect("digitizing_systems.db3")
conn.row_factory = sqlite3.Row
c = conn.cursor()
c.execute("INSCRIRE ICI UNE REQUETE SQL ")
justesse = []
for ligne in c:
        if not None in ligne:
                justesse.append(float(ligne["qual.trueness"]))
c.close()
```

11. À partir de requêtes SQL intégrées à Python, calculer le temps de numérisation de la surface conique pour chacun des systèmes qualifiés présents dans la base de données. Trier les résultats par ordre croissant de temps de numérisation.

Corrections des exercices

Corrigé exo 6.0

0. On utilise GROUP BY pour agréger les données de chaque prénom.

```
SELECT prenom , SUM ( nombre ) as nombreF FROM baseprenoms
    WHERE sexe="F" GROUP BY prenom
```

La table des prénoms masculins est construite sur le même modèle :

```
SELECT prenom , SUM ( nombre ) as nombreM FROM baseprenoms
    WHERE sexe="M" GROUP BY prenom
```

1. On joint les deux tables pour obtenir les prénoms donnés à des filles et aussi à des garçons.

```
SELECT *, nombreF / (nombreF + nombreM) as TauxF FROM feminin JOIN masculin
    ON masculin.prenom = feminin.prenom
```

2. Une différence ensembliste permet d'obtenir les prénoms exclusivement féminins.

```
SELECT prenom FROM baseprenoms WHERE sexe="F"
EXCEPT SELECT prenom FROM baseprenoms WHERE sexe="M"
```

On procède de même pour les prénoms exclusivement masculins.

```
SELECT prenom FROM baseprenoms WHERE sexe="M"
EXCEPT SELECT prenom FROM baseprenoms WHERE sexe="F"
```

3. On fait l'union entre les épicènes, les purement féminins et les purement masculins.

```
SELECT prenom , 0 AS TauxF FROM prenommasculin
UNION
SELECT prenom , 1 AS TauxF FROM prenomfeminin
UNION
SELECT prenom , nombreF / nombreF + nombreH as TauxF FROM epicene
```

Corrigé exo 6.1

0. La colonne ref.

1.

```
SELECT standard, (rem_min+rem_max)/2 as rem_moy, temp FROM standards;
```

2.

```
SELECT ref, aimants.standard, materiau
    FROM aimants JOIN standards ON aimants.standard = standards.standard
    WHERE forme = "cylindre" AND rem_min >= 50 AND temp >= 100;
```

3. L'opération demandée s'appelle « division ensembliste ». On utilise ici le fait qu'une requête qui ne renvoie qu'une valeur est assimilée à cette valeur (d'où le test du `HAVING`).

```
SELECT standard FROM aimants GROUP BY standard
    HAVING COUNT(DISTINCT forme) = (SELECT COUNT(DISTINCT forme) FROM aimants);
```

Corrigé exo 6.2

0. Le couple (Reference, Ingredient) est une clef primaire possible.

1. Une condition `WHERE` permet de ne garder que les lignes voulues.

```
SELECT Ingredient FROM recette WHERE Reference = "W08";
```

2. On fait une jointure pour obtenir la référence du ChocoPlusPlus.

```
SELECT Ingredient FROM recette JOIN nomenclature ON recette.Reference = nomenclature.Reference
    WHERE Nom = "ChocoPlusPlus";
```

3. On regroupe par référence après avoir fait une jointure pour avoir les prix. On divise par 1000 pour convertir les kg en g.

```
SELECT Reference, SUM(Prix*Quantite/1000)
    FROM recette JOIN matieres_premieres ON recette.Ingredient = matieres_premieres.Ingredients
    GROUP BY Reference;
```

4. On commence par écrire la table dans laquelle le prix du cacao Criollo a été remplacé par celui du cacao Forastero :

```
SELECT * FROM  matieres_premieres WHERE Ingredient <> "Cacao Criollo"
UNION
SELECT "Cacao Criollo" as Ingredient, prix FROM matieres_premieres WHERE  Ingredient = "Cacao Forastero";
```

Cette requête sert alors de sous-requête :

```
SELECT Reference, SUM(Prix*Quantite)
  FROM recette JOIN
    (SELECT * FROM  matieres_premieres WHERE Ingredient <> "Cacao Criollo"
    UNION
    SELECT "Cacao Criollo" as Ingredient, prix FROM matieres_premieres
          WHERE  Ingredient = "Cacao Forastero")
  ON recette.Ingredient = matieres_premieres.Ingredients GROUP BY Reference;
```

Corrigé exo 6.3

0. On obtient :

```
SELECT Cote FROM prets WHERE Rendu="Non";
```

1. Il suffit de compter le nombre de lignes de la table des inscrits.

```
SELECT COUNT(*) FROM inscrits;
```

2. Dans la table `prets`, on compte combien d'inscrits apparaissent.

```
SELECT COUNT(DISTINCT NumeroInscription) FROM prets;
```

3. On joint les trois tables pour faire le lien entre Roland Franklyn et les livres qu'il a empruntés.

```
SELECT livres.* FROM
    livres JOIN prets ON livres.Cote = prets.Cote JOIN inscrits ON NumeroInscription = Numero
    WHERE Nom = "Franklyn" AND Prenom = "Roland";
```

Corrigé exo 6.4

0. Non, car cela impliquerait deux lignes identiques dans la table `commande`. Visiblement, la base de données n'a pas été très bien conçue !

1. Le prix de la table `pizza` peut évoluer (l'inflation) et on connaîtra toujours le prix de vente de chaque pizza dans les commandes passées.

2. On retire de la table des pizzas toutes les pizzas contenant l'un ou l'autre ingrédient problématique.

```
SELECT * FROM pizza
EXCEPT
SELECT pizza.* FROM pizza JOIN ingredients ON pizza.nom_pizza = ingredients.nom_pizza
               WHERE ingredient = "tomate" OR ingredient="purée de tomates";
```

3. On retire de la table des pizzas toutes celles commandées il y a moins d'un an.

```
SELECT * FROM pizza
EXCEPT
SELECT pizza.* FROM pizza JOIN commande ON pizza.nom_pizza = commande.nom_pizza
    WHERE date("now","-1 year") >= date;
```

4.

```
SELECT numero_client, strftime("%Y",X) as annee, SUM(prix) as depense
    FROM commande
    GROUP BY numero_client, annee;
```

Correction d'un TP

Corrigé TP 6.0

0. On obtient :

```
SELECT SUM(Volume) FROM orange;
```

1.

```
SELECT *, "Orange" as entreprise FROM orange WHERE Volume = 0
UNION
SELECT *, "Airbus" as entreprise FROM airbus WHERE Volume = 0;
```

2.

```
SELECT *, "Orange" as entreprise FROM orange WHERE Volume = 0
UNION
SELECT *, "Airbus" as entreprise FROM airbus WHERE Volume = 0
ORDER BY Date;
```

3.

```
SELECT * FROM airbus WHERE date > date("now", "-3 years");
```

4.

```
SELECT MIN(date), MAX(date) FROM airbus;
```

5.

```
SELECT MIN(date) FROM airbus WHERE Close>open;
```

6.

```
SELECT strftime("%Y-%m",date) as mois, MIN(low), MAX(high) FROM orange GROUP BY mois;
```

7.

```
SELECT date FROM orange EXCEPT SELECT date FROM airbus;
```

8.

```
SELECT orange.date FROM airbus JOIN orange ON orange.date = airbus.date
    WHERE orange.open > airbus.open;
```

9.

```
SELECT orange.date, orange.high-orange.low as gain_orange, airbus.high-airbus.low as gain_airbus
    FROM orange JOIN airbus ON orange.date = airbus.date;
```

10. Avec ce que nous avons vu en cours, une solution est de faire deux requêtes avec deux `WHERE` pour distinguer les cas, puis de faire une union.

```
SELECT orange.date, orange.high-orange.low as gain_orange,
          airbus.high-airbus.low as gain_airbus, "Orange" as entreprise
    FROM orange JOIN airbus ON orange.date = airbus.date
    WHERE gain_orange>gain_airbus
UNION
SELECT orange.date, orange.high-orange.low as gain_orange,
          airbus.high-airbus.low as gain_airbus, "Airbus" as entreprise
    FROM orange JOIN airbus ON orange.date = airbus.date
    WHERE gain_orange <= gain_airbus;
```

Il est aussi possible d'utiliser des fonctionnalités de SQL que nous n'avons pas abordées en cours, comme le CASE.

```
SELECT orange.date, orange.high-orange.low as gain_orange,
          airbus.high-airbus.low as gain_airbus, CASE orange.high-orange.low>airbus.high-airbus.low
          WHEN 1 THEN "orange" WHEN 0 THEN "airbus" END AS entreprise
    FROM orange JOIN airbus ON orange.date = airbus.date;
```

11.

```
SELECT airbus.date, airbus2.high-airbus.low as gain
    FROM airbus JOIN airbus as airbus2 ON date(airbus.date,"+1 day")=airbus2.date;
```

12.

```
SELECT M.month, airbus.open as open, airbus2.close as close, M.low, M.high FROM
        (
                SELECT strftime("%Y-%m",date) as month,  MIN(date) as mindate,
        MAX(date) as maxdate, MIN(low) as low, MAX(high) as high
                FROM airbus
                GROUP BY month
        ) as M
        JOIN airbus ON M.mindate = airbus.date
        JOIN airbus as airbus2 ON M.maxdate = airbus2.date;
```

13. On fait attention au fait que les jours d'achat-vente ne sont pas les mêmes pour les deux entreprises.

```
SELECT air.annee, gain_orange, gain_airbus FROM
    (SELECT strftime("%Y",date) as annee, SUM(high-low) as gain_orange
        FROM orange GROUP BY annee) as ora
    JOIN
    (SELECT strftime("%Y",date) as annee, SUM(high-low) as gain_airbus
        FROM airbus GROUP BY annee) as air
    ON air.anne=ora.annee
```

CHAPITRE 7

Piles et récursivité

L'essentiel du cours

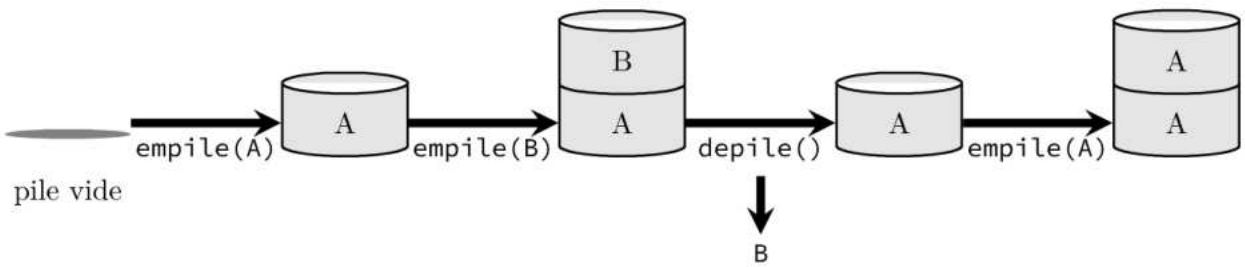
Piles

■ 0 Définition

Définition

Une **pile** est une structure de données **linéaire** (les données sont rangées sur une ligne) ayant pour maxime « dernier entré premier sorti » (*Last In, First Out*), LIFO.
Les méthodes (= fonctions) disponibles pour cette structure de données sont :

- **construction** (d'une pile vide),
- **test** d'une pile **vide**,
- ajout d'un élément (**empiler** = *push*) mis en premier dans la pile,
- retirer le premier élément (**dépiler** = *pop*) si la pile est non vide et renvoyer cet élément,
- lire le **sommet** de la pile.



Structure de Pile

Exemples d'utilisation des piles

- la fonction annuler (ctrl-Z)
- le calcul en notation polonaise inversée (ancienne calculatrice HP)
- le traitement des fonctions récursives – on met dans une pile les informations de chaque fonction appelée (variables locales, etc.).

■ 1 Les listes vues comme des piles

Le type `list` de `Python` possède déjà toutes les méthodes d'une pile :

```
pile = []       # création d'une pile
pile.append(4)  # on empile 4
pile.append(7)
pile.append("salut")
pile.append(5)
print("Pour l'instant, notre pile vaut:", pile)
pile.pop()      # on dépile 5
a = pile.pop()  # on dépile en récupérant le contenu
print('Ancienne tête:', a)
print('Tête actuelle:', pile[-1])  # on lit la tête
print('Pile actuelle:', pile)
print('Pile  vide?', pile == [])
```

```
Pour l'instant, notre pile vaut: [4, 7, 'salut', 5]
Ancienne tête: salut
Tête actuelle: 7
Pile actuelle: [4, 7]
Pile  vide? False
```

Les piles peuvent être implémentées de plusieurs manières. Nous utiliserons les listes.

Effet	Nom classique	Python
Ajouter un élément	push	`.append(x)`
Retirer un élément et renvoyer sa valeur	pop	`.pop()`
Test du vide		`== []`
Pile vide		`[]`

La méthode `.pop()` a deux effets :
- Elle modifie la pile en retirant le dernier élément,
- Elle renvoie la valeur du dernier élément.

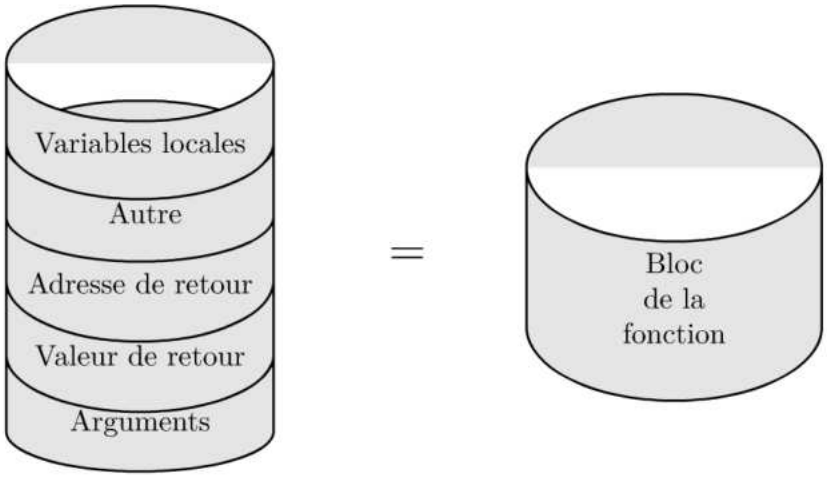
■ 2 Définir sa propre pile (construction d'un objet de type pile)

On peut créer un type (= une classe) `Pile` personnelle en Python. Le TP d'initiation à la programmation orientée objet et son application à la construction d'une structure de pile en propose un exemple d'implémentation.

voir le TP **7.0** p. 293.

■ 3 Pile d'exécution

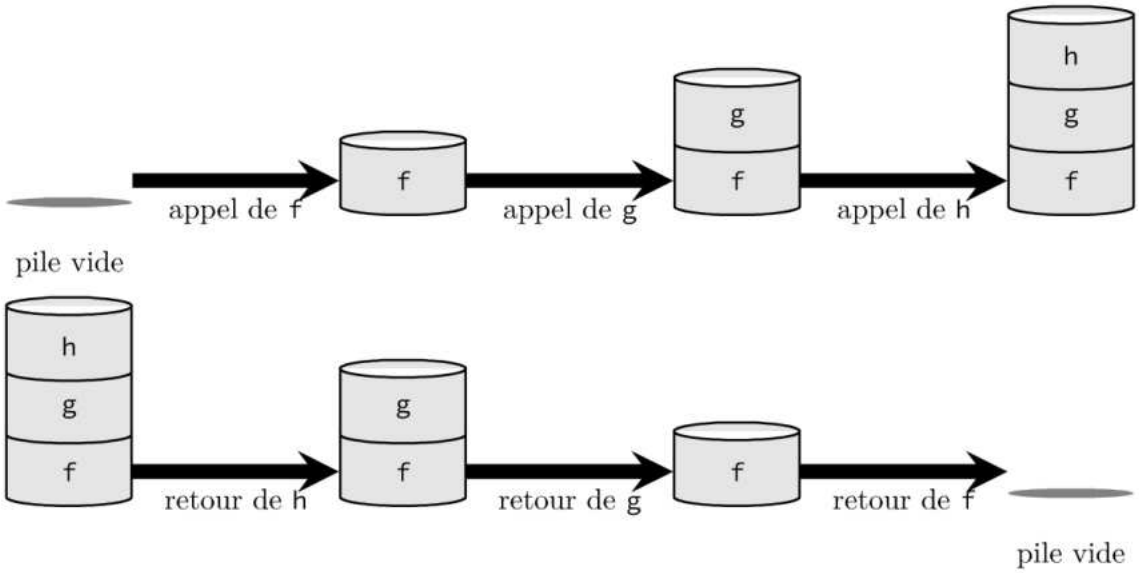
Lors de l'appel d'une fonction, que se passe-t-il ?
Un bloc est créé pour la fonction contenant :
- les arguments de la fonction,
- une place pour écrire la valeur de retour,
- l'*adresse de retour*, i.e., une information disant ce qu'il faudra faire une fois la fonction terminée,
- de la place pour les variables locales,
- éventuellement d'autres informations.



Un bloc de la fonction appelée est créé « au dessus » du bloc de la fonction appelante. Ces blocs sont empilés dans la pile d'exécution. Pour plus de détails, on peut consulter le Dragon's book, section 7.2.2 « Blocs d'activation » [ALS+07].
Comment sont gérés les appels de fonctions ?
Considérons le code suivant.

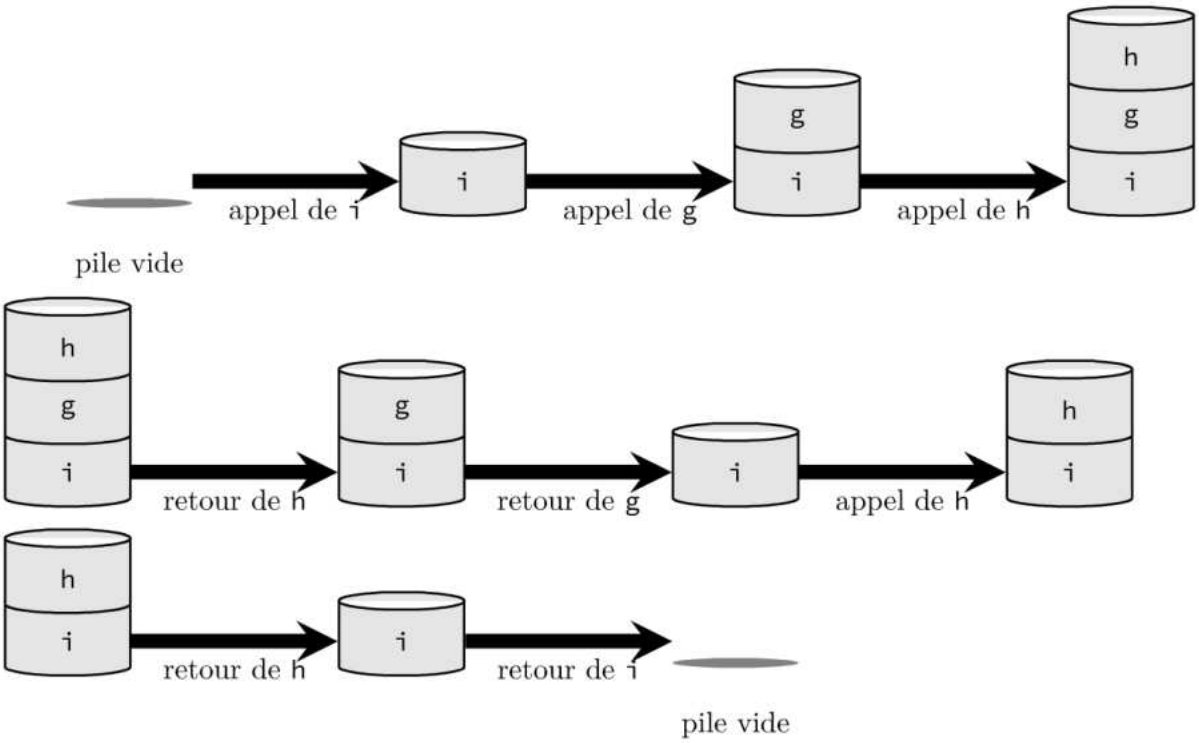
```
def h(x):
    return x + 1


def g(x):
    return h(x) * 2


def f(x):
    return g(x) + 1
```

Que se passe-t-il lors de l'appel à `f(5)` ?



Un autre exemple :

```
def h(x):
    return x + 1


def g(x):
    return h(x) * 2


def i(x):
    a = g(x)
    b = h(x)
    return a + b
```



Récursivité

■ 0 Définitions et exemples

Définition

Une **fonction récursive** est une fonction qui s'appelle elle-même.

Voici deux exemples classiques :

```
def expo(a, n):
    if n == 0:
        return 1
    else:
        return a * expo(a, n - 1)
```

```
def exporapide(a, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return exporapide(a, n // 2)**2
    else:
        return a * exporapide(a, n // 2)**2
```

Détaillons un peu.
Pour la première version.

```
def expo(a, n):
    if n == 0:
        print('ça y est n = 0')
        return 1
    else:
        print('Je rentre avec n =', n)
        return a * expo(a, n-1)
```

```
>>> expo(2, 7)
Je rentre avec n = 7
Je rentre avec n = 6
Je rentre avec n = 5
Je rentre avec n = 4
Je rentre avec n = 3
Je rentre avec n = 2
Je rentre avec n = 1
ça y est n = 0
128
```

Pour la seconde version.

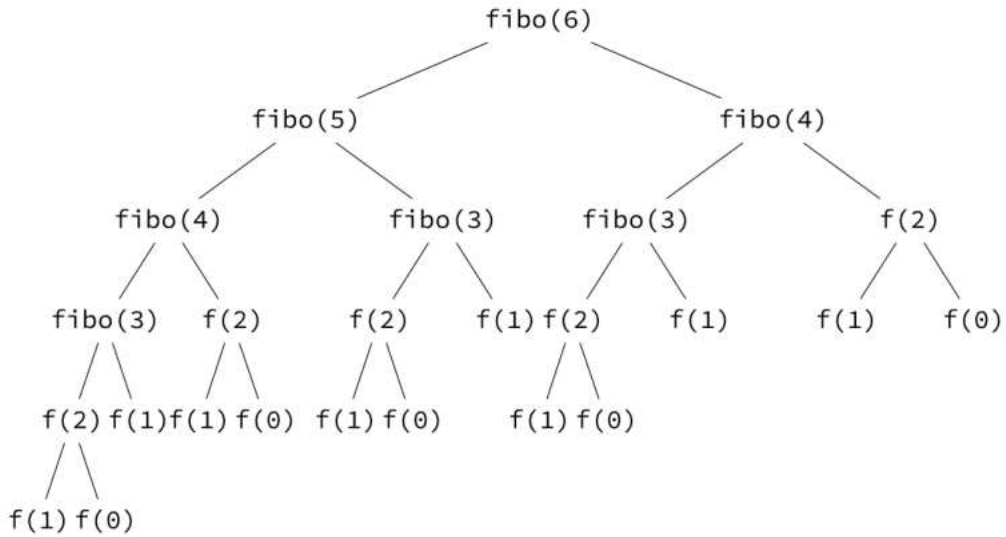
```
def exporapide(a, n):
    if n == 0:
        print('ça y est n = 0')
        return 1
    elif n % 2 == 0:
        print('(P) Je rentre avec n =', n)
        return exporapide(a, n//2)**2
    else:
        print('(I) Je rentre avec n =', n)
        return a * exporapide(a, n//2)**2
```

```
>>> exporapide(2, 17)
(I) Je rentre avec n = 17
(P) Je rentre avec n = 8
(P) Je rentre avec n = 4
(P) Je rentre avec n = 2
(I) Je rentre avec n = 1
ça y est n = 0
131072
```

Étudions la suite de Fibonacci : $u_n = \begin{cases} n & \text{si } n \in \{0,1\} \\ u_{n-1} + u_{n-2} & \text{pour } n \geqslant 2 \end{cases}$

```
def fibo(n):
    if n < 2:
        return n
    else:
        return fibo(n-1) + fibo(n-2)

for i in range(10):
    print(fibo(i), end=' ')
```

On peut se rendre compte que le programme récursif de calcul de la suite de Fibonacci est gourmand en temps et en espace. En effet, on peut représenter les appels récursifs suivant un arbre.



Cet exemple montre qu'il faut se méfier de la récursivité car elle peut cacher une complexité spatiale ou temporelle gourmande.
Dans notre exemple, il est bien préférable d'écrire le programme **itératif** suivant.

```
def fiboiter(n):
    if n < 2:
        return n
    a, b = 0, 1
    for i in range(n-1):
        a, b = b, a+b
    return b

for i in range(10):
    print(fiboiter(i), end=' ')

# 0 1 1 2 3 5 8 13 21 34
```

■ 1 Intérêt de la récursivité

En résumé, la récursivité fournit des algorithmes **concis et élégants**. La **preuve de la correction** du programme est souvent assez facile, mais il faut se méfier de la complexité. De plus, il faut bien étudier les **cas d'arrêt** pour s'assurer de la **terminaison** du programme.

■ 2 Pile d'appels récursifs

Lors de l'appel d'une fonction récursive, une pile est utilisée. En Python, la taille de la pile est limitée (1000 par défaut). En cas de dépassement, on a le droit à :

```
RuntimeError: maximum recursion depth exceeded
```

Il est possible de changer la taille de la pile.

```
import sys

sys.setrecursionlimit(10**4)
```

Un exemple très classique de programmation récursive est la résolution des tours de Hanoï.

Pour un exemple de dérécursivation utilisant explicitement une pile, voir l'exercice **7.15** p. 292.

■ 3 Encapsuler une fonction récursive

On est parfois amené à utiliser une fonction récursive à l'intérieur d'une fonction pour bénéficier de variable « semi-globale ».
Voici un exemple.
La fonction `uplets(n, k)` renvoie la liste de tous les k-uplets parmi $[\![1, n]\!]$.

```
def uplets(n, k):
    S = []

    def upint(L, nb):
        if nb == 0:
            S.append(L)          # on a fini un kuplet, on le met dans S
        else:
            for i in range(n):
                upint(L + [i], nb - 1)
    upint([], k)
    return S
```

On remarquera que la variable `S` est utilisée comme une variable globale dans la fonction interne récursive `upint`.

Pour d'autres programmes récursifs générant des objets combinatoires, voir l'ex. **7.13** p. 291.

■ 4 Récursivité croisée (☕)

Donnons un exemple de couple de fonctions qui s'appellent l'une l'autre.

```
def a(n):
    if n == 0:
        return True
    else:
        return b(n - 1)


def b(n):
    if n == 0:
        return False
    else:
        return a(n - 1)
```

Que calculent ces deux fonctions (d'argument entier naturel) [1] ?

Pour un exemple d'utilisation de cette notion, voir le TP **7.1** p. 296.

1. `a(n)` resp. `b(n)` teste si `n` est un entier pair resp. impair.

Les méthodes à maîtriser

Méthode 7.0 : Écrire une fonction récursive simple

Une fonction récursive simple s'écrit sous la forme :

```
def fonction(args):
    if condition d'arrêt:
        return valeur
    appel récursif
```

Pour écrire une fonction récursive on :

- détermine le type de données à renvoyer ;
- détermine pour quelle valeur de l'argument le problème est simple ; on conjecture alors la condition d'arrêt de la fonction récursive ;
- écrit la condition d'arrêt à l'aide des deux étapes précédentes ;
- détermine une règle permettant de réduire la valeur de l'argument du problème ;
- écrit l'appel récursif en prenant garde à ce que le type de données qu'il renvoie soit cohérent avec celui renvoyé par la condition d'arrêt.

Prenons pas à pas quelques exemples, du plus simple au plus complexe :

Exemple d'application

Écrire une fonction qui renvoie l'inverse d'une chaîne de caractères.

Le type de données à renvoyer est une chaîne de caractères. L'inverse d'une chaîne vide ou de longueur un est elle-même.
Lorsque la chaîne à inverser est plus longue, on obtient son inverse en plaçant le dernier caractère de la chaîne au début de la chaîne inversée et en inversant la chaîne où le dernier caractère a été exclu.

```
def inverse(s):
    if len(s) <= 1:
        return s
    return s[-1] + inverse(s[:len(s) - 1])
```

La fonction précédente a une complexité cachée : en effet, le *slicing* a ici un coût de $\mathcal{O}(n)$, et l'appel récursif est répété n fois, pour une complexité totale de $\mathcal{O}(n^2)$. Si l'on souhaite écrire une fonction récursive (une fonction itérative est plus simple ici), on peut écrire :

```
def inverse(s, i=0):
    if i == len(s):
        return ""
    return s[-i - 1] + inverse(s, i + 1)
```

Exemple d'application

Écrivons une fonction `swap` qui prend en argument une liste L et qui renvoie cette liste où l'ordre des éléments est modifié de sorte que cette liste contienne : le second élément de L puis le premier, puis le quatrième, puis le troisième, etc. `swap([1, 4, 9, 16, 25]` renverra `[4, 1, 16, 9, 25]`.

Le type de données à renvoyer est une liste. Lorsque L est vide ou composée d'un seul élément, `swap(L)` renvoie `L`. Ceci donne la condition d'arrêt.
Lorsque `L` est plus longue, on réduit la taille du problème en inversant les deux premiers éléments puis en appliquant `swap` sur la liste privée des deux premiers éléments. La remarque précédente sur la complexité reste valable.

```
def swap(L):
    if len(L) < 2:
        return L
    return [L[1], L[0]] + swap(L[2:])
```

Exemple d'application

Écrivons une fonction qui renvoie le nombre de façons de rendre une somme n en fonction d'une liste de type de pièces disponibles stockées dans une liste `L` ; on suppose que l'on dispose de suffisamment de pièces de chaque type pour rendre n.

Le type de données à renvoyer est un entier naturel. La condition d'arrêt n'est pas évidente ici, nous pouvons la trouver en réfléchissant à comment nous allons réduire la taille du problème : à chaque appel récursif, soit n va diminuer – puisqu'on essayera de rendre la somme due à l'aide du premier type de pièces disponibles, soit la longueur de L va diminuer – puisqu'on renoncera à rendre la somme due à l'aide du premier type de pièces. La condition d'arrêt porte donc à la fois sur n et à la fois sur `len(L)`.
La remarque précédente donne également l'appel récursif :

```
def pieces(n, L):
    if n == 0:  # Il y a succès dans le rendu de monnaie
        return 1
    if n < 0 or L = []:  # On est dans une situation d'impasse
        return 0
    return pieces(n, L[1:]) + pieces(n - L[0], L)  # la partition présentée.
```

Méthode 7.1 : Analyser la complexité d'une fonction récursive

Pour trouver la complexité T_n d'une fonction récursive :
- on cherche une relation de récurrence impliquant T_n,
- on résout la relation de récurrence.

Exemple d'application

Évaluer la complexité de la suite définie par $u_n = u_{n-1}^2 - 3$ et $u_0 = 0$ (issue d'un exercice filière PT).

Il est possible de la programmer récursivement comme suit.

```
def u(n):
    if n == 0:
        return 0
    else :
        return u(n - 1)**2 - 3
```

Ce qui donne une relation de récurrence sur la complexité de la forme $T_n = T_{n-1} + C$ avec C une constante. On reconnait une suite arithmétique d'où une complexité en $\mathcal{O}(n)$.

Exemple d'application

Prenons un cas plus compliqué :
évaluer la complexité du calcul récursif du nombre de 2 en base trois.

```
def nb_deux(n):
    if n == 0:
        return 0
    else:
        r = (n % 3) // 2
        return nb_deux(n // 3) + r
```

La relation de récurrence est alors $T_n = T_{n//3} + C$ avec C une constante qu'on ne cherche pas à évaluer. En itérant k fois cette relation de récurrence, on obtient $T_n = T_{n//3^k} + k \times C$. En choisissant pour k le nombre de chiffres de n en base trois, on obtient $T_n = T_0 + \text{nb\_chiffres} \times C$. Ce nombre de chiffres valant (à une unité près) $\log_3(n)$ (voir exercice 2.0 page 69), la complexité est en $\mathcal{O}(\log_3(n))$, i.e., en $\mathcal{O}(\ln(n))$.

Vrai/Faux sur le cours

0. Dans une pile, on peut ajouter en temps constant un élément à une extrémité et en enlever un à l'autre extrémité. ☐ **Vrai** ☐ **Faux**

1. Les piles sont utilisées en interne pour les fonctions récursives. ☐ **Vrai** ☐ **Faux**

2. Les instructions suivantes `copiepile` permettent d'obtenir une copie de la pile `p1` dans la pile `p2` mais vide la pile `p1`. ☐ **Vrai** ☐ **Faux**

```
def copiepile(p1):
    p2 = Pile()
    while not(p1.estvide()):
        p2.empile(p1.depile())
    return p2

p2 = copiepile(p1)    # p1 est une pile
```

3. Le code suivant permet d'afficher des entiers dans l'ordre croissant. ☐ **Vrai** ☐ **Faux**

```
def prog(n):
    if n != 0:
        print('Valeur ', n)
        prog(n-1)

prog(10)
```

4. Le code suivant affiche un message d'erreur. ☐ **Vrai** ☐ **Faux**

```
def listentierdecr(n):
    if n != 0:
        return [n] + listentierdecr(n-1)

listentierdecr(10)
```

5. Le programme suivant permet d'obtenir l'inverse d'une liste. ☐ **Vrai** ☐ **Faux**

```
def inverse(L):
    return inverse(L[1:]) + L[0]
```

6. Le programme suivant a une complexité en $\mathcal{O}(n)$. ☐ **Vrai** ☐ **Faux**

```
def fun(n):
    if n == 1:
        return 2
    return fun(n - 1)**n
```

7. Le programme suivant a une complexité en $\mathcal{O}(n^2)$. ☐ **Vrai** ☐ **Faux**

```
def fibo(n):
    if n <= 1:
        return 1
    return fibo(n - 1) + fibo(n - 2)
```

Vrai/Faux sur le cours – corrigé

0. Non, on ajoute ou enlève un élément à la même extrémité (sommet de la pile). Pour modifier le fond de la pile, on est obligé de vider toute la pile (temps linéaire). ☐ **Vrai** ☑ **Faux**

1. Oui, les entrées successives et les environnements sont empilés. ☑ **Vrai** ☐ **Faux**

2. Non, la pile `p1` est bien vidée mais la pile `p2` est renversée par rapport à la pile initiale. ☐ **Vrai** ☑ **Faux**

3. Non, le programme affiche les valeurs par ordre décroissant. ☐ **Vrai** ☑ **Faux**

4. En effet, il y a bien un message d'erreur même si un test d'arrêt est présent. Si `n` vaut `0`, le retour implicite est `None` donc il y a un conflit pour sommer une liste avec un objet de type `None`, ce qui explique le message d'erreur ☑ **Vrai** ☐ **Faux**

```
TypeError: can only concatenate list (not "NoneType") to list
```

Le code suivant, en revanche, fonctionnera :

```
def listentierdecr(n):
    if n!= 0:
        return [n] + listentierdecr(n-1)
    else:
        return [0]

listentierdecr(10)
```

5. Il manque la condition d'arrêt, le code suivant est en revanche correct. ☐ **Vrai** ☑ **Faux**

```
def inverse(L):
    if len(L) <= 1:
        return L
    return inverse(L[1:]) + L[0]
```

6. Si on note T_n la complexité de cet algorithme, on a $T_n = T_{n-1} + C$, donc T_n est une suite arithmétique. ☑ **Vrai** ☐ **Faux**

7. Cet algorithme a une complexité exponentielle, on peut montrer en effet que le nombre d'appels récursifs pour calculer F_n est de l'ordre de $\left(\frac{1+\sqrt{5}}{2}\right)^n$. ☐ **Vrai** ☑ **Faux**

Exercices

Piles Dans ces exercices, la règle du jeu est bien sûr d'appliquer les méthodes de pile et de ne pas utiliser les fonctionnalités élaborées des listes Python...

Exercice 7.0 Copier une pile

0. Écrire une fonction `deversepile` qui déverse une pile dans une pile donnée.

1. On veut écrire une fonction qui copie une pile.
Au premier essai, on écrit :

```
def copiepilebof(p):
    pnew, paux = Pile(), Pile()
    deversepile(p, paux)
    deversepile(paux, pnew)
    return pnew
```

Quel est le défaut de cette fonction ?

2. Proposer une amélioration de cette fonction.

Exercice 7.1 Échanger le sommet et le k^e élément

À chercher après avoir traité l'exercice précédent.
On utilisera en particulier la fonction `deversepile`.

0. Écrire une fonction `echange` qui échange les deux premiers éléments d'une pile.

1. Écrire une fonction `echk` qui échange le premier et le k^e élément.

Exercice 7.2 Rotation d'une pile

0. Écrire une fonction `rotpile(p)` qui met le premier élément en dernière position (rotation d'un cran de la pile).

1. Construire une fonction rapide `rotk(p, k)` pour une rotation de k crans de la pile en évitant d'utiliser la fonction `rotpile`.

Exercice 7.3 Séparer les positifs des négatifs

0. Écrire une fonction `separe(p)` qui, à partir d'une pile `p` d'entiers relatifs, renvoie une pile où tous les éléments strictement positifs de la pile `p` sont au-dessus des autres. On ne demande pas de garder la pile `p` intacte et on pourra utiliser des piles auxiliaires. De plus, l'ordre des éléments apparaissant dans la pile de sortie n'est pas important.

1. Écrire une fonction `extraitpos(p)` qui, à partir d'une pile `p` d'entiers comme précédemment, renvoie une pile de ses entiers positifs en **respectant** l'ordre d'apparition dans la pile `p`. Cette fonction doit préserver (à la sortie) la pile d'entrée `p` et n'utiliser que deux piles supplémentaires.

Pour des exercices de tri utilisant explicitement une pile voir **8.3** et **8.4** p. 363 et suiv.

Exercice 7.4 Expressions bien parenthésées

L'objectif de cet exercice est d'étudier quelles expressions sont bien parenthésées, dans plusieurs contextes.

0. On considère les expressions ne contenant que des parenthèses ouvrantes `'('` et fermantes `')'`. Écrire une fonction utilisant un compteur `n_ouv` qui compte le nombre de parenthèses ouvertes mais non fermées et qui renvoie un booléen indiquant si une expression `s` est bien parenthésée ou non.

1. On s'intéresse désormais aux expressions pouvant contenir des parenthèses `'('` et `')'`, des crochets `'['` et `']'` et des accolades `'{'` et `'}'`.
Donner un exemple d'expression contenant autant de parenthèses ouvrantes que de parenthèses fermantes et autant de crochets ouvrant que de crochets fermants mais qui n'est pas correctement parenthésée.

2. En utilisant une pile, écrire une fonction qui renvoie un booléen indiquant si une expression (comportant les trois types de parenthèses précédentes) est bien parenthésée ou non.

Exercice 7.5 Évaluation en notation polonaise inversée (postfixe)

On considère l'expression en notation polonaise inversée[1] codée par une liste Python de la manière suivante

```
L = [7, 8, '-', 6, '*', 10, 3, '+', '*']
```

L'expression infixe traditionnelle est `(7-8)*6*(10+3)`.
Certaines calculatrices de marque Hewlett-Packard des années 1990 utilisaient cette approche et affichaient donc une pile permettant d'effectuer les calculs[2].

Pour calculer l'expression correspondant à la liste `L`, on peut utiliser une pile.
On parcourt les éléments de la liste `L` :

- si l'élément est un nombre, on l'empile ;
- s'il s'agit d'une opération, on effectue l'opération à partir des deux derniers éléments de la pile puis on empile le résultat (attention à l'ordre si l'opération n'est pas commutative).

Écrire une fonction `evalNPI(L)` utilisant une pile qui calcule l'expression post-fixée (notation polonaise inversée) correspondant à la liste `L` et renvoie la valeur.

Pour aller plus loin, voir TP **7.1** p. 296.

Exercice 7.6 Déplacement d'un cavalier sur un échiquier

Considérons un échiquier $E_{n,p}$ à n lignes et p colonnes, où n et p sont deux entiers naturels non nuls. Nous allons dans cet exercice nous intéresser au déplacement d'un cavalier sur cet échiquier, selon les règles du jeu d'échec : le cavalier se déplace d'une case dans une direction puis de deux cases dans une direction perpendiculaire. L'échiquier $E_{n,p}$ sera représenté par une matrice E de

1. Du mathématicien polonais Jan Łukasiewicz.
2. Le lecteur intéressé pourra par exemple télécharger l'application pour smartphone Android `Droid48` qui émule la célèbre calculatrice HP48.

taille $n \times p$, où $E[x, y]$ pourra prendre différentes valeurs entières en fonction des problèmes que nous aurons à étudier.
Nous utiliserons les bibliothèques `numpy` et `matplotlib` pour tracer quelques figures :

```
import numpy as np
import matplotlib.pyplot as plt
```

La matrice E sera initialisée par l'instruction `E = np.zeros((n, p), dtype=int)` et nous utiliserons aussi la variable globale `Dep` qui code les huit déplacements possibles du cavalier :

```
Dep = [(1, 2), (1, -2), (2, 1), (2, -1), (-1, 2), (-1, -2), (-2, 1), (-2, -1)]
```

Un `chemin` du cavalier dans l'échiquier sera représenté par une liste $[(x_0, y_0), \ldots, (x_k, y_k)]$ de cases.

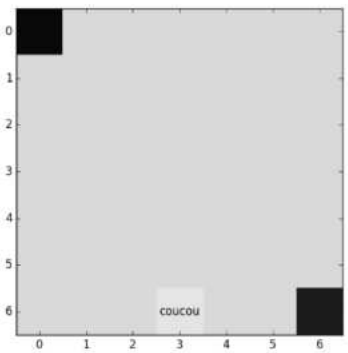
0. Écrire une fonction `Distance` qui, appliquée à (x_i, y_i, n, p), renvoie la matrice E tel que $E[x, y]$ contient le nombre minimum de déplacements nécessaires au cavalier pour passer de la case (x_i, y_i) à la case (x, y) dans l'échiquier $E_{n,p}$ (et -1 s'il n'est pas possible d'atteindre (x, y) depuis (x_i, y_i)). On utilisera une pile permettant de stocker toutes les cases situées à une même distance de la case initiale (x_i, y_i).

1. Modifier la fonction précédente en une fonction `Chemin_Minimaux` qui, appliquée à (x_i, y_i, n, p), renvoie cette fois-ci une matrice $Pred$ telle que $Pred[x, y]$ est le prédécesseur de (x, y) dans un chemin de longueur minimal de (x_i, y_i) à (x, y) si un tel chemin existe.

2. Écrire le code d'une fonction qui, appliquée à la matrice des prédécesseurs calculée précédemment et à une case finale accessible (x_f, y_f), renvoie une représentation graphique d'un chemin minimal de (x_i, y_i) à (x_f, y_f).
Voici quelques indications pour numéroter les cases d'une grille (utilisant la méthode `annotate`) :

```
A = np.zeros((7, 7), dtype=int)
A[0, 0] = -5
A[-1, -1] = 5
A[6, 3] = 1
fig = plt.figure()
ax = fig.add_subplot(111)
ax.annotate('coucou', xy=(3, 6), va="center", ha="center")
plt.imshow(A, interpolation='nearest')
plt.show()
```

ce qui donne



Exercice 7.7 Le problème du cavalier d'Euler (par *backtracking*)

Nous reprenons les notations de l'exercice précédent et nous cherchons maintenant à construire un *parcours du cavalier*, c'est-à-dire un chemin du cavalier passant une et une seule fois par chaque case de l'échiquier. Un tel parcours sera dit *cyclique* si le cavalier peut rejoindre en un coup la case de départ depuis la case d'arrivée. La matrice E permettra maintenant de stoker les instants de passage du cavalier par chaque case de l'échiquier. Au début du calcul, chaque case de E contient la valeur 0.

0. Écrire le code d'une fonction `cases_accessibles` qui, appliquée à la matrice E et à une position (x, y), renvoie la pile des cases accessibles depuis (x, y) (on entend par accessibles celles qui n'ont pas encore été visitées par le cavalier).

1. Écrire le code d'une fonction `parcours` qui, appliquée à quatre entiers x_i, y_i, n, p renvoie la matrice E associée à un parcours partant de (x_i, y_i) dans l'échiquier $E_{n,p}$ (si un tel parcours existe), en utilisant l'analyse suivante :

- **a.** On initialise la matrice E et on copie la valeur 1 dans la case (x_i, y_i).
- **b.** On initialise une variable N à la valeur 1 : cette variable va compter le nombre de cases déjà visitées par le cavalier.
- **c.** On initialise une pile *Coups* qui contient pour seul élément le triplet (x_i, y_i, L_i) où L_i est la liste des cases accessibles depuis (x_i, y_i). Cela signifie que le premier coup joué sera (x_i, y_i) et que L_i est la pile des cases accessibles depuis (x_i, y_i) non encore étudiées.
- **d.** Tant que $N \neq n \times p$ et que *Coups* est non vide, on dépile l'élément (x, y, L) qui est en tête de *Coups*. Si L est vide, on a atteint un cul-de-sac et on revient d'un coup en arrière ; sinon, on empile dans *Coups* le triplet (x, y, L') où L' est la liste L privée d'un de ses éléments (xs, ys) (s comme « suivant »), puis on empile dans *Coups* le triplet (xs, ys, Ls) où Ls est la liste des cases accessibles depuis (xs, ys).

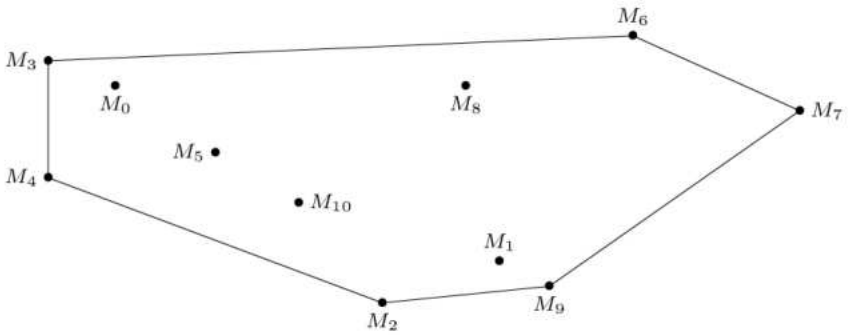
Quelle amélioration pourrait-on proposer pour accélérer, en cas d'existence, la recherche d'un parcours ?

2. Reprendre la question précédente en renvoyant cette fois-ci la représentation graphique d'un parcours cyclique (si un tel parcours existe).

Exercice 7.8 Enveloppe convexe d'une famille de points du plan

Nous considérons une famille $(M_i)_{0 \leqslant i < n}$ de points distincts d'un plan euclidien orienté, rapporté au repère orthonormé direct $(O, \overrightarrow{u}, \overrightarrow{v})$. Nous représenterons chaque point M_i par le couple (x_i, y_i) de ses coordonnées dans $(O, \overrightarrow{u}, \overrightarrow{v})$ et la famille $(M_i)_{0 \leqslant i < n}$ par la liste $L = [(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})]$. L'enveloppe convexe $\mathscr{C}$ de cette famille est le plus petit polygone convexe contenant tous les points M_i. Le but de cet exercice est de calculer la liste E dite « bien ordonnée » des sommets de $\mathscr{C}$, i.e. la liste des sommets de $\mathscr{S}$ parcourus dans le sens trigonométrique direct, en commençant par le point M_{i_0} situé le plus bas parmi ceux situés le plus à gauche. Ainsi, avec les points représentés sur

la figure ci-dessous, $i_0 = 4$ et nous cherchons à construire la liste $E = [M_4, M_2, M_9, M_7, M_6, M_3]$:



Pour s'épargner le traitement de cas particulier, nous supposerons que trois points quelconques de la famille (M_i) ne sont jamais alignés.

0. Si A, B et C sont trois points non alignés du plan, nous dirons que « (A, B, C) tourne à gauche » si l'angle $\overrightarrow{AB}$ et $\overrightarrow{AC}$ admet une mesure dans $]0, \pi[$. Écrire le code d'une fonction `tourne_gauche` qui prend en argument trois points A, B et C et qui renvoie le booléen `True` si (A, B, C) tourne à gauche et `false` sinon.

1. Écrire le code d'une fonction `point_depart(L)` qui renvoie l'indice i_0.

2. En s'inspirant de l'exercice 8.4, écrire le code de la fonction `tri_sommets` qui, appliquée à L et i_0, renvoie la pile P des points M_i pour $i \neq i_0$, triée dans l'ordre décroissant des angles que font les vecteurs $\overrightarrow{u}$ et $\overrightarrow{M_{i_0} M_i}$. Dans l'exemple proposé, cette fonction devra renvoyer $[M_3, M_0, M_6, M_8, M_5, M_7, M_{10}, M_1, M_9, M_2]$. On utilisera la fonction `tourne_gauche`.

3. Écrire le code d'une fonction `enveloppe_convexe` qui, appliquée à L, renvoie la liste E en utilisant la méthode suivante :

(a) On calcule i_0 puis P ;

(b) On initialise une pile E qui ne contient que M_{i_0} au début du calcul ;

(c) On vide P en modifiant E de sorte qu'à chaque tour de boucle, la propriété $\mathscr{P}$ soit vérifiée :

$\mathscr{P}$: E est la liste bien ordonnée des sommets de l'enveloppe convexe des points de L qui ne sont pas présents dans P.

Quel est le temps de calcul de la partie (c) de cet algorithme ? Quel est le temps de calcul de la fonction `enveloppe_convexe` ? Comment pourrait-on améliorer ce temps ?

Soit $n \in \mathbb{N}^* \geqslant 3$ et $X_0, X_1, \ldots, X_{n-1}, Y_0, Y_1, \ldots, Y_{n-1}$ des variables aléatoires indépendantes et identiquement distribuées, de loi uniforme sur $[0, 1]$. On cherche à estimer l'espérance du nombre N de sommets de l'enveloppe convexe des points $(X_i, Y_i)_{0 \leqslant i < n}$.

4. Écrire une fonction `tracer` qui, appliquée à un entier n, simule le tirage des points (X_i, Y_i), et trace le nuage de ces points ainsi que le contour de leur enveloppe convexe.

5. Écrire une fonction `estimation` qui, appliquée à un entier n, renvoie une estimation de l'espérance et de la variance de N. Que peut-on conjecturer quant au comportement de l'espérance quand n tend vers l'infini ?

Récursivité

Exercice 7.9 Bézout récursif

0. Écrire une fonction récursive `pgcd(a, b)` qui renvoie le plus grand commun diviseur (pgcd) des entiers supposés positifs a et b (c'est une question de cours).

1. Écrire une fonction récursive `bezoutrec(a, b)` qui renvoie le pgcd de a et de b ainsi que deux entiers u et v tels que $au + bv = d$.

Exercice 7.10 Figures récursives

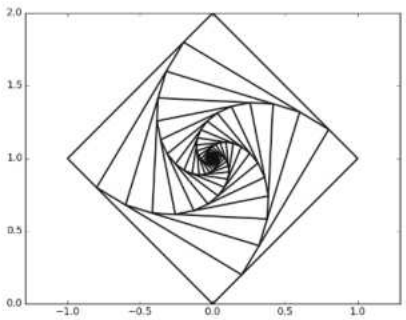
0. On définit la fonction suivante

```
def carre(a, b):
    return [a, b, b - 1j * (a - b), a + 1j * (b - a), a]
```

On peut alors tracer facilement un carré

```
import matplotlib.pyplot as plt

L = carre(1, 1 + 1j)
plt.axis('equal')
for a in L:
    plt.plot([a.real for a in L], [a.imag for a in L], 'b', lw=2)
plt.show()
```

Écrire une fonction récursive qui permet d'obtenir la figure suivante



1. À partir d'un segment $[a, b]$ où $a, b \in \mathbb{C}$, on trace le segment $\left[a, \frac{1}{4}(3a + b)\right]$ puis on recommence avec $a \leftarrow a$ et $b \leftarrow a + \frac{1}{2}(b-a)e^{i}$; $a \leftarrow a$ et $b \leftarrow a + \frac{1}{2}(b-a)e^{-i}$ et enfin $a \leftarrow \frac{1}{4}(3a+b)$ et $b \leftarrow b$
On obtient un figure assez jolie en partant de $a = 0$ et $b = 2$. À vous de la réaliser !

Exercice 7.11 Arborescence de fichiers – parcours en profondeur

On propose de parcourir tous les répertoires d'un système de fichiers récursivement à l'aide de Python.
Pour cela, on dispose dans le module `os` des fonctions suivantes :

— `os.chdir(chaine)`, par exemple `os.chdir('..')` pour revenir au répertoire parent ;

— `os.listdir()` qui liste les fichiers et répertoires du répertoire de travail ;
— `os.path.isdir(chaine)` qui dit si chaîne est un répertoire `(directory)` ;
— `os.path.abspath(chaine)` qui donne le chemin absolu du répertoire courant ;
— (pour information) `os.getcwd()` donne le répertoire de travail courant.

Écrire un programme en Python qui liste les sous-répertoires à partir d'un répertoire de départ donné. Préciser pour chaque répertoire la profondeur de l'arbre.

Exercice 7.12 Terminaison de la fonction d'Ackermann

La fonction `ack` calcule la *fonction d'Ackermann* (les entrées `m` et `n` sont des entiers naturels) :

```
def ack(m, n):
    if m == 0:
        return n + 1
    elif n == 0:
        return ack(m - 1, 1)
    else:
        return ack(m - 1, ack(m, n - 1))
```

On considère l'ordre lexicographique strict $\prec$ sur $\mathbb{N} \times \mathbb{N}$ défini par

$$(a,b) \prec (a',b') \text{ si et seulement si } \begin{cases} a < a' \\ \text{ou} \\ a = a' \text{ et } b < b' \end{cases}$$

0. Existe-t-il une suite strictement décroissante $(a_n, b_n)_{n\in\mathbb{N}}$ de couples, c'est-à-dire tel que pour tout $n \in \mathbb{N}$, $(a_{n+1}, b_{n+1}) \prec (a_n, b_n)$?

1. On considère l'ensemble $\mathscr{A} = \{(m,n) \in \mathbb{N} \times \mathbb{N} \mid$ `ack(m, n)` ne termine pas$\}$.
et on suppose que l'ensemble $\mathscr{A}$ est non vide. Montrer que $\mathscr{A}$ possède un minimum (m_0, n_0) minimal (raisonner par l'absurde).

2. Montrer que le calcul de `ack` termine toujours.

Exercice 7.13 Génération d'objets combinatoires

0. Programmer une fonction récursive qui liste tous les **k-uplets croissants** de $[\![1, n]\!]$.

1. Même question pour les **k-arrangements**.

2. On appelle **partition d'un entier** $n \geqslant 1$, toute décomposition du type

$$n = n_1 + n_2 + \cdots + n_p \text{ avec } n_i \geqslant 1.$$

Programmer une fonction récursive qui liste toutes les partitions d'entiers sans répétition (on s'interdit d'écrire $3 = 1 + 1 + 1$ par exemple) puis celles avec répétitions.

Exercice 7.14 Génération d'objets aléatoires = difficile

Cet exercice est une version récursive de l'exercice 3.6 p. 92.

On considère le programme suivant

```
from random import random


def combalea(n, p, L):
```

```
    if n <= 0 or p <= 0 or n < p:
        return L

    r = random()
    if r < p / n:
        L.append(n)
        return combalea(n - 1, p - 1, L)
    else:
        return combalea(n - 1, p, L)
```

0. Expliquer ce qu'affiche la suite d'instructions suivantes

```
for i in range(20):
    L = []
    print(combalea(27, 5, L))
```

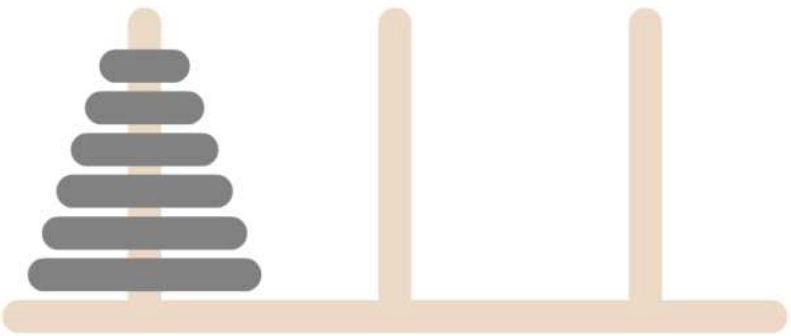
1. ☕ Montrer que les sorties de la fonction `combalea` sont équiprobables.

Exercice 7.15 Tour de Hanoï et pile d'appels récursifs

On propose, sur le célèbre exemple de la tour de Hanoï, d'expliciter la pile d'appels récursifs.

Principe des tours de Hanoï

On dispose de trois piquets sur lesquels peuvent s'insérer des disques de diamètres différents. Au début, tous les disques sont empilés par diamètre décroissant sur un piquet de départ (a).



Le but du jeu est de déplacer toute la pyramide sur un piquet d'arrivée (b) avec la règle du jeu suivante :

- on ne peut déplacer qu'un disque à la fois ;
- on ne peut empiler un disque que sur un emplacement vide ou sur un disque de diamètre inférieur.

On codera les piquets `a`, `b`, piquet intermédiaire `c` par trois nombres 0, 1 et 2. Remarquons que si `a` et `b` sont les piquets de départ et d'arrivée, `c = 3 - (a + b)` nous donne le numéro du piquet intermédiaire.

0. Écrire une fonction récursive `hanoi(n, a, b)` qui affiche les opérations successives de déplacement de n disques pour au final déplacer tous les disques de la tige a vers la tige b.

1. Écrire une version itérative de la fonction précédente [1] en utilisant une pile d'appels. Pour cela on pourra empiler deux types de données :

- soit un tuple du type `('appel', (n, a, b))`,
- soit un tuple du type `('affiche', ("1 -> 2", 0, 0))` (les zéros ne sont pas utilisés).

1. Dans un but purement pédagogique...

Travaux pratiques

TP 7.0 – Initiation à la programmation orientée objet

Ce TP est consacré à la *Programmation Orientée Objet*, POO en abrégé, et à son application à la construction de la structure de pile.

Principe de la POO

Généralités La POO consiste en la définition et l'interaction d'éléments logiciels appelés objets ; un objet représente un concept, ou une entité du monde physique, comme un personnage dans un jeu vidéo ou un livre. Il possède une structure interne et un comportement. Il peut interagir avec ses pairs. La POO est un style de programmation où on cherche à représenter ces objets et leurs relations.

Concrètement, un objet est une structure de données qui répond à un ensemble de messages. Cette structure de données définit son état tandis que l'ensemble des messages qu'il comprend décrit son comportement :

- les données qui décrivent sa structure interne sont appelées ses *attributs* ;
- l'ensemble des messages forme ce que l'on appelle l'interface de l'objet ; c'est seulement au travers de celle-ci que les objets interagissent entre eux. La réponse à la réception d'un message par un objet est appelée une *méthode* (méthode de mise en œuvre du message) ; elle décrit quelle réponse doit être donnée au message.

La structure interne des objets et les messages auxquels ils répondent sont définis dans une classe (c'est une des représentations possibles). Une classe décrit les méthodes de création d'objets de ce type (on parle d'instance de classe), et les méthodes auxquelles répondront les objets de ce type lors de la réception de messages.

En Python En Python, la création d'une classe commence par le mot clé `class` suivi du nom de la classe, de deux points et d'une indentation. Toute la partie du script indentée contient les méthodes de création des instances de cette classe et les méthodes de cette classe.
Lors de la programmation de méthodes dans une classe, on devra faire référence à l'instance de l'objet sur lequel on travaille : le mot clé `self` permet ceci en Python.
La méthode de création d'un objet est également conventionnelle en Python : elle est définie par `def __init__` suivi au minimum de l'argument `self` et éventuellement d'autres arguments. Les attributs des objets de cette classe sont définis dans cette méthode. Ils sont codés sous la forme `self.attribut`.

Par exemple, le code suivant permettrait de définir un objet de type Carte en Python, et une méthode qui renvoie un booléen indiquant si une carte est un honneur (10, V, D, R ou As) :

```
Class Carte:
    def __init__(self, h, v):
        self.hauteur = h
        self.valeur = v

    def honneur(self):
        return self.valeur in ('10', 'Valet', 'Dame', 'Roi', 'As')
```

Le code suivant créerait alors le Valet de Carreau, `C = Carte('Valet', 'Carreau')`
L'expression `C.honneur()` serait évaluée à `True`.

On verra dans ce TP une autre méthode spéciale, `def __repr__(self):` appelée lors de l'instruction `print`.
Il ne s'agit là que d'une micro-introduction à la programmation orientée objet ; un semestre entier (en école...) ne suffira même pas à poser complètement les principes, techniques et intérêts de ce paradigme de programmation.

Premiers pas en Programmation orientée objet : classe Vecteur

0. Ouvrir le fichier `vecteur.py` et l'exécuter.
Taper dans la console `V = Vecteur(1, 2, 3)` puis `print(V)`.
Créer une instance `W` de la classe `Vecteur` représentant le vecteur (4, 5, 6).
Tester dans la console `V.norme()` puis `V.somme(W)`.

1. Écrire une méthode `multipliescalaire(self, k)` qui renvoie à partir du vecteur $\overrightarrow{u}$ et du scalaire k le vecteur $k\overrightarrow{u}$.

2. Écrire une méthode `produitvectoriel` qui calcule le produit vectoriel de deux vecteurs.

3. Écrire une méthode `testorthogonal` qui teste si deux vecteurs $\overrightarrow{u}$ et $\overrightarrow{v}$ sont orthogonaux.

Création d'une classe Pile

La notion de classe permet de créer de nouveaux types de structures. Nous allons nous en servir pour créer une structure pile de deux manières différentes.
La structure Pile, que nous étudions en cours, est une structure de données linéaire qui fonctionne sur le modèle de la pile d'assiettes. Les seules opérations possibles sont le fait de déposer un élément en haut de la pile (on parlera de la fonction `push`) ou d'enlever l'élément du haut de la pile (on parlera de la fonction `pop`). Les piles fonctionnent donc sur le principe du LIFO (*Last in First Out*) :

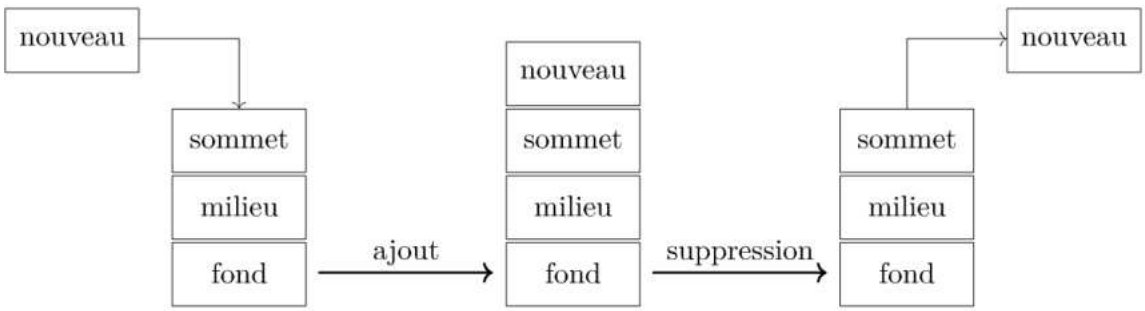


FIGURE 0. Empiler - dépiler

Dans un premier temps, on va se servir de la structure `list` pré-implémentée en Python. Les piles seront représentées en mémoire par des listes. Le sommet de la pile sera le dernier élément de la liste. Ainsi, empiler un élément consistera à ajouter un élément à la fin d'une liste, et effectuer un `pop` consistera à supprimer le dernier élément de la liste en renvoyant sa valeur. Le seul attribut d'un objet pile sera une liste de valeurs.

On s'inspirera évidemment de la partie précédente.

4. Créer une classe `Pile`, et créer une méthode de constructeur de pile vide à l'aide de la syntaxe `def __init__(self)`.
5. Créer une méthode `estvide(self)` qui renvoie un booléen indiquant si la pile est vide ou non.
6. Créer une méthode `push(self, x)` qui empile l'élément `x` sur la pile. Il s'agira donc de rajouter un élément à la fin de la liste qui représente la pile.
7. Créer une méthode `pop(self)` qui renvoie une erreur si la pile est vide, et qui dépile le sommet de la pile en renvoyant sa valeur dans le cas contraire.
8. Créer la méthode `__repr__(self)` qui va être appelée lors de l'affichage d'une pile `P` à l'aide de l'instruction `print(P)`. On souhaite que la pile contenant les valeurs 1, 2 et 3, empilées dans cet ordre, soit affichée de la manière suivante :

```
sommet
|3|
|2|
|1|
fond
```

9. Créer une fonction (en dehors de la classe `Pile`, il ne s'agit pas d'une méthode) qui inverse les deux premiers éléments d'une pile, en ne se servant que des méthodes définies sur les piles.
10. Créer la pile obtenue en empilant successivement 1 puis 2 puis 3 puis 4. Afficher la. Tester la fonction précédente, et afficher le résultat.

Une classe pile alternative

Il est également possible de créer une classe Pile sans se servir de classes déjà pré-implémentées comme la classe List. C'est l'objectif de cette partie.
Nous allons créer à la main une structure de liste chaînée ; pour ce faire, nous allons créer une classe Cellule, qui est donnée dans le document joint `PilesAlt.py` ; et une deuxième classe Pile, qui se servira de cette structure, dont certains éléments sont également donnés dans le fichier joint.
On rappelle qu'une liste chaînée est une structure linéaire, constituée de cellules. Une cellule contient une valeur et un pointeur qui dirige soit vers la cellule suivante, soit vers la valeur `None`, auquel cas la cellule est la « dernière » de la liste chaînée.

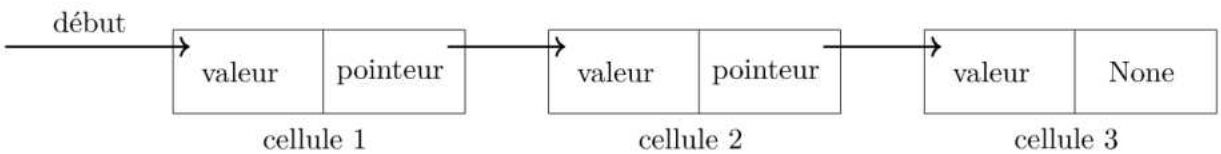


FIGURE 1. Représentation chaînée

11. Compléter la méthode `push` fournie dans le fichier joint.
12. Créer une méthode `pop`.
13. Créer la méthode `__repr__(self)` qui va être appelée lors de l'affichage d'une pile `P` à l'aide de l'instruction `print(P)`.

TP 7.1 – Expressions arithmétiques ☕ = difficile

Avec une pile

On se donne une expression arithmétique écrite en notation classique (infixe) mais **complètement parenthésée** sous la forme d'une liste `Python`, par exemple

```
L = ['(', 4, '+', '(', 5, '-', 7, ')', ')']
```

Chaque élément est soit un nombre (entier), soit une parenthèse (ouvrante ou fermante) soit une opération `+`, `-`, `*` ou `^`.

0. Écrire en utilisant une pile une fonction `verif_parenthese(L)` qui vérifie si l'expression est bien parenthésée.

1. Écrire une fonction `transf_inf_post(L)` toujours avec une pile qui transforme une expression infixe complètement parenthésée en expression postfixe (notation polonaise inversée).
Voici un exemple.

```
>>> L = ['(', 4, '+', '(', 5, '-', 7, ')', ')']
>>> transf_inf_post(L)
[4, 5, 7, '-', '+']
```

2. Écrire une fonction `transf_post_inf(L)` qui transforme une expression postfixe en une expression infixe (classique) complètement parenthésée.
Voici un exemple.

```
>>> L = ['8', '4', '7', '*', '+', '9', '6', '^', '+']
>>> transf_post_inf(L)
'((8+(4*7))+(9^6))'
```

3. Utilisation d'une expression régulière
Utiliser le module `re` (expression régulière) et taper
`import re` (début de fichier)

```
decompose = re.compile("\s*(\d*\.\d+|^[-+]\d+|(?<=\()[-+]\d+|\d+|.)")
```

Puis taper

```
S = '-3+ ((84 + (-47*765))+(3^6))+9+6-5+9+(-6)'
```

Que fait `S = S.replace(' ','')` ?
Expérimenter `LL = decompose.findall(S)`.
Autres expressions que l'on peut tester

```
S = '-3+ ((84 + (-47*765))+(3^6))+9+6-5+9+(-6)'
S = '12+3-(2*2^6^3)^3^2-6^3^(2^1)'
```

4. (☕) Écrire une fonction `convinfpost(L)` qui transforme une expression infixe (non complètement) parenthésée en expression postfixe.
Faire quelques tests, on pourra utiliser `print(eval(S.replace('^','**')))` pour utiliser l'évaluation de l'interpréteur `Python`.
Indication Il serait judicieux d'utiliser un dictionnaire pour gérer les priorités des opérations.

5. (☕) Soit `L = ['2', '^', '5', '^', '2']` représentant une expression infixe.
Que donne votre fonction appliquée à L ? Essayer de corriger ce problème.

Même thème sans piles explicites mais avec des fonctions récursives croisées

On va reprendre l'activité précédente pour résoudre le problème avec des fonctions récursives croisées.
On reprend l'expression régulière :

```
decompose = re.compile("\s*(\d*\. \d+|^[-+]\d+|(?<=\()[-+]\d+|\d+|.)")
L = decompose.findall(S)
```

On va évaluer l'expression au moyen de fonctions récursives croisées qui lisent la liste `L` en la réduisant petit à petit par l'utilisation de `L.pop(0)`.

6. **Évaluation d'une expression avec des + ou des -** sans parenthèse.
   À partir de la simple fonction :

```
def lire_nombre(L):
    """
    version basique
    """
    a = L.pop(0)
    return float(a)
```

   définir la fonction `calcul_expression(L)` qui calcule la valeur de l'expression supposée ne contenir que des + ou des - (pas de parenthèses, pas d'autres opérateurs).
7. **Avec aussi des opérateurs * et ou /**
   Écrire la fonction `lire_terme(L)` qui calcule des expressions avec des opérateurs * et / (et utilise `lire_nombre(L)`).
8. Adapter alors la fonction `calcul_expression(L)` pour qu'elle calcule des expressions avec les quatre opérateurs en tenant compte des priorités (pour l'instant il n'y a pas de parenthèses).
9. Améliorer `lire_nombre(L)` en définissant la fonction `lire_nombrepuiss(L)` pour qu'elle traite l'**opérateur puissance (^).**
   Indication Un nombre est... un nombre ou une expression avec des puissances.
   En principe, l'aspect récursif de votre programme augmente...
10. **Avec des parenthèses**
    Écrire une fonction `lire_nombrepuissparenth(L)` qui étend la calcul de `lire_nombre` dans le cas où `L[0] == '('`. Cette fonction utilise `lire_nombre()`.
    Au final, vous avez défini `lire_nombre`, `lire_nombrepuiss`, `lire_nombrepuissparenth`, `lire_terme` (avec * et /), `calcul_expression` (niveau + et -) avec des jolis appels récursifs croisés.
    Vous pouvez tester vos expressions avec l'instruction `eval` de Python (penser à transformer votre chaîne de caractères par `Spyt = S.replace('^', '**')`.

TP 7.2 – Arbre de barème et lecture d'un fichier texte

Lecture d'un fichier Latex, extraction du barème

Nous disposons d'un fichier texte qui est un énoncé d'un devoir et est écrit dans le langage scientifique Latex.

Cet énoncé comporte des questions, des sous-questions, des sous-sous-questions, etc.

On peut représenter la structure de l'énoncé par un arbre où les feuilles sont les « vraies » questions posées au candidat. Ces dernières suivent un certain barème qui peut éventuellement être précisé dans le fichier à l'aide de la syntaxe `\brm{1,5}` par exemple (1,5 points).

Les questions d'un certain niveau (qui correspondent à une certaine profondeur de l'arbre) sont encadrées par les balises `\begin{enumerate}` et `\end{enumerate}` et chaque question proprement dite commence par le code Latex `\item`.

Le but du TP est de construire l'arbre du barème en lisant le fichier Latex puis d'effectuer quelques opérations sur cet arbre (calcul complet du barème et liste ordonnée des feuilles).

Pour mieux comprendre le but du TP, voici le code Latex du fichier `testbarsimple.tex` (téléchargeable sur la page dédiée à cet ouvrage sur le site de Dunod).

```
\documentclass[12pt,french,a4paper, couleur, python, DS]{lupersoMarc}
\def \llhead{\small Devoir}
\def \rrhead{\small PT \annees}
\def \ttitle{Déterminant}
\begin{document}

\Titre

$\mathbb{\mathbb{K}}$ désigne le corps commutatif $\mathbb{R}$ ou $%
\mathbb{C}$.

\EXO{Problème}
\begin{enumerate}
\item \brm{1} Soit $M=(m_{ij})\in \mathfrak{M}_{n}(\mathbb{K})$ On consid%
ère $M(x)=(m_{ij}+x)$ où $x\in \mathbb{K}$ Montrer que $\det(M(x))$
est une fonction affine.% commentaire bidon

\item \brm{1,5} En déduire une méthode de calcul des déterminants du type
$\left\vert
\begin{array}{cccc}
a &  &  & (c) \\
& a &  &  \\
&  & \ddots  &  \\
(b) &  &  & a%
\end{array}%
\right\vert .$

\item On pose $S_{n}(x)=\dsum\limits_{k=0}^{n-1}\cos \left( a+k\pi x\right) $
% autre commentaire

\begin{enumerate}
\item \brm{2} Calculer $S_{n}(0).$

\item Sous-problème bidon

\begin{enumerate}
\item \brm{1} Sous-sous question 1.

\item \brm{1} Sous-sous question 2.

\item \brm{1} Sous-sous question 3.
\end{enumerate}

\item \brm{1,5} Calculer $S_{n}(1).$
\end{enumerate} % penser à rajouter une indication

\item \brm{3} Calculer $S_{n}(x)$ dans le cas général en utilisant
les nombres complexes.
\end{enumerate}
\end{document}
```

Compilé, on obtient :

DÉTERMINANT

$\mathbb{K}$ désigne le corps commutatif $\mathbb{R}$ ou $\mathbb{C}$.

I Problème — 12 points

1 pt **1)** Soit $M = (m_{ij}) \in \mathfrak{M}_n(\mathbb{K})$ On considère $M(x) = (m_{ij} + x)$ où $x \in \mathbb{K}$ Montrer que $\det(M(x))$ est une fonction affine.

1,5 pt **2)** En déduire une méthode de calcul des déterminants du type $\begin{vmatrix} a & & & (c) \\ & a & & \\ & & \ddots & \\ (b) & & & a \end{vmatrix}$.

3) On pose $S_n(x) = \sum_{k=0}^{n-1} \cos(a + k\pi x)$

2 pts **a)** Calculer $S_n(0)$.

b) Sous-problème bidon

1 pt i. Sous-sous question 1.

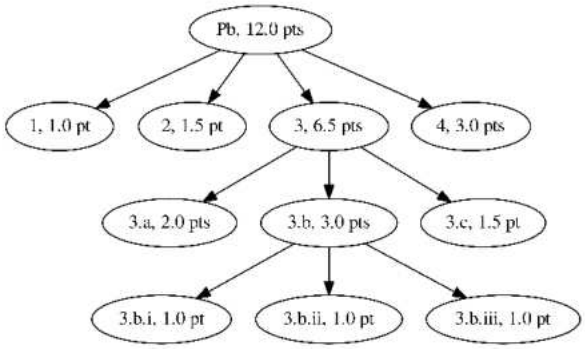
1 pt ii. Sous-sous question 2.

1 pt iii. Sous-sous question 3.

1,5 pt **c)** Calculer $S_n(1)$.

3 pts **4)** Calculer $S_n(x)$ dans le cas général en utilisant les nombres complexes.

ce qui donne l'arbre de barème suivant



Nous allons utiliser le module `re` qui gère les *expressions régulières* pour la recherche et le remplacement de motifs dans les chaînes de caractères.

```
import re

REGBEGINENUM = r"\\begin\{enumerate\}"
# permet de rechercher \begin{enumerate}
regbeginenum = re.compile(REGBEGINENUM)

REGENDENUM = r"\\end\{enumerate\}"
# permet de rechercher \end{enumerate}
```

```
regendenum = re.compile(REGENDENUM)
# permet de rechercher \item
REGITEM = r"\\item"  # très améliorable...
regitem = re.compile(REGITEM)

REGBRM = r"\\brm\{(.*?)\}"
# permet de rechercher \brm{pts} et on récupèrera pts...
regbrm = re.compile(REGBRM)
```

Nous nous contenterons d'utiliser le code suivant pour tester par exemple si la chaîne de caractères contient `\begin{enumerate}`.
(le `r` devant les guillemets permet d'éviter que Python n'interprète les caractères échappés).

```
ligne = r"""
Ceci est du texte
où il y a bien
\begin{enumerate}
"""
m = regbeginenum.search(ligne)
if m:
    print("Oui, c'est bien présent")
else:
    print("Non, pas de enumerate...")
```

```
Oui, c'est bien présent
```

On utilisera aussi le code plus sophistiqué suivant

```
ligne = r"""
Ceci est du texte
où il y a bien
\brm{12,7}
"""
m = regbrm.search(ligne)
if m:
    nb = regbrm.findall(ligne)
    nb = nb[0]
    nb = nb.replace(',', '.')
    nb = float(nb)
    print(r"J'ai trouvé le nombre suivant dans \brm :", nb)
```

```
J'ai trouvé le nombre suivant dans \brm : 12.7
```

0. Après avoir récupéré le fichier `testbarsimple.tex`, le lire et récupérer la liste des lignes de ce fichier en écrivant une fonction `litfichier(nomfichier)`.
On utilisera la fonction ainsi

```
Llignes = litfichier('testbarsimple.tex')
```

1. Les commentaires en langage Latex débutent par le caractère `%` (l'équivalent de `#` en Python). Proposer une fonction `nettoiecommentaire(L)` qui à partir d'une liste de chaînes de caractères (les lignes du fichiers texte) renvoie la liste nettoyée de tous les commentaires éventuels.
Indication Utiliser la méthode `split` de la classe (type) `str`.

```
L = ["ceci est un %test vraiment", "oui, c'est %coucou voilà"]
nettoiecommentaire(L)
```

```
['ceci est un ', "oui, c'est "]
```

2. Pour cette question, l'utilisation d'une **pile** sera très utile...
Écrire une fonction `trouvequestion(Llignes)` qui à partir d'une liste de lignes de textes en Latex renvoie une liste de couples (n° question, profondeur, pt éventuel), où pt éventuel = 0 par défaut sauf si `\brm` est trouvé, la profondeur commence à 0.
À titre de vérification, voici ce que donne le fichier `testbarsimple.tex`

```
Llignes = nettoiecommentaire(Llignes)
L = trouvequestion(Llignes)
print(L)
```

```
[[1, 0, 1.0], [2, 0, 1.5], [3, 0, 0], [1, 1, 2.0],
 [2, 1, 0], [1, 2, 1.0], [2, 2, 1.0], [3, 2, 1.0],
 [3, 1, 1.5], [4, 0, 3.0]]
```

Affichage des questions avec leur référence

3. On se propose de lister les questions comme figurant sur l'image précédente.
Afin de respecter les types de numérotations suivant la profondeur, on utilisera la fonction suivante

```
Lalpha = ['i', 'ii', 'iii', 'iv', 'v', 'vi', 'vii', 'viii', 'ix', 'x']
# on va se limiter à 10...


def cv(num, prof):
    if prof == 1:
        return chr(ord('a') + num - 1)
    elif prof == 2:
        return Lalpha[num - 1]
    elif prof == 3:
        return chr(ord('A') + num - 1)
    else:
        return str(num)
```

Comprendre ce que fait la fonction `cv` et écrire la fonction `affichequest(L)` qui à partir de la liste des questions sous la forme (numéro, profondeur, points), obtenue à partir de la fonction `trouvequestion`, affiche la liste de toutes les questions et sous-questions avec leur référence absolue (par exemple 3.a.ii).

```
affichequest(L)
```

```
1
2
3
3.a
3.b
3.b.i
3.b.ii
3.b.iii
3.c
4
```

Construction de l'arbre de barème

4. Structure d'un arbre avec un type nœud

On va utiliser le type `Arbre` construit comme suit

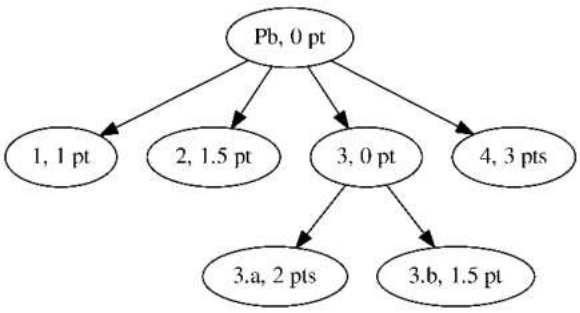
```
class Arbre:
    """ Classe pour représenter des arbres """

    def __init__(self, nom='', pts=0,  fils=[]):
        """
        Crée un arbre à partir
        d'une double étiquette nom, points
        et éventuellement d'une liste de
        sous-arbres (fils)

        si fils == [], le noeud est une feuille.
        """
        # initialisation
        self.nom = ''
        self.pts = 0
        self.fils = []  # ne pas mettre fils !
        if nom != '':
            self.nom = nom
        if pts != 0:
            self.pts = pts
        if fils != []:
            self.fils = fils
```

```
    def ajoute(self, a):
        """ Ajoute l'arbre a aux fils de self
(à la fin de la liste)
        """
        # petite précaution
        assert(isinstance(a, Arbre))
        self.fils.append(a)

    def __repr__(self):
        if isinstance(self.nom, str):
            nomstr = '"{}"'.format(self.nom)
        else:
            nomstr = str(self.nom)
        # si c'est une feuille
        if len(self.fils) == 0:
            return 'Arbre({}, {})'.format(nomstr,
                                          self.pts)
        # sinon
        L = []
        for l in self.fils:  # subtilement récursif
            L.append(repr(l))

        return 'Arbre({}, {}, [{}])'.format(nomstr,
                            self.pts, ", ".join(L))
```

Par exemple, voici un arbre élémentaire

```
a = Arbre("Pb", 0,
          [Arbre("1", 1),
           Arbre("2", 1.5),
           Arbre("3", 0, [Arbre("3.a", 2), Arbre("3.b", 1.5)]),
           Arbre("4", 3)]
          )
```

qui représente l'arbre suivant



Arbre à barème

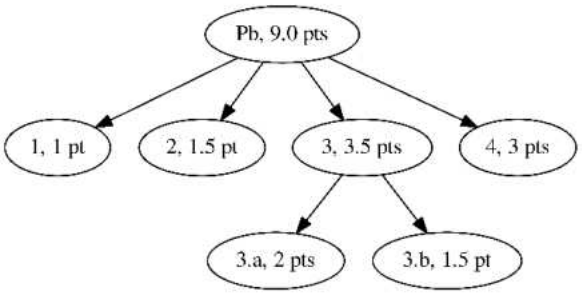
Calcul général du barème

Écrire une fonction **récursive** `calculebareme(a)` qui calcule le barème pour chaque question (= cellule) de l'arbre à partir des feuilles et modifie l'arbre `a` (effet de bord).

```
calculebareme(a)
```

```
Arbre("Pb", 9.0, [
                  Arbre("1", 1), Arbre("2", 1.5), Arbre("3", 3.5,
                            [Arbre("3.a", 2), Arbre("3.b", 1.5)]),
                  Arbre("4", 3)
                 ])
```

dont voici une représentation graphique :



Arbre avec barème rempli

5. Extraction des feuilles
Écrire une fonction `listefeuilles(a)` qui liste les feuilles de l'arbre `a` (de gauche à droite). Ceci donne avec l'exemple précédent

```
listefeuilles(a)
```

```
[('Pb.1', 1), ('Pb.2', 1.5), ('Pb.3.a', 2), ('Pb.3.b', 1.5), ('Pb.4', 3)]
```

6. Écrire la fonction `definitarbre(L)` qui construit un arbre de barème à partir de la liste `L` (obtenue à partir de la fonction `trouvequestion`).
Pour l'instant, le nom d'un nœud sera un entier qui représente le numéro de la question.

```
a = definitarbre(L)
print(a)
```

```
Arbre("Pb", 0, [Arbre(1, 1.0), Arbre(2, 1.5),
                Arbre(3, 0,
                  [Arbre(1, 2.0),
                   Arbre(2, 0, [Arbre(1, 1.0), Arbre(2, 1.0), Arbre(3, 1.0)]), Arbre(3, 1.5)]),
                Arbre(4, 3.0)])
```

7. Écrire la fonction `calculequestabs(a)` qui renomme les nœuds de l'arbre en replaçant le numéro de la question par sa dénomination « absolue » sous forme de chaîne de caractères, par exemple 3.b.ii.

```
calculequestabs(a)
print(a)
```

```
Arbre("Pb", 0, [Arbre("1", 1.0), Arbre("2", 1.5),
                Arbre("3", 0,
                  [Arbre("3.a", 2.0),
```

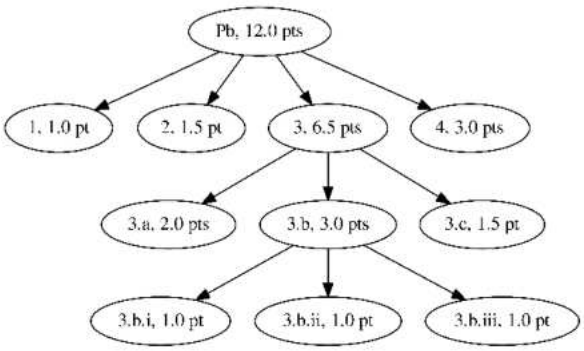
```
                Arbre("3.b", 0,
                      [Arbre("3.b.i", 1.0), Arbre("3.b.ii", 1.0), Arbre("3.b.iii", 1.0)]),
                Arbre("3.c", 1.5)]),
            Arbre("4", 3.0)])
```

8. Calculons le barème complet

```
calculebareme(a)
```

```
Arbre("Pb", 12.0, [Arbre("1", 1.0), Arbre("2", 1.5),
                   Arbre("3", 6.5,
                     [Arbre("3.a", 2.0),
                      Arbre("3.b", 3.0,
                            [Arbre("3.b.i", 1.0), Arbre("3.b.ii", 1.0), Arbre("3.b.iii", 1.0)]),
                      Arbre("3.c", 1.5)]),
                   Arbre("4", 3.0)])
```

ou graphiquement



Et listons les feuilles de cette arbre

```
listefeuilles(a)
```

```
[('Pb.1', 1.0),
 ('Pb.2', 1.5),
 ('Pb.3.a', 2.0),
 ('Pb.3.b.i', 1.0),
 ('Pb.3.b.ii', 1.0),
 ('Pb.3.b.iii', 1.0),
 ('Pb.3.c', 1.5),
 ('Pb.4', 3.0)]
```

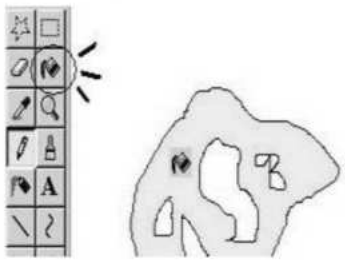
Tester maintenant le programme sur le fichier Latex `testbar.tex` (particulièrement compliqué, téléchargeable sur la page dédiée à cet ouvrage sur le site de Dunod).

TP 7.3 – Coloriage récursif

Un **GUI** (*Graphical User Interface*) est un module permettant de gérer des **interfaces graphiques** (fenêtres, boutons, menus, barres de défilement, etc.).
Le module Python `tkinter` est un « GUI » modeste mais facile d'utilisation (le logiciel IDLE l'utilise).
Il existe d'autres modules plus complets comme `PyQt4` (par exemple le logiciel `Spyder` l'utilise).
Notre programme d'informatique ne demande pas une maîtrise de tels modules (il nous faudrait beaucoup de temps et une culture encyclopédique), on va juste utiliser quelques fonctionnalités.
Les interfaces graphiques utilisent toutes une **programmation orientée objet**.
Avec le module `tkinter`, on crée une fenêtre principale avec `Tk()`.
On écrit `fen = Tk()`.
L'objet `fen` est de type `tkinter.Tk` est la classe « application graphique » de `tkinter`.
Elle possède de nombreuses méthodes, par exemple `title()` ou `mainloop()`, et des données (parfois cachées et gérées en interne).
Dans l'objet `fen` on va mettre des objets graphiques comme des boutons, des zones de textes etc. appelés **widgets** (*windows gadget*) qui eux-mêmes possèdent des données (**attributs**) et des fonctions (**méthodes**).
Quand la construction graphique et le comportement (actions des widgets quand on clique dessus, etc.) auront été définis, on lance la **boucle d'événements** (`fen.mainloop()`) et la gestion des événements est contrôlée en interne par le module `tkinter`.
Télécharger les fichiers `remplissage.py` et `labyrinthe.py` qui possèdent une interface graphique déjà construite : on peut modifier une grille et appeler une fonction avec un bouton.
Il ne vous reste qu'à compléter une fonction récursive pour les deux activités proposées ci-après.

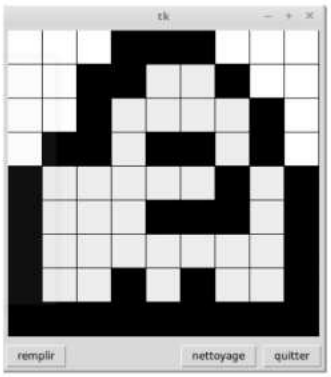
Colorier une zone

On propose de colorier une zone d'un graphique délimitée par une frontière en partant d'un point de coordonnées (i_0, j_0).
On code le graphique par un tableau (par exemple un `array` de `numpy`).
On choisit `tab[i, j] = BLANC` pour une case coloriable (blanche) et `tab[i, j] = NOIR` pour une case en noire qui est susceptible de délimiter la partie à colorier.
On doit définir une fonction `remplit(`i_0`, `j_0`)` qui, partant de la case (i_0, j_0), colorie (par exemple en jaune, codage `tab[i, j] = JAUNE`) autour d'elle mais ne traverse pas les cases noires.
On cherche à imiter la fonction du logiciel paint.



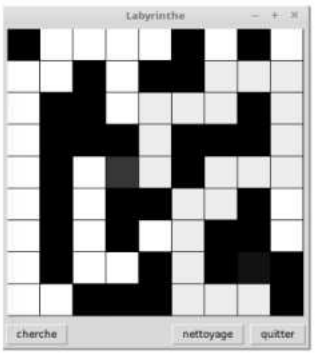
0. Récupérer le fichier `remplissage.py` et compléter la fonction récursive `remplir(i, j)`.
Attention : il faut éventuellement régler les problèmes du bord du cadre.

```
def remplir(i, j):
    '''
    remplit en jaune à partir de la case (i, j)
    une case est déjà jaune si elle est déjà passée par
    cette fonction
    '''
    if tab[i, j] != NOIR and tab[i, j] != JAUNE:
        '''
        si ce n'est pas une case noire et qu'on n'est pas déjà passé
        par cette case alors on remplit...
        '''
        tab[i, j] = JAUNE  # on met en jaune
        # petite animation avec du rouge
        carre(i, j, 'red')
        can.update()
        sleep(0.05)  # Time in seconds.
        carre(i, j, 'yellow')
        can.update()
        sleep(0.01)  # Time in seconds.
        # fin de la petite animation
        pass
        #### A FAIRE ####
```



Trouver le chemin dans un labyrinthe

On veut maintenant trouver un chemin pour aller d'une case à une autre dans un labyrinthe. Proposer un algorithme récursif qui trouve – quand c'est possible – un chemin reliant ces deux cases en reprenant l'exemple précédent.



1. Pour cela, récupérer le fichier `labyrinthe.py` et compléter la fonction `labyrinthe`.

```
def labyrinthe(i, j):
    '''
    construit le labyrinthe avec la case courante (i, j)
    renvoie True si chemin trouvé
```

```
    False si échec
    '''
    pass
    # A VOUS DE JOUER !!!!!
```

TP 7.4 – Transformée de Fourier discrète (récursivité)

On note pour $n \geqslant 2$, $\omega_n = \exp\left(-\frac{2i\pi}{n}\right)$.

À tout vecteur $\mathbf{x} = {}^t(x_0, x_1, \ldots, x_{n-1}) \in \mathbb{C}^n$, on associe le polynôme $P_{\mathbf{x}}(X) = \sum_{k=0}^{n-1} x_k X^k$ ainsi que le vecteur $\widehat{\mathbf{x}} \in \mathbb{C}^n$ défini par $\boxed{\widehat{\mathbf{x}} = {}^t\left(P_{\mathbf{x}}(1), P_{\mathbf{x}}(\omega_n), \ldots, P_{\mathbf{x}}(\omega_n^{n-1})\right)}$

L'application $\mathscr{F}_n : \mathbf{x} \to \widehat{\mathbf{x}}$ est linéaire.

On l'appelle **la transformation de Fourier discrète d'ordre** n (notée aussi DFT_n chez les anglo-saxons).

On a pour tout $k \in [\![0, n-1]\!]$, $\widehat{\mathbf{x}}[k] = P_{\mathbf{x}}\left(\omega_n^k\right) = \sum_{\ell=0}^{n-1} x_\ell \omega_n^{k\ell}$.

Quelques résultats

L'application $\mathscr{F}_n$ est un automorphisme de $\mathbb{C}^n$.

$$\mathrm{mat}_{can.}\left(\mathscr{F}_n\right) = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & \omega_n & & \omega_n^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \cdots & \omega_n^{(n-1)^2} \end{bmatrix} = \left[\omega_n^{k\ell}\right]_{(k,\ell)\in[\![0,n-1]\!]^2} = \mathbf{M}_n$$

La matrice $\mathbf{M}_n$ est inversible d'inverse $\frac{1}{n}\overline{\mathbf{M}_n}$. En conséquence $\mathscr{F}_n^{-1}$ est définie par :

$$\forall \mathbf{x} \in \mathbb{C}^n,\ \mathscr{F}_n^{-1}(\mathbf{x}) = \frac{1}{n} \cdot {}^t\left(P_{\mathbf{x}}(1), P_{\mathbf{x}}(\overline{\omega_n}), \ldots, P_{\mathbf{x}}(\overline{\omega_n}^{n-1})\right)$$

On observera que le calcul effectif de $\mathscr{F}_n^{-1}$ n'est pas plus compliqué que le calcul de $\mathscr{F}_n$.

Une application directe de la méthode de Hörner montre que le nombre d'opérations (dans $\mathbb{C}$) nécessaires à évaluer $\mathscr{F}_n(\mathbf{x})$ est un $\mathscr{O}\left(n^2\right)$. On va voir qu'il est possible de faire beaucoup mieux en adoptant une stratégie « diviser pour régner ».

Méthode naïve

0. Définir la fonction `TFD(x)` où `x` est un tableau (`array`) de nombres flottants qui calcule de manière brutale (avec une matrice de Vandermonde et un produit matriciel) le vecteur $\widehat{\mathbf{x}}$. Définir de même la transformée inverse `TFI(x)`.

Algorithme rapide

L'algorithme suivant s'appelle la transformation de Fourier « rapide ». Elle a été (re)découverte par deux américains Cooley et Tuckey en 1965. On considère aujourd'hui qu'il s'agit de l'un des algorithmes les plus importants en informatique et en traitement du signal.

1. On se limite au cas où $n = 2^p$, avec $p \geqslant 1$. On posera $m = \frac{n}{2}$.

Pour tout vecteur $\mathbf{x} = {}^t(x_0, x_1, \ldots, x_{n-1}) \in \mathbb{C}^n$, on pose

$$\mathbf{x}^{[0]} = {}^t(x_0, x_2, \ldots, x_{n-2}) \in \mathbb{C}^m \quad \text{et} \quad \mathbf{x}^{[1]} = {}^t(x_1, x_3, \ldots, x_{n-1}) \in \mathbb{C}^m$$

On notera aussi

$$\widehat{\mathbf{x}^{[0]}} = \mathscr{F}_m\left(\mathbf{x}^{[0]}\right) \quad \text{et} \quad \widehat{\mathbf{x}^{[1]}} = \mathscr{F}_m\left(\mathbf{x}^{[1]}\right)$$

On dispose alors des relations suivantes pour $j \in [\![0, n-1]\!]$

◦ Si $j \in [\![0, m-1]\!]$,

$$\widehat{\mathbf{x}}[j] = \widehat{\mathbf{x}^{[0]}}[j] + \omega_n^j \cdot \widehat{\mathbf{x}^{[1]}}[j]$$

◦ Si $j \in [\![m, n-1]\!]$,

$$\widehat{\mathbf{x}}[j] = \widehat{\mathbf{x}^{[0]}}[j-m] + \omega_n^j \cdot \widehat{\mathbf{x}^{[1]}}[j-m]$$

Le lecteur intéressé pourra lire la démonstration de ce résultat dans le corrigé.
On voit donc que pour calculer la transformation de Fourier discrète d'un vecteur de longueur n, il suffit de calculer deux transformations de Fourier discrètes de vecteurs de longueur $\frac{n}{2}$, puis d'effectuer $\frac{n}{2}$ multiplications et n additions.
En effet, une fois calculé les $\widehat{\mathbf{x}}[j]$ pour $j = [\![0, m-1]\!]$, les autres coefficients s'obtiennent en remarquant que

$$\boxed{\widehat{\mathbf{x}}[j] = \widehat{\mathbf{x}^{[0]}}[j-m] + \omega_n^j \cdot \widehat{\mathbf{x}^{[1]}}[j-m] = \widehat{\mathbf{x}^{[0]}}[j-m] - \omega_n^{j-m} \cdot \widehat{\mathbf{x}^{[1]}}[j-m]}$$

car $\omega_n^m = -1$ et les produits $\omega_n^{j-m} \cdot \widehat{\mathbf{x}^{[1]}}[j-m]$ ont déjà été calculés.
Il reste donc à effectuer $\frac{n}{2}$ additions.
Notons $C(n)$ le nombre de multiplications dans $\mathbb{C}$ nécessaires au calcul de $\mathscr{F}_n(x)$.
On a la relation

$$C(n) = 2 \cdot C\left(\frac{n}{2}\right) + \frac{n}{2}$$

Montrer que $\boxed{C(n) = n \cdot \log_2(n)}$

2. Définir la fonction `FFT(x)` où `x` est un tableau (`array`) de nombres flottants qui calcule de manière récursive le vecteur $\widehat{\mathbf{x}}$. Définir de même la tranformée inverse `FFTI(x)`.
Indication On pourra utiliser la fonction `np.concatenate` qui concatène les tableaux. Pour effectuer les tests, on pourra comparer les résultats de `FFT(x)` avec la fonction `fft(x)` du sous-module `numpy.fft`.

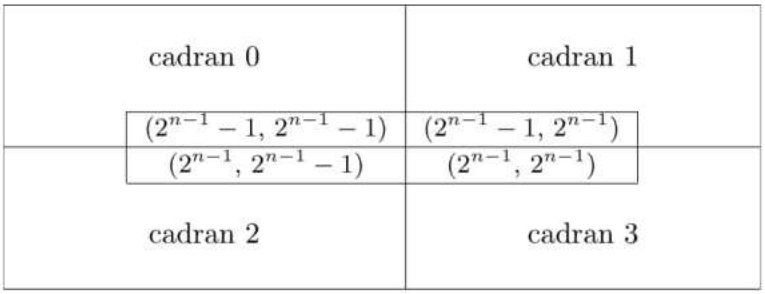
Remarque Il existe une manière itérative reprenant l'analyse précédente mais on se contentera de cette approche.

TP 7.5 – Pavage par des triminos (récursivité)

0. On considère une grille carrée de taille 2^n ($n \geqslant 1$). On enlève une case que l'on appellera la **graine** à cette grille.
Montrer que l'on peut paver la grille (moins cette case) par des « triminos[1] », dont voici un exemplaire :

Indication On pourra raisonner à partir du centre de la grille en s'appuyant sur la figure ci-dessous.

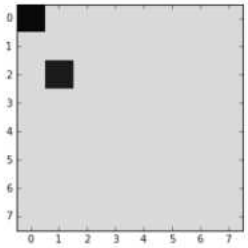
1. Merci à Laurent Schwald, lycée Poincaré, Nancy pour cette suggestion de TP.



Résolution par coloriage d'une grille

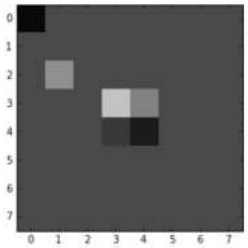
1. On va utiliser un tableau `array` de taille $2^n \times 2^n$ représentant la grille et la fonction suivante `dessine(grille)` pour afficher le tableau (les cases seront coloriées en fonction de leur valeur).

```
def creation_grille(n):
    return np.zeros([2**n, 2**n])


def dessine(grille):
    plt.imshow(grille, interpolation='nearest')


grille = creation_grille(3)
grille[2, 1] = 1
grille[0, 0] = -2
dessine(grille)
plt.show()
```



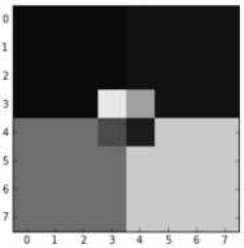
Écrire une fonction `quatremilieux(grille)` qui à partir d'une grille renvoie la liste (sous forme de couples) des quatre cases du milieu.

```
sachet = quatremilieux(grille)
for i in range(4):
    grille[sachet[i]] = i + 5

dessine(grille)
```



2. Écrire la fonction `quatrecadrans(grille)` qui renvoie une vue sur les quatre cadrans en respectant l'ordre de la figure du début de l'énoncé.

```
Lcadrans = quatrecadrans(grille)
for i in range(4):
    Lcadrans[i][:, :] = i

for i in range(4):
    grille[sachet[i]] = i + 5

dessine(grille)
```



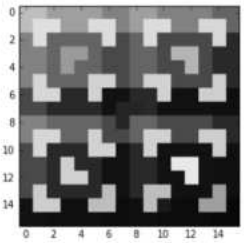
3. Écrire la fonction `detectioncadran(n, graine)` qui renvoie le numéro de cadran correspondant à la graine `graine` (couple) en utilisant les numéros de la figure du début de l'énoncé pour une grille de taille $n \times n$.

4. Écrire la fonction `selection_pavage(n, graine, num_graine)` qui renvoie la liste constituée de la graine et des trois cases du milieu de la grille, correspondant aux cadrans où ne figure pas la graine (qui est dans le cadran `num_graine`) pour une grille de taille $n \times n$.

5. Écrire la fonction `coloriage_trimino(grille, sachet, num_graine, couleur)` qui remplit les trois cases du milieu de la grille avec la valeur `couleur`.

6. Enfin, écrire la fonction `pavage_trimino(grille, graine, couleur)` qui colorie récursivement le tableau `grille`. On pourra, une fois colorié le trimino du milieu de la grille, appeler récursivement la fonction sur les quatre cadrans en baissant la couleur de `couleur - 5 - i` où `i` est le numéro du cadran.
Voici un petit test :

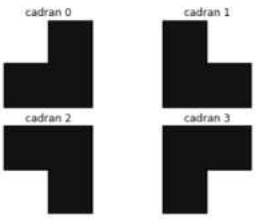
```
taille = 4
T = creation_grille(taille)
deb = 4  # couleur de la graine de départ
graine = (14, 10)
fig = plt.figure()

T[graine] = 5  # on marque la graine par une couleur plus chaude que les autres
pavage_trimino(T, graine, deb)
dessine(T)
```



Amélioration du tracé : liste des triminos

7. Nous allons améliorer le tracé en constituant la liste des triminos que nous tracerons ensuite avec la fonction `plt.fill`. Voici pour gagner du temps les quatre triminos fondamentaux :

```
Trim0 = np.array([(0, 1), (0, 2), (2, 2), (2, 0), (1, 0), (1, 1), (0, 1)])
Trim3 = 2 - Trim0
Trim2 = np.array([2 - Trim0[:, 1], Trim0[:, 0]]).T
Trim1 = 2 - Trim2


def desTrim(T):
    '''
    on garde l'orientation des axes comme pour imshow
    c'est-à-dire la représentation matricielle (i, j)
    i numéro de ligne (en descendant)
    j numéro de colonne
    '''
    plt.fill(T[:, 1], -T[:, 0], lw=2)
    # c'est pourquoi on inverse x et y et on décroit l'axe des y...


Polytrim = [Trim0, Trim1, Trim2, Trim3]
Lfig = ['221', '222', '223', '224']

for i in range(4):
    plt.subplot(Lfig[i], title='cadran {}'.format(i))
    plt.axis('equal')
    plt.axis('off')
    desTrim(Polytrim[i])

plt.show()
```

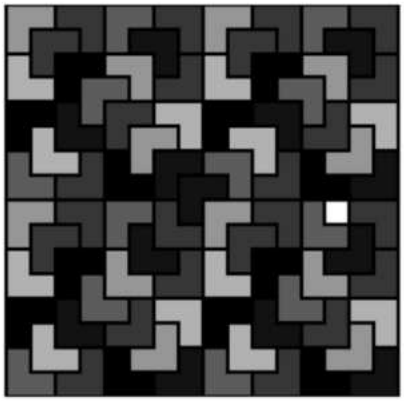


Réécrire la fonction `quatrecadrans(grille)` où maintenant `grille` est un triplet `(i, j, n)` où `(i, j])` est la coordonnée du coin en haut à gauche et `n` est la taille de la grille carrée.

8. Écrire la fonction `rajliste_trimino(grille, num_graine, Ltrim)` qui rajoute le trimino de numéro `num_graine` avec les bonnes coordonnées absolues dans la liste de triminos `Ltrim`. `grille` représente un triplet comme précédemment.

9. Réécrire la fonction `pavage_trimino(grille, graine, Ltrim)` qui remplit cette fois-ci la liste de triminos `Ltrim` et la tester comme ci-dessous :

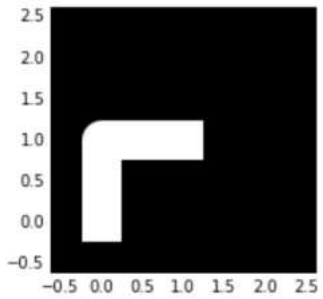
```
Ltrim = []

taille = 4
graine = (8, 13)

fig = plt.figure()
plt.axis('equal')
plt.axis('off')
grille = (0, 0, 2**taille)
pavage_trimino(grille, graine, Ltrim)
for trim in Ltrim[:]:
    desTrim(trim)

# plt.savefig('monpavage.png')  # si vous voulez sauver votre figure
plt.show()
```



TP 7.6 – Labyrinthe parfait (pile/récursivité) ☕ = difficile

On va se servir de `matplotlib` pour tracer un labyrinthe de la manière suivante :
Exemple de tracé case (0,0) → case (0,1) → case (1,1)

```
N = 3
fig = plt.figure(figsize=(N, N), facecolor='w')
plt.subplot('111', axisbg='white')
eps = .6
ax = plt.axis([-eps, N - 1 + eps, -eps, N - 1 + eps],
                                        axisbg='b')
plt.plot([0, 0, 1], [0, 1, 1], 'w', lw=25)
plt.show()
```



On définit un tableau `array` représentant les cases déjà construites d'un labyrinthe en devenir. Chaque case est donc un booléen, `True` voulant dire que la case a déjà été traitée.

```
N = 10  # taille du labyrinthe carré, var globale
T = np.zeros((N, N), dtype=bool)
# var globale, dit si la case a déjà été traitée pour construire le
# labyrinthe ou non
```

Labyrinthe parfait avec une pile

0. Définir une fonction `dirpossible(c)` qui renvoie la liste des cases non déjà traitées (dans `T`) valables à partir d'une case `c = (i,j)`.
On pourra se servir pour la suite de la fonction suivante.

```
import random as rd


def choix(L):
    n = len(L)
    return L[rd.randint(0, n - 1)]
```

1. Un labyrinthe est dit parfait si deux cases quelconques sont reliées par un chemin et un seul. Il n'y a donc pas de cycle (c'est un arbre...).
On se propose de générer un tel labyrinthe en utilisant une pile.
Pour cela, on commence par la case (0, 0), que l'on marque comme construite et on met cette case dans une pile des cases dont il faut visiter les voisins.
Ensuite, tant que notre pile est non vide :

- on dépile une case et on visite aléatoirement les cases accessibles à partir de celle-ci ;
- on lance un `plot` pour visualiser à terme ce chemin ;
- on réempile ensuite cette case, puis la nouvelle case s'il y en a.

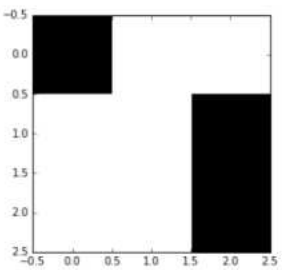
S'il n'y a pas de voisin, on ne fait rien de plus.
Écrire cet algorithme et le tester en visualisant les labyrinthes obtenus.

Labyrinthe parfait avec une fonction récursive

2. Réécrire l'algorithme précédent avec une fonction récursive mais pas de pile (ou plutôt : la pile est cachée dans les appels récursifs...).

3. On se propose de modifier le code précédent pour récupérer le labyrinthe sous forme d'un tableau `array` de cases noires ou blanches. Pour chaque case, on indique donc s'il s'agit d'un couloir ou d'un mur.
On pourra alors afficher le labyrinthe à l'aide des fonctions suivantes :

```
def affiche(Tab):
    plt.imshow(Tab, interpolation='nearest', cmap='Greys')
    plt.show()

NOIR = 2
BLANC = 0
GRIS = 1

Tab = np.array([[NOIR, BLANC, BLANC],
                [BLANC, BLANC, NOIR],
                [BLANC, BLANC, NOIR]], dtype=int)
affiche(Tab)
```

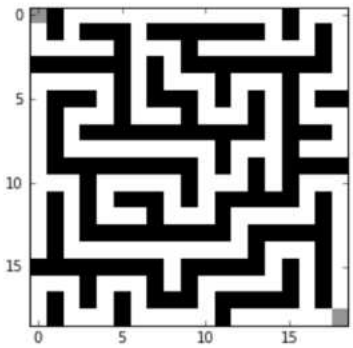


Écrire le code.

4. À présent, on va écrire un programme permettant de trouver le chemin entre deux cases blanches (couloir).

Par exemple, en supposant les cases `[0, 0]` et `[2*N-2, 2*N-2]` blanches, on peut visualiser le labyrinthe.

```
Tlab2[0, 0] = GRIS  # case de départ
Tlab2[-1, -1] = GRIS  # case d'arrivée
affiche(Tlab2)
```



Écrire un algorithme récursif qui permet de trouver un chemin entre deux cases du labyrinthe.

5. Adapter le code précédent en écrivant un code itératif qui utilise une pile stockant les cases à visiter et qui détermine s'il existe bien un chemin.

6. (☕) Améliorer le code précédent pour tracer le chemin.
Indication Utiliser une deuxième pile représentant le chemin et modifier la pile des cases à visiter en mettant le nombre de possibilités de visite pour chaque case à visiter : (`True/False`, nombre de visites possibles (1 par défaut)).

Corrections des exercices

Corrigé exo 7.0

0. On obtient :

```
def deversepile(p1, p2):
    while not(p1.estvide()):
        p2.empile(p1.depile())
```

1. La pile de départ est vidée.

2. Modifions la fonction précédente pour qu'elle préserve l'ancienne pile

```
def copiepile(p):
    ''' copie la pile en préservant l'ancienne '''
    paux = Pile()
    deversepile(p, paux)
    p2 = Pile()
    while not(paux.estvide()):
        a = paux.depile()
        p.empile(a)
        p2.empile(a)
    return p2
```

Corrigé exo 7.1

0. On obtient :

```
def echange(p):
    a, b = p.depile(), p.depile()
    p.empile(a)
    p.empile(b)
```

1.

```
def echk(p, k):
    ''' échange le premier et le kieme, k >= 2 '''
    paux = Pile()
    premier = p.depile()
    for i in range(k - 2):
        paux.empile(p.depile())
    kieme = p.depile()
    p.empile(premier)
    deversepile(paux, p)
    p.empile(kieme)
```

Corrigé exo 7.2

0. On obtient :

```
def rotpile(p):
    ''' rotation d'un cran de la pile '''
    if p.estvide():
```

```
        return
    paux = Pile()
    premier = p.depile()
    deversepile(p, paux)
    p.empile(premier)
    deversepile(paux, p)
```

1.

```
def rotk(p, k):
    ''' rotation de k crans de la pile p
    '''
    p2 = Pile()
    p3 = Pile()
    for i in range(k):
        p2.empile(p.depile())
    deversepile(p, p3)
    deversepile(p2, p)
    deversepile(p3, p)
```

Corrigé exo 7.3

0.

```
def separe(p):
    pos = Pile()  # pile des positifs stricts
    neg = Pile()  # pile des négatifs
    while not p.estvide():
        s = p.depile()
        if s > 0:
            pos.empile(s)
        else:
            neg.empile(s)
    while not pos.estvide():  # on met les positifs sur les négatifs
        neg.empile(pos.depile())
    return neg
```

1.

```
def deversepile(p1, p2):
    while not p1.estvide():
        p2.empile(p1.depile())


def extraitpos(p):
    '''
    on récupère les nombres positifs dans l'ordre
    on n'utilise que deux nouvelles piles
    on préserve la pile p d'entrée
    '''
    pos = Pile()  # pile auxiliaire
    # on va y mettre les positifs dans l'ordre inverse
    pret = Pile()  # pile de retour,
    # servira de sauvegarde inversée de p
    while not p.estvide():
        s = p.depile()
        if s > 0:
            pos.empile(s)
        pret.empile(s)  # on sauve

    deversepile(pret, p)
    deversepile(pos, pret)
```

```
    return pret
```

Corrigé exo 7.4

0. Il suffit de parcourir la chaîne `s` en s'assurant que `n_ouv` ne devient jamais négatif et qu'il est nul à la fin du parcours de la chaîne :

```
def expressionbienparentheseev1(s):
    n_ouv = 0
    for ch in s:
        if ch == '(':
            n_ouv += 1
        if ch == ')':
            n_ouv -= 1
            if n_ouv < 0:
                return False
    return n_ouv == 0
```

1. L'expression `'(a+[b+c)+d]'` est par exemple mal parenthésée. Ceci explique pourquoi utiliser trois compteurs pour résoudre ce problème est insuffisant et pourquoi l'usage d'une (seule et non trois !) pile est adapté.

2. On adapte le code précédent en empilant les ouvrants, en dépilant les fermants et en vérifiant leur correspondance avec l'ouvrant dépilé.

```
def expressionbienparenthesee(s):
    P = []
    for ch in s:
        if ch in '([{':
            P.append(ch)
        if ch in ')]}':
            if P == []:
                return False
            x = P.pop()
            if ch == ')' and x != '(' or ch == '[' and x != ']' or ch == '{' and x != '}':
                return False
    return P == []
```

Remarque Le connecteur logique `and` est prioritaire sur le connecteur `or`.
Avec un nombre de parenthèses quelconque, la structure de dictionnaire permettrait d'avoir des tests bien plus simples ; ceci s'écrirait :

```
def expressionbienparenthesee(s):
    dic = {'(': ')', '[': ']', '{': '}'}
    P = []
    for ch in s:
        if ch in dic.keys():
            P.append(ch)
        if ch in dic.values():
            if P == []:
                return False
            x = P.pop()
            if ch != dic[x]:
                return False
    return P == []
```

Corrigé exo 7.5

Voici un code possible de la fonction `evalNPI`.

```
def evalNPI(L):
    p = Pile()
    for a in L:
        if a == '+':
            deuxieme, premier = p.depile(), p.depile()
            p.empile(premier + deuxieme)
        elif a == '-':
            deuxieme, premier = p.depile(), p.depile()
            p.empile(premier - deuxieme)
        elif a == '*':
            deuxieme, premier = p.depile(), p.depile()
            p.empile(premier * deuxieme)
        elif a == '/':
            deuxieme, premier = p.depile(), p.depile()
            p.empile(premier / deuxieme)
        else:
            p.empile(a)
    return p.depile()
```

ce qui donne

```
>>> L = [7, 8, '-', 6, '*', 10, 3, '+', '*']
>>> evalNPI(L)
-78
>>> eval('(7-8)*6*(10+3)')
-78
```

Pour ceux qui connaissent, on peut utiliser un dictionnaire pour factoriser le code et éventuellement utiliser l'instruction Python `isinstance` qui teste le type de la variable proposée :

```
def evalNPI(L):
    dico = {'+': lambda x, y: x + y,
            '-': lambda x, y: x - y,
            '*': lambda x, y: x * y,
            '/': lambda x, y: x / y}
    p = Pile()
    for a in L:
        if a in dico:  # pareil que dico.keys()
            deuxieme, premier = p.depile(), p.depile()
            r = dico[a](premier, deuxieme)
            p.empile(r)
        elif isinstance(a, int) or isinstance(a, float):
            p.empile(a)
        else:
            pass  # par exemple une fonction 'sin'? à traiter
            # ultérieurement...
    return p.depile()   # en principe il ne reste plus qu'un nombre
```

Corrigé exo 7.6

0. Au début du calcul, la matrice E ne contient que des -1 : aucune case n'a été découverte. Nous utilisons une variable k et une pile P dans laquelle sont stockées les cases situées à la distance k de (x_i, y_i) ; k est initialisé à la valeur 0 et P à la valeur $[(x_i, y_i)]$ (on n'oublie pas de modifier $E[x_i, y_i]$). Tant que P est non vide, c'est-à-dire tant que l'on a trouvé de nouvelles cases, on incrémente k et on calcule la nouvelle pile NP ; celle-ci contient les cases accessibles depuis

les cases de P et qui n'ont pas encore été rencontrées. Le calcul de NP se fait en utilisant la fonction auxiliaire `ajout_cases_acc`, qui va dans le même temps mettre à jour la matrice E. Une fois NP calculée, on affecte sa valeur à la variable P.

```
def Distance(xi, yi, n, p):
    E, P, k = np.zeros((n, p), dtype=int), [(xi, yi)], 0
    E[:, :] = -1
    E[xi, yi] = 0

    def ajout_cases_acc(x, y, pile):
        for a, b in Dep:
            nx, ny = x + a, y + b
            if 0 <= nx < n and 0 <= ny < p and E[nx, ny] == -1:
                E[nx, ny] = k
                pile.append((nx, ny))
    while P != []:
        k += 1
        NP = []
        while P != []:
            (x, y) = P.pop()
            ajout_cases_acc(x, y, NP)
        P = NP
    return E
```

```
>>> Distance(0, 0, 6, 3)
array([[0, 3, 2],
       [3, 4, 1],
       [2, 1, 4],
       [3, 2, 3],
       [2, 3, 2],
       [3, 4, 3]])
```

1. La matrice des prédécesseurs $Pred$ est initialement remplie de $(-2, -2)$. Nous reprenons la même méthode, la fonction auxiliaire mettant cette fois-ci à jour la matrice $Pred$. Par convention, le prédécesseur de (x_i, y_i) est $(-1, -1)$.

```
def Chemins_Minimaux(xi, yi, n, p):
    Pred = [[(-2, -2) for x in range(p)] for i in range(n)]
    Pred[xi][yi] = (-1, -1)
    P = [(xi, yi)]

    def ajout_cases_acc(x, y, pile):
        for a, b in Dep:
            nx, ny = x + a, y + b
            if 0 <= nx < n and 0 <= ny < p and Pred[nx][ny] == (-2, -2):
                Pred[nx][ny] = (x, y)
                pile.append((nx, ny))
    while P != []:
        NP = []
        while P != []:
            (x, y) = P.pop()
            ajout_cases_acc(x, y, NP)
        P = NP
    return Pred
```

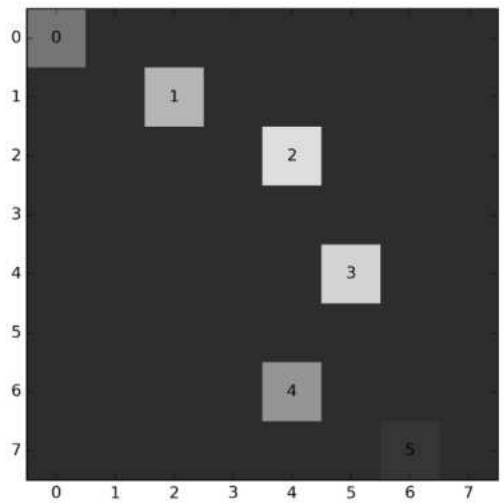
2. On commence par construire le chemin minimal (inversé) L en partant de (x_f, y_f) et en remontant tant que le prédécesseur n'est pas égal à $(-1, -1)$. On parcourt ensuite cette pile L pour annoter le graphique tout en remplissant la matrice E avec les instants de passage. On utilise enfin la fonction `imshow` de la bibliothèque `matplotlib.pyplot` pour tracer l'échiquier (la case $(0, 0)$ est située en haut à gauche, conformément à cette représentation matricielle).

```
def Chemin_Minimal(Pred, xf, yf, n, p):
    if Pred[xf][yf] == (-2, -2):
        print('le cavalier ne peut pas atteindre la case', (xf, yf))
    else:
        x, y = xf, yf
        L = [(xf, yf)]
        nx, ny = Pred[x][y]
        while (nx, ny) != (-1, -1):
            x, y = nx, ny
```

```
            L.append((x, y))
            nx, ny = Pred[x][y]
        E = np.zeros((n, p), dtype=int)
        E[:, :] = -1
        fig = plt.figure()
        ax = fig.add_subplot(111)
        k = 0
        while L != []:
            x, y = L.pop()
            E[x, y] = k
            ax.annotate(str(k), xy=(y, x), va="center", ha="center")
            k += 1
        plt.imshow(E, interpolation='nearest', cmap="rainbow")
        plt.show()
```

Nous obtenons ainsi un chemin minimal de $(0,0)$ à $(7,6)$ dans l'échiquier usuel $E_{8,8}$:

```
>>> Pred = Chemins_Minimaux(0, 0, 8, 8)

>>> Chemin_Minimal(Pred, 7, 6, 8, 8)
```



Corrigé exo 7.7

0. On remplit une pile P initialement vide en testant tour à tour chaque déplacement (a, b) :

```
def cases_accessibles(x, y, E, n, p):
    P = []
    for a, b in Dep:
        if 0 <= x + a < n and 0 <= y + b < p and E[x + a, y + b] == 0:
            P.append((x + a, y + b))
    return P
```

1. Il suffit de traduire l'algorithme proposé, sans oublier de mettre à jour la matrice E et le compteur N :

```
def parcours(xi, yi, n, p):
    E = np.zeros((n, p), dtype=int)
    E[xi, yi] = 1
    N = 1
    taille = n * p
    # pile des coups déjà joués
    Coups = [(xi, yi, cases_accessibles(xi, yi, E, n, p))]
    while N != taille and Coups != []:
        # le calcul n'est pas terminé et il reste des possibilités à explorer
        (x, y, L) = Coups.pop()  # on récupère le dernier coup joué
        if L == []:
            E[x, y] = 0  # 1er cas: on ne peut plus jouer depuis (x,y)
            N -= 1  # et on revient en arrière
        else:
            xs, ys = L.pop()                # 2ème cas : on teste un coup possible
            Coups.append((x, y, L))  # on réempile le coup (x,y)
            # on joue en (xs,ys)
            Coups.append((xs, ys, cases_accessibles(xs, ys, E, n, p)))
            N += 1                                  # on met à jour N
            E[xs, ys] = N                          # on met à jour E
    if N == taille:  # on a trouvé un parcours
        return E
    else:  # on a exploré en vain toutes les possibilités
        return "Il n'existe pas de tour partant de ", (xi, yi)
```

Cet algorithme nous donne un parcours partant de $(0, 0)$ dans l'échiquier $E_{5,5}$:

```
>>> parcours(0, 0, 5, 5)
array([[1, 12,  3, 18, 21],
       [4, 17, 20, 13,  8],
       [11,  2,  7, 22, 19],
       [16,  5, 24,  9, 14],
       [25, 10, 15,  6, 23]]
```

Le lecteur pourra vérifier qu'il n'existe pas de parcours partant de $(1, 2)$ dans l'échiquier $E_{5,5}$, ce qui prouve qu'il n'existe pas de parcours cyclique dans cet échiquier.

Le temps de calcul devient beaucoup trop long quand on veut s'attaquer à l'échiquier usuel $E_{8,8}$. On peut espérer, en cas d'existence d'un parcours, accélérer le calcul en choisissant mieux la case suivante (xs, ys). Cela se fait en définissant une *heuristique*, c'est-à-dire une règle permettant de faire de meilleurs choix. Ici, il peut sembler naturel d'essayer de visiter en premier une case difficilement accessible (si on ne choisit pas cette case comme case suivante, on lui enlève encore une possibilité d'être atteinte). Nous allons donc choisir la case (xs, ys) de L depuis laquelle le nombre de cases accessibles est minimal, ce qui sera fait par la fonction `meilleur_choix` :

```
def meilleur_choix(L, E, n, p):
    P = L
    NP = []
    (x, y) = P.pop()
    t = len(cases_accessibles(x, y, E, n, p))
    while P != []:
        (xs, ys) = P.pop()
        ts = len(cases_accessibles(xs, ys, E, n, p))
        if ts < t:
            NP.append((x, y))
            (x, y) = (xs, ys)
            t = ts
        else:
            NP.append((xs, ys))
    return (x, y), NP
```

Cette fonction parcourt la liste L et retient le meilleur choix (x, y), qui possède t cases accessibles : à chaque étape du calcul, le sommet qui n'a pas été retenu est ajouté à la pile NP initialement vide. Nous obtenons alors la nouvelle fonction :

```
def parcours_heuristique(xi, yi, n, p):
    E = np.zeros((n, p), dtype=int)
    E[xi, yi] = 1
    N = 1
    taille = n * p
    Coups = [(xi, yi, cases_accessibles(xi, yi, E, n, p))]
    while N != taille and Coups != []:
        (x, y, L) = Coups.pop()
        if L == []:
            E[x, y] = 0
            N -= 1
        else:
            # on choisit le meilleur coup
            (xs, ys), NP = meilleur_choix(L, E, n, p)
            Coups.append((x, y, NP))
            Coups.append((xs, ys, cases_accessibles(xs, ys, E, n, p)))
            N += 1
            E[xs, ys] = N
    if N == taille:
        return E
    else:
        return "Il n'existe pas de tour partant de ", (xi, yi)
```

Cette fois, nous obtenons un parcours de l'échiquier $E_{8,8}$ en une fraction de seconde (et une seconde suffit pour l'échiquier $E_{100,100}$) :

```
>>> parcours_heuristique(0, 0, 8, 8)
array([[1, 22,  3, 18, 25, 30, 13, 16],
       [4, 19, 24, 29, 14, 17, 34, 31],
       [23,  2, 21, 26, 49, 32, 15, 12],
       [20,  5, 56, 39, 28, 35, 50, 33],
       [57, 40, 27, 48, 61, 54, 11, 36],
       [6, 43, 60, 55, 38, 47, 64, 51],
       [41, 58, 45,  8, 53, 62, 37, 10],
       [44,  7, 42, 59, 46,  9, 52, 63]])
```

2. Nous reprenons la même méthode que précédemment, mais en ajoutant un booléen *Cyclique* qui teste si la case initiale est accessible depuis la case où se trouve le cavalier. Dans le cas où on ajoute un déplacement (xs, ys), on met à jour ce booléen, qui prend la valeur `True` si et seulement si $(xs, ys) \in \{(2, 1), (1, 2)\}$. On sort de la boucle si la pile *Coups* est vide ou si $N = n \times p$ et *Cyclique* prend la valeur `True`. Cela donne, en utilisant l'heuristique précédente et la représentation utilisée dans l'exercice précédent :

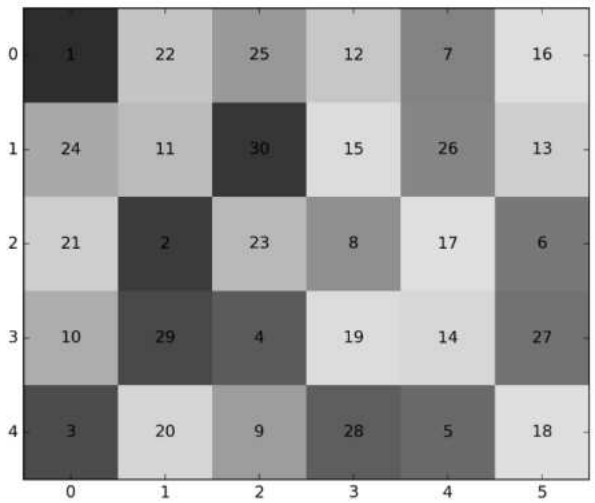
```
def parcours_cyclique_heuristique(n, p):
    E = np.zeros((n, p), dtype=int)
    E[0, 0] = 1
    cyclique = False
    N = 1
    taille = n * p
    Coups = [(0, 0, cases_accessibles(0, 0, E, n, p))]
    while (N != taille or (not cyclique)) and Coups != []:
        (x, y, L) = Coups.pop()
        if L == []:
            E[x, y] = 0
            N -= 1
            cyclique = False
        else:
            (xs, ys), NP = meilleur_choix(L, E, n, p)
```

```
            Coups.append((x, y, NP))
            Coups.append((xs, ys, cases_accessibles(xs, ys, E, n, p)))
            N += 1
            E[xs, ys] = N
            cyclique = ((xs, ys) == (2, 1) or (xs, ys) == (1, 2))
    if N == taille:
        fig = plt.figure()
        ax = fig.add_subplot(111)
        for x in range(n):
            for y in range(p):
                ax.annotate(str(E[x, y]), xy=(y, x), va="center", ha="center")
        plt.imshow(E, interpolation='nearest', cmap="rainbow")
        plt.show()
        return E
    else:
        return "Il n'existe pas de parcours cyclique"
```

L'heuristique n'est malheureusement pas très efficace dès que la taille de l'échiquier augmente, mais cette fonction permet de voir qu'il n'existe pas de parcours cyclique pour $n, m \leqslant 5$, mais qu'il en existe un dans l'échiquier $E_{6,5}$:

```
>>> parcours_cyclique_heuristique(6, 5)
array([[1, 10, 19, 22, 29],
       [18, 27, 30,  9, 20],
       [11,  2, 21, 28, 23],
       [26, 17,  6, 13,  8],
       [3, 12, 15, 24,  5],
       [16, 25,  4,  7, 14]])
```

que l'on peut représenter avec la même méthode que dans l'exercice précédent :



Corrigé exo 7.8

0. En notant (x_M, y_M) les coordonnées d'un point M, on tourne à gauche en B pour aller de A vers C en passant par B si et seulement si $(x_B - x_A)(y_C - y_A) - (y_B - y_A)(x_C - x_A) \geqslant 0$ (il faut éviter de calculer des taux d'accroissement, qui obligent à traiter à part de nombreux cas particuliers).

```
def tourne_gauche(A, B, C):
    return (B[0] - A[0]) * (C[1] - A[1]) >= (B[1] - A[1]) * (C[0] - A[0])
```

1. La fonction calcule le minimum pour l'ordre lexicographique de $\mathbb{R}^2$, avec une variable i_0 qui contient l'indice du point minimal parmi les points déjà étudiés :

```
def point_depart(L):
    i0 = 0
    for i in range(1, len(L)):
        if (L[i][0] < L[i0][0]) or (L[i][0] == L[i0][0] and L[i][1] < L[i0][1]):
            i0 = i
    return i0
```

2. Comme à l'exercice 8.4, nous insérons chaque point M_i (pour $i \neq i_0$) dans la pile triée P :

```
def tri_sommets(L, i0):
    P = []
    for i in range(len(L)):
        if i != i0:  # on va insérer L[i] dans P
            D = []
            while P != [] and tourne_gauche(L[i0], P[-1], L[i]):
                # on commence par dépiler P pour trouver la place de L[i]
                D.append(P.pop())
            P.append(L[i])
            while D != []:
                # on redéverse dans P ce que l'on avait versé dans D
                P.append(D.pop())
    return P
```

3. Pour faciliter la rédaction, notons $M_{i_0} = N_0$ et $P = [N_{n-1}, \ldots, N_1]$. Nous commençons par dépiler N_1 de P et par initialiser E à la pile $[N_0, N_1]$: la propriété $\mathscr{P}$ est clairement vérifiée. Nous allons ensuite traiter chaque élément N_i à tour de rôle, en dépilant P, et en modifiant E de sorte que la propriété $\mathscr{P}$ soit toujours vérifiée. Cela se fait en remarquant que si l'on note B le dernier élément de E et A l'avant-dernier, deux cas se présentent :
a) Si (A, B, N_i) tourne à gauche, il suffit d'ajouter N_i à E pour que $\mathscr{P}$ soit à nouveau vérifiée ;
b) Sinon, on peut supprimer le point B de E sans modifier l'enveloppe convexe.
Autrement dit, on va supprimer la tête de E tant que (A, B, N_i) ne tourne pas à gauche, puis empiler N_i dans E. Cela donne donc :

```
def enveloppe_convexe(L):
    i0 = point_depart(L)  # étape (a)
    P = tri_sommets(L, i0)  # étape (b)
    E = [L[i0], P.pop()]  # étape (c)
    while P != []:  # on va insérer la tête de P
        C = P.pop()
        B = E[-1]
        A = E[-2]
        while not(tourne_gauche(A, B, C)):
            E.pop()  # le point B peut être supprimé
            A, B = E[-2], A
```

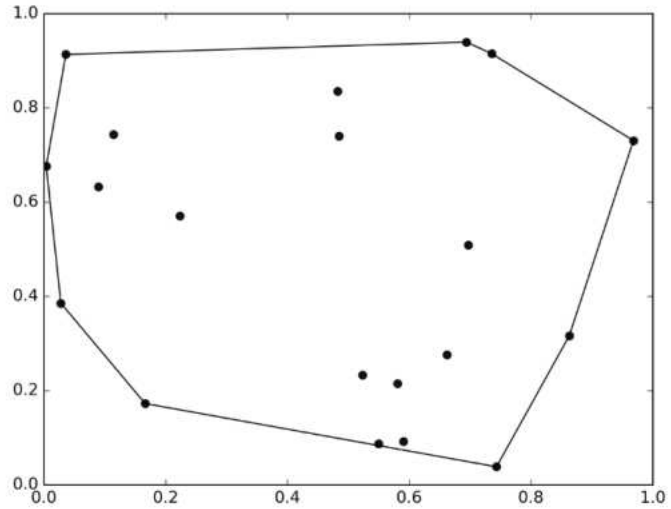
```
        # à la fin de la boucle on a un virage à gauche
        # on empile C au dessus de A et B
        E.append(C)
    return E
```

L'analyse du temps de calcul semble délicate car la seconde boucle, imbriquée dans la première, est difficile à étudier. En revanche, une analyse globale est aisée : chaque point de P étant empilé une et une seule fois dans E, il ne peut être dépilé qu'une fois. On en déduit que le temps de calcul de l'étape (c) est un $\Theta(n)$. Le calcul de i_0 demande également un temps linéaire, mais le tri effectué par l'étape (b) demande un temps quadratique dans le pire des cas, ce qui donne une complexité totale en $\Theta(n^2)$. Il suffirait d'écrire une fonction `tri_sommets` qui travaille en un temps quasi-linéaire, i.e. en $O(n \ln n)$, pour obtenir une fonction `enveloppe_convexe` de complexité quasi-linéaire (ce travail pourra être fait après la lecture du chapitre sur les tris).

4. On utilise la fonction `random` de la bibliothèque `numpy.random` pour simuler les tirages aléatoires :

```
import numpy.random as rd
import matplotlib.pyplot as plt


def tracer(n):
    fig = plt.figure()
    L = [(rd.random(), rd.random()) for i in range(n)]
    # on trace le nuage de point
    x, y = [L[i][0] for i in range(len(L))], [L[i][1] for i in range(len(L))]
    plt.plot(x, y, ".")
    E = enveloppe_convexe(L)
    E.append(E[0])  # on ajoute à fin du calcul le point de départ du contour
    # on trace le contour de l'enveloppe
    X, Y = [E[i][0] for i in range(len(E))], [E[i][1] for i in range(len(E))]
    plt.plot(X, Y)
    plt.show()
```

Nous obtenons ainsi, avec $n = 20$, la figure ci-dessous.



5. Nous commençons par écrire la fonction `N` qui simule la variable N :

```
def N(n):
    L = [(rd.random(), rd.random()) for i in range(n)]
    return len(enveloppe_convexe(L))
```

et on utilise la moyenne et la variance empirique comme estimateurs de l'espérance et de la variance, en effectuant P simulation de la variable N :

```
def estimation(n, P):
    S = 0
    S2 = 0
    for i in range(P):
        a = N(n)
        S += a
        S2 += a**2
        m, m2 = S / P, S2 / P
    return m, m2 - m**2
```

Le problème est de choisir P suffisamment grand, pour que l'estimation soit assez précise, mais pas trop grand pour que le temps de calcul reste raisonnable. Notre propos n'étant pas de rentrer dans des calculs probabilistes précis, contentons-nous de quelques résultats :

```
>>> estimation(100, 10)
(12.08, 3.5936000000000092)
>>> estimation(1000, 100)
(18.01, 6.949899999999957)
```

En multipliant ce type de calcul (il faut améliorer le temps de calcul de la fonction `tri_sommets` si on souhaite augmenter n), on peut conjecturer que le nombre moyen de sommets de l'enveloppe convexe est de l'ordre de $\ln(n)$.

Corrigé exo 7.9

0.

```
def pgcd(a, b):
    if b == 0:
        return a
    else:
        q, r = a // b, a % b
        return pgcd(b, r)
```

1. Partant de la division euclidienne, $a = bq + r$, si on a $bu' + rv' = d$, alors puisque $r = a - bq$ (r est une combinaison entière de a et de b), il vient $d = av' + b(u' - qv')$ donc on a $u = v'$ et $v = u' - qv'$ d'où le code suivant

```
def bezoutrec(a, b):
    if b == 0:
        return a, 1, 0
    else:
        q, r = a // b, a % b
        d, u, v = bezoutrec(b, r)
        return d, v, u - q * v
```

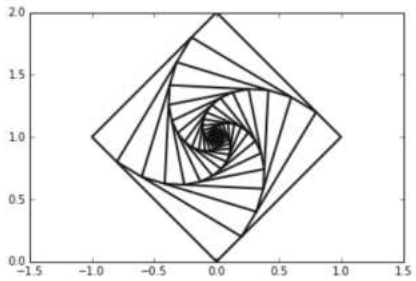
Corrigé exo 7.10

0.

```
def carrerec(a, b, l, eps=0.01):
    """ l entre 0 et 1
    """
    L = [carre(a, b)]
    if abs(b - a) > eps:
        L.extend(carrerec(a + (b - a) * l, b + (b - a) * 1j * l, l, eps))
    return L
```

On teste alors notre fonction

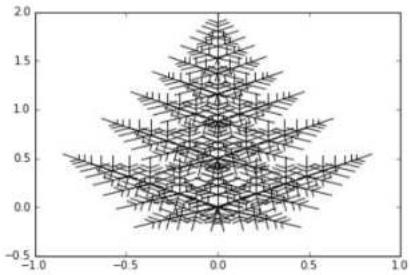
```
BigL = carrerec(0j, 1 + 1j, .2, eps=0.01)
plt.axis('equal')
for L in BigL:
    plt.plot([a.real for a in L], [a.imag for a in L], 'b', lw=2)

plt.show()
```



1.

```
def sapin(a, b, eps=0.1):
    L = []
    if abs(a - b) < eps:
        return [(a, b)]
    L.extend(sapin(a, a + (b - a) * .5 * np.exp(1j), eps))
    L.extend(sapin(a, a + (b - a) * .5 * np.exp(-1j), eps))
    L.append((a, (3 * a + b) / 4))
    L.extend(sapin((3 * a + b) / 4, b, eps))
    return L
```

```
L = sapin(0., 2j)

for a, b in L:
    plt.plot([a.real, b.real], [a.imag, b.imag], 'b')

plt.show()
```



Corrigé exo 7.11

```
import os


def parcours(rep, p):
    ''' parcours récursif de l'arborescence des fichiers
    (pour info: os.walk le fait pour nous) '''
    os.chdir(rep)
    print('p={}, je suis dans {}'.format(p, rep))
    #print('Chemin abs:', os.path.abspath('.'))
    liste = os.listdir()
    # liste des répertoires
    lrep = [l for l in liste if os.path.isdir(l)]
    for l in lrep:
        parcours(l, p + 1)
    os.chdir('..')  # on pourrait tester si p != 0


parcours(r'/home/monnom/DOSSIERpython', 0)
```

Corrigé exo 7.12

0. On utilise le fait qu'une suite décroissante d'entiers naturels est stationnaire (constante à partir d'un certain rang).
Supposons en raisonnant par l'absurde que la suite $(a_n, b_n)_{n\in\mathbb{N}}$ existe. On remarque que la suite d'entiers naturels (a_n) est décroissante, elle est donc stationnaire.
Il existe $n_0 \in \mathbb{N}$ tel que pour tout $n \geqslant n_0$, $a_n = a_{n_0}$.
Mais alors la suite $(b_n)_{n\geqslant n_0}$ est elle-même décroissante.
Il existe $n_1 \geqslant n_0$ tel que pour tout $n \geqslant n_1$, $b_n = b_{n_1}$.
Ainsi, la suite de couples $(a_n, b_n)_{n\geqslant n_1}$ est constante, ce qui contredit le caractère **strictement** décroissant.

1. Supposons qu'il n'existe pas d'élément minimal. En prenant un élément $(a_0, b_0) \in \mathscr{A}$, il existe un élement strictement plus petit $(a_1, b_1) \prec (a_0, b_0)$ et ainsi de suite, on construit ainsi une suite infinie strictement décroissante, ce qui est en contradiction avec ce que l'on a prouvé précédemment.

2. Remarquons que $m_0 \geqslant 1$ (sinon la fonction $\mathsf{ack}(m_0, n_0)$ termine).
Par ailleurs, pour tout $(a, b) \preceq (m_0, n_0)$ distinct de (m_0, n_0), comme $(a, b) \notin \mathscr{A}$, `ack(a, b)` termine. C'est le cas en particulier si $a = m - 1$ ou si $(a, b) = (m_0, n_0 - 1)$.

Or, en examinant le code de la fonction, l'appel ack(m_0, n_0) termine puisqu'il fait appel à $(m_0, n_0 - 1)$ et/ou à $(m_0 - 1, ...)$.

Corrigé exo 7.13

0.

```
# uplets croissants
def upletscrois(n, k):
    S = []

    def upint(L, nb):
        if nb == 0:
            S.append(L)
        else:
            for i in range(n):
                if len(L) == 0 or i > L[-1]:
                    upint(L + [i], nb - 1)
    upint([], k)
    return S
```

1.

```
# uplets arrangements
def upletsarrang(n, k):
    S = []

    def upint(L, nb):
        if nb == 0:
            S.append(L)
        else:
            for i in range(n):
                if not(i in L):
                    upint(L + [i], nb - 1)
    upint([], k)
    return S
```

2.

Partitions d'entiers (sans doublons)

```
# partitions d'entiers distincts
def partition(n):
    ''' partition sans répétition de l'entier n
    '''
    S = []

    def partitionentiersimple(n, L):
        if n == 0:
            S.append(L)
        else:
            if L == []:
                M = n
            else:
                M = min(L[-1] - 1, n)  # le -1 pour distincts
            for d in range(M, 0, -1):
                partitionentiersimple(n - d, L + [d])

    partitionentiersimple(n, [])
    return S
```

```
partition(9)
```

```
[[9], [8, 1], [7, 2], [6, 3], [6, 2, 1], [5, 4], [5, 3, 1], [4, 3, 2]]
```

Partitions d'entiers avec doublons

```
# partitions d'entiers doublons
def partitionrepet(n):
    ''' partition avec répétions de l'entier n '''
    S = []

    def partitionentiermult(n, L):
        if n == 0:
            S.append(L)
        else:
            if L == []:
                M = n
            else:
                M = min(L[-1], n)  # c'est là que se situe la
                # petite différence
            for d in range(M, 0, -1):
                partitionentiermult(n - d, L + [d])

    partitionentiermult(n, [])
    return S
```

```
partitionrepet(7)
```

```
[[7],
 [6, 1],
 [5, 2],
 [5, 1, 1],
 [4, 3],
 [4, 2, 1],
 [4, 1, 1, 1],
 [3, 3, 1],
 [3, 2, 2],
 [3, 2, 1, 1],
 [3, 1, 1, 1, 1],
 [2, 2, 2, 1],
 [2, 2, 1, 1, 1],
 [2, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1]]
```

Corrigé exo 7.14

0. Renvoie une combinaison de cinq nombres parmi $[\![1, 27]\!]$ et ceci 20 fois de suite. On montre que chaque combinaison est équiprobable (question suivante).

1. Tout repose sur l'écriture des probabilités conditionnelles.
En effet, on tire une p combinaison aléatoire parmi $[\![1, n]\!]$ que l'on note $C_{n,p}$ grâce à la fonction `combalea(27, 5, [])`. Fixons A une p combinaison parmi $[\![1, n]\!]$ arbitraire.
On a

$$\mathbb{P}(C_{n,p} = A) = \mathbb{P}(C_{n,p} = A \mid n \in C_{n,p}) \times \mathbb{P}(n \in C_{n,p}) + \mathbb{P}(C = A \mid n \notin C_{n,p}) \times \mathbb{P}(n \notin C_{n,p}).$$

- Si $n \in A$, $\mathbb{P}(C_{n,p} = A) = \mathbb{P}(C_{n,p} = A \mid n \in C_{n,p}) \times \mathbb{P}(n \in C_{n,p})$,
- Si $n \notin A$, $\mathbb{P}(C_{n,p} = A) = \mathbb{P}(C = A \mid n \notin C_{n,p}) \times \mathbb{P}(n \notin C_{n,p})$ et A est en réalité une p combinaison parmi $[\![1, n-1]\!]$.

Or, d'après le programme `combalea`,

$$\mathbb{P}(n \in C_{n,p}) = \frac{p}{n}$$
$$\text{si } n \in A,\ \mathbb{P}(C_{n,p} = A \mid n \in C_{n,p}) = \mathbb{P}(C_{n-1,p-1} = A \setminus \{n\})$$
$$\text{si } n \notin A,\ \mathbb{P}(C_{n,p} = A \mid n \notin C_{n,p}) = \mathbb{P}(C_{n-1,p} = A)$$

On montre alors par récurrence sur $p+n$ que pour une p combinaison A parmi $[\![1, n]\!]$ fixée quelconque,

$$\boxed{\mathbb{P}(C_{n,p} = A) = \frac{1}{\binom{n}{p}}}$$

En effet,

- $n + p = 1$ donne $n = 1$, $p = 0$, $A = \{\}$ seule possibilité et $\mathbb{P}(C_{n,p} = \{\}) = 1$.
- En supposant le résultat vrai au rang $n + p - 1$, il vient :

◦ Si $n \in A$,

$$\begin{aligned}\mathbb{P}(C_{n,p} = A) &= \frac{p}{n} \times \mathbb{P}(C_{n-1,p-1} = A \setminus \{n\}) \\ &= \frac{p}{n} \times \frac{1}{\binom{n-1}{p-1}} \text{ (hypothèse de récurrence)} \\ &= \frac{p}{n} \times \frac{(p-1)!(n-p)!}{(n-1)!} \\ &= \frac{1}{\binom{n}{p}}.\end{aligned}$$

◦ Si $n \notin A$,

$$\begin{aligned}\mathbb{P}(C_{n,p} = A) &= \left(1 - \frac{p}{n}\right) \times \mathbb{P}(C_{n-1,p} = A) \\ &= \left(1 - \frac{p}{n}\right) \times \frac{1}{\binom{n-1}{p}} \text{ (hypothèse de récurrence)} \\ &= \frac{n-p}{n} \times \frac{p!(n-p-1)!}{(n-1)!} \\ &= \frac{1}{\binom{n}{p}}.\end{aligned}$$

Corrigé exo 7.15

0. Voici le code.

```
def hanoi(n, a, b):
    """ n disques à déplacer de a vers b
    a et b entre 0 et 2 """
    c = 3 - (a + b) # astuce
    if n > 0:
        hanoi(n-1, a, c)
        print(a+1,"->", b+1)
        hanoi(n-1, c, b)

hanoi(4, 0, 2)
```

Résultat pour `hanoi(4, 0, 2)` (15 étapes) :

```
1 -> 2                          2 -> 3
1 -> 3                          2 -> 1
2 -> 3                          3 -> 1
1 -> 2                          2 -> 3
3 -> 1                          1 -> 2
3 -> 2                          1 -> 3
1 -> 2                          2 -> 3
1 -> 3
```

1. Voici une écriture itérative de ce programme avec une pile d'appel.

```
from MaPile import *


def hanoiiter(n0, a0, b0):
    """ n disques à déplacer de a vers b
    a et b entre 0 et 2 """
    p = Pile()
    p.empile(('appel', (n0, a0, b0)))
    while not(p.estvide()):
        etat, (n, a, b) = p.depile()
        if etat == 'appel':
            if n > 0:
                c = 3 - (a + b)
                # attention à inverser
                p.empile(('appel', (n - 1, c, b)))
                p.empile(('affiche',
                          ("{} -> {}".format(a + 1, b + 1), 0, 0)))
                p.empile(('appel', (n - 1, a, c)))
        elif etat == 'affiche':
            print(n)

hanoiiter(4, 0, 2)
```

Corrections des TP

Corrigé TP 7.0

1.

```
def multipliescalaire(self, k):
    return Vecteur(k * self.x, k * self.y, k * self.z)
```

2.

```
def produitvectoriel(self, other):
    return Vecteur(self.y * other.z - other.y * self.z,
                   -self.x * other.z + other.x * self.z, self.x * other.y - other.x * self.y)
```

3. On implémente un produit scalaire, dont on teste la nullité en faisant attention au test d'égalité entre flottants :

```
def testorthogonal(self, other):
    return -1e-8 < self.x * other.x + self.y * other.y + self.z * other.z < 1e-8
```

4. à 8. Dans ces questions, on n'utilise pas de paramètres, uniquement `self`. Il est important de comprendre la différence entre l'instance et sa représentation. Pour définir une pile, on tapera `P = Pile()`, qui va créer une pile vide.
La représentation de `P` sera `P.liste`. Les méthodes vont donc travailler sur `P.liste` et non sur `P`.

```
class Pile:

    def __init__(self):
        self.liste = []

    def est_vide(self):
        return self.liste == []

    def push(self, x):
        self.liste.append(x)

    def pop(self):
        assert not self.est_vide()
        return self.liste.pop()

    def __repr__(self):
        ch = "sommet\n"
        for i in range(len(self.liste) - 1, -1, -1):
            ch += "  |" + str(self.liste[i]) + "|\n"
        return ch + " fond"
```

9.

```
def echange(p):  # Tout se fait en place
    if P.est_vide():
        return None
    a = P.pop()
    if P.est_vide():
        P.push(a)  # pas besoin de nouveau "return None" du fait du test if/else
    else:
```

```
        b = P.pop()
        P.push(a)
        P.push(b)
```

10.

```
P = Pile()
for i in range(1, 5):
    P.push(i)
print(P)
echange(P)
print(P)
```

11.

```
def push(self, x):
    C = Cell(x)
    C.next = self.listecell
    self.listecell = C
```

12.

```
def pop(self):
    assert not self.estvide()
    x = self.listecell
    self.listecell = x.next
    return x.val
```

13.

```
def __repr__(self):
    L = []
    while not self.estvide():
        L.append(self.pop())
    n = len(L)
    ch = "sommet\n"
    for i in range(len(L)):
        ch += "  |" + str(L[i]) + "|\n"
        self.push(L[n - 1 - i])
    return ch + " fond"
```

Corrigé TP 7.1

0. On peut par exemple empiler les `'('` et dépiler lorsque l'on rencontre un `')'`.

```
def verif_parenthese(L):
    """ vérifie l'emplacement des parenthèses
    en utilisant une pile
    """
    p = Pile()
    for a in L:
        if a == '(':
            p.empile(a)
        elif a == ')':
            if p.estvide() or p.sommet() != '(':
                return False
            else:
                p.depile()
    return p.estvide()     # True si tout va bien
```

1. On utilise une pile d'opérateurs `p` et on construit petit à petit l'expression post-fixée `Lr`.
   - si on lit un opérateur, on l'empile ;
   - si on lit `')'`, il nous faut mettre l'opérateur en attente en dépilant la pile `p` et en mettant cet opérateur dans `Lr` ;
   - si c'est un nombre, on le place dans `Lr`.

```
def transf_inf_post(L):
    """
    transforme une liste infixe parsée avec parenthèses complètes
    en une liste postfixe
    pas de gestion des priorités
    """
    p = Pile()
    Lr = []
    for a in L:
        if operateur(a):
            p.empile(a)
        elif a == '(':
            pass
        elif a == ')':
            Lr.append(p.depile())
        else:
            Lr.append(a)
    return Lr
```

2. On reprend l'évaluation d'une expression post-fixée (notation polonaise inversée) de l'exercice **7.5** p. 286 mais en manipulant des chaînes de caractères plutôt que des nombres.

```
def transf_post_inf(L):
    """ prend une liste parsée postfixe et renvoie une chaîne
    de caractères en notation infixe avec toutes les parenthèses possibles
    """
    s = ""
    p = Pile()
    for a in L:
        if operateur(a):
            c, d = p.depile(), p.depile()
            s = '(' + str(d) + a + str(c) + ')'
            p.empile(s)
        else:
            p.empile(a)
    return s
```

3. On récupère dans la liste `LL` la liste des objets : nombre (sous forme de chaîne de caractères), opérateurs et parenthèses.
4. On reprend le code de la question 1 en ajoutant la gestion des priorités des opérateurs :
   - si on lit un opérateur, on va l'empiler **mais on calcule à quel niveau on va le placer dans la pile `p` en fonction des priorités** et on met les opérateurs dépilés dans `Lr`,
   - si on lit `'('`, on l'empile (il nous faut repérer le début de cette « sous-évaluation »),
   - si on lit `')'`, on dépile `p` jusqu'à tomber sur `'('` et on met les opérateurs dans `Lr`,
   - si c'est un nombre, on le place dans `Lr`.

```
def convinfpost(L):
    """ convertit une liste infixe parsée
    non complètement parenthésée
    en une liste postfixe
    en gérant les priorités (sauf ^ correctement)
    """
    prio = {'(': 0, '+': 1, '-': 1, '*': 2, '/': 2, '^': 3}
    p = Pile()
```

```
    Lr = []  # liste postfixe à retourner
    for a in L:
        if operateur(a):
            placement = not p.estvide()
            while placement:
                if prio[p.sommet()] >= prio[a]:
                    e = p.depile()
                    Lr.append(e)
                    placement = not p.estvide()
                else:
                    placement = False
            p.empile(a)
        elif a == '(':
            p.empile(a)
        elif a == ')':
            e = p.depile()
            while e!= '(':
                Lr.append(e)
                e = p.depile()
        else:  # c'est un nombre
            Lr.append(a)
    while not p.estvide():
        Lr.append(p.depile())
    return Lr
```

5. Dans le cas de la puissance (`'^'`), on l'empile directement sans attendre[1].

```
def convinfpost2(L):
    """ convertit une liste infixe parsée en une liste postfixe
    en gérant les priorités et 2^3^4 = 2^(3^4) lecture à droite
    pour la puissance
    """
    prio = {'(': 0, '+': 1, '-': 1, '*': 2, '/': 2, '^': 3}
    p = Pile()
    Lr = []
    for a in L:
        if a == '^':
            placement = not p.estvide()
            while placement:
                if prio[p.sommet()] > prio[a]:  # pour l'instant
                                                # ce cas n'existe pas
                    e = p.depile()
                    Lr.append(e)
                    placement = not p.estvide()
                else:
                    placement = False
            p.empile(a)
        elif operateur(a):
            placement = not p.estvide()
            while placement:
                if prio[p.sommet()] >= prio[a]:
                    e = p.depile()
                    Lr.append(e)
                    placement = not p.estvide()
                else:
                    placement = False
            p.empile(a)
        elif a == '(':
            p.empile(a)
        elif a == ')':
            e = p.depile()
            while e!= '(':
                Lr.append(e)
                e = p.depile()
        else:  # c'est un nombre
```

1. Dans le code, on a rajouté l'éventualité d'un opérateur strictement plus prioritaire que (`'^'`).

```
            Lr.append(a)
    while not p.estvide():
        Lr.append(p.depile())
    return Lr
```

Version avec des fonctions récursives croisées

6. Avec la fonction

```
def lire_nombre(L):
    """
    version basique
    """
    a = L.pop(0)
    return float(a)
```

on peut écrire la fonction suivante :

```
def calcul_expression(L):
    """
    version que + et -
    """
    nb = lire_nombre(L)
    while L != [] and L[0] in ['+', '-']:
        op = L.pop(0)
        nb2 = lire_nombre(L)
        if op == '+':
            nb += nb2
        elif op == '-':
            nb -= nb2
    return nb
```

7. On adapte le code précédent pour les opérateurs `*` et `/`.

```
def lire_terme(L):
    """
    version sans puissance, seulement * et /
    """
    nb = lire_nombre(L)
    while L != [] and L[0] in ['*', '/']:
        op = L.pop(0)
        nb2 = lire_nombre(L)
        if op == '*':
            nb *= nb2
        elif op == '/':
            nb /= nb2
    return nb
```

8. On change simplement la ligne `nb2 = lire_nombre(L)` par `nb2 = lire_terme(L)`

```
def calcul_expression(L):
    """
    version avec *, / etc.
    """
    nb = lire_terme(L)  # changement
    while L != [] and L[0] in ['+', '-']:
        op = L.pop(0)
        nb2 = lire_terme(L)  # changement
        if op == '+':
            nb += nb2
        elif op == '-':
            nb -= nb2
    return nb
```

9. Voici le code de la fonction qui lit un nombre vu comme une expression uniquement constituée de nombres d'opérateurs puissance.

```
def lire_nombrepuiss(L):
    """ version incluant les puissances
    """
    nb = lire_nombre(L)
    liste = [nb]
    while L != [] and L[0] == '^':
        L.pop(0)  # élément inutile
        nb2 = lire_nombre(L)
        liste.append(nb2)   # on empile les exposants
    n = 1
    while liste != []:
        nb = liste.pop()
        n = nb ** n
    return n
```

et on adapte la fonction `lire_terme`,

```
def lire_terme(L):
    """
    version avec puissance
    """
    nb = lire_nombrepuiss(L)  # changement
    while L != [] and L[0] in ['*', '/']:
        op = L.pop(0)
        nb2 = lire_nombrepuiss(L)  # changement
        if op == '*':
            nb *= nb2
        elif op == '/':
            nb /= nb2
    return nb
```

10. Voici la version complète avec en plus de l'opérateur puissance, la gestion des parenthèses :

- on ne change pas `calcul_expression` qui utilise `lire_terme` combiné avec les opérateurs + et − ;
- on adapte la fonction `lire_terme` qui utilise maintenant `lire_nombrepuissparenth` combiné avec les opérateurs * et / ;
- on écrit la fonction `lire_nombrepuissparenth` et on adapte la fonction `lire_nombrepuiss`.

```
def lire_terme(L):
    """
    version avec puissance et parenthèse
    nombrecomplet = avec les puissances et
    surtout les parenthèses !!!
    """
    nb = lire_nombrepuissparenth(L)  # changement
    while L != [] and L[0] in ['*', '/']:
        op = L.pop(0)
        nb2 = lire_nombrepuissparenth(L)   # changement
        if op == '*':
            nb *= nb2
        elif op == '/':
            nb /= nb2
    return nb


def lire_nombrepuissparenth(L):
    """ lit un nombre avec puissance et
    avec les parenthèses (supposées bien placées)
    """
    if L[0] == '(':     # gestion des parenthèses
```

```
        L.pop(0)  # on oublie '('
        resultat = calcul_expression(L)
        # en principe L[0] == ')'
        L.pop(0)
        L.insert(0, str(resultat))   # on insère le résultat (au bon endroit)
        resultat = calcul_expression(L)
        return resultat
    else:     # fin gestion des parenthèses
        return lire_nombrepuiss(L)


def lire_nombrepuiss(L):
    """ version incluant les puissances modifiées
        au début un vrai nombre puis ^ puis éventuellement une parenthèse
    """
    nb = lire_nombre(L)
    liste = [nb]
    while L != [] and L[0] == '^':
        L.pop(0)  # élément inutile
        if L[0] == '(':     # gestion des parenthèses
            L.pop(0)  # on oublie '('
            nb2 = calcul_expression(L)
            # en principe L[0] == ')'
            L.pop(0)
        else:
            nb2 = lire_nombre(L)
        liste.append(nb2)   # on empile les exposants
    n = 1
    while liste != []:
        nb = liste.pop()
        n = nb ** n
    return n
```

On peut effectuer quelques tests de la manière suivante

```
def test(s):
    s = s.replace(' ', '')  # on enlève tous les blancs
    L = decompose.findall(s)
    spyt = s.replace('^', '**')
    resultat = calcul_expression(L)
    return resultat, eval(spyt)
```

Cela donne par exemple

```
>>> test('(5+6^2^3)-8*9/5-(8-7*(5+8))')
(1679689.6, 1679689.6)
```

Corrigé TP 7.2

0. On peut utiliser les méthodes de fichier `read` ou `readlines` et la méthode de chaîne de caractères `splitline`.

```
def litfichier(nomf):
    Llignes = []
    try:
        f = open(nomf, "r")
        # Llignes = f.readlines()  # avec les \n
        Llignes = f.read().splitlines()  # sans les \n
        f.close()
    except:
        print("Aïe, pas pu ouvrir", nomf)
```

```
        return []

    return Llignes
```

1. La méthode de chaîne de caractères `split('%')[0]` nous fournit ce que l'on veut.

```
def nettoiecommentaire(L):
    Lret = []
    for ligne in L:
        ligne = ligne.split('%')[0]   # commentaires
        Lret.append(ligne)
    return Lret
```

2. On utilise deux piles. La pile `Pquest` où on empile les questions que l'on trouve au fur et à mesure et la pile `Pilenum` qui empile *à l'avance* le numéro de question à utiliser pour une prochaine question. On démarre à 1 puis on incrémente tant que l'on reste avec des questions de même profondeur. La présence d'un `\begin{enumerate}` indique que l'on progresse en profondeur et on recommence alors à numéroter les questions à partir de 1.

```
def trouvequestion(Llignes):
    '''
    à partir d'une liste de lignes de textes en Latex
    renvoie une liste de couples (num question, profondeur, pt éventuel)
    (pt éventuel = 0 par défaut sauf si \brm est trouvé)
    '''
    # MANIPULATION POUR ELIMINER LES COMMENTAIRES
    Llignes = nettoiecommentaire(Llignes)

    Pquest = []   # liste (numéro question, profondeur, pt)
    prof = -1     # profondeur initiale
    Pilenum = []  # Pile des numéros de question, commencera à 1 en prof 0
    for ligne in Llignes:

        m = regbeginenum.search(ligne)
        if m:  # \begin{enumerate} repéré
            # on démarre le numéro de question potentielle à 1
            Pilenum.append(1)
            prof += 1        # en principe prof = len(Pilenum)
            continue

        m = regendenum.search(ligne)
        if m:  # \end{enumerate} répéré
            prof -= 1
            Pilenum.pop()
            continue

        m = regitem.search(ligne)
        if m:  # \item repéré
            quest = Pilenum.pop()
            Pquest.append([quest, prof, 0])    # 0 est le bareme par défaut
            Pilenum.append(quest + 1)   # attention pas de continue

        # lecture d'un bareme éventuel
        m = regbrm.search(ligne)
        if m:
            nb = regbrm.findall(ligne)
            nb = nb[0]
            nb = nb.replace(',', '.')
            nb = float(nb)
            Pquest[-1][2] = nb  # on met le barême

    return Pquest
```

3. On construit le chemin absolu en suivant l'évolution d'une pile `P` qui s'adapte au parcours de la liste des questions en scrutant les différences relatives de profondeur.

```
def affichequest(L):
    '''
    affiche avec chemin absolu les questions
    '''
    # initialisation
    numa, profavant = 1, 0
    P = ['0']

    for num, prof, bar in L:
        if profavant == prof - 1:
            pass  # on rentre dans un lot de sous-questions
        elif profavant == prof + 1:
            P.pop()
            P.pop()  # on a terminé un lot de sous-questions
        else:
            P.pop()  # on passe à la question suivante, on enlève le numéro précédent

        P.append(cv(num, prof))
        profavant = prof

        print('.'.join(P))
        # on peut aussi bien sûr empiler cette affichage...
```

4. Cela s'écrit très rapidement en utilisant la récursivité.

```
def calculebareme(a):
    '''
    calcule le bareme pour chaque question (= cellule) de l'arbre
    à partir des feuilles et modifie l'arbre a
    (effet de bord)
    '''
    if a.fils == []:
        return a.pts
    else:
        total = sum(calculebareme(b) for b in a.fils)
        a.pts = total
        return total
```

5. On utilise une fonction interne récursive `listeint`[1].

```
def listefeuilles(a, nomexo='Pb'):
    ''' liste les feuilles de l'arbre (de gauche à droite)
    '''
    L = []

    def listeint(arb):
        if arb.fils == []:
            L.append((nomexo + '.' + arb.nom, arb.pts))
        else:
            for arbf in arb.fils:
                listeint(arbf)
                # les questions commencent à 1...
    listeint(a)
    return L
```

6. On utilise une pile des arbres nouvellement créés à chaque nouvelle profondeur.
On rajoute les questions en tenant compte du niveau hiérarchique :

- si le niveau ne bouge pas, on rajoute la question aux fils de l'arbre courant (sommet de la pile) ;

1. Pour ceux qui connaissent, il s'agit d'un parcours en profondeur de l'arbre.

- si le niveau progresse, on créé une nouvelle ramification avec un nouvel arbre que l'on relie à l'arbre courant et on empile ce nouvel arbre ;
- si on redescend à un niveau hiérarchique plus bas, on dépile le nombre nécessaire pour pouvoir rajouter la question au bon niveau de l'arbre ;

et on renvoie ensuite la racine de l'arbre principal.

```
def definitarbre(L):
    '''
    à partir d'une liste de questions hiérarchiques
    crée l'arbre
    '''
    a = Arbre('Pb')
    racine = a

    Pile = [a]
    nivanc = 0  # niveau de départ
    pts = -1  # pas de pts pour l'instant

    for quest, niv, bar in L:
        if niv == nivanc + 1:
            b = Arbre(quest, bar)
            a = Pile[-1]
            a = a.fils[-1]
            a.ajoute(b)
            Pile.append(a)
        elif niv == nivanc:
            a = Pile[-1]
            a.ajoute(Arbre(quest, bar))
        elif niv < nivanc:
            desc = nivanc - niv
            for i in range(desc):
                b = Pile.pop()
            b = Pile[-1]
            b.ajoute(Arbre(quest, bar))

        nivanc = niv

    return racine
```

7. On reprend l'idée de la question 5 en utilisant la récursivité.

```
def calculequestabs(a):
    '''
    renomme les questions en notation absolue
    (parcours en profondeur)
    '''

    def calculequestrec(a, pred='', prof=-1):
        etiq = str(cv(a.nom, prof))
        if pred!= '':
            a.nom = pred + '.' + etiq
        else:
            # Petite subtilité pour éviter de commencer la référence de la
            # question par un point.
            a.nom = etiq

        # on poursuit le parcours de l'arbre
        for b in a.fils:
            calculequestrec(b, a.nom, prof + 1)

    # on oublie la racine...
    for b in a.fils:
        calculequestrec(b, '', 0)  # '' au lieu de a.nom
```

8. On exécute le code suivant :

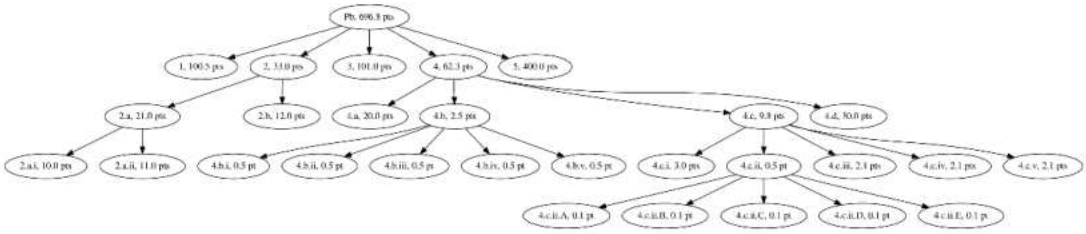
```
Llignes = litfichier('testbar.tex')
Llignes = nettoiecommentaire(Llignes)
L = trouvequestion(Llignes)
a = definitarbre(L)
calculequestabs(a)
print(calculebareme(a))
listefeuilles(a)
```

On obtient

```
[('Pb.1', 100.5),
 ('Pb.2.a.i', 10.0),
 ('Pb.2.a.ii', 11.0),
 ('Pb.2.b', 12.0),
 ('Pb.3', 101.0),
 ('Pb.4.a', 20.0),
 ('Pb.4.b.i', 0.5),
 ('Pb.4.b.ii', 0.5),
 ('Pb.4.b.iii', 0.5),
 ('Pb.4.b.iv', 0.5),
 ('Pb.4.b.v', 0.5),
 ('Pb.4.c.i', 3.0),
 ('Pb.4.c.ii.A', 0.1),
 ('Pb.4.c.ii.B', 0.1),
 ('Pb.4.c.ii.C', 0.1),
 ('Pb.4.c.ii.D', 0.1),
 ('Pb.4.c.ii.E', 0.1),
 ('Pb.4.c.iii', 2.1),
 ('Pb.4.c.iv', 2.1),
 ('Pb.4.c.v', 2.1),
 ('Pb.4.d', 30.0),
 ('Pb.5', 400.0)]
```

```
Arbre("Pb", 696.8,
[Arbre("1", 100.5), Arbre("2", 33.0,
    [Arbre("2.a", 21.0,
        [Arbre("2.a.i", 10.0), Arbre("2.a.ii", 11.0)]),
     Arbre("2.b", 12.0)]),
 Arbre("3", 101.0), Arbre("4", 62.3,
    [Arbre("4.a", 20.0),
     Arbre("4.b", 2.5,
        [Arbre("4.b.i", 0.5), Arbre("4.b.ii", 0.5),
         Arbre("4.b.iii", 0.5), Arbre("4.b.iv", 0.5),
         Arbre("4.b.v", 0.5)]),
     Arbre("4.c", 9.799999999999999,
        [Arbre("4.c.i", 3.0), Arbre("4.c.ii", 0.5,
            [Arbre("4.c.ii.A", 0.1), Arbre("4.c.ii.B", 0.1),
             Arbre("4.c.ii.C", 0.1), Arbre("4.c.ii.D", 0.1),
             Arbre("4.c.ii.E", 0.1)]),
         Arbre("4.c.iii", 2.1),
         Arbre("4.c.iv", 2.1),
         Arbre("4.c.v", 2.1)]),
     Arbre("4.d", 30.0)]),
Arbre("5", 400.0)])
```

Voici l'arbre obtenu graphiquement



On pourra noter le problème de l'approximation décimale (0.1 n'est pas exact en binaire...)

Corrigé TP 7.3

0.

```
def remplir(i, j):
    '''
    remplit en jaune à partir de la case (i, j)
    une case est déjà jaune si elle est déjà passée par
    cette fonction
    '''
    if tab[i][j] != NOIR and tab[i][j] != JAUNE:
        '''
        si ce n'est pas une case noire et qu'on n'est pas déjà passé
        par cette case alors on remplit...
        '''
        tab[i][j] = JAUNE  # on met en jaune
        # petite animation avec du rouge
```

```
        carre(i, j, 'red')
        can.update()
        sleep(0.05)  # Time in seconds.
        carre(i, j, 'yellow')
        can.update()
        sleep(0.01)  # Time in seconds.
        # fin de la petite animation
        if i < long - 1:  # attention au bord si on n'a pas mis de case noire
            remplir(i + 1, j)
        if j > 0:
            remplir(i, j - 1)
        if i > 0:
            remplir(i - 1, j)
        if j < larg - 1:
            remplir(i, j + 1)
```

1.

```
def labyrinthe(i, j):
    '''
    construit le labyrinthe avec la case courante (i, j)
    renvoie True si chemin trouvé
    False si échec
    '''
    if tab[i][j] == BLEU:
        print('Chemin trouvé !')
        return True
    elif tab[i][j] == BLANC:
        '''
        si ce n'est pas une case noire et qu'on n'est pas déjà passé
        par cette case alors on remplit...
        '''
        tab[i][j] = JAUNE  # on met en jaune
        # petite animation avec du rouge
        carre(i, j, 'red')
        can.update()
        sleep(0.1)  # Time in seconds.
        carre(i, j, 'yellow')
        can.update()
        sleep(0.005)  # Time in seconds.
        # fin de la petite animation

        # attention au bord si on n'a pas mis de case noire
        if i < long - 1 and labyrinthe(i + 1, j):
            return True

        if j > 0 and labyrinthe(i, j - 1):
            return True

        if i > 0 and labyrinthe(i - 1, j):
            return True

        if j < larg - 1 and labyrinthe(i, j + 1):
            return True

        tab[i][j] = BLANC  # on remet en blanc
        carre(i, j, 'white')
        return False
    else:  # barrière
        return False
```

Corrigé TP 7.4

0.

```
def TFD(x):
    N = len(x)
    omega = np.exp(-2j * np.pi / N)
    M = [[(omega ** k) ** i for i in range(N)] for k in range(N)]
    Mat = np.array(M)
    return np.dot(Mat, x)


def TFI(x):
    N = len(x)
    omegabar = np.exp(2j * np.pi / N)
    M = [[(omegabar ** k) ** i for i in range(N)] for k in range(N)]
    Mat = np.array(M)
    return np.dot(Mat, x) / N
```

1. $n = 2^p$ et $C(1) = 0$. On pose alors $u_p = C\left(2^p\right)$ et on a

$$u_p = 2u_{p-1} + 2^{p-1}$$

d'où

$$\forall k \geqslant 1,\ \frac{u_k}{2^k} - \frac{u_{k-1}}{2^{k-1}} = 1$$

d'où, par sommation

$$\frac{u_p}{2^p} = \sum_{k=1}^{p} \left(\frac{u_k}{2^k} - \frac{u_{k-1}}{2^{k-1}}\right) = p$$

d'où

$$C(n) = C(2^p) = 2^p \cdot p = \boxed{n \cdot \log_2(n)}$$

Voici la démonstration du résultat technique de l'énoncé.

Démonstration

Pour tout $j \in [\![0, n-1]\!]$,

$$\widehat{\mathbf{x}}[j] = \sum_{k=0}^{n-1} x_k \omega_n^{kj} = \sum_{\ell=0}^{m-1} x_{2\ell} \omega_n^{2\ell j} + \sum_{\ell=0}^{m-1} x_{2\ell+1} \omega_n^{(2\ell+1)j}$$

Or $\omega_n^2 = \exp(\frac{4i\pi}{2m}) = \omega_m$, donc :

$$\widehat{\mathbf{x}}[j] = \sum_{\ell=0}^{m-1} x_{2\ell} \omega_m^{\ell j} + \sum_{\ell=0}^{m-1} x_{2\ell+1} \omega_n^j \cdot \omega_m^{\ell j}$$

On distingue alors deux cas.

1er cas $j \in [\![0, m-1]\!]$. La relation précédente montre que : $\widehat{x}[j] = \widehat{\mathbf{x}^{[0]}}[j] + \omega_n^j \widehat{\mathbf{x}^{[1]}}[j]$.

2^{e} cas $j \in [\![m, n-1]\!]$. Dans ce cas,

$$\begin{aligned}
\widehat{\mathbf{x}}[j] &= \sum_{\ell=0}^{m-1} x_{2\ell} \cdot \omega_m^{\ell j} + \sum_{\ell=0}^{m-1} x_{2\ell+1} \cdot \omega_n^j \cdot \omega_m^{\ell j} \\
&= \sum_{\ell=0}^{m-1} x_{2\ell} \omega_m^{\ell(j-m)} + \omega_n^j \sum_{\ell=0}^{m-1} x_{2\ell+1} \cdot \omega_m^{\ell(j-m)} \\
&= \widehat{\mathbf{x}^{[0]}}[j-m] + \omega_n^j \cdot \widehat{\mathbf{x}^{[1]}}[j-m]
\end{aligned}$$

2.

```
import numpy as np
from numpy.fft import fft   # pour comparer


def FFT(x, inv=False):
    """ FFT récursive
    """
    n = len(x)

    if n == 1:
        return x
    elif n % 2 > 0:
        raise ValueError("n n'est pas une puissance de 2!")
    else:
        X_pair = FFT(x[::2], inv)
        X_impair = FFT(x[1::2], inv)
        if not(inv):
            omega = np.exp(-2j * np.pi * np.arange(n // 2) / n)  # la moitié
        else:
            omega = np.exp(2j * np.pi * np.arange(n // 2) / n)  # conjug
        X1 = omega * X_impair
        return np.concatenate([X_pair + X1, X_pair - X1])


def FFTI(x):
    n = len(x)
    return 1 / n * FFT(x, inv=True)
```

Testons notre fonction en la confrontant à la fonction `fft` du sous-module `numpy.fft`

```
# test
def f(t):
    return np.sin(t) + t**2

T = np.linspace(0, 1, 2**15)
X = f(T)
FX, fx = FFT(X), fft(X)

print(FX - fx)
print('FFT et fft', sum(abs(FX - fx)**2))

Z = FFTI(FFT(X)) - X
print('inverse:', sum(abs(Z)**2))
```

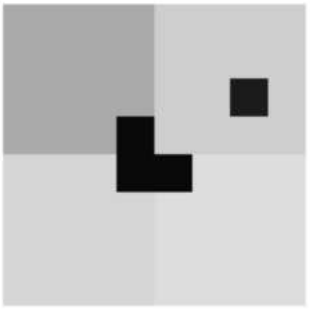
ce qui donne

```
[  0.00000000e+00 +0.00000000e+00j  -9.09494702e-12 +7.27595761e-12j
  -2.55795385e-12 +9.09494702e-13j {\ldots},   2.64890332e-11 +3.63797881e-12j
   3.81987775e-11 +4.54747351e-12j   8.68567440e-11 +1.81898940e-11j]
FFT et fft 1.27528190625e-20
inverse: 3.79052911205e-27
```

Corrigé TP 7.5

0. On raisonne par récurrence sur n, le cas $n = 1$ étant évident.
Supposons que l'on sache paver par des triminos une grille de taille $2^{n-1} \times 2^{n-1}$ moins une graine, montrons que l'on peut le faire pour une grille de taille $2^n \times 2^n$ moins un graine.
On coupe la grille en quatre cadrans et on repère le quadrant contenant la graine : ce cadran est pavable par des triminos par hypothèse de récurrence. Pour les trois autres, il suffit d'enlever

le bon trimino au centre de la grande grille pour obtenir trois cadrans moins une graine (dans un coin à chaque fois) qui eux aussi sont pavables.
Il reste maintenant à implémenter ce raisonnement...
Voici une figure explicative :



1.

```
def quatremilieux(grille):
    ''' on renvoie les quatre cases du milieu
    '''
    dimension, _ = grille.shape
    nt = dimension // 2
    sachet = [(nt - 1, nt - 1), (nt - 1, nt), (nt, nt - 1), (nt, nt)]
    return sachet
```

2.

```
def quatrecadrans(grille):
    ''' on renvoie une vue sur les quatre cadrans
    '''
    n, _ = grille.shape
    nt = n // 2
    Lcadrans = [grille[:nt, :nt], grille[:nt, nt:],
                grille[nt:, :nt], grille[nt:, nt:]]
    return Lcadrans
```

3.

```
def detectioncadran(n, graine):
    nt = n // 2
    ancre = [graine[0] // nt, graine[1] // nt]
    # on utilise un petite astuce la numérotation qu'on a choisie des zones
    # fait qu'ancre donne en base 2 le numéro de la case.
    return 1 * ancre[1] + 2 * ancre[0]
```

4.

```
def renvoie_sachet(n, graine, num_graine):
    # par defaut on renvoie les quatre cases centrales
    # comme dans selection pour le tapis
    nt = n // 2
    sachet = [(nt - 1, nt - 1), (nt - 1, nt), (nt, nt - 1), (nt, nt)]
    # il faut maintenant modifier l'une de ces cases pour
    # qu'elle soit remplacée par la graine
    sachet[num_graine] = graine
    # le sachet contient maintenant les 3 cases du trimino et la graine,
    # dans l'ordre où ils doivent être transmis récursivement aux cadrans.
    return sachet
```

5.

```
def coloriage_trimino(grille, sachet, num_graine, couleur):
    # on colorie les cases de sachet sauf la case num_graine
    for k in range(4):
        if k != num_graine:
            grille[sachet[k]] = couleur
```

6.

```
def pavage_trimino(grille, graine, couleur):
    n, _ = grille.shape
    if n > 1:
        nt = n // 2
        Lcadrans = quatrecadrans(grille)
        num_graine = detectioncadran(n, graine)
        sachet = renvoie_sachet(n, graine, num_graine)
        coloriage_trimino(grille, sachet, num_graine, couleur)
        for i in range(4):
            grille = Lcadrans[i]
            sousgraine = sachet[i]
            # on obtient facilement les coordonnées des graines
            # dans les cadrans, il suffit de calculer modulo nt.
            sousgraine_relative = (sousgraine[0] % nt,
                                   sousgraine[1] % nt)
            pavage_trimino(grille, sousgraine_relative, couleur - 5 - i)
```

7.

```
# réécriture de la fonction


def quatrecadrans(grille):
    ''' on renvoie une vue sur les quatre cadrans
    '''
    i, j, n = grille
    nt = n // 2
    Lcadrans = [(i, j, nt), (i, j + nt, nt),
                (i + nt, j, nt), (i + nt, j + nt, nt)]
    return Lcadrans
```

8.

```
def rajliste_trimino(grille, num_graine, Ltrim):
    global Polytrim
    # on rajoute le trimino de numéro num_graine avec les bonnes coordonnées absolues
    # dans la liste de triminos Ltrim
    trim = Polytrim[num_graine].copy()
    x, y, n = grille
    nt = n // 2
    trim[:, 0] = x + nt - 1 + trim[:, 0]
    trim[:, 1] = y + nt - 1 + trim[:, 1]
    Ltrim.append(trim)
```

9.

```
def pavage_trimino(grille, graine, Ltrim):
    """
    maintenant grille = (x, y, taille)
    """
    x, y, n = grille
    if n > 1:
        nt = n // 2
        Lcadrans = quatrecadrans(grille)
```

```
        num_graine = detectioncadran(n, graine)
        sachet = renvoie_sachet(n, graine, num_graine)
        rajliste_trimino(grille, num_graine, Ltrim)
        for i in range(4):
            grille = Lcadrans[i]
            sousgraine = sachet[i]
            # on obtient facilement les coordonnées des graines
            # dans les cadrans, il suffit de calculer modulo nt.
            sousgraine_relative = (sousgraine[0] % nt,
                                   sousgraine[1] % nt)
            pavage_trimino(grille, sousgraine_relative, Ltrim)
```

Corrigé TP 7.6

0.

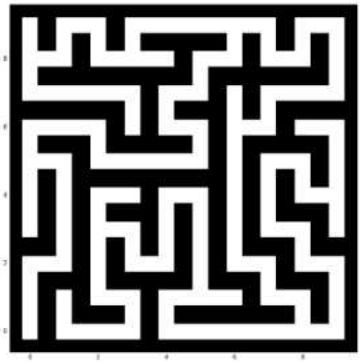
```
def cochecase(c):
    i, j = c
    if 0 <= i < N and 0 <= j < N:
        T[i, j] = True


def caseimpossible(c):
    i, j = c
    if 0 <= i < N and 0 <= j < N:
        return T[i, j]
    else:
        # on est au bord, donc on n'y va pas
        return True

DIR = [(-1, 0), (1, 0), (0, -1), (0, 1)]


def dirpossible(c):
    i, j = c
    L = []

    def metle(cc):
        if not caseimpossible(cc):
            L.append(cc)

    for a, b in DIR:
        metle((i + a, j + b))

    return L
```

1.

```
# labyrinthe avec pile
from pile import Pile

# petite partie graphique
N = 10
fig = plt.figure(figsize=(N, N), facecolor='w')
plt.subplot('111', axisbg='white')
eps = .6
ax = plt.axis([-eps, N - 1 + eps, -eps, N - 1 + eps], axisbg='b')
###

pile = Pile()
c = (0, 0)  # case de départ
pile.empile(c)
cochecase(c)
while not pile.estvide():
```

```
    c = pile.depile()
    L = dirpossible(c)
    if L:
        capres = choix(L)
        plt.plot([c[0], capres[0]], [c[1], capres[1]], 'w',
                 lw=25)
        # plt.pause(.1)   # si on veut une animation
        cochecase(capres)

        pile.empile(c)
        pile.empile(capres)

plt.show()
```

Ce qui donne

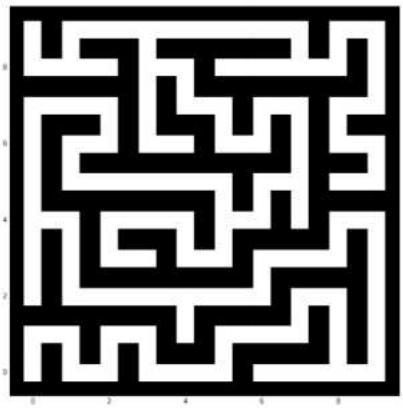


2. C'est bien sûr beaucoup plus court en version récursive.

```
# version récursive (la pile est cachée dans les appels récursifs)

N = 10  # taille du labyrinthe carré, var globale
T = np.zeros((N, N), dtype=bool)

fig = plt.figure(figsize=(N, N), facecolor='w')
plt.subplot('111', axisbg='white')
eps = .6
ax = plt.axis([-eps, N - 1 + eps, -eps, N - 1 + eps], axisbg='b')


def laby(c):
    cochecase(c)
    L = dirpossible(c)
    if L:
        capres = choix(L)
        plt.plot([c[0], capres[0]], [c[1], capres[1]], 'w',
                 lw=25)
        # plt.pause(.1)   # si on veux une animation
        laby(capres)
        laby(c)

laby((0, 0))

plt.show()
```

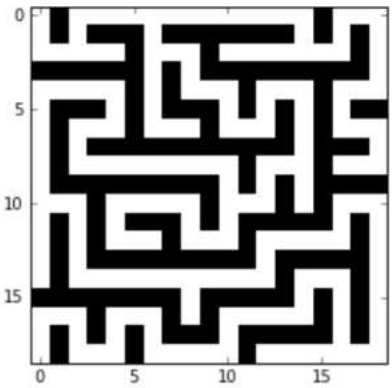
3. Les cases sont quasi-dédoublées (faire une figure), on définit un tableau de taille $(2N-1) \times (2N-1)$.

```
N = 10  # taille du labyrinthe carré, var globale
T = np.zeros((N, N), dtype=bool)
Tlab = np.zeros((2 * N - 1, 2 * N - 1), dtype=int)


NOIR = 2
BLANC = 0
GRIS = 1


Tlab[:, :] = NOIR    # tout est fermé au début



fig = plt.figure(figsize=(N, N), facecolor='w')
plt.subplot('111', axisbg='black')


eps = .6
ax = plt.axis([-eps, N - 1 + eps, -eps, N - 1 + eps],
                                          axisbg='b')



def laby(c):


    cochecase(c)
    L = dirpossible(c)
    if L:


        capres = choix(L)
        plt.plot([c[0], capres[0]], [c[1], capres[1]], 'w',
                 lw=25)


        Tlab[2 * c[0], 2 * c[1]] = BLANC
        Tlab[2 * capres[0], 2 * capres[1]] = BLANC
        i, j = (c[0] + capres[0]), (c[1] + capres[1])


        Tlab[i, j] = BLANC  # la jonction
        laby(capres)
        laby(c)


laby((0, 0))
```



Affichons le nouveau tableau (en faisant attention à l'orientation des axes) :

```
Tlab2 = Tlab.copy()
for i in range(2 * N - 1):
    for j in range(2 * N - 1):
        Tlab2[i, j] = Tlab[j, 2 * N - 2 - i]

affiche(Tlab2)
```



4. On écrit une version adaptée de la fonction `dirpossible`.

```
DIR = [(-1, 0), (1, 0), (0, -1), (0, 1)]


def dirpossiblelab(T, c):
    i, j = c
    N, _ = T.shape
    L = []

    def caseimpossible(c):
        i, j = c

        if 0 <= i < N and 0 <= j < N:
            return T[i, j] != BLANC
            # si c'est blanc c'est pas impossible
        else:
            # on est au bord, donc on n'y va pas
            return True

    def metle(cc):
        if not caseimpossible(cc):
            L.append(cc)

    for a, b in DIR:
        metle((i + a, j + b))

    return L
```

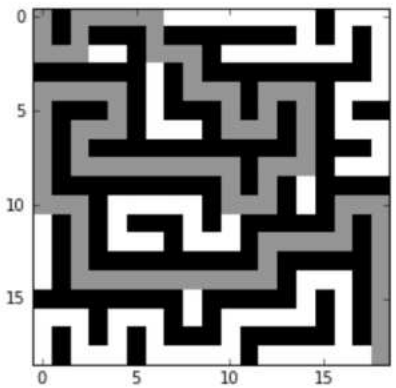
Puis la fonction récursive `trouverchemin` qui opère par *backtracking*.

```
def trouverchemin(debut, arrivee, T):
    if debut == arrivee:
        return True

    L = dirpossiblelab(T, debut)
    for i, j in L:
        if T[i, j] == BLANC:
            T[i, j] = GRIS  # visité
            res = trouverchemin((i, j), arrivee, T)
            if res:
                return True
            else:
                T[i, j] = BLANC  # plus visité
    return False
```

Un exemple de test.

```
T = Tlab2.copy()
N, _ = T.shape

print(trouverchemin((0, 0), (N - 1, N - 1), T))

affiche(T)
```

```
True
```

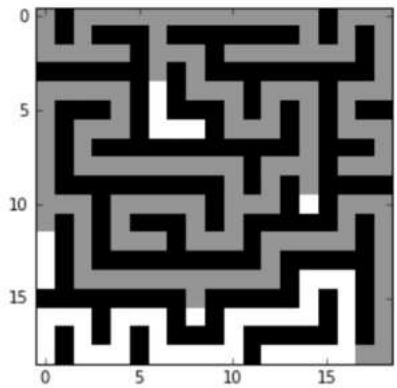


5. On présente maintenant une version itérative en stockant dans une pile les cases à visiter.

```
def trouvercheminiter(debut, arrivee, T):
    N, _ = T.shape
    #Tvisite = np.zeros(T.shape, dtype=bool)
    p = Pile()
    p.empile(debut)
    c = debut

    while not p.estvide() and c != arrivee:

        c = p.depile()
        L = dirpossiblelab(T, c)

        for i, j in L:
            if T[i, j] == BLANC:
                T[i, j] = GRIS  # visité
                p.empile((i, j))

    return c == arrivee
```

```
T = Tlab2.copy()
N, _ = T.shape

print(trouvercheminiter((0, 0), (N - 1, N - 1), T))

affiche(T)
```

```
True
```



La figure présente beaucoup de cases grises inutiles : on a grisé toutes les cases visitées même celles inutiles pour le chemin recherché...

6. Écrivons une fonction `taille(p)` qui donne le nombre d'éléments de la pile `p` (en utilisant seulement des méthode de pile).

```
def deversepile(p1, p2):
    while not(p1.estvide()):
        p2.empile(p1.depile())


def taille(p):
    """
    Retourne le nombre d'éléments dans p
    """
    c = 0
    p_aux = Pile()
    while not p.estvide():
        c = c + 1
        p_aux.empile(p.depile())
    deversepile(p_aux, p)
    return c
    # return len(c.pile)  # si on veut tricher
```

Écrivons maintenant une version améliorée de la version itérative de `trouverchemin`.
On va utiliser deux piles :

- la pile `pchemin` qui suit le chemin courant,
- la pile `p` des cases à visiter comme précédemment,

mais on crée également un tableau `Tvis` permettant d'associer à chaque case du labyrinthe le couple :

(visité ou pas, nombre de chemins encore à visiter à partir de la case).

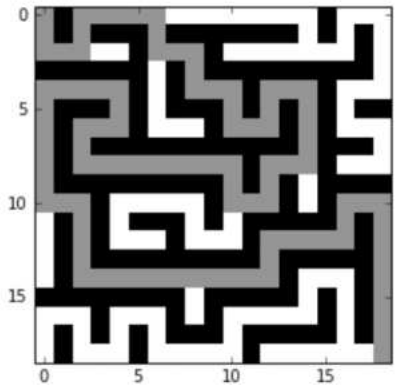
Une case découverte la première fois est initialisé à `(True , 1)` puis quand elle sera traitée (sommet de la pile `p`), on mettra à jour le nombre de possibilités de poursuite à partir de cette case.
Si on tombe dans un cul-de-sac, on remonte la pile `pchemin` jusqu'à la première bifurcation.

```
def dirpossibleiter(T, Tvis, c):
    ''' donne la liste des directions possibles à partir du
    labyrinthe T et des cases visités Tvis
    '''
    i, j = c
    N, _ = T.shape
    L = []

    def caseimpossible(c):
        i, j = c

        if 0 <= i < N and 0 <= j < N:
            return not (T[i, j] == BLANC) or Tvis[i][j][0]
        else:
            # on est au bord, donc on n'y va pas
            return True

    def metle(cc):
        if not caseimpossible(cc):
            L.append(cc)

    for a, b in DIR:
        metle((i + a, j + b))

    return L


def trouvercheminiter2(debut, arrivee, T):
    N, _ = T.shape
    Tvis = [[(False, 1) for j in range(N)] for i in range(N)]
    # tableau (visite, nb de possibilités de poursuite)
    p = Pile()   # pile de cases à traiter
    pchemin = Pile()  # chemin gagnant courant... s'il existe
    p.empile(debut)
    c = debut
    Tvis[c[0]][c[1]] = (True, 1)

    while not p.estvide() and c != arrivee:

        c = p.depile()
        pchemin.empile(c)  # on met la case dans le chemin courant
        t = taille(p)

        L = dirpossibleiter(T, Tvis, c)

        for i, j in L:
            Tvis[i][j] = (True, 1)  # visité
            p.empile((i, j))

        t2 = taille(p)
        if t2 == t and not pchemin.estvide():
            # on est dans une impasse!!
            # on va revenir à la dernière bifurcation...

            i, j = pchemin.sommet()

            while Tvis[i][j] == (True, 1) and not pchemin.estvide() and (i, j) != arrivee:
```

```
                pchemin.depile()   # on dépile jusqu'à une intersection (_, 1)
                if not pchemin.estvide():
                    i, j = pchemin.sommet()
            if not pchemin.estvide():
                _, tt = Tvis[i][j]
                Tvis[i][j] = (True, tt - 1)
                # on décrémente les possibilités non visitées de la case i, j

        elif t2 > t + 1:
            Tvis[c[0]][c[1]] = (True, t2 - t)
            # on met à jour le nombre de possibilités de la case c

    return pchemin
```

Faisons le test.

```
T = Tlab2.copy()
N, _ = T.shape

pchemin = trouvercheminiter2((0, 0), (N - 1, N - 1), T)

while not pchemin.estvide():
    i, j = pchemin.depile()
    T[i, j] = GRIS

affiche(T)
```



C'est mieux ainsi.

CHAPITRE

8 Tris

L'essentiel du cours

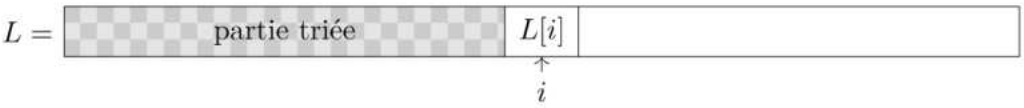
Dans de très nombreuses situations, la résolution d'un problème nécessite de trier une famille de données. Nous considérons une liste $L = [c_0, c_1, \ldots, c_{n-1}]$ où les c_i sont des éléments d'un ensemble totalement ordonné, et notre but est, ou bien de renvoyer une nouvelle liste contenant les valeurs c_i rangées dans l'ordre croissant, ou bien de modifier L en triant les c_i par ordre croissant. Dans le second cas, nous utiliserons systématiquement la fonction qui échange deux valeurs du tableau L :

```
def echange(L, i, j):
    L[i], L[j] = L[j], L[i]
```

Nous présentons ici le tri par insertion, le tri fusion et le tri rapide ; le tri à bulles est étudié à l'intention des élèves de BCPST dans l'exercice **8.2** p. 363.

■ 0 Tri par insertion

Un des algorithmes de tri les plus simples à implémenter est le **tri par insertion**, qui fonctionne à la manière d'un joueur qui classe ses cartes en les insérant les unes après les autres dans son jeu. Il consiste à faire varier un indice i de 1 à $n-1$ en assurant l'invariant :



Dans le corps de la boucle, nous ferons descendre la valeur $L[i]$ pour la placer au bon endroit à l'aide d'échanges successifs, comme dans l'exemple ci-dessous où $i = 3$:

$$\begin{aligned} L = [2,\ 5,\ 7,\ 4,\ 10,\ 1] \quad &: \quad L[2] > L[3] \text{ et on échange ces deux valeurs ;} \\ L = [2,\ 5,\ 4,\ 7,\ 10,\ 1] \quad &: \quad L[1] > L[2] \text{ et on échange ces deux valeurs ;} \\ L = [2,\ 4,\ 5,\ 7,\ 10,\ 1] \quad &: \quad L[0] < L[1] \text{ et la liste } L[0:4] \text{ est triée.} \end{aligned}$$

La descente se fait grâce à une boucle `while` : on initialise une variable j à la valeur i et tant que $j > 0$ et que $L[j] < L[j-1]$, on échange les contenus des cases j et $j-1$ et on décrémente j.
On obtient ainsi les fonctions `tri_insertion`, ci-dessous. Celle de gauche s'applique à une liste (ou à un tableau) dont les éléments peuvent être comparés par la fonction $<$; celle de droite prend en argument, en plus de la liste, une fonction `inferieur` qui, appliquée à deux valeurs a et b, renvoie le booléen `True` si a est strictement inférieur à b pour l'ordre considéré, et le booléen `False` sinon :

```
def descendre(L, i):  # L[0: i] est croissant
    j = i
    while j > 0 and L[j] < L[j - 1]:
        echange(L, j - 1, j)
        j -= 1

def tri_insertion(L):
    for i in range(1, len(L)):
        descendre(L, i)
    return L
```

```
def descendre_bis(L, i, inferieur):
    j = i
    while j > 0 and inferieur(L[j], L[j - 1]):
        echange(L, j - 1, j)
        j -= 1

def tri_insertion_bis(L, inferieur):
    for i in range(1, len(T)):
        descendre_bis(L, i, inferieur)
    return L
```

Ces fonctions renvoient le tableau trié, mais on peut supprimer la dernière ligne : elles trieront alors le tableau passé en argument sans rien renvoyer. Les exemples ci-dessous trient respectivement une liste de mots et une liste de points par abscisses croissantes :

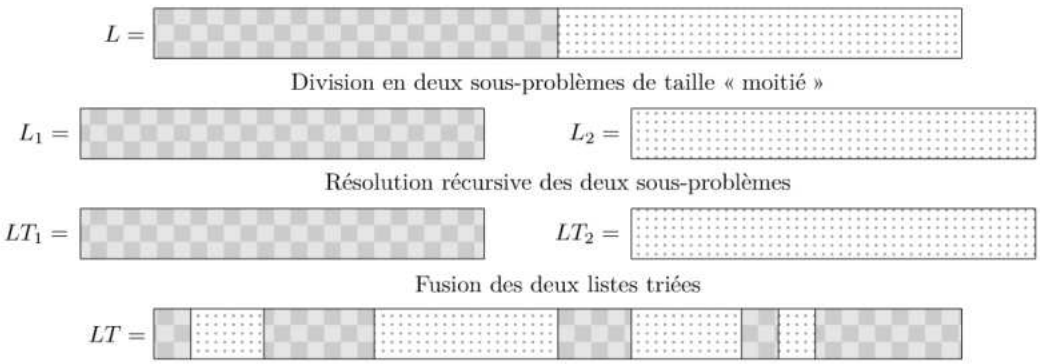
```
>>> L = ['Waller', 'Tatum', 'Garner', 'Peterson', 'Monk', 'Powell', 'Lewis', 'Evans', 'Jamal', 'Jarrett']
>>> tri_insertion(L)
['Evans', 'Garner', 'Jamal', 'Jarrett', 'Lewis', 'Monk', 'Peterson', 'Powell', 'Tatum', 'Waller']
>>> def inferieur(A, B): return A[0] < B[0]
>>> L_bis = [(3.2, 2.3), (1.7, -1.3), (4., 4.2), (3.2, 3.5), (-3.2, 4.2), (1.2, 2.3)]
>>> tri_insertion_bis(L_bis, inferieur)
[(1.7, -1.3), (3.2, 2.3), (3.2, 3.5), (4.0, 4.2), (-3.2, 4.2), (1.2, 2.3)]
```

Dans le pire des cas, le tableau est strictement décroissant : dans chaque boucle `while`, j va varier de i à 0 et le temps de calcul sera de l'ordre de $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$, soit un temps de calcul total de l'ordre n^2 ; dans le meilleur des cas, le tableau est déjà croissant et le temps de calcul est de l'ordre de n, puisqu'on sort de la boucle `while` quand $j = i$. Cet algorithme n'est cependant que très rarement efficace : on peut montrer que si l'on choisit uniformément et indépendamment n réels $a_0, a_1, \ldots, a_{n-1}$ dans l'intervalle $[0, 1]$, le temps que mettra la fonction `tri_insertion` pour trier la liste $[a_0, a_1, \ldots, a_{n-1}]$ est une variable aléatoire dont la moyenne est de l'ordre de n^2. Le temps moyen est de l'ordre du temps dans le pire des cas, ce qui traduit que l'on est presque toujours dans le pire des cas.

Le tri par insertion a cependant l'avantage de trier le tableau **en place**, c'est-à-dire en n'utilisant qu'une quantité d'espace mémoire constante (en dehors de l'espace utilisé pour stocker le tableau).

■ 1 Tri fusion (*merge sort*)

Le **tri fusion** propose de trier une liste L de longueur n de façon récursive, selon l'analyse suivante : si $n = 0$ ou $n = 1$, la liste est triée ; sinon, on la découpe en deux sous-listes L_1 et L_2 de longueurs environ égales à $n/2$. On trie ensuite (récursivement) les listes L_1 et L_2, ce qui donne deux listes triées LT_1 et LT_2, que l'on fusionne en une liste triée LT contenant les mêmes valeurs que L.



Nous avons choisi de traiter la fusion de façon itérative, en remplissant une liste LT initialement vide avec les éléments des deux listes LT_1 et LT_2 ;

```
def fusionne(LT1, LT2):
    LT = []
    i, j = 0, 0
    while(i < len(LT1) and j < len(LT2)):   # aucune liste n'a encore été totalement recopiée
        if LT1[i] < LT2[j]:
            LT.append(LT1[i])  # le plus petit élément est LT1[i]
            i += 1
        else:
            LT.append(LT2[j])  # le plus petit élément est LT2[j]
            j += 1
    LT.extend(LT1[i:])  # on déverse dans LT la fin de LT1 ... sans effet si i = len(LT1)
    LT.extend(LT2[j:])  # on déverse dans LT la fin de LT1 ... sans effet si j = len(LT2)
    return LT
```

Il reste à écrire le code de la fonction `tri_fusion` (si $n \geqslant 2$, on pose $L_1 = L[0 : p]$ et $L_2 = L[p :]$ où p est la partie entière de $n/2$) :

```
def tri_fusion(L):
    n = len(L)
    if n > 1:
        p = n // 2
        return(fusionne(tri_fusion(L[0:p]), tri_fusion(L[p:])))
    else:
        return L
```

Le calcul de la complexité temporelle du tri fusion est caractéristique des algorithmes de type **diviser pour régner** ; si nous notons $T(n)$ le temps que met notre fonction, dans le pire des cas, pour trier une liste de longueur n, nous avons :

$$\forall n \geqslant 2,\ T(n) = f(n) + T(p) + T(n-p)$$

où $f(n)$ est le temps nécessaire à la création des listes L_1 et L_2 et à la fusion des deux listes triées.

La création de L_1 et de L_2 prend un temps de l'ordre de n, de même que la fusion des listes LT_1 et LT_2 (on effectue un simple parcours des deux listes). Il existe ainsi une constante M telle que $f(n) \leqslant M\,n$. En se limitant aux n qui sont des puissances de 2, on obtient par récurrence :

$$\forall k \geqslant 1,\ T(2^k) = 2^k T(1) + 2^k \sum_{i=1}^{k} \frac{f(2^i)}{2^i} \leqslant 2^k\,(T(1) + k\,M)$$

On peut ensuite travailler sur un entier n quelconque. En effet, en supposant que f est croissante, on montre facilement que T l'est également ; en écrivant $2^{k-1} < n \leqslant 2^k$ où k est la partie entière supérieure de $\log_2(n)$, nous obtenons :

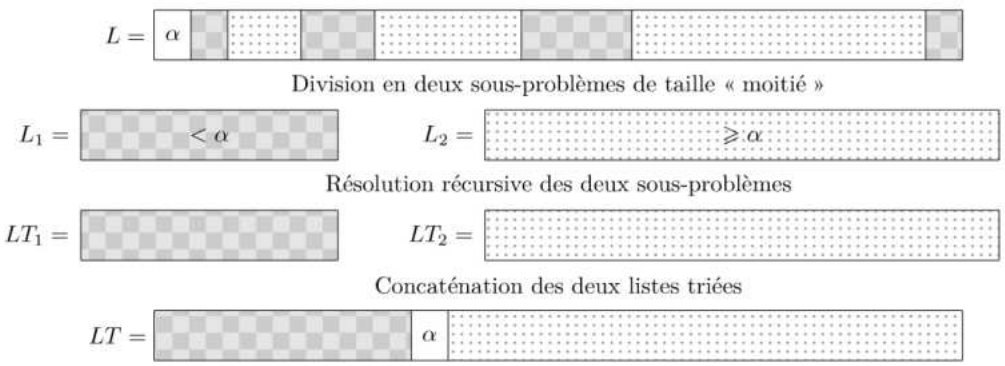
$$\forall n \geqslant 1,\ T(n) \leqslant T(2^k) \leqslant 2^k\,(T(1) + k\,M) = O\,(n\,\ln(n))$$

Ainsi, le temps de calcul est un $\mathcal{O}(n\,\ln(n))$ (on dit aussi qu'il est **quasi-linéaire**) : cet ordre de grandeur est optimal pour le problème posé.

En revanche, l'algorithme ne trie pas la liste en place, puisque la création des listes L_1 et L_2 demande déjà un espace mémoire de l'ordre de n (nous ne rentrerons pas dans l'analyse précise de la complexité en mémoire).

■ 2 Tri rapide (*quick sort*)

Le tri rapide [**Hoa62**] reprend le paradigme « diviser pour régner » du tri fusion : on choisit une des valeurs α de la liste L (par exemple $\alpha = L[0]$) qu'on appelle *pivot* et on répartit les autres éléments de L dans deux listes L_1 et L_2, en plaçant dans L_1 les éléments $< \alpha$ et dans L_2 les éléments $\geqslant \alpha$. On trie ensuite récursivement les listes L_1 et L_2, et il reste à concaténer les deux listes triées en insérant α entre les deux.



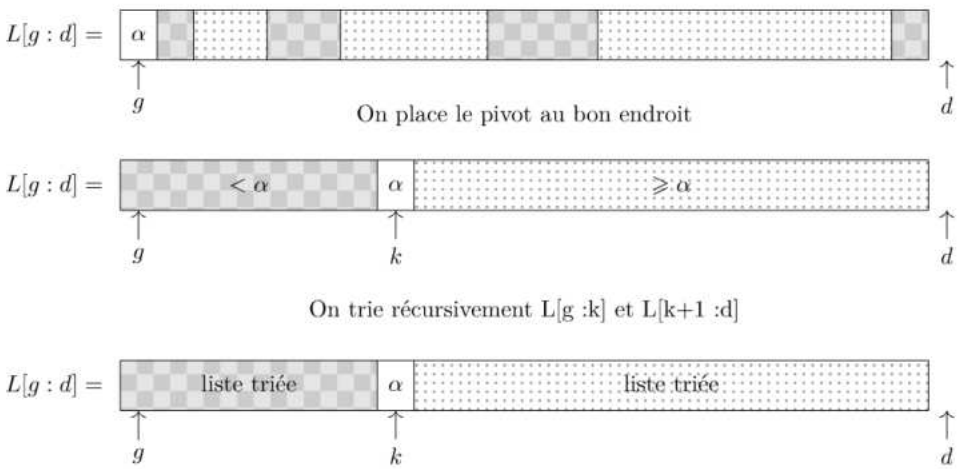
Une première mise en place élémentaire peut se faire en créant de nouvelles listes L_1 et L_2 :

```
def tri_rapide(L):
    if len(L) <= 1:
        return L[:] # On renvoie une copie de L
    pivot = L[0]
    L1, L2 = [], []
    for i in range(1, len(L)):
        if L[i] < pivot:
            L1.append(L[i])
        else:
            L2.append(L[i])
    return tri_rapide(L1) + [pivot] + tri_rapide(L2)
```

Dans le pire des cas, une des listes L_1 ou L_2 est systématiquement vide (cela arrive par exemple quand L est strictement monotone) ; le temps de calcul T_n dans le pire des cas vérifie donc une relation de récurrence de la forme : $T(n) = f(n) + T(n-1)$ où f est de l'ordre de n ($f(n)$ est le temps mis à créer L_1 et L_2 et à calculer la concaténation finale). On en déduit que $T(n)$ est de l'ordre de n^2.

Dans le meilleur des cas, on peut admettre que le pivot est systématiquement une médiane de la liste et nous nous retrouvons dans la même situation que pour le tri fusion, avec un temps de calcul de l'ordre de $n\ln(n)$.
Si l'on souhaite être plus rigoureux, on peut remarquer que l'on a $T(n) = \inf(T(p)+T(n-1-p)+f(n), 0 \leqslant p \leqslant n-1)$. Comme cette borne inférieure est minorée par tout p, on peut notamment choisir $p = \dfrac{n-1}{2}$. En travaillant avec $n = 2^k - 1$ on retrouve alors, par un raisonnement et des calculs analogues à ceux effectués dans le tri fusion, le résultat annoncé.

Il est remarquable que le temps de calcul moyen est également un $\mathcal{O}(n\ln(n))$: on en déduit que presque toutes les listes sont triées en un temps quasi-linéaire. Cependant, il arrive souvent que celles que nous avons à trier présentent de longues parties croissantes ou décroissantes. Il y a deux façons naturelles de remédier à ce problème : on peut choisir un pivot au hasard dans la liste à trier, ce qui revient paradoxalement à mélanger la liste avant de la trier ; on peut également choisir un bon pivot, voire même calculer un pivot qui coupe exactement la liste en deux parties de tailles égales (à une unité près). Sans être explicitement au programme, le lecteur est invité à faire le TP **8.0** p. 368 pour approfondir la technique de recherche de pivot et d'analyse de la complexité.
Cette première approche a le gros défaut de ne pas travailler en place. Au lieu de créer deux nouvelles listes L_1 et L_2, il est naturel de modifier L en place. Le travail récursif va alors se faire sur des portions de la liste L, au moyen de la fonction récursive `tri_rapide_rec` : quand on l'applique à deux entiers g et d tels que $0 \leqslant g \leqslant d \leqslant n$, la sous-liste $L[g:d]$ est triée sur place. On utilise pour cela la fonction auxiliaire `place_pivot`, qui met le pivot en place et renvoie la position k qu'il occupe à la fin du calcul, comme détaillé dans le schéma :



Le lecteur trouvera dans les exercices **8.8** p. 365 et **8.9** p. 366 une mise en place de la fonction `place_pivot`, qui permettra de compléter le code de la fonction :

```
def tri_rapide(L):
    def tri_rapide_rec(g, d): # trie en place la sous-liste L[g:d]
        if g < d:
            k = place_pivot(L, g, d) # k est l'indice où se retrouve placé le pivot
            tri_rapide_rec(g, k) # on trie récursivement la sous-liste située à gauche du pivot
            tri_rapide_rec(k+1, d) # on trie récursivement la sous-liste située à droite du pivot
    tri_rapide_rec(0, len(L))  # on trie l'ensemble de la liste
```

QCM sur le cours

Parmi les affirmations suivantes lesquelles sont vraies ?

(a) ☐ Le tri par insertion est toujours le tri (parmi ceux du cours) le moins efficace.
☐ ($\mathscr{P}_n$) : « Après n itérations de la boucle for `[L[0], ..., L[n]]` est triée » est un invariant de boucle du tri par insertion.
☐ ($\mathscr{P}_n$) : « Après n itérations de la boucle for `[L[0], ..., L[n]]` est composée des n plus petits éléments de `L` dans l'ordre croissant » est un invariant de boucle du tri par insertion.
☐ Le tri rapide est strictement plus efficace dans le pire des cas que le tri par insertion dans le pire des cas.
☐ Le tri fusion est strictement plus efficace dans le pire des cas que le tri par insertion dans le pire des cas.

(b) On considère la fonction suivante :

```
def mystere(L):
    if len(L) <= 1:
        return L
    pivot = L[0]
    Lg, Ld = [], []
    for i in range(1, len(L)):
        if L[i] < pivot:
            Lg.append(L[i])
        else:
            Ld.append(L[i])
    return mystere(Lg) + [pivot] + mystere(Ld)
```

☐ Cette fonction réalise le tri rapide.
☐ Le tri qu'effectue cette fonction est en place.
☐ L'algorithme écrit est en $\mathscr{O}(n^2)$ dans le pire des cas.
☐ L'algorithme écrit est en $\mathscr{O}(n)$ dans le meilleur des cas.

(c) On considère la fonction suivante :

```
def mystere2(L):
    for i in range(len(L) - 1):
        if L[i] < L[i + 1]:
            L[i], L[i + 1] = L[i + 1], L[i]
```

☐ Cette fonction réalise le tri par insertion.
☐ Cette fonction ne réalise pas de tri.
☐ L'action sur les listes étant globales, le `return` est inutile.
☐ L'algorithme écrit est en $\mathscr{O}(n^2)$ dans tous les cas.

(d) On considère la fonction suivante :

```
def mystere3(L):
    for i in range(len(L) - 1, 0, -1):
        for j in range(i):
            if T[j + 1] < T[j]:
                T[j + 1], T[j] = T[j], T[j + 1]
```

☐ Cette fonction réalise le tri de `L`.
☐ Cette fonction ne réalise pas de tri.
☐ ($\mathscr{P}_n$) : « Après n itérations de la boucle for, `[L[len(L)-1-n], ..., L[len(L)-1]]` est composée des n plus grands éléments de `L` dans l'ordre croissant » est un invariant de boucle.
☐ L'algorithme écrit est en $\mathscr{O}(n^2)$ dans tous les cas.

Exercices

Exercice 8.0 Données satellites

Plusieurs joueurs ont joué au pendu[1], nous disposons d'une liste de scores contenant les noms et scores (le taux de réussite) de chaque joueur. La liste est de cette forme :
`[['Marc', 0.87], ['Maryam', 0.99], ['Jean-Loup', 0.91], ['Hubert', 0.84]]`.
Écrire une fonction `tri_score(L)` qui trie une liste de scores par score décroissant, en utilisant le tri fusion.

Exercice 8.1 Tri par sélection

On considère dans cet exercice des listes L dont les éléments sont comparables par le biais de $<$.

0. Écrire une fonction `indice_max(L,i)` qui, quand $0 \leqslant i \leqslant$ `len`(L), calcule la position du maximum de la sous-liste $L[: i+1]$.
En déduire le code d'une fonction `tri_selection(L)` qui trie une liste L de longueur n en plaçant le maximum de L en position $n-1$, puis le maximum de $L[: n-1]$ en position $n-2$, et ainsi de suite.

1. Étudier la complexité en temps et en mémoire de cet algorithme de tri.

Exercice 8.2 Tri à bulles

L'algorithme de tri à bulles reprend l'idée du tri par sélection : on place en position $n-1$ le maximum de la liste $L = L[: n]$, puis en position $n-2$ le maximum de la sous-liste $L[: n-1]$, et ainsi de suite jusqu'à placer en position 1 le maximum de la sous-liste $L[: 2]$. La différence est que l'on va cette fois-ci arrêter le calcul dès que la liste est triée.

0. Écrire le code d'une fonction `remonter(L, i)` qui remonte le maximum de la sous-liste $L[: i+1]$ jusqu'à la position i ; on procédera pour cela à des échanges successifs éventuels des contenus des cases 0 et 1, 1 et 2, ..., $i-1$ et i, comme si une bulle remontait le long de la liste. Cette fonction renverra un booléen égal à `True` si aucun échange n'a été fait (i.e. si la sous-liste $L[: i+1]$ était croissante) et à `False` sinon.

1. En déduire le code d'une fonction `tri_bulles(L)` qui trie la liste L, en arrêtant le calcul dès que la liste est triée.

2. Étudier la complexité en temps et en mémoire de cet algorithme de tri.

Exercice 8.3 Tri crêpes

On empile un tas de crêpes de diamètres différents. On ne s'autorise qu'à donner un coup de spatule à l'intérieur du tas de crêpes ce qui a pour effet de retourner tout ou partie de la pile (à partir du sommet).

0. Écrire une fonction `retourne(p, k)` qui retourne les k premiers éléments de la pile `p` (en partant du sommet), c'est-à-dire qui donne un coup de spatule sur le tas de crêpes au dessous de la k^e crêpe.

1. Écrire une fonction `taille(p)` qui retourne le nombre d'éléments de pile `p`.

1. Par exemple avec celui programmé dans le TP 1.0 p. 37.

2. Écrire une fonction `trouve_max(p, n)` qui renvoie le numéro de la crêpe de diamètre maximal parmi les n premiers éléments de la pile `p` (= tas de crêpes).
3. Concevoir un algorithme (simple) à base de coup de spatules sur le tas de crêpes pour trier le tas par diamètre croissant (la crêpe la plus petite est au sommet).

Exercice 8.4 Tri par insertion avec une pile

Mettre en place le tri par insertion à l'aide de deux piles et d'une variable auxiliaire.

Exercice 8.5 Tri fusion avec un tableau (*buffer*), tri itératif

On souhaite programmer un tri fusion récursif avec un tableau auxiliaire à n éléments (dans la version du cours, on a, par *slicing*, création de tout un tas de petites listes éparpillées).
Enfin, on cherchera même à programmer une version *itérative* du tri fusion.

0. Écrire la fonction `fusionnebuff(t, buffer, i, j, k)` qui fusionne les sous-listes `t[i:j]` et `t[j:k]` dans `t` en utilisant une liste tampon `buffer` (qui possède au moins `len(t)` éléments).
1. Écrire alors la fonction `trifusionbuff(t, buffer, i, j)` qui trie la liste `t[i:j]` par la méthode du tri fusion récursif en utilisant le tampon `buffer`.
2. Écrire une fonction `trifusioniter(t)` qui trie la liste `t` en utilisant en interne la fonction `fusionnebuff(t, buffer, i, j, k)` mais sans appels récursifs, donc en programmant de manière *itérative*.

Exercice 8.6 Tri casier ; d'après l'oral de la banque PT

On considère M un entier strictement positif et `L` une liste d'entiers compris entre 0 et M-1.

0. Écrire une fonction `comptage` qui renvoie une liste `L1` de longueur M telle que `L1[i]` soit égal au nombre d'éléments de L égaux à i.
1. Utiliser la fonction précédente pour écrire une fonction permettant de trier `L`.
2. Quelle est la complexité de cette fonction de tri ?

Exercice 8.7 Le tri par baquets

Nous souhaitons trier (dans l'ordre lexicographique) une liste L de n mots, écrits sur l'alphabet $A = \{a, b, c, \ldots, z\}$, de longueur maximale N (on suppose que N est petit devant n). Quitte à compléter les mots de L à l'aide du caractère « espace », noté _, nous pouvons supposer que les mots de L sont tous de longueur N et écrits sur l'alphabet $\{\_, a, b, c, \ldots, z\}$, où la lettre _ est placée avant la lettre a dans l'ordre lexicographique. Nous coderons les lettres $\_, a, b, c, \ldots, z$ par les entiers $0, 1, 2, 3, \ldots, 26$ et, pour tout mot u de L et pour chaque entier $i \in \{0, 1, \ldots, N-1\}$, nous noterons $c_i(u)$ le code de la i-ème lettre de u. Ainsi, $c_2(\texttt{"exercice"}) = 5$ et $c_{20}(\texttt{"exercice"}) = 0$.

0. Écrire le code d'une fonction `c` qui, appliquée à un entier naturel i et à un mot u sur l'alphabet A, renvoie $c_i(u)$.

On utilisera la fonction `ord` qui, appliquée à un caractère, renvoie son code Unicode[1]. Par exemple, celui du caractère `'a'` est 97.

1. Cf. le chapitre 2 partie 3 p. 64.

Nous allons trier les mots de L lettre par lettre, en commençant pas la lettre d'indice $N-1$. Pour cela, nous utilisons 27 listes (appelées *baquets*), initialement vides, stockées dans une liste B, initialisée par l'instruction `B = [[] for i in range(27)]`. On parcourt L de gauche à droite, et on ajoute chaque mot u étudié au baquet $B[c_{N-1}(u)]$. À la fin de ce traitement, les n mots de L ont été répartis dans les 27 baquets. Le baquet $B[5]$, par exemple, contient tous les mots de L de longueur N dont la dernière lettre est un 'e', tandis que le baquet $B[0]$ contient tous les mots de L de longueur strictement inférieure à N. On recopie alors les contenus des baquets $B[0], B[1], \dots, B[26]$ (dans cet ordre) dans la liste L, en prenant soin de préserver l'ordre des mots dans chaque baquet. On recommence cette opération pour les lettres d'indice $N-2$, puis $N-3$, et ainsi de suite.

1. Décrire l'évolution de L et de B dans le cas où `L = ["ba", "ae", "ce", "abc", "ee", "a", "dbe"]` (on n'utilisera que six baquets).

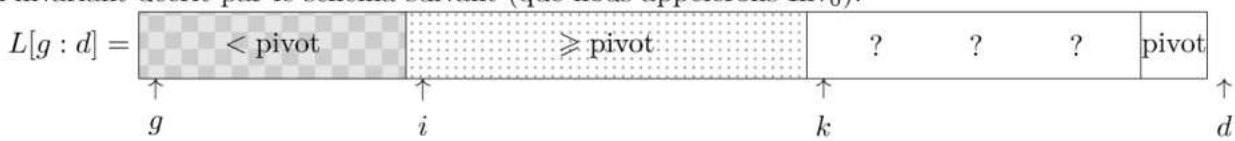
2. Écrire le code d'une fonction `mettre_en_baquets` qui, appliquée à la liste L et à un entier i, renvoie la liste des 27 baquets obtenus en rangeant les mots de L selon leur i-ème lettre.
Écrire le code d'une fonction `recopie_baquets` qui, appliquée à la liste B des baquets et à la liste L, recopie dans L les contenus des baquets $B[0], B[1], \dots, B[26]$.
Écrire le code d'une fonction `tri_baquets` qui trie une liste de mots sur l'alphabet A par la méthode des baquets. Donner quelques éléments permettant de justifier que cette fonction fait bien ce que l'on attend d'elle. Donner un ordre de grandeur de la complexité en temps et en place de cette fonction de tri, en fonction de n et de N.

Exercice 8.8 Le tri rapide version Lomuto

Dans cet exercice nous allons programmer la fonction `place_pivot` du cours selon une méthode attribuée à Nico Lomuto par [Ben16].
Dans un premier temps, nous allons réordonner la sous-liste $L[g:d]$ en mettant le pivot α à la fin et non au début. On souhaite donc modifier une liste L de façon à obtenir ceci :

0. Écrire une fonction `check` qui, prenant en argument une liste L et deux indices g et d, renvoie `True` si $L[g:d]$ est de la forme attendue (d'abord les éléments plus petits que le pivot, puis les éléments plus grands, puis le pivot) et qui renvoie `False` sinon.
Par exemple `check([1, 1, 5, 1, 1, 7, 6, 7, 6], 0, 9)` renvoie `True`.

Pour arriver à nos fins, étant donné une sous-liste $L[g:d]$ et deux entiers i et k, considérons l'invariant décrit par le schéma suivant (que nous appelerons Inv_0).



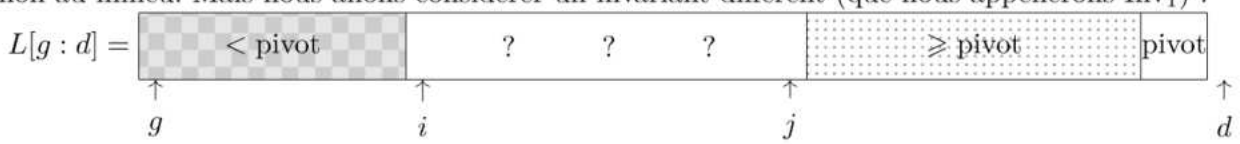
Cet invariant dit « Pour tout $p \in [\![0, i[\![$, `L[p]` $<$ pivot ET pour tout $p \in [\![i, k[\![$, `L[p]` $\geqslant$ pivot ET le dernier élément de la liste est le pivot ». Cet invariant ne dit rien sur les éléments contenus dans la zone de points d'interrogation (entre l'indice k et le pivot).
Considérons le code suivant :

```
def partition0(L, g, d):
    pivot = L[d-1]
    i = g
    for k in range(g, d-1):
        # A compléter
    return i
```

1. Compléter la fonction `partition0(L, g, d)` de sorte à maintenir l'invariant Inv_0.
2. En déduire une fonction `place_pivot` permettant d'implémenter le tri rapide en place comme vu dans le cours.
3. Quelle est la complexité de la fonction `partition0` ? De la fonction `place_pivot` ?

Exercice 8.9 Le tri rapide version Hoare

Dans cet exercice nous allons programmer une version de la fonction `place_pivot` différente de la version de l'exercice 8.8 et inspirée de la méthode originale de C.A.R. Hoare [**Hoa62**].
À l'instar de l'exercice 8.8, nous allons d'abord réordonner la liste en mettant le pivot à la fin et non au milieu. Mais nous allons considérer un invariant différent (que nous appellerons Inv_1) :



Cet invariant dit « Pour tout $p \in [\![0, i[\![$, `L[p]` $<$ pivot ET pour tout $p \in]\!]j,$ `len(L)-1`$[\![$, `L[p]` $\geqslant$ pivot ET le dernier élément de la liste est le pivot ».
Considérons l'algorithme suivant prenant en entrée[1] une sous-liste $L[g : d]$.

(0) Choisir pour pivot le dernier élément de la liste. Initialiser judicieusement les variables i et j.
(1) Augmenter i tant que ça ne brise pas l'invariant.
(2) Diminuer j tant que ça ne brise pas l'invariant.
(3) Si la zone de points d'interrogation est vide, arrêter l'algorithme et renvoyer i.
(4) Échanger $L[i]$ et $L[j]$, augmenter i de 1 et diminuer j de 1.
(5) Retourner à l'étape (1).

0. Écrire une fonction `partition1` implémentant cet algorithme en Python.
1. Pourquoi l'étape (4) de l'algorithme ne brise-t-elle jamais l'invariant Inv_1 ?
2. Quelle est la complexité de la fonction `partition1` ?
3. Utiliser `partition1` pour écrire une fonction `place_pivot` permettant d'implémenter le tri rapide comme vu dans le cours.

Exercice 8.10 Analyse en moyenne

Pour un tableau T, on note $E_i(T)$ (resp. $E_r(T)$) les nombres d'échanges effectués par l'algorithme de tri par insertion (resp. de tri rapide) appliqué au tableau T.

0. Calculer $E_i(T)$ et $E_r(T)$ pour $T = [3, 4, 1, 2, 0]$.
1. Comment peut-on modifier les fonctions `tri_insertion` et `tri_rapide` pour qu'elles calculent $E_i(T)$ et $E_r(T)$?

1. La fonction Python prendra comme arguments L, g et d.

2. Écrire une fonction `tableau_aléatoire` qui, appliquée à un entier n, génère un tableau pseudo-aléatoire $T = [x_0, x_1, \ldots, x_{n-1}]$ où les x_i simulent des variables aléatoires indépendantes de loi uniforme sur $[0, 1]$.
3. Pour différentes valeurs de n, estimer les espérances des quantités $E_i(T)$ et $E_r(T)$. Que peut-on conjecturer quant au comportement asymptotique de ces espérances quand n tend vers l'infini ?

Exercice 8.11 Quick Select et calcul de la médiane

L'objectif de cet exercice est de programmer une fonction `quick_select(L, k)` qui, étant donné une liste `L`, renvoie le k^e plus petit élément de la liste `L` (on suppose dans tout l'exercice que k est toujours strictement plus petit que la longueur de `L`). Par exemple `quick_select([5, 7, 2, 3], 0)` renvoie 2 tandis que `quick_select([5, 7, 2, 3], 2)` renvoie 5.
Une première solution est de trier la liste, et de prendre le k^e élément de la liste triée.

0. Programmer cette première solution en utilisant le tri rapide (*Quick Sort*) comme vu dans le cours.
1. Écrire une fonction `rang(L, x)` qui renvoie le rang de x dans la liste `L`. Si x est le plus petit élément, la fonction renvoie 0, si c'est le second plus petit élément, la fonction renvoie 1, etc. Si `x` apparaît plusieurs fois dans la liste, la fonction renvoie le plus petit rang.

Dans le tri rapide, il y a deux appels récursifs.
Ici, pour notre fonction `quick_select(L, k)`, si le pivot est le q^e plus petit élément de `L`, alors :

- Si $q > k$, il n'est pas besoin de trier les éléments plus grands que le pivot ;
- Si $q < k$, il n'est pas besoin de trier les éléments plus petits que le pivot.

2. Modifier l'algorithme du tri rapide pour programmer `quick_select` avec un seul appel récursif.
3. Quelle est la complexité de cette fonction ?

La complexité « en moyenne » est nettement meilleure, elle est en $\mathcal{O}(n)$. Le problème de cette méthode est la complexité en mémoire. En effet, cette méthode crée de nombreuses nouvelles listes. À présent, nous allons programmer une version « en place » de `quick_select`, c'est-à-dire une version qui ne crée aucune nouvelle liste. En contrepartie, cette fonction va modifier `L` (`L` va être « un peu » triée).

4. En utilisant la fonction `place_pivot` de l'exercice 8.8, modifier la fonction `quick_select` de façon à ce qu'elle travaille en place.

Travaux pratiques

TP 8.0 – Recherche de la médiane (quickselect) ☕ = difficile

Notre objectif est d'améliorer la complexité de la fonction `quick_select` définie dans l'exercice 8.11. L'article [BFP+73] (1972) donne une méthode pour choisir plus judicieusement le pivot et ainsi obtenir une meilleur complexité : on choisit comme pivot une médiane de médianes. Il existe plusieurs définitions d'une médiane d'une liste. Nous prendrons la définition suivante :

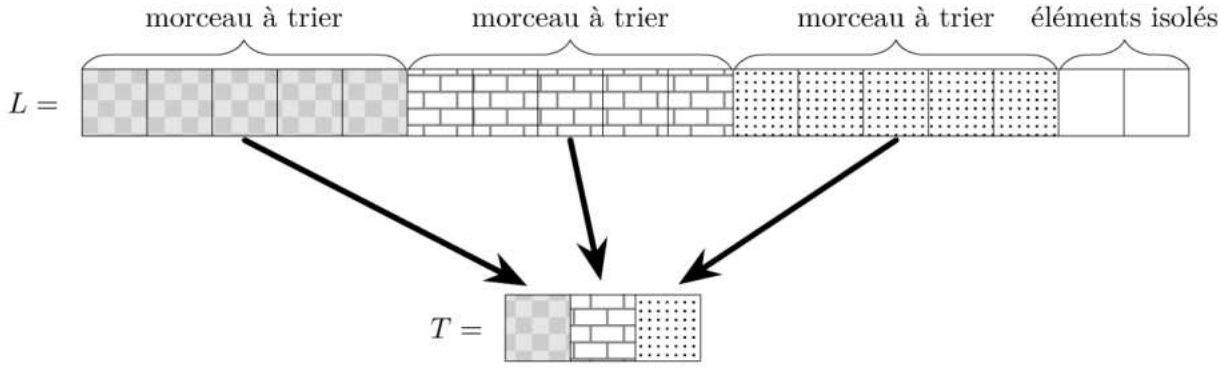
Définition

La **médiane** d'une liste `L` constituée de n éléments est l'élément d'indice $n//2$ dans la liste triée (par ordre croissant).

La méthode pour programmer `quick_select` (pas en place) est la suivante :

- On découpe la liste `L` en morceaux de 5 éléments (le dernier morceau peut éventuellement être plus petit si la longueur de la liste n'est pas divisible par 5).
- On trie chacun des morceaux avec, par exemple, le tri rapide [1].
- On crée la liste T des médianes des morceaux de 5 élements (s'il y a un morceau à moins de 5 éléments, on ne prend pas sa médiane).
- On appelle récursivement `quickselect` pour trouver la médiane m de T.
- On choisit m comme pivot
- On continue comme à la question 2 de l'exercice 8.11.

Le dessin ci-après résume la construction du tableau T.



Dans un premier temps, nous nous autorisons à faire des copies de listes.

0. Écrire une fonction `creation_T(L)` qui crée la liste `T`.
1. Programmer `quick_select` avec cette méthode.
2. ☕ Quelle est la complexité de cette fonction dans le cas où tous les éléments de la liste sont différents ?

À l'instar du `quick_select` de l'exercice 8.11, le problème de cette fonction est la complexité en mémoire. Nous allons programmer une version « en place » qui ne crée aucune nouvelle liste mais qui, en contrepartie, modifie `L`.

1. Dans l'article de 1972, c'est le tri fusion qui est utilisé.

3. Écrire une version en place de la fonction `place_pivot(L, g, d)` du tri rapide (voir l'exercice 8.8 ou l'exercice 8.9). Cette fonction devra choisir comme pivot `L[g]`.
4. Modifier la fonction `place_pivot`, pour qu'elle prenne un argument supplémentaire `p` et travaille sur `L[g:d:p]` au lieu de `L[g:d]`.
5. Écrire une fonction `tri5(L, i, p)` qui trie en place la sous-liste `L[i:i+5*p:p]` (en utilisant le tri rapide en place).
6. Écrire une fonction `tri_par5(L, g, d, p)` qui découpe la liste `L[g:d:p]` en bloc de 5 (en ignorant les éléments en trop à la fin) et qui trie chaque bloc en place.

Nous allons définir une fonction `quick_select_aux(L, k, g, d, p)` qui renvoie l'indice dans `L` du k^e plus petit élément de la liste `L[i:j:p]`. Cette fonction modifie `L` et renvoie l'indice de l'élément recherché après modification.

7. Écrire la fonction `quick_select_aux` en utilisant la méthode de la question 1 mais en travaillant en place.
8. En déduire une fonction `quick_select(L, k)` qui travaille en place.
9. Comment faire si je ne souhaite ni modifier `L` ni utiliser trop de mémoire ?

QCM sur le cours – corrigé

Parmi les affirmations suivantes lesquelles sont vraies ?

(a) ☐ Le tri par insertion est toujours le tri (parmi ceux du cours) le moins efficace.
☑ ($\mathscr{P}_n$) : « Après n itérations de la boucle for `[L[0], ..., L[n]]` est triée » est un invariant de boucle du tri par insertion.
☐ ($\mathscr{P}_n$) : « Après n itérations de la boucle for `[L[0], ..., L[n]]` est composée des n plus petits éléments de `L` dans l'ordre croissant » est un invariant de boucle du tri par insertion.
☐ Le tri rapide est strictement plus efficace dans le pire des cas que le tri par insertion dans le pire des cas.
☑ Le tri fusion est strictement plus efficace dans le pire des cas que le tri par insertion dans le pire des cas.

(b) (voir l'énoncé de la fonction `mystere`)
☑ Cette fonction réalise le tri rapide.
☐ Le tri qu'effectue cette fonction est en place.
☑ L'algorithme écrit est en $\mathscr{O}(n^2)$ dans le pire des cas.
☐ L'algorithme écrit est en $\mathscr{O}(n)$ dans le meilleur des cas.

(c) On considère la fonction suivante :

```
def mystere2(L):
    for i in range(len(L) - 1):
        if L[i] < L[i + 1]:
            L[i], L[i + 1] = L[i + 1], L[i]
```

☐ Cette fonction réalise le tri par insertion.
☑ Cette fonction ne réalise pas de tri.
☑ L'action sur les listes étant globales, le `return` est inutile.
☐ L'algorithme écrit est en $\mathscr{O}(n^2)$ dans tous les cas.

(d) On considère la fonction suivante :

```
def mystere3(L):
    for i in range(len(L) - 1, 0, -1):
        for j in range(i):
            if T[j + 1] < T[j]:
                T[j + 1], T[j] = T[j], T[j + 1]
```

☑ Cette fonction réalise le tri de `L`.
☐ Cette fonction ne réalise pas de tri.
☑ ($\mathscr{P}_n$) : « Après n itérations de la boucle for, `[L[len(L)-1-n], ..., L[len(L)-1]]` est composée des n plus grands éléments de `L` dans l'ordre croissant » est un invariant de boucle.
☑ L'algorithme écrit est en $\mathscr{O}(n^2)$ dans tous les cas.

Corrections des exercices

Corrigé exo 8.0

Il suffit de modifier la fonction `fusion` du cours, plus précisément cette ligne :

```
if LT1[i] < LT2[j]:
```

On la remplace par

```
if LT1[i][1] > LT2[j][1]:
```

Corrigé exo 8.1

0. Les codes s'écrivent sans problème :

```
def indice_max(L, i):
    pos = 0 # pos est la position du maximum
    for j in range(1, i + 1):
        if L[j] > L[pos]:
            pos = j
    return pos
```

```
def echange(L, i, j):
    L[i], L[j] = L[j], L[i]

def tri_selection(L):
    for i in range(len(L) - 1, 0, -1):
        echanger(L, i, indice_max(L, i))
    return L
```

1. Le temps de calcul de l'appel `indice_max(L,i)` est de l'ordre de i, donc le tri par sélection a un temps de calcul de l'ordre de $1+2+\cdots+(n-1)=\mathcal{O}(n^2)$ dans tous les cas. Comme le tri par insertion, il trie la liste en place : la complexité en mémoire est donc constante.

Corrigé exo 8.2

0. Un indice j varie de 0 à $i-2$: si $L[j]>L[j+1]$, on échange les contenus des cases j et $j+1$, ce qui assure l'invariant de boucle : « le maximum de la sous-liste $L[:j+1]$ est l'élément $L[j+1]$ ». Ainsi, à la sortie de la boucle, $j=i$ et le maximum de $L[:i+1]$ est en position i. Une variable booléenne b est initialisée à la valeur `True` et dès qu'un échange doit être fait, on lui affecte la valeur `False` : il suffit de renvoyer `b` à la fin de la boucle.

```
def echange(L, i, j):
    L[i], L[j] = L[j], L[i]
```

```
def remonter(L, i):
    b = True
    for j in range(i):
        if L[j] > L[j + 1]:
            b = False
            echange(L, j, j + 1)
    return b
```

1. Le tri à bulles va ensuite se faire, pour une liste L de longueur n, en appliquant la fonction `remonter` pour i variant de $n-1$ à 1, au moyen d'une boucle `for`, dont on sortira si `remonter(L, i)` renvoie le booléen `True`. Cela donne :

```
def tri_bulle(L):
    for i in range(len(L) - 1, 0, -1):
        if remonter(L, i):
            return L
    return L
```

2. Le temps de calcul de l'appel `remonter(L,i)` est de l'ordre de i, donc le tri par sélection a un temps de calcul dans le pire des cas de l'ordre de $1 + 2 + \cdots + (n-1) = \mathcal{O}(n^2)$. Dans le meilleur des cas (quand la liste est croissante), le temps de calcul est de l'ordre de n, puisqu'on n'applique qu'une fois la fonction `remonter`. Il faut toutefois remarquer que dans presque tous les cas, le temps de calcul est un $\mathcal{O}(n^2)$.
Le tri à bulles trie la liste en place, donc la complexité en mémoire est constante.

Corrigé exo 8.3

0.

```
def retourne(p, k):
    """
    Retourne les k premiers éléments de p, c'est-à-dire inverse leur ordre:
    un coup de spatule en dessous du k-ième élément sur le tas de crêpes (k <= n).

    Cette opération modifie la pile p
    """
    p_aux = Pile()
    for i in range(k):
        p_aux.empile(p.depile())
    p_rev = Pile()
    deversepile(p_aux, p_rev)
    deversepile(p_rev, p)
    return p
```

1.

```
def taille(p):
    """
    Retourne le nombre d'éléments dans p
    """
    c = 0
    p_aux = Pile()
    while not p.estvide():
        c = c + 1
        p_aux.empile(p.depile())
    deversepile(p_aux, p)
    return c
```

2.

```
def trouve_max(p, n):
    """
    Retourne la position du plus grand parmi les n > 0 premiers éléments de p.
    La position est comptée à partir de 1
    """
    p_aux = Pile()
    M = p.depile()
    p_aux.empile(M)
    pos = 1
    for i in range(1, n):
        el = p.depile()
        if el > M:
```

```
            M = el
            pos = i + 1
        p_aux.empile(el)
    deversepile(p_aux, p)
    return pos
```

3.

```
def tricrepe(p):
    """
    Tri de la pile p par opérations de retournement
    """
    long = taille(p)
    for n in range(long, 1, -1):
        k = trouve_max(p, n)
        if k != n:
            retourne(p, k)
            retourne(p, n)
    return p
```

Corrigé exo 8.4

```
def triparinsertion(p):
    '''
    Tri par insertion
    à partir d'une pile p
    on renvoie une pile triée par ordre décroissant (le sommet est minimal)
    Attention: la pile p est vide! (effet de bord)
    (faire une copie)

    on construit petit à petit la pile s qui est triée
    '''
    s = Pile()
    while not(p.estvide()):  # on va insérer element dans s
        element = p.depile()
        while not(s.estvide()) and element > s.sommet():
            # on commence par dépiler s pour trouver la place de element
            p.empile(s.depile())
        s.empile(element)  # on le met
        while not(p.estvide()) and p.sommet() < s.sommet():
            # on redeverse dans s ce que l'on avait versé dans p
            s.empile(p.depile())
    return s
```

Corrigé exo 8.5

0.

```
def fusionnebuff(t, buffer, i, j, k):
    ''' fusionne t[i:j] et t[j:k] '''
    i1, j1 = i, j
    for pos in range(i, k):
        if j1 >= k:
            buffer[pos] = t[i1]
            i1 += 1
        elif i1 >= j:
            buffer[pos] = t[j1]
            j1 += 1
        elif t[i1] <= t[j1]:
```

```
            buffer[pos] = t[i1]
            i1 += 1
        else:
            buffer[pos] = t[j1]
            j1 += 1
    for pos in range(i, k):
        t[pos] = buffer[pos]
```

1.

```
def trifusionbuff(t, buffer, i, j):
    ''' trie t[i:j] avec le buffer
    '''
    n = j - i
    if n >= 2:  # au moins deux éléments
        trifusionbuff(t, buffer, i, i + n // 2)
        trifusionbuff(t, buffer, i + n // 2, j)
        fusionnebuff(t, buffer, i, i + n // 2, j)
```

2.

```
def trifusioniter(t):
    n = len(t)
    buffer = [0] * n
    lg = 1  # longueur des sous-listes
    while lg < n:  # lg est une puissance de 2
        for i in range(0, n - lg, 2 * lg):
            # on s'attaque à t[i: i+lg], t[i+lg, i+2*lg]
            bornemax = min(i + 2 * lg, n)
            fusionnebuff(t, buffer, i, i + lg, bornemax)
        lg *= 2

    return t
```

Il apparaît plus clairement que la complexité en nombre de comparaisons est en $\mathcal{O}(n \ln n)$.

Corrigé exo 8.6

0. On commence par créer une liste L1 de M zéros, puis on parcourt L, en incrémentant L1[i] chaque fois que l'on lit i dans L :

```
def comptage(L, M):
    L1 = M * [0]
    for x in L:
        L1[x] += 1
    return L1
```

1. On parcourt désormais la liste C renvoyée par la fonction précédente, en ajoutant C[j] fois l'élément j dans une liste initialement vide :

```
def tri\_casier(L):
    C = comptage(L)
    LTriee = []
    for j in range(len(C)):
        for i in range(C[j]):
            LTriee.append(j)
    return LTriee
```

2. La fonction comptage parcourt L et ne comporte que des opérations élémentaires : elle a une complexité temporelle en $\mathcal{O}(n)$ où n=len(L).

La fonction tri casier parcourt C, qui est de longueur M (pour accéder aux C[j]) et réalise $\sum_{j=0}^{M-1} C[j] = n$ opérations élémentaires dans l'ensemble des boucles internes.
La complexité de l'algorithme du tri casier est donc en $\mathcal{O}(n+M)$. En pratique, avec M faible, cela revient à une complexité en $\mathcal{O}(n)$. Elle est donc meilleure que les complexités des algorithmes du programme, et meilleure que toute complexité d'un tri par comparaison. Ceci est possible à cause de la nature particulière des données à trier.

Corrigé exo 8.7

0.

```
def c(i, u):
    if i < len(u):
        return ord(u[i]) - 96
    else:
        return 0   # u n'a pas de i-ème lettre
```

1. Nous obtenons successivement :

```
B = [["ba", "ae", "ce", "ee", "a"], [], [], ["abc"], [], ["dbe"]]
L = ["ba", "ae", "ce", "ee", "a", "abc", "dbe"]
B = [["a"], ["ba"], ["abc", "dbe"], [], [], ["ae", "ce", "ee"]
L = ["a", "ba", "abc", "dbe", "ae", "ce", "ee"]
B = [[], ["a", "abc", "ae"], ["ba"], ["ce"], ["dbe"], ["ee"]
L = ["a", "abc", "ae", "ba", "ce", "dbe", "ee"].
```

2.

```
def mettre_en_baquets(L, i):
    B = [[] for j in range(27)]    # on crée 27 baquets vides
    for u in L:   # chaque mot u de L est ajouté au bon baquet
        B[c(i, u)].append(u)
    return B


def recopie_baquets(B, L):
    k = 0    # k est l'indice à partir duquel on copie dans L
    for baquet in B:    # on prend chaque baquet (de  gauche à droite)
        # on recopie le baquet dans L depuis la position k
        L[k:k + len(baquet)] = baquet
        k += len(baquet)    # on modifie k


def tri_baquets(L):
    N = 0    # on calcule N, maximum des longueurs des mots de L
    for u in L:
        N = max(N, len(u))
    for i in range(N - 1, -1, -1):  # on fait varier i de N-1 à 0
        recopie_baquets(mettre_en_baquets(L, i), L)
```

On peut montrer que cette fonction est correcte grâce au variant de boucle suivant : au début de chaque boucle, si nous notons $L = [u_0, \ldots, u_{n-1}]$, alors les mots $u_0[i+1:N], u_1[i+1:N], \ldots, u_n[i+1:N]$ sont croissants pour l'ordre lexicographique (d'où l'importance de remplir les baquets en parcourant L de gauche à droite). À la fin de la boucle, la liste a été triée.

La fonction `mettre_en_baquets` prend un temps de l'ordre de n, puisque l'on fait une opération de coût constant pour chaque élément u de L. La fonction `recopie_baquets` demande

également un temps de l'ordre n, puisque chaque passage dans la boucle demande un temps de l'ordre de la longueur du baquet étudié (et la somme des longueurs des baquets est égale à n). On en déduit que le tri par baquets est de complexité temporelle en $\mathcal{O}(nN)$ (un temps de l'ordre de n pour calculer N, puis $N-1$ boucles qui demandent chacune un temps de l'ordre de n).

La fonction `mettre_en_baquets` utilise un espace mémoire de l'ordre de n (il faut créer les baquets qui vont contenir les n mots de L). Comme la fonction `recopie_baquets` recopie directement les mots dans la liste L, elle n'utilise qu'une quantité constante de mémoire. La fonction `tri_baquet` a donc une complexité en place en $\mathcal{O}(n)$.

Ces analyses sont toutefois contestables car elles négligent en partie le rôle joué par la longueur des mots (l'espace mémoire utilisé pour stocker un mot et le temps de copie du mot dans un baquet dépend de N) ; si l'on veut se rapprocher de cette situation, dans le cas où N n'est pas « petit », il faut simplement travailler sur la liste des indices des mots plutôt que sur la liste des mots elle-même.

Corrigé exo 8.8

0. Cette fonction sert à vérifier si on a bien compris l'objectif. Elle ne sert pas à programmer le tri.

```
def check(L,g,d):
    pivot = L[d-1]
    i = g
    while L[i]<pivot :
        i +=1
    for k in range(i,d-1):
        if L[k] < pivot:
            return False
    return True
```

1. Bien évidemment, on utilise la fonction `echange` du cours.

```
def partition0(L, g, d):
    pivot = L[d-1]
    i = g
    for k in range(g, d-1):
        if L[k] < pivot:
            echange(L,k,i)
            i += 1
    return i
```

2. Il reste juste à mettre le pivot au bon endroit.

```
def place_pivot(L, g, d):
    i = partition0(L, g, d)
    echange(L, i, d-1)
    return i
```

3. La boucle `for` nous assure une complexité linéaire pour `partition0`, c'est-à-dire en $\mathcal{O}(n)$ avec $n =$ `len(L[g:d])`. La fonction `place_pivot` a la même complexité que la fonction `partition0`.

Corrigé exo 8.9

0. On fera attention à ne pas déborder de la sous-liste (le pivot à la fin sert de sentinelle pour `i` ; pour `j` il faut vérifier nous-mêmes).

```
def partition1(L, g, d):
    pivot = L[d-1]
    i, j = g, d-2
    while True:  # La boucle est ici, mais la condition de sortie est plus loin.
        while L[i]<pivot:
            i +=1
        while  j >= g and L[j]>= pivot:
            j -= 1
        if i <= j: # condition de sortie de boucle.
            echange(L, i, j)
        else :
            return i
```

1. D'après l'étape (1), on est sûr que i pointe sur une valeur supérieure ou égale au pivot, d'après l'étape (2), on sait que j pointe sur une valeur inférieure ou égale au pivot, et enfin l'étape (3) assure que i et j pointent encore dans l'intervalle des points d'interrogation. Après l'échange, il y a en i une valeur strictement plus petite que le pivot, et en j une valeur supérieure ou égale, on peut donc incrémenter i et décrémenter j, l'invariant reste vrai.

2. La complexité est linéaire, c'est-à-dire en $\mathcal{O}(n)$ avec $n =$ `len(L[g:d])`. Il y a la même complexité qu'avec la méthode Lomuto décrite dans l'exercice 8.8.

3. Il suffit de déplacer le pivot.

```
def place_pivot(L, g, d):
    i = partition1(L, g, d)
    echange(L, i, d-1)
    return i
```

Corrigé exo 8.10

0. Tri par insertion : quand $i = 1$, on ne fait aucun échange (car $3 < 4$) ; pour $i = 2$, on procède à deux échanges pour faire descendre 1, obtenant ainsi le tableau $T = [1, 3, 4, 2, 0]$; il faut ensuite deux échanges pour placer correctement la valeur 2, puis quatre échanges pour placer la valeur 0, soit $E_i(T) = 8$.

Tri rapide : on place le pivot 3 au bon endroit en effectuant deux échanges ($4 \leftrightarrow 0$ et $3 \leftrightarrow 2$), ce qui donne $T = [2, 0, 1, 3, 4]$. Le tri de la partie gauche $[2, 0, 1]$ commence par placer le pivot 2 au bon endroit en effectuant un échange, pour donner $[1, 0, 2]$. Le tri de la partie gauche $[1, 0]$ demande un nouvel échange pour placer le pivot 1 au bon endroit. Les autres appels récursifs se font sur des sous-tableaux de longueur 1 et ne nécessitent pas de nouveaux échanges. Nous avons donc $E_r(T) = 4$.

1. Nous utilisons ici un compteur N, qui est une variable globale. N est remis à zéro à chaque nouveau calcul et nous modifions simplement la fonction `echange` pour qu'elle incrémente N à chacun de ses appels. Cela donne :

```
N = 0  # compteur d'échanges


def echange(T, i, j):


    global N
    T[i], T[j] = T[j], T[i]
    N += 1  # chaque échange incrémente N



def E_i(T):
    global N
    N = 0  # on met le compteur à 0


    for i in range(1, len(T)):
        j = i
        while j > 0 and T[j] < T[j - 1]:
            echange(T, j - 1, j)


            j -= 1
    return N
```

```
def tri_partie(T, i, j):
    if i < j:
        a, b = i + 1, j
        while(a != b + 1):


            if T[a] > T[i]:
                echange(T, a, b)
                b -= 1
            else:


                a += 1
        echange(T, i, b)
        tri_partie(T, i, b - 1)
        tri_partie(T, b + 1, j)



def E_r(T):
    global N


    N = 0  # on met le compteur à 0
    tri_partie(T, 0, len(T) - 1)
    return N
```

2. On peut par exemple utiliser le module `numpy.random` :

```
import numpy.random as rd

def tableau_aleatoire(n):
    return [rd.random() for i in range(n)]
```

3. On estime l'espérance d'une variable aléatoire par la moyenne empirique d'un assez grand nombre de simulations :

```
def estimation_i(n, M):
    a = 0
    for i in range(M):
        a += E_i(tableau_aleatoire(n))
    return a / M
```

```
def estimation_r(n, M):
    a = 0
    for i in range(M):
        a += E_r(tableau_aleatoire(n))
    return a / M
```

Quelques calculs faits avec $M = 1000$ laissent penser que le nombre moyen d'échanges faits pas le tri par insertion est de l'ordre de n^2. Nous obtenons effectivement :

```
>>> [estimation_i(n, 1000) / (n * n) for n in [10, 20, 30, 50, 100, 200]]
[0.22576000000000002, 0.237515, 0.24150666666666665,
    0.24514760000000002, 0.2473208, 0.248337025]
```

On peut même conjecturer que ce nombre moyen d'échange est équivalent au voisinage de l'infini à αn^2 où α est une constante (avec peut-être $\alpha = 1/4$).

Pour le tri rapide, le nombre moyen est plutôt de l'ordre de n :

```
>>> [estimation_r(n, 1000) / n for n in [10, 20, 40, 80, 160, 320, 640]]
[1.8405999999999998, 2.4133, 3.0486, 3.699425, 4.368025, 4.9878875, 5.7284078125]
```

Cette suite croit de façon arithmétique : quand on multiplie n par deux, le nombre moyen augmente d'environ 0,6. On peut donc conjecturer que le nombre moyen d'échanges faits pas le tri rapide est équivalent au voisinage de l'infini à $\beta n \ln(n)$:

```
>>> [estimation_r(n, 1000) / (n * ln(n)) for n in [40, 80, 160, 320, 640]]
[0.8238545167451093, 0.8338293681484955, 0.8569755752886695,
    0.8732182429987781, 0.8890788487550568]
```

Corrigé exo 8.11

0. On réutilise la fonction `tri_rapide` du cours.

```
def quick_select(L, k):
    return tri_rapide(L)[k]
```

1.

```
def rang(L, x):
    R = 0
    for y in L:
        if x > y:
            R += 1
    return R
```

2. On modifie le code du cours. En fonction du rang, on cherche dans l'une ou l'autre moitié de la liste.

```
def quick_select(L, k):
    pivot = L[0]
    R = rang(L, pivot)
    if k == R:
        return pivot
    elif k < R:
        L1 = [x for x in L if pivot > x]
        return quick_select(L1, k)
    else:
        L2 = [x for x in L[1:] if pivot <= x]
        return quick_select(L2, k - R - 1)
```

Il est possible d'avoir un code plus compact avec l'opérateur xor (ou exclusif, qui se note ^ en Python).

```
def quick_select(L, k):
    pivot = L[0]
    R = rang(L, pivot)
    if k == R:
        return pivot
    L1 = [x for x in L[1:] if (pivot <= x) ^ (k < R)]
    if k >= R:
        k = k - R - 1
    return quick_select(L1, k)
```

3. Au pire, la liste sur laquelle est fait l'appel récursif ne diminue que de 1 élément, on fait donc n appels récursifs (où n est la longueur de la liste). À chaque appel récursif, la fonction `rang` et la création de la nouvelle liste coûte un temps $\mathcal{O}(n)$. D'où une complexité en $\mathcal{O}(n^2)$.

4. On commence par définir une fonction auxiliaire qui travaille sur la sous-liste `L[g:d]`.

```
def quick_select_aux(L, g, d, k):
    q = place_pivot(L, g, d, 1)
    if q == k :
```

```
        return L[q]
    elif q>k:
        return quick_select_aux0(L, g, q, k)
    else :
        return quick_select_aux0(L, q+1, d, k)
```

Puis on écrit la fonction recherchée.

```
def quick_select0(L, k):
    return quickselect_aux0(L, 0, len(L), k)
```

Correction d'un TP

Corrigé TP 8.0

0. On utilise la fonction `tri_rapide` du cours.

```
def creation_T(L):
    return [tri_rapide(L[5 * i:5 * i + 5])[2] for i in range(len(L) // 5)]
```

1.

```
def quick_select(L, k):
    if len(L) < 5: # Cas d'arrêt
        return tri_rapide(L)[k]
    T = creation_T(L)
    pivot = quick_select(T, len(L) // 10)
    # A partir de là, on fait comme dans l'exercice
    R = rang(L, pivot)
    if k == R:
        return pivot
    elif k < R:
        L1 = [x for x in L if pivot > x]
        return quick_select(L1, k)
    else:
        L2 = [x for x in L if pivot <= x]
        L2.remove(pivot)
        return quick_select(L2, k - R - 1)
```

2. Notons n la taille de la liste. Le pivot trouvé est une médiane de médianes de blocs de 5 éléments. Il y a $\lfloor \frac{n}{10} - 1 \rfloor$ médianes de bloc plus petites que le pivot, chacune de ces médianes a deux éléments de plus petits qu'elle. Il y a donc au moins $P_n = 3\lfloor \frac{n}{10} - 1 \rfloor$ éléments de la liste qui sont plus petits que le pivot. Pour visualiser ce calcul, on pourra regarder la figure 1 de [**BFP+73**] (le dessin est fait pour des blocs de 7, mais l'idée est la même).
De même, il y a au moins P_n éléments plus grands que le pivot. Dans tous les cas, la complexité du dernier appel récursif est majorée par $T(n - P_n)$. De plus, la complexité du calcul de la médiane de T est majorée par $T(\lfloor \frac{n}{5} \rfloor)$. Et la complexité des autres opérations (dont la création de liste) est un $\mathcal{O}(n)$ donc majorée par $C \times n$ avec C une constante.
On en déduit la relation de récurrence suivante sur la complexité $T(n)$:

$$T(n) \leqslant T(n - P_n) + T(\lfloor \frac{n}{5} \rfloor) + C \times n$$

Comme $\frac{n-P_n}{n}$ converge vers 70%, on a, pour n assez grand, $n - P_n < \lfloor 75\%n \rfloor$. On en déduit alors que (pour n assez grand) $T(n) \leqslant T(\lfloor 75\%n \rfloor) + T(\lfloor 20\%n \rfloor) + C \times n$.
Par une récurrence évidente, comme $(75\% + 20\%) \times 20 + 1 = 20$, on montre que si C a été choisi suffisamment grand [1], alors $T(n) \leqslant 20 \times C \times n$.
La complexité est donc linéaire (i.e., en $\mathcal{O}(n)$).

3. On adapte, par exemple, la version de Lomuto. Contrairement à l'exercice 8.8, le pivot est choisi à gauche et non à droite. On procède donc à un échange au début pour le mettre à droite.

1. Si C n'est pas assez grand, on ne peut pas initialiser la récurrence.

```
def place_pivot(L, g, d):
    echange(L, g, d-1)
    pivot = L[f]
    i = g
    for k in range(g, d-1, p):
        if L[k] < pivot:
            echange(L, k, i)
            i += p
    echange(L, i, d-1)
    return i
```

4. On commence par calculer l'indice du dernier élément de la liste (dans la fonction précédente, c'est `d-1`).

```
def place_pivot(L, g, d, p):
    e = d - g - 1
    f = g + e - e % p  # Indice du dernier élément de la sous-liste L[g:d:p]
    echange(L, g, f)
    pivot = L[f]
    i = g
    for k in range(g, f, p):
        if L[k] < pivot:
            echange(L, k, i)
            i += p
    echange(L, i, f)
    return i
```

5. On adapte la fonction `tri_rapide_rec` vue dans le cours.

```
def tri_rapide_rec(L, g, d, p):
    if g < d:
        k = place_pivot(L, g, d, p)
        tri_rapide_rec(L, g, k, p)
        tri_rapide_rec(L, k + 1, d, p)


def tri5(L, i, p):
    tri_rapide_rec(L, i, i + 5 * p, p)
```

6. Une simple boucle `for`.

```
def tri_par5(L, g, d, p):
    for i in range(g, d - 5 * p + 1, 5 * p):
        tri5(L, i, p)
```

7.

```
def quick_select_aux(L, k, g, d, p):
    if (d - g) // p < 10:
        tri_rapide_rec(L, g, d, p)
        return g + k * p
    tri_par5(L, g, d, p)
    m = (d - g) // p // 10  # Rang de la médiane
    ipivot = quick_select_aux(L, m, g + 2, d, p * 5)
    pivot = L[ipivot]
    echange(L, g, ipivot)
    Ip = place_pivot(L, g, d, p)
    R = (Ip - g) // p
    if k == R:
        return pivot
    elif k < R:
        return quick_select_aux(L, k, g, g + R * p, p)
```

```
    else:
        return quick_select_aux(L, k, g + (R + 1) * p, d, p)
```

8.

```
def quick_select(L, k):
    return L[quick_select_aux(L, k, 0, len(L), 1)]
```

9. Il suffit de travailler sur une copie.

```
def quick_select(L, k):
    L2 = L.copy()
    return L2[quick_select_aux(L2, k, 0, len(L), 1)]
```

CHAPITRE

9 Graphes

L'essentiel du cours

■ 0 Définitions

Définition

On appelle **graphe** la donnée d'un ensemble **fini** V de points (ou **sommets du graphe** ; *vertices* en anglais) et d'un ensemble E de liens entre ces points.

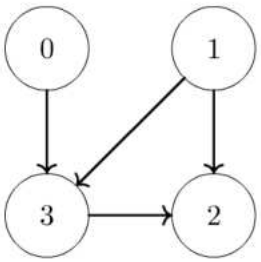
Définition

L'ensemble E de liens peut être vu comme une relation $\mathscr{R}$ sur $V \times V$. Lorsque cette relation est symétrique (c'est-à-dire lorsque l'existence d'un lien entre un sommet s_1 et un sommet s_2 équivaut à l'existence d'un lien entre le sommet s_2 et le sommet s_1) le graphe est dit non orienté. Un lien est alors appelé une **arête** (ou *edge* en anglais). Lorsque cette relation n'est pas symétrique, le graphe est dit orienté. On parle alors d'**arc** entre deux sommets.

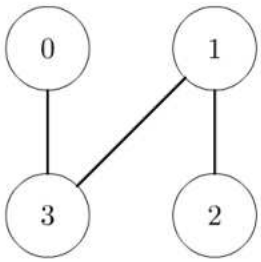
Nous noterons généralement $G = (V, E)$ un graphe.

Définition

On appelle **ordre d'un graphe** le cardinal de son ensemble de sommets.



Un graphe orienté d'ordre 4



Un graphe non orienté d'ordre 4

Définition

On appelle **graphe pondéré** (ou valué) un graphe où les arêtes sont affectées d'un poids qui est un nombre réel. Il peut être orienté ou non.

On considérera pour certains algorithmes le seul cas où le poids affecté est strictement positif. Cela représentera par exemple des situations de distances (ou de coûts de transports) dans un réseau routier.

Définition

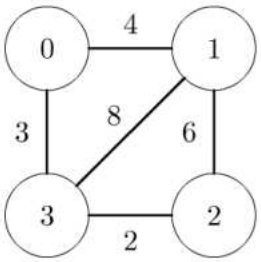
Soit $G = (V, E)$ un graphe. Un chemin $P = (S, A)$ est défini par :
$S = \{s_1, s_2, \ldots, s_k\}$, $A = \{s_0s_1,\ s_1s_2, \ldots, s_{k-1}s_k\}$ avec $S \subset V$ et $A \subset E$.
Autrement dit, un **chemin** est une suite consécutive d'arcs dans un graphe orienté. Dans le cas d'un graphe non orienté on parle de **chaîne**.

Définition

Un **cycle** est un chemin ou une chaîne pour lequel $s_0 = s_k$ (le sommet de départ et le sommet d'arrivée sont identiques).

Définition

La **longueur d'une chaîne** (resp. d'un chemin) dans un graphe non orienté (resp. orienté) est son nombre d'arêtes (resp. d'arcs).
Dans le cas d'un graphe non orienté (resp. orienté) pondéré, le **poids** d'une chaîne (resp. d'un chemin) est la somme du poids de ses arêtes (resp. arcs).



Exemple de graphe pondéré

Ce graphe est pondéré.
[0,1,2,3,1] est un exemple de chemin sur ce graphe.
Sa longueur est 20.

Définition

G est un **graphe connexe** s'il existe un chemin entre tout couple de sommets. Quand on parle de connexité pour un graphe orienté, on considère non pas ce graphe mais le graphe non-orienté correspondant.

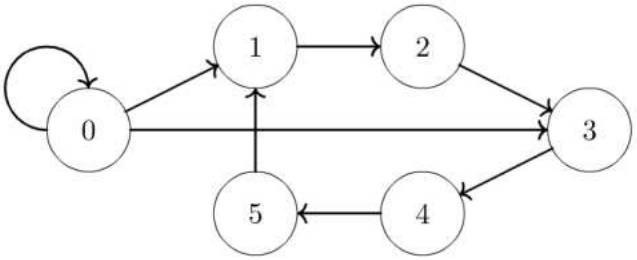
Un graphe est connexe s'il est « en un seul morceau ».

1 Graphes et matrices

La représentation informatique des graphes peut être faite de plusieurs manières. On peut notamment écrire la liste des sommets et la liste des arcs (ou des arêtes). Cette structure peut être pratique si on désire rajouter des sommets à un graphe mais rend l'exploration de celui-ci plus complexe. Alternativement, si les sommets V d'un graphe G sont numérotés, on peut supposer que V est de la forme $\{0, 1, \ldots, n-1\}$ et adopter une représentation matricielle du graphe.

Définition

Soit $G = (V, E)$ un graphe non pondéré où V est de la forme $\{0, 1, \ldots, n-1\}$. On appelle **matrice d'adjacence** A, la matrice carrée d'ordre n, dont chaque élément A_{ij} est égal à 0 s'il n'y a pas d'arête liant i à j et à 1 sinon.



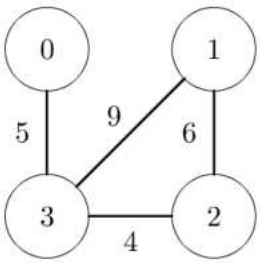
La matrice d'adjacence est :

$$A = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Définition

Lorsque $G = (V, E)$ est un graphe pondéré, on appelle **matrice des poids** la matrice carrée A d'ordre $\text{card}(V)$ dont chaque élément A_{ij} est égal au poids de l'arête liant i à j.

Si G est non orienté, sa matrice d'adjacence (ou sa matrice des poids) est symétrique.



Exemple de graphe pondéré

La matrice des poids est :

$$M = \begin{pmatrix} 0 & 0 & 0 & 5 \\ 0 & 0 & 6 & 9 \\ 0 & 6 & 0 & 4 \\ 5 & 9 & 4 & 0 \end{pmatrix}$$

2 Parcours de graphes et algorithme de Dijkstra

L'exploration des graphes est un enjeu majeur : on peut par exemple chercher quels sont les éléments atteignables depuis un point du graphe (ce problème est appelé recherche de composante connexe), quel est le plus court chemin entre deux points dans un graphe, pour résoudre un problème de minimisation ou d'orientation dans un labyrinthe.

Le principe consiste à partir d'un sommet initial puis d'explorer ses voisins, ainsi que les voisins de ses voisins non encore explorés, etc. On peut privilégier les voisins du premier voisin (et ainsi de suite récursivement) aux premiers voisins non encore explorés. Cela revient alors à « partir » le plus loin possible du premier sommet choisi. Cette méthode est appelée parcours en profondeur (*Deep First Search* ou DFS en anglais). Alternativement, on peut privilégier l'exploration de tous les voisins du premier sommet, avant d'explorer les voisins des voisins. Cela revient à « faire grossir le diamètre des sommets explorés » et est appelé parcours en largeur (*Breadth First Search* ou BFS en anglais).

Pour une étude des algorithmes DFS et BFS voir les exercices **9.4** et **9.5** p. 393.

L'**algorithme de Dijkstra** est un parcours de graphe destiné à trouver la plus courte distance à un sommet donné dans un graphe pondéré non orienté, sans arête de poids négatif.

Implémenter un algorithme de Dijkstra est à proscrire lorsque le graphe n'est pas pondéré. Un parcours en largeur est nettement mieux adapté.

L'algorithme de Dijkstra prend en entrée un graphe et un sommet initial I et se déroule comme suit :

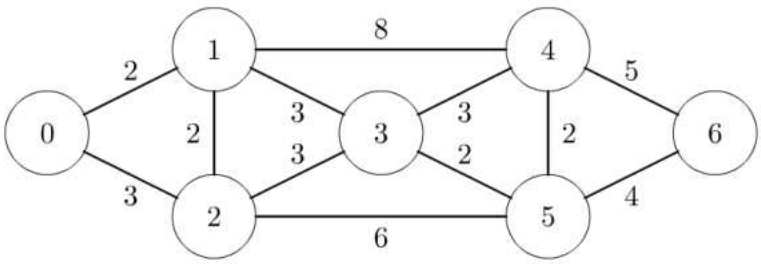
- On initialise les variables suivantes :
  $E = []$
  $V = [I]$
  $D =$ la liste composée de n fois `[np.inf, None]` sauf $D[I]$ qui vaut `[0, None]`

`None` est une valeur spéciale en Python qui désigne le fait qu'une variable n'a pas de valeur ; `np.inf` désigne la valeur $+\infty$.

E désigne la liste des sommets déjà explorés ; V désigne la liste des sommets dont on sait d'ores et déjà qu'ils sont accessibles depuis I mais qui n'ont pas encore été explorés. Enfin D a une structure plus complexe : c'est une liste de n listes de deux éléments. Chaque sous-liste représente un sommet ; la première composante est la meilleure distance (depuis I) trouvée à ce stade de l'exploration du graphe et la seconde composante est le sommet précédent celui-ci qui a permis de trouver cette distance.

- Tant que la liste V n'est pas vide :
  - On sélectionne le sommet S tel que $D[S][0] = \min\{D[i][0], i \in V\}$.
    Il s'agit du sommet non encore exploré qui a la plus courte distance à I parmi les sommets accessibles.
  - On supprime S de V
  - On ajoute S à E
  - Pour chaque voisin VP de S on regarde s'il est dans la liste E. S'il n'y est pas, on ajoute VP à V puis on compare $D[VP][0]$ à la somme de $D[S][0]$ et de la distance d de S à VP : si cette dernière est strictement inférieure, on fait $D[VP] = [D[S][0] + d, S]$.
- E est composé de tous les sommets de la composante connexe de I.
- $D[S][0]$ est la plus petite distance de I à S. Si elle vaut ∞, S n'est pas accessible depuis I.
- Si S est accessible, pour connaître le plus court chemin de I à S, on refait le parcours inversé :
  - on initialise une liste `Parcours = [S]` et une variable `prec = D[S][1]`
  - tant que `suivant` ne vaut pas `None`, on ajoute `prec` à la fin de `Parcours` et on affecte `D[prec][1]` à `prec`.
  - on retourne `Parcours` inversé.

Voici un exemple d'utilisation de l'algorithme de Dijkstra :



V	E	S	0	1	2	3	4	5	6
(Initial)	[0]	[ ]	0,N	+∞,N	+∞,N	+∞,N	+∞,N	+∞,N	+∞,N
[0]	[]	0	0,N	2,0	3,0	+∞,N	+∞,N	+∞,N	+∞,N
[1,2]	[0]	1	0,N	2,0	3,0	5,1	10,1	+∞,N	+∞,N
[2,3,4]	[0,1]	2	0,N	2,0	3,0	5,1	10,1	9,2	+∞,N
[3,4,5]	[0,1,2]	3	0,N	2,0	3,0	5,1	8,3	7,3	+∞,N
[4,5]	[0,1,2,3]	5	0,N	2,0	3,0	5,1	8,3	7,3	11, 4
[5,6]	[0,1,2,3,4]	4	0,N	2,0	3,0	5,1	8,3	7,3	11, 4
[6]	[0,1,2,3,4,5]	6	0,N	2,0	3,0	5,1	8,3	7,3	11, 4

Pour coder de manière simple (une implémentation optimisée mais plus complexe est proposée dans le TP **9.1**) cet algorithme en Python, nous créons les fonctions suivantes :

- `Voisins` qui renvoie la liste des voisins d'un sommet S,
- `indicesommetdmin` qui renvoie l'indice du sommet à distance minimale de V,
- `Dijkstra` qui est la fonction centrale du code et qui renvoie une liste des meilleures distances au sommet initial et des prédécesseurs codées sous forme de listes,
- `Chemin` qui reconstitue le meilleur chemin allant du sommet initial à un sommet S.

```
import numpy as np


def Voisins(M, S):
    L = []
    for i in range(len(M)):
        if M[S][i] != 0:
            L.append(i)
    return L


def indicesommetdmin(V, D):
    dmin = min([D[i][0] for i in V])
    for i in V:
        if D[i][0] == dmin:
            return i


def Dijkstra(M, I):
    E = []
    V = [I]
    D = [[np.inf, None] for i in range(len(M))]
    D[I][0] = 0
    while len(V) > 0:
        S = indicesommetdmin(V, D)
        V.remove(S)
        E.append(S)
        for VP in Voisins(M, S):
            if VP not in E:
                V.append(VP)
                if D[VP][0] > D[S][0] + M[S][VP]:
                    D[VP] = [D[S][0] + M[S][VP], S]
    return D


def Chemin(D, S):
    Parcours = [S]
    prec = D[S][1]
    while prec is not None:
        Parcours.append(prec)
        prec = D[prec][1]
    Parcours.reverse()
    return Parcours
```

Exercices

Graphes et matrices

Exercice 9.0 Matrice d'adjacence VS listes d'adjacence

Nous considérons des graphes orientés $G = (V, E)$ où V est un ensemble de la forme $\{0, 1, \ldots, n-1\}$. Un tel graphe peut être représenté soit par sa matrice d'adjacence, notée A, soit par un vecteur de taille n, noté L, où pour tout $i \in V$, $L[i]$ est la liste d'adjacence du sommet i, i.e. la liste des successeurs de i (sans contrainte sur l'ordre de ces successeurs). Ainsi, pour l'exemple du paragraphe 1, nous avons :

$$A = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \text{ et } L = [\,[0,\ 1,\ 3],\ [2],\ [3],\ [4],\ [5],\ [1]\,]$$

0. Comparer les espaces mémoires nécessaires pour stocker un graphe sous forme de matrice d'adjacence ou de liste de listes d'adjacence.

1. Pour chacune de ces deux structures, écrire une fonction qui teste s'il existe un arc d'un sommet i à un sommet j. Quels sont les temps de calcul de ces deux fonctions dans le pire des cas ?

2. Écrire une fonction `Matrice` (resp. une fonction `Liste`) qui convertit la liste des listes d'adjacence (resp. la matrice d'adjacence) d'un graphe en matrice d'adjacence (resp. en liste de listes d'adjacence). Quelles sont les complexités de ces deux fonctions ?

3. Pour chaque sommet i de G, les degrés entrant et sortant de i, notés $d^-(i)$ et $d^+(i)$, sont les nombres d'arcs qui respectivement arrivent en i et partent de i. Écrire une fonction qui, appliquée à la liste L, renvoie une matrice d de taille $n \times 2$ dont la ligne i contient le couple $(d^-(i), d^+(i))$.

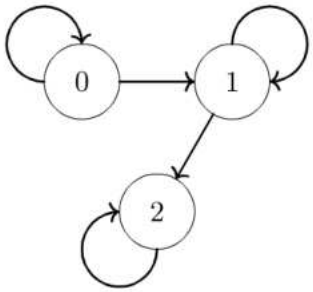
Exercice 9.1

On considère un graphe $G = (V, E)$ non pondéré et A sa matrice d'adjacence.

0. Soient i et j deux sommets de G. Montrer qu'il y a $A^n_{i,j}$ chemins de longueur n allant de i à j.

1. Écrire en Python une fonction `CheminLongueurN(A, i, j, n)` qui prend en entrée un graphe représenté par sa matrice d'adjacence A, deux sommets i et j et qui renvoie le nombre de chemins de longueur n allant de i à j. On pourra se servir du module `numpy`.

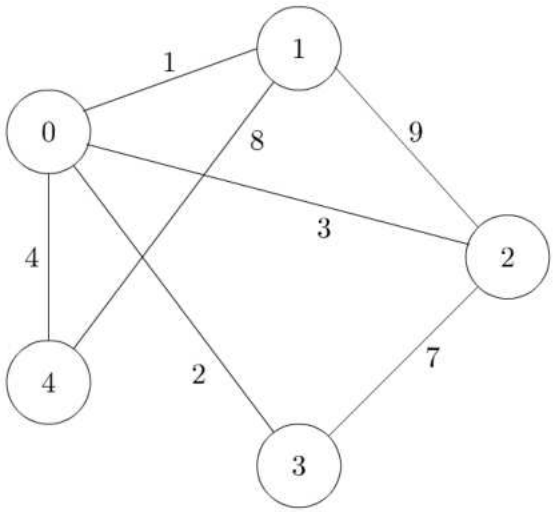
2. Combien y-a-t-il de chemins de longueur 100 partant de 0 et allant à 2 dans le graphe suivant :



3. Retrouver le résultat précédent par un calcul matriciel.
4. Retrouver le résultat précédent avec un dénombrement.

Exercice 9.2

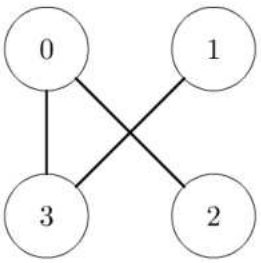
On considère le graphe suivant :



0. Écrire la matrice des poids de ce graphe, puis la créer sous Python.
1. On considère une liste L d'entiers. Écrire une fonction `test_chaine(M, L)` qui prend en argument une matrice M représentant un graphe G et la liste L, et qui teste si la liste de sommets apparaissant dans L dans cet ordre correspond à un chemin possible sur G.
2. Adapter cette fonction pour qu'elle renvoie la longueur du chemin lorsqu'il est possible ou -1 si le chemin est impossible.

Exercice 9.3

Un graphe G non orienté et non pondéré peut être représenté par un nombre n de sommets et une liste de listes d'arêtes.
Par exemple le graphe suivant :

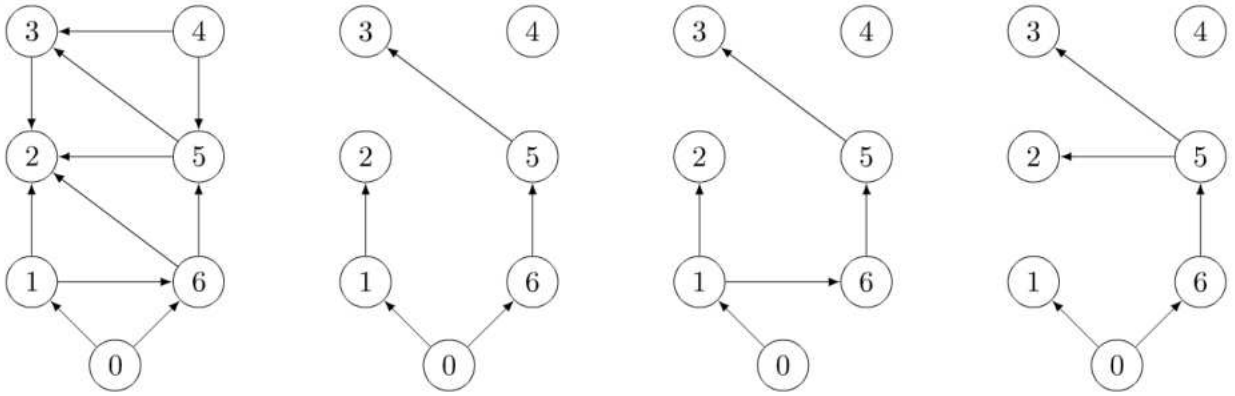


est représenté par $n = 4$ et $L = [[0, 2], [0, 3], [1, 3]]$.

0. Écrire une fonction `ListeMatrice(n, L)` qui prend en arguments l'entier n et la liste de listes L précédemment décrite et qui renvoie une matrice carrée d'ordre n correspondant à la matrice d'adjacence du graphe. On prendra bien garde à la nature non orientée du graphe G.
1. Tester cette fonction sur l'exemple proposé plus haut.

Parcours de graphes

Nous considérons des graphes orientés $G = (V, E)$ où V est un ensemble de la forme $\{0, 1, \ldots, n-1\}$. Un tel graphe sera représenté, comme à l'exercice 9.0, par la liste L de ses listes d'adjacence. Très souvent, la résolution d'un problème modélisé par un graphe nécessite de parcourir le graphe, c'est-à-dire d'explorer le graphe en suivant ses arcs. Pour un sommet i du graphe, l'**exploration du graphe** à partir de i se fait en partant de i et en suivant les arcs tant que de nouveaux sommets peuvent être atteints. Voici trois exemples d'arbres obtenus en explorant le même graphe G_0, à partir du sommet 0 :



Le graphe G_0 Parcours 1 Parcours 2 Parcours 3

Une exploration depuis un sommet i va permettre de définir un arbre de racine i dans le graphe G, en ne conservant que les arcs qui ont été suivis pendant notre exploration. Cet arbre sera caractérisé par une liste π de longueur n, appelée **liste des pères** : si une arête (j, k) a été conservée, nous dirons que k est un **fils** de j et que j est le **père** de k, noté $\pi[k]$. Le père de k est l'origine de l'arc qui a permis la découverte du sommet k. Par convention, nous poserons $\pi[i] = -1$ (la racine n'a pas de père) et $\pi[j] = -2$ si j n'est pas un successeur de i. Ainsi, le parcours 2 nous donne la liste des pères $\pi = [-1, 0, 1, 5, -2, 6, 1]$.
Si l'on souhaite parcourir tous les sommets, il suffit ensuite d'explorer le graphe depuis un sommet qui n'a pas encore été atteint, et ainsi de suite jusqu'à épuisement des sommets. Les sommets du graphe seront alors recouverts par un ensemble d'arbres disjoints, appelé **forêt couvrante**, toujours représentée par une liste π (pour chaque racine i, nous aurons $\pi[i] = -1$).

L'exploration depuis un sommet i, que nous appellerons la **source de l'exploration**, laisse une grande latitude dans le choix des arcs à suivre. Deux types de parcours sont usuellement utilisés :

- Les **parcours en largeur d'abord** (ou *Breadth First Search* en anglais) traitent les sommets « niveau par niveau », dans l'ordre de leur distance au sommet source. Ainsi, on utilise tous les arcs qui partent de i pour atteindre les sommets situés à la distance 1 de la source, puis on atteint tous les successeurs de ces sommets qui n'ont pas encore été découverts, et ainsi de suite jusqu'à avoir traité tous les sommets que l'on peut relier à la source. Le parcours 1 ci-dessus est un parcours en largeur d'abord : 0 a deux successeurs 1 et 6, donc on conserve les arcs $(0, 1)$ et $(0, 6)$, puis les sommets 1 et 6 permettent d'atteindre les sommets 2 et 5 (situés à la distance 2 de la source) ; on choisit de conserver les arcs $(1, 2)$ et $(6, 5)$, et 3 est le dernier sommet que l'on peut atteindre, qui sera relié à 5.
- Les **parcours en profondeur d'abord** (ou *Depth First Search* en anglais) consistent à descendre le plus profondément possible (on suit les arcs tant que l'on peut découvrir un nouveau nœud), puis à remonter quand on arrive à un cul-de-sac. Le parcours 2 est un

parcours en profondeur d'abord : on part de la source 0, on « descend » en 1, puis en 2 ; on « remonte » alors en 1, pour redescendre en 6, puis en 5, puis en 3 ; on remonte en 5, qui ne possède plus de nouveau successeur (2 a déjà été découvert), on remonte en 6, on remonte en 1 et on remonte en 0 où le calcul s'achève puisque 6 a déjà été découvert.

Exercice 9.4 Parcours en largeur

Pour parcourir un graphe en largeur d'abord à partir d'un sommet i, deux approches simples sont possibles :

- on initialise une liste D à la valeur $[i]$ (D contient tous les sommets qui sont à la distance $\delta = 0$ de i) ; tant que D est non vide (il contient tous les sommets qui sont situés à une même distance δ de i), on construit la liste ND des successeurs des éléments de D qui n'ont pas encore été rencontrés et on remplace D par ND (D contient alors tous les sommets situés à la distance $\delta + 1$ de i) ;
- on peut également créer une file d'attente qui ne contient que i au début du calcul. Tant que la file n'est pas vide, on retire l'élément j qui est en tête de la file, puis on ajoute à la queue de la file tous les successeurs de j qui n'ont pas encore été rencontrés.

0. Proposer une structure permettant de gérer une file d'attente. Cette structure de type « *First In, First Out* » devra être associée à quatre fonctions, permettant :
- de créer une file d'attente vide ;
- de tester si une file d'attente est vide ;
- d'ajouter un élément à la queue d'une file ;
- de supprimer l'élément i qui est en tête d'une file et de renvoyer cet élément.

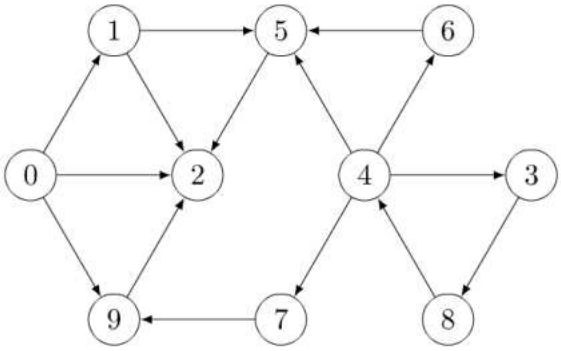
1. Pour chacune de ces deux approches, écrire le code d'une fonction qui, quand on l'applique à la liste L et à un sommet i, effectue un parcours en largeur d'abord du graphe depuis le sommet i et renvoie deux listes d et π de longueur n telles que, pour tout $j \in \{0, 1, \ldots, n-1\}$:
- $d[j]$ est la distance de i à j (avec par convention $d[j] = -1$ s'il n'existe pas de chemin de i à j) ;
- $\pi[j]$ est le père de j dans l'arbre associé à l'exploration choisie (avec par convention $\pi[i] = -1$ et $\pi[j] = -2$ si j n'est pas atteignable depuis i).

Ainsi, pour le graphe G_0 exploré par le parcours 1, nous avons $d = [0,\ 1,\ 2,\ 3,\ -1,\ 2,\ 1]$ et $\pi = [-1,\ 0,\ 1,\ 5,\ -2,\ 6,\ 0]$.

2. En déduire le code d'une fonction qui, appliquée à L et à deux sommets i et j, renvoie une liste $[i_0, i_1, \ldots, i_k]$ telle que $(i_0, i_1, \ldots, i_k)$ soit un plus court chemin de i à j dans le graphe défini par L (par convention, la fonction renverra la liste vide si j n'est pas connecté à i).

Exercice 9.5 Parcours en profondeur

On peut mettre en place un parcours en profondeur d'abord à l'aide d'une fonction récursive `explorer` qui s'applique à un sommet j du graphe : pour explorer j, on prend l'un après l'autre chaque successeur k de j qui n'a pas déjà été rencontré, on pose $\pi[k] = j$ (j sera le père de k dans l'exploration), et on applique la fonction `explorer` à k. Lors d'un parcours en largeur, on se contente en général d'explorer le graphe depuis un sommet source i fixé, en construisant un arbre recouvrant tous les sommets atteignables depuis i. Les parcours en profondeur sont plutôt utilisés pour parcourir l'ensemble du graphe, comme dans l'exemple :



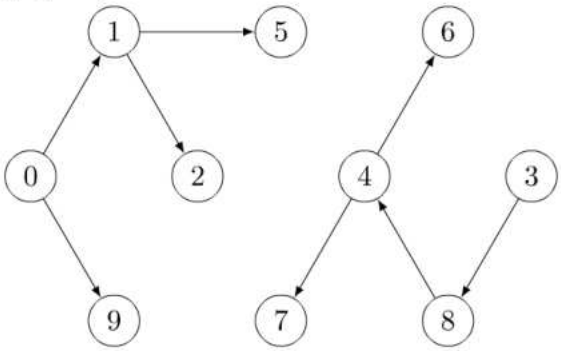
On commence par explorer le graphe en profondeur depuis le sommet 0. Cette exploration peut être représentée par une succession de déplacements dans le graphe, en suivant ($\rightarrow$) ou remontant ($\leftarrow$) les arcs :

$$0 \rightarrow 1 \rightarrow 2 \leftarrow 1 \rightarrow 5 \leftarrow 1 \leftarrow 0 \rightarrow 9 \leftarrow 0$$

Une fois cette exploration terminée, on poursuit le parcours à partir d'un sommet non encore atteint, par exemple le sommet 3. On obtient ainsi le parcours complet :

$$0 \rightarrow 1 \rightarrow 2 \leftarrow 1 \rightarrow 5 \leftarrow 1 \leftarrow 0 \rightarrow 9 \leftarrow 0 \;||\; 3 \rightarrow 8 \rightarrow 4 \rightarrow 6 \leftarrow 4 \rightarrow 7 \leftarrow 4 \leftarrow 8 \leftarrow 3$$

qui permet de recouvrir le graphe par une forêt constituée de deux arbres, de racines 0 et 3 :



Cette forêt est définie par la **liste des pères** : $\pi = [-1,\ 0,\ 1,\ -1,\ 8,\ 1,\ 4,\ 4,\ 3,\ 0]$; les racines des arbres n'ont pas de père (d'où la convention $\pi[0] = \pi[3] = -1$) et chaque autre sommet j a pour père le sommet i depuis lequel il a été découvert (le père de 1 est 0, celui de 2 est 1, etc).

0. Écrire le code d'une fonction `Parcours_Profondeur` qui, appliquée à la liste des listes d'adjacence L, applique la méthode précédente et renvoie la liste π ainsi construite.

1. Écrire le code d'une fonction `Parcours_Profondeur_Itérative` qui fait le même travail sans utiliser de fonction récursive (on pourra stocker les sommets découverts dans une pile).

Lors d'un parcours en profondeur, les instants d_i et f_i de début et de fin de traitement de chaque sommet i jouent un rôle très important. Sur l'exemple du parcours défini ci-dessus, voici représentés les vingt instants intéressants du parcours :

$$0 \rightarrow 1 \rightarrow 2 \leftarrow 1 \rightarrow 5 \leftarrow 1 \leftarrow 0 \rightarrow 9 \leftarrow 0 \quad 3 \rightarrow 8 \rightarrow 4 \rightarrow 6 \leftarrow 4 \rightarrow 7 \leftarrow 4 \leftarrow 8 \leftarrow 3$$

$$d_0 \quad d_1 \quad d_2\ f_2 \quad d_5\ f_5 \quad f_1 \quad d_9\ f_9 \quad f_0 \quad d_3 \quad d_8 \quad d_4 \quad d_6\ f_6 \quad d_7\ f_7 \quad f_4 \quad f_8 \quad f_3$$

Ces instants seront numérotés de 1 à 20 : $d_0 = 1$, $d_1 = 2$, $d_2 = 3$, $f_2 = 4$, etc. Nous dirons par exemple que le traitement du sommet 4 a débuté à l'instant 13 et s'est terminé à l'instant 18.

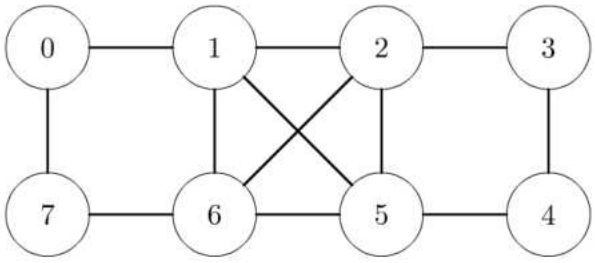
2. Modifier le code de la fonction `Parcours_Profondeur` pour qu'elle renvoie, en plus de π, les listes d et f de longueur n telles que $d[i] = d_i$ et $f[i] = f_i$ pour tout $i \in \{0, 1, \ldots, n-1\}$.

Exercice 9.6

Définition

On considère un graphe non orienté $G = (V, E)$. On considère une chaîne $P = (S, A)$.
Celle-ci est appelée **chaîne eulérienne** si l'on a $E = A$ et $\text{card}(E) = \text{card}(A)$. Cela revient à dire qu'une chaîne eulérienne est une chaîne pour laquelle toute arête est parcourue une fois et une seule.
Un cycle eulérien est une chaîne eulérienne qui est un cycle. Un graphe est dit eulérien s'il contient un cycle eulérien.

0. Le graphe suivant est-il eulérien ?



Définition

Soit $G = (V, E)$ un graphe non orienté et $s \in V$. On appelle degré du sommet s le nombre d'arêtes reliées à s.

On admet le théorème d'Euler :

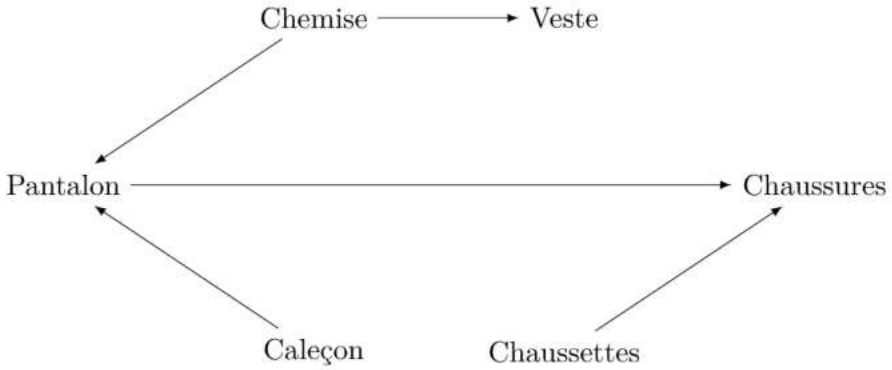
Un graphe est eulérien si, et seulement si, il est connexe et a tous ses sommets de degré pair.

1. Écrire une fonction `degre(M, i)` qui prend en argument une matrice M représentant un graphe non orienté G et un sommet i et qui renvoie le degré de ce sommet.
2. Écrire une fonction `est_connexe(M)` qui prend en argument une matrice M représentant un graphe non orienté G et qui renvoie un booléen indiquant si G est connexe.
3. Écrire une fonction `est_eulerien(M)` qui prend en argument une matrice M représentant un graphe non orienté G et qui renvoie un booléen indiquant si G est eulérien.

Exercice 9.7 Graphe de tâches

Chaque matin, avant d'aller travailler, un employé doit enfiler ses vêtements (pantalon, chemise, chaussures, caleçon, veste, chaussettes), mais l'ordre dans lequel il doit le faire est assujetti à des **règles de préséance** évidentes : il ne peut pas mettre ses chaussures avant d'avoir enfiler ses chaussettes, ni mettre sa veste avant sa chemise. Plus généralement, nous considérons un ensemble $\{T_0, T_1, \ldots, T_{n-1}\}$ de tâches à accomplir, assujetties à des règles de préséance : certaines tâches ne peuvent être exécutées que si d'autre tâches ont déjà été effectuées. Nous représentons cette

situation par la donnée d'un graphe orienté $G = (V, E)$ où $V = \{0, 1, \ldots, n-1\}$ et où $(i, j) \in E$ si et seulement si la tâche T_i doit être exécutée avant la tâche T_j. Dans notre exemple, cela donne le graphe :



Nous cherchons à ordonner les sommets du graphe en une liste $[i_0, i_1, \ldots, i_{n-1}]$ de sorte que s'il existe un chemin d'un sommet i à un sommet j (ce qui signifie que T_i doit nécessairement être effectuée avant T_j), alors j apparaît avant i dans la liste. Si ce calcul est possible, on dit que la liste trouvée définit un **ordre topologique** sur le graphe orienté G : elle donne un mode opératoire pour accomplir toutes les tâches sans jamais rencontrer d'incompatibilité (on effectue T_{i_0}, puis T_{i_1}, et ainsi de suite jusqu'à $T_{i_{n-1}}$). Il est clair qu'un graphe possédant un cycle ne peut pas être muni d'un ordre topologique ; nous supposerons donc que le graphe G est **acyclique**, c'est-à-dire qu'il ne contient pas de cycle.

0. Donner un ordre topologique pour le graphe ci-dessus.

1. On suppose qu'un graphe acyclique G est défini par une liste L de listes d'adjacence. On effectue un parcours en profondeur de ce graphe, ce qui permet en particulier de construire les listes d et f des instants de débuts et de fins de traitement (voir exercice **9.5** p. 393). On considère deux sommets i et j du graphe tels que $f[i] < f[j]$. Montrer que l'on est dans l'un des cas suivants :
- $d[i] < f(i) < d(j) < f(j)$ et il n'existe pas de chemin de i à j ;
- $d[j] < d(i) < f(i) < f(j)$ et il existe un chemin de j à i

En déduire que si $f[i] < f[j]$, il n'existe pas de chemin de i à j.

2. Expliquer comment la connaissance de la liste des instants de fins de traitement permet de construire un ordre topologique sur G.

Écrire le code d'une fonction `Ordre_Topologique` qui, quand on l'applique à la liste L représentant un graphe orienté acyclique G, renvoie un ordre topologique $[i_0, i_1, \ldots, i_{n-1}]$.

Travaux pratiques

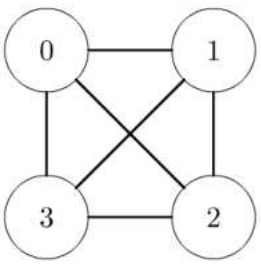
TP 9.0 – Algorithme génétique et voyageur de commerce

Ce TP est consacré à l'étude du parcours de graphes non orientés et à la résolution du problème du voyageur de commerce, tant de manière exacte par force brute dans des cas simples que de manière approchée à l'aide d'algorithmes génétiques dans des cas plus complexes.
Un graphe $\mathscr{G}$ est dit **graphe hamiltonien** s'il possède au moins un cycle passant par tous les sommets de $\mathscr{G}$ exactement une fois ; un tel cycle est appelé **cycle hamiltonien**.

Un représentant de commerce part de sa ville d'origine et doit passer visiter ses clients dans des villes différentes, une fois et une seule. Il a bien sûr intérêt à minimiser la longueur du cycle qu'il va faire et il souhaite rentrer dans sa ville d'origine. Ce problème est appelé le « problème du voyageur de commerce » (*salesman problem*) et est apparu dans les années 1930. Avec le vocabulaire introduit plus haut, il consiste à trouver, dans un graphe hamiltonien (pondéré), un cycle hamiltonien de longueur minimale.
Nous nous restreindrons aux graphes complets pour la suite de ce TP.

Définition

On appelle **graphe complet** d'ordre n et on note K_n l'unique (au nom des sommets près) graphe non orienté d'ordre n tel que toute paire de sommets (distincts) est reliée.



Exemple du graphe K_4

Cela revient dans notre analogie du représentant commercial à dire qu'il est possible d'aller de n'importe quelle ville à n'importe quelle autre sans faire nécessairement étape dans une ville intermédiaire déjà visitée. Par ailleurs, le graphe est non orienté, chaque chemin entre deux villes pouvant être emprunté indifféremmment dans les deux sens. Un graphe complet est bien sûr toujours hamiltonien.

Le problème du voyageur de commerce est connu pour être particulièrement difficile : il est dans la classe de complexité des problèmes NP-complets ; c'est une classe de problèmes algorithmiquement « très difficiles » pour lesquels nous ne savons pas écrire dans l'état actuel des connaissances informatiques d'algorithme ayant une complexité en temps polynomiale.

Résolution par force brute

L'approche naïve (dite par force brute) de résolution du problème du voyageur de commerce consiste à tester toutes les boucles hamiltoniennes d'origine donnée, puis à sélectionner celle qui a la plus petite longueur.

0. Quelle est la complexité de cet algorithme en fonction du nombre n de sommets du graphe ?

1. Écrire une fonction récursive `genere_permutations(L)` prenant en argument une liste `L` et renvoyant la liste de toutes les permutations de ses éléments, chaque permutation étant elle-même codée dans une liste. Par exemple, `genere_permutations([0,1,2])` renverra (dans cet ordre ou dans un autre) :
   `[[0, 1, 2], [0, 2, 1], [1, 0, 2], [1, 2, 0], [2, 0, 1], [2, 1, 0]]`.
2. Écrire une fonction `genere_boucle(n, depart)` prenant en argument un entier `n` et un entier `depart` et renvoyant une liste de toutes les boucles hamiltoniennes possibles d'origine (et d'arrivée) `depart` sur un graphe complet d'ordre n dont les sommets sont numérotés par des nombres de 0 à $n-1$. On se servira de la fonction précédente.
3. Écrire une fonction `distances_boucles(LB, T)` prenant en argument une liste de boucles hamiltoniennes `LB` codées par des listes d'entiers et la matrice des poids `T` du graphe complet où les boucles sont situées, et qui renvoie la liste des distances associées à chacune des boucles, dans le même ordre que celui des boucles.
4. Écrire une fonction `meilleure_boucle(LB, T)` de mêmes arguments que `distances_boucles` renvoyant un tuple contenant l'indice de la boucle la plus courte et son poids total.
5. On dispose d'une liste de coordonnées de points repérés dans un repère orthonormé du plan affine euclidien, codée sous forme de tuples de longueur deux.
   Écrire une fonction `coord_vers_matrice(LC)` qui prend en argument une telle liste et renvoie une matrice des distances « à vol d'oiseau » entre les couples de points.
6. On dispose des coordonnées des points suivants :
   $A(0,0)$, $B(1,1)$, $C(2,4)$, $D(1,-3)$, $E(0,-5)$, $F(0,4)$, $G(-1,-5)$, $H(-2,3)$ et $I(-3,0)$.
   Résoudre le problème du voyageur de commerce sur le graphe ayant ces points comme sommets, la distance étant la distance à vol d'oiseau. Le voyageur part du point A. Représenter les sommets et la solution obtenue.

Problème des 250 villes : création de la matrice des poids

Le problème du voyageur de commerce ne peut être résolu par force brute lorsque le nombre de villes devient relativement important (et il le devient très vite). On cherche alors des solutions approchées, accessibles en un temps raisonnable.
La recherche est encore active en ce domaine, et des défis sur des nombres de villes assez élevés ont même été organisés. Nous nous intéresserons dans la suite de ce TP au challenge du problème du voyageur de commerce pour deux cent cinquante villes tel que décrit ci-dessous :

`http://labo.algo.free.fr/defi250/defi_des_250_villes.html`.

Nous allons chercher une réponse approchée par un algorithme génétique : pour commencer nous allons créer une matrice de poids associée à ce graphe.

7. On fournit un fichier externe `'villes250.txt'` contenant deux cent cinquante lignes de la forme `x, y` correspondant aux coordonnées des deux cent cinquante villes. Écrire une fonction sans argument qui renvoie une liste de deux cent cinquante tuples de la forme (x,y) correspondant à ces coordonnées. On prendra garde à ce que les coordonnées x et y soient de type numérique (flottant).
8. Créer la matrice des poids associée à ce problème.

Une des méthodes pour donner une solution approchée au problème du voyageur de commerce s'appuie sur des **algorithmes génétiques**.

On part d'une génération initiale d'individus aléatoires. **Un individu est une boucle hamiltonienne sur le graphe** et correspond donc à une solution approchée (pas forcément performante et encore moins optimale !) du problème du voyageur de commerce.

On écrit ensuite une fonction permettant de créer la génération suivante à partir des principes suivants :

- plus un individu est performant (*i.e.* plus la route qu'il emprunte est courte), plus il a de chances de se reproduire. Un tirage au sort entre individus reproducteurs est conduit selon le principe de la roulette, principe qui sera détaillé plus loin ;
- à chaque nouvelle génération, les individus peuvent voir leur code génétique muter avec une probabilité p, qui est un paramètre (inconnu...). Une mutation consiste à sélectionner deux indices dans le code génétique d'un individu et à inverser le code génétique de cet individu entre ces deux indices. Ici, cela signifie inverser le sens du trajet entre deux villes dans la boucle hamiltonienne définissant l'individu (ce qui préserve évidemment le caractère hamiltonien du parcours) ;
- deux parents créent deux enfants par un système dit de *cross-over* : le premier parent donne le début (dont l'indice de fin est déterminé aléatoirement) de son parcours à un fils et le second parent donne le début de son parcours à l'autre fils. Les parcours des fils sont alors complétés par les parcours du parent dont le « code génétique » (ici, la boucle hamiltonienne qui le définit) n'a pas été utilisé, en ajoutant à la fin du code génétique du fils les sommets qui n'y apparaissent pas encore, pris dans l'ordre d'apparition dans le code du second parent.

La fonction `random.shuffle(L)` mélange aléatoirement la liste `L`. La liste `L` est modifiée et la fonction renvoie `None`. Voir le chapitre 1 section 8 p. 23.

9. Utiliser `random.shuffle` pour écrire une fonction `cree_initiale(N)` qui renvoie la génération initiale, codée comme liste de listes. Cette génération initiale comportera N individus, qui seront chacun représentés par une liste, commençant par l'élément 0, contenant les indices des deux cent quarante-neuf villes restantes dans un ordre aléatoire et finissant par 0.

10. Écrire une fonction `meilleur_individu(population)` qui prend en argument une liste de listes d'individus représentés par leur parcours, et qui renvoie un tuple contenant la distance totale parcourue par le meilleur individu de `population` et son parcours.

11. Écrire une fonction `mutation(individu)` qui prend en argument un individu décrit par son cycle hamiltonien et codé sous forme d'une liste et qui renvoie l'individu une fois son code génétique muté, comme décrit en début de cette partie. Les indices de mutation sont aléatoires, générés par `random.randint`.

12. Écrire une fonction `crossover(p1, p2)` prenant en argument deux individus parents `p1` et `p2` et renvoyant leurs deux fils, obtenus par le procédé de *cross-over* décrit en début de cette partie.

13. La roulette est une des façons de sélectionner les individus reproducteurs. Un individu donné a une probabilité proportionnelle à $f(d)$ de se reproduire où f est une fonction donnée décroissante et d la distance de parcours de l'individu.
Plus précisément, on dispose au départ d'une génération, codée sous forme de liste de listes. On calcule la liste D des distances totales parcourues par chaque individu, puis la liste F des $f(d)$. À partir de cette liste F, on crée une liste R, représentant la fonction de répartition de

la variable aléatoire X, où $\mathbf{P}(X = x_k)$ est par définition la probabilité que l'individu d'indice k se reproduise. On a ainsi :

$$R_i = \frac{\sum_{j=0}^{i} F_j}{\sum_{j=0}^{N-1} F_j}.$$

On tire ensuite au hasard un flottant x entre 0 et 1. L'entier k tel que $R_k \leqslant x < R_{k+1}$ donne l'indice de l'individu choisi pour se reproduire.

Écrire une fonction `genere_roulette(population, T)`, qui prend en argument une génération et la matrice des poids T et qui renvoie la liste R décrite ici. On pourra prendre $f(d) = \frac{1}{(d-0.99\,m)^3}$, m étant la distance parcourue par l'individu le plus performant de la génération considérée (il s'avère empiriquement que cette fonction donne des résultats convenables).

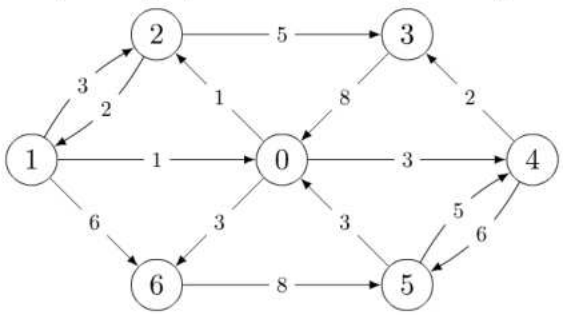
14. Écrire une fonction `indiceroulette(R)`, qui prend en argument la liste précédemment construite et qui renvoie l'indice d'un individu tiré au sort pour se reproduire.

15. Créer une fonction `generation suivante` en utilisant les éléments précédents (génération par *cross-over* et mutation aléatoire).

16. La tester sur plusieurs générations en prenant soin de ne pas lancer de calculs trop longs.

TP 9.1 – Algorithme de Dijkstra

Le but de ce TP est de programmer de façon efficace l'algorithme de Dijkstra, qui calcule, dans un graphe orienté pondéré à masses positives, les plus courts chemins d'une origine I fixée à chaque sommet du graphe. Nous considérons des graphes de la forme $G = (S, A, P)$ où $S = \{0, 1, \ldots, n-1\}$ est un ensemble fini, A une partie de $S \times S$ ne contenant aucun point de la diagonale et P une application de A dans $\mathbb{R}^+$: S est l'ensemble des sommets de G, A l'ensemble des arcs de G et chaque arc $(i, j) \in A$ est de poids $P(i, j)$. Le graphe G sera défini par une liste L de longueur n où, pour tout $i \in \{0, \ldots, n-1\}$, $L[i]$ est la liste des couples $(j, P(i, j))$ où j décrit l'ensemble des successeurs de i. Voici un exemple de graphe et une liste qui le représente :



$L = [[(2,1),(4,3),(6,3)],\ [(0,1),(2,3),(6,6)],\ [(1,2),(3,5)],\ [(0,8)],\ [(3,2),(5,6)],\ [(0,3),(4,5)],\ [(5,8)]]$

Nous utiliserons l'élément `float('inf')` pour représenter le poids $+\infty$, qui est accepté par les opérations usuelles (addition, comparaison des nombres et calcul du minimum), comme on le voit ci-dessous :

```
>>> Inf = float('inf')
>>> 2 < Inf, 3.6 > Inf, Inf <= Inf, 3.4 + Inf, Inf + Inf, min(2, Inf)
(True, False, True, inf, inf, 2)
```

Pour un sommet I fixé, nous allons construire deux listes de longueur n, notées d et π, telles que :

- pour tout i, $d[i]$ est la distance de I à i, c'est-à-dire le poids minimal d'un chemin reliant I à i ; s'il n'existe pas de tel chemin, nous poserons $d[i] = +\infty$;
- pour tout $i \neq I$ tel que $d[i] \neq +\infty$, $\pi[i]$ est le sommet qui précède i dans un plus court chemin qui relie I à i. Par convention, nous poserons $\pi[I] = -1$ et $\pi[j] = -2$ si $d[i] = +\infty$.

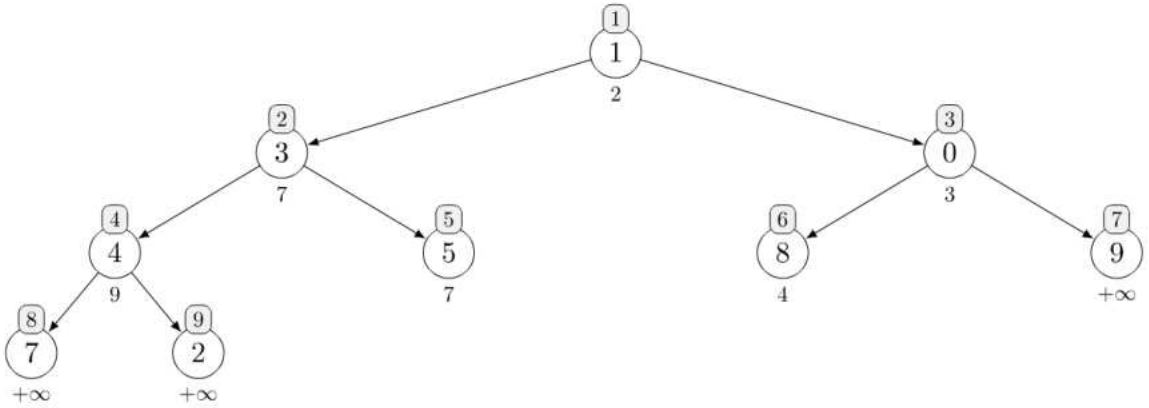
Au début du calcul, chaque case de la liste d contient la valeur $+\infty$, exceptée $d[I]$ qui contient 0, et l'ensemble V des sommets non encore découverts est $\{0, 1, \ldots, n-1\}$. L'algorithme de Dijkstra consiste à extraire de V un élément i tel que $d[i]$ soit minimal, puis à mettre à jour les distances $d[j]$ pour tous les successeurs de i. Pour que ces deux opérations se fassent de façon optimale, nous avons besoin d'une structure de file d'attente efficace.

La structure de tas

Pour gérer cet ensemble dynamique V, partie de $\{0, 1, \ldots, n-1\}$, nous utiliserons une structure de **file de priorité**, c'est-à-dire une file d'attente permettant de classer (partiellement) les éléments en fonction d'une notion de priorité. Dans notre cas, notre structure doit pouvoir rapidement :

- déterminer un élément i de V tel que $d[i]$ est minimal, puis le supprimer de V ;
- pour un élément j de V, prendre en compte une diminution de la valeur $d[j]$ en remettant en ordre la structure de V.

La structure de **tas** permet de faire chacune de ces opérations en un temps de l'ordre du logarithme de la taille de V. Un tas est un arbre binaire dont les nœuds contiennent les éléments de V ; cet arbre est **complet** dans le sens où chaque niveau de l'arbre est rempli, excepté éventuellement le dernier niveau, qui est **tassé à gauche**. Enfin, on impose aux éléments d'être stockés dans cet arbre en respectant une condition raisonnable : si un nœud contient la valeur i et si un de ses fils contient la valeur j, on doit avoir $d[i] \leqslant d[j]$. Les nœuds du tas sont numérotés de 1 à k (k, cardinal de V, est appelé la **taille** du tas) en les parcourant de haut en bas et de gauche à droite. Ainsi, quand $n = 13$, $V = \{0, 1, 2, 3, 4, 5, 7, 8, 9\}$ et $d = [3, 2, +\infty, 7, 9, 7, 0, +\infty, 4, +\infty, 1, 1, 1]$ (les sommets 6, 10, 11 et 12 ne sont plus dans le tas), le tas pourrait être celui représenté ci-dessous, où les nœuds sont numérotés de 1 à 9, chaque sommet i étant également accompagné de la valeur de $d[i]$:



Pour représenter ce tas, nous utiliserons une liste T de taille $n+1$: $T[0] = k$ et pour $x \in \{1, \dots, k\}$, $T[x]$ est la valeur stockée dans le x-ième nœud. Dans l'exemple précédent, nous pouvons avoir :

$$T = [9, \underbrace{1, 3, 0, 4, 5, 8, 9, 7, 2}_{\text{éléments de } V}, \underbrace{12, 10, 11, 6}_{\text{éléments de } S\backslash V}].$$

0. Si $x \in \{1, \dots, k\}$, à quelle condition le x-ième nœud possède-t-il un fils gauche (resp. un fils droit) ? Quel est alors le numéro de ce fils gauche (resp. de ce fils droit) ?

1. Si $x \in \{2, \dots, k\}$, quel est le numéro du père du x-ième nœud ?

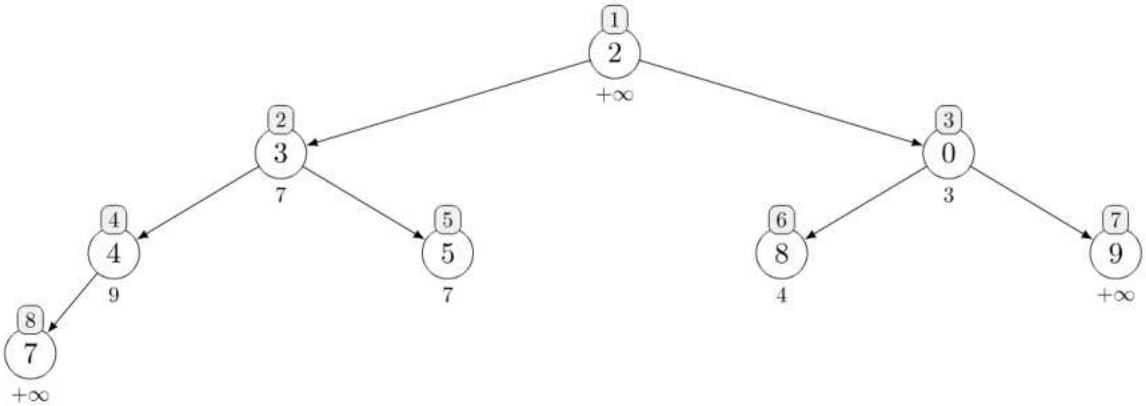
Pour $j \in \{0, \dots, n-1\}$, nous aurons également besoin de calculer l'endroit où la valeur j est stockée dans T. Ceci se fera en définissant une liste *Position* de longueur n telle que $Position[j]$ est le numéro du nœud qui contient la valeur j, ce qui s'écrit $Position[j] = p$ où $T[p] = j$. Dans l'exemple ci-dessus, $Position = [3, 1, 9, 2, 4, 5, 13, 8, 6, 7, 11, 12, 10]$.

2. Écrire une fonction `echange` qui prend en arguments T, *Position*, x et y, avec $x, y \in \{1, \dots, k\}$, et qui échange dans le tas les contenus des cases x et y (sans oublier de modifier *Position* en conséquence).

3. Si $j \in V$, on va être amené à diminuer la valeur $d[j]$. Comment peut-on modifier le tas pour lui redonner une structure valide ? On expliquera la méthode sur l'exemple ci-dessus, après avoir modifié $d[7] = +\infty$ en $d[7] = 3$.

4. Écrire une fonction `remettre_en_forme` qui, appliquée à $(T, Position, d, j)$, remet en forme le tas après que l'on ait diminué la valeur $d[j]$.

La valeur $i = T[1]$ est un élément de V pour lequel $d[i]$ est minimal. Pour supprimer cette valeur de V, comme nous ne pouvons supprimer que le dernier nœud du tas, la première idée est d'appliquer la fonction `echange` avec $x = 1$ et $y = k$, puis de supprimer le dernier nœud du tas (en décrémentant k). Dans l'exemple précédent, cela nous donne le nouvel arbre :



5. Expliquer comment remettre ce tas en forme en effectuant un minimum d'échanges.

6. Écrire une fonction `match_a_trois` qui prend en argument trois « nombres » $a, b, c \in \mathbb{R} \cup \{+\infty\}$ et qui renvoie -1 si $a = \min(a, b, c)$, 0 si $b < a$ et $b \leqslant c$ et 1 sinon.

7. Écrire une fonction `extraire` qui, appliquée à T, *Position* et d supprime (quand $T[0] \geqslant 1$) la valeur stockée à la racine de T et renvoie cette valeur. La fonction devra remettre le tas en forme.

Mise en place de l'algorithme de Dijkstra

Nous rappelons que le graphe G est défini pas une liste L où pour tout sommet i, $L[i]$ est la liste des couples $(j, P(i, j))$ où j décrit l'ensemble des successeurs de i.

8. Comment doit-on initialiser la structure $(T, Position, d, \pi)$ au début de l'algorithme de Dijkstra appliqué au sommet source I ?

9. Écrire une fonction `Dijkstra` qui, appliquée à la liste L et à un sommet I, applique l'algorithme de Dijkstra de calcul des plus courts chemins d'origine I et renvoie les listes d et π.

10. Écrire une fonction `plus_court_chemin` qui, appliquée à la liste L et à deux sommets I et J, renvoie une liste $[i_p, i_{p-1}, \ldots, i_0]$ telle que $(i_0, i_1, \ldots, i_{p-1}, i_p)$ soit un plus court chemin de I à J dans le graphe G (la fonction renverra la liste vide s'il n'existe pas de chemin de I à J).

11. Écrire une fonction `Graphe_Alea` qui, appliquée à un entier $n \geqslant 1$, construit la liste L associée à un graphe aléatoire pondéré sur l'ensemble $\{0, 1, \ldots, n-1\}$. Pour tous sommets distincts i et j, il y aura des arêtes de i à j et de j à i de même poids, choisi aléatoirement dans l'ensemble $[0, 1[$ à l'aide de la fonction `random` du module `numpy.random`. Estimer, en fonction de n, le temps de calcul d'un chemin minimal entre deux sommets dans un tel graphe aléatoire.

TP 9.2 – Puissance 4, algorithme min-max, et α-β

Présentation

On s'intéresse au jeu « **puissance 4** ». Le jeu se déroule sur une grille au départ vide de largeur 7 et hauteur 6 mais on peut bien sûr imaginer un jeu avec une grille d'une autre taille.

Le joueur qui **débute** la partie a des jetons **jaunes** et l'adversaire des jetons **rouges.** À chaque tour, le joueur choisit une colonne et son jeton descend en bas de celle-ci. Le premier qui a 4 jetons de sa couleur alignés horizontalement, verticalement ou diagonalement a gagné. Bien sûr, il peut y avoir égalité si toute la grille est complétée sans qu'aucun des joueurs n'ait réussi à aligner 4 jetons.

Nous allons sur l'exemple de ce jeu nous familiariser avec l'algorithme **minmax** qui parcourt un arbre et son amélioration, l'algorithme α-β; ces algorithmes sont très utilisés dans les simulations de jeux de stratégie pour faire jouer « intelligemment » l'ordinateur.

Ce T.P. se divise en deux parties :

- la mise en place des outils élémentaires de simulation du jeu (grille, présentation sommaire, simulation d'un coup, test de fin de partie). Ce sera un prétexte pour utiliser les tableaux `array` de `numpy`.

- une rédaction élémentaire et récursive de l'algorithme **minmax** avec une profondeur bornée. Cela nécessitera une évaluation heuristique d'une grille dont on vous fournira le code. Puis on pourra accélérer l'algorithme avec sa version α-β et constater que l'on peut encore améliorer l'intelligence artificielle de l'ordinateur...

On fournit également une intergace graphique très élémentaire (`puissance4_gui.py`, `GUI` = Graphical User Interface) écrite à l'aide du package `tkinter` pour que vous testiez votre programme une fois terminé.

Outils élémentaires

Structure de données – grille On commence par fixer les constantes et initialiser la grille.

```
import numpy as np

V = 0  # vide
J = 1  # jaune commence joueur n°1
R = 2  # rouge joueur n°2

INFINI = 100000


def initgrille(lignes, cols):
    """ crée une grille vide et indique qui doit jouer le prochain coup
    ici J (joueur jaune = n°1)
    """
    return np.zeros((lignes, cols), dtype=int), J
```

Une grille à jouer sera donc un `tuple (g, j)` où `g est un` tableau `array` (en général 6×7) représentant la grille et j un entier (`int`) valant `J` ou `R` suivant que c'est au joueur « jaune » ou « rouge » de jouer le coup suivant.

Affichage

0. Pour pouvoir effectuer des tests facilement, écrire une fonction `affichegrille(grille, joueur=None)` qui affiche (avec `print`) la grille d'un puissance 4 de taille quelconque (raisonnable).

Par exemple

```
grille = np.array([[R, J, R, J, J, J, R],
                   [R, R, J, J, R, 0, 0],
                   [R, R, R, J, J, 0, 0],
                   [J, 0, J, 0, 0, 0, 0],
                   [0, 0, 0, 0, 0, 0, 0],
                   [0, 0, 0, 0, 0, 0, 0]], dtype=int)
affichegrille(grille, R)  # au rouge de jouer
```

donnera quelque chose du genre

```
  | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
  | - | - | - | - | - | - | - |
6 |   |   |   |   |   |   |   |
5 |   |   |   |   |   |   |   |
4 | X |   | X |   |   |   |   |
3 | O | O | O | X | X |   |   |
2 | O | O | X | X | O |   |   |
1 | O | X | O | X | X | X | O |
C'est au joueur Rouge de jouer (R = O)
```

Il s'agit donc de construire une chaîne de caractère.
Rappel `'\n'` désigne le retour à la ligne, la méthode du type `str` `.join(s)` est particulièrement utile.

1. **Colonne pleine ?**
   On dispose d'un tableau array `grille` de taille
   `lignes` × `cols` (`lignes, cols = grille.shape`).
   Écrire la fonction `colpleine(grille, col)` qui renvoie `True` si la colonne numéro `col` (en démarrant de 0) est pleine.
   On pourra dans un premier temps rédiger un programme « universel » puis utiliser la machinerie de `numpy`.
   Indication Tester le retour de `np.array([1, 2, 3, 3, 4, 1, 5]) == 1` puis les méthodes `.any()`, `.all()`.
   Tester votre fonction.
2. **Blocs de 4 jetons, test de fin de partie**
   On se donne une grille de taille `lignes` × `cols` d'indices de 0 à `lignes` × `cols`−1 de la manière suivante

```
A = np.arange(lignes * cols, dtype=int).reshape((lignes, cols))
```

   Écrire la fonction `listeind(lignes, cols)` qui renvoie une liste d'`array` à quatre éléments qui constituent tous les blocs de quatre indices horizontaux, verticaux ou diagonaux (dans les deux sens).
   Vous pouvez utiliser les *fancy indexing* `[deb:fin:step]`,
   les fonctions/méthodes `np.transpose()`, `np.diagonal(d)`.
3. On peut aplatir un tableau `array` bidimensionnel par la méthode `.flatten()`. À partir de là, on récupère facilement les blocs de quatre éléments d'une grille donnée `G` à partir d'une liste d'indices `L` obtenue par la fonction précédente

```
def creepaquets(grille):
    """ crée la liste des paquets de 4 horizontaux, verticaux ou diagonaux
    de la grille
    """
    lignes, cols = grille.shape
    L = listeind(lignes, cols)
    G = grille.flatten()
    return [G[l] for l in L]
```

   Écrire une fonction `testfin(grille)` qui teste si la grille `grille` est gagnante (blocs de 4 `J` ou 4 `R`). La fonction renvoie le numéro du joueur gagnant, 0 s'il n'y a pas de gagnant et −1 si la grille est pleine. Tester votre fonction.
4. **Coup suivant**
   Écrire la fonction `coupsuivant_col(grille, joueur, col)` qui, à partir d'une grille et du joueur devant jouer le coup suivant, renvoie la grille et le joueur du coup correspondant au choix de la colonne numéro `col`.
   Tester votre fonction.
5. Écrire la fonction `coupsuivants(grille, joueur)` qui renvoie la liste des coups suivants possibles (sous la forme d'une liste de `tuple` (grille, joueur)).
6. **Fonction d'évaluation d'une grille**
   On fournit une fonction d'évaluation d'une grille écrite de façon empirique et peu raffinée. Plus la note est élévée, plus le joueur jouant les jetons jaunes a de chance de gagner et c'est

l'opposé pour le joueur jouant les jetons rouges. Si une grille est gagnante, on lui attribue la note maximale « ±INFINI-1 » mais c'est plus compliqué quand la grille est incomplète. Essayer de comprendre comment est calculée cette note.

```
def evaluegrille(grille, joueur):
    """ attibue une note d'évaluation de la grille pour le joueur
    (si J, on prend le max, sinon l'opposé)
    c'est un calcul très très mauvais... À améliorer durant
    les longues soirées d'hiver
    """
    fini = testfin(grille)
    if fini != 0:
        if fini == J:   # + pour J et - pour R
            return INFINI - 1    # subtilité pour l'algorithme alpha-beta !
        elif fini == R:
            return - INFINI + 1  # idem, subtil...
        else:
            return 0  # math nul (fini = -1)

    # en principe on n'a pas 4 pareils
    scoreJ = np.array([0, 0, 0, 0], dtype=int)   # score pour 0 à 3 éléments J
    scoreR = np.array([0, 0, 0, 0], dtype=int)
    coeff = np.array([0, 0, 1, 4], dtype=int)

    T = creepaquets(grille)
    for b in T:
        nb = (b == 0).sum()
        if nb != 4:   # si pas que des 0
            nbJ = (b == J).sum()
            nbR = (b == R).sum()
            if nbR == 0:
                scoreJ[nbJ] += 1
            elif nbJ == 0:
                scoreR[nbR] += 1

    score = (coeff * scoreJ).sum() - (coeff * scoreR).sum()

    return score
```

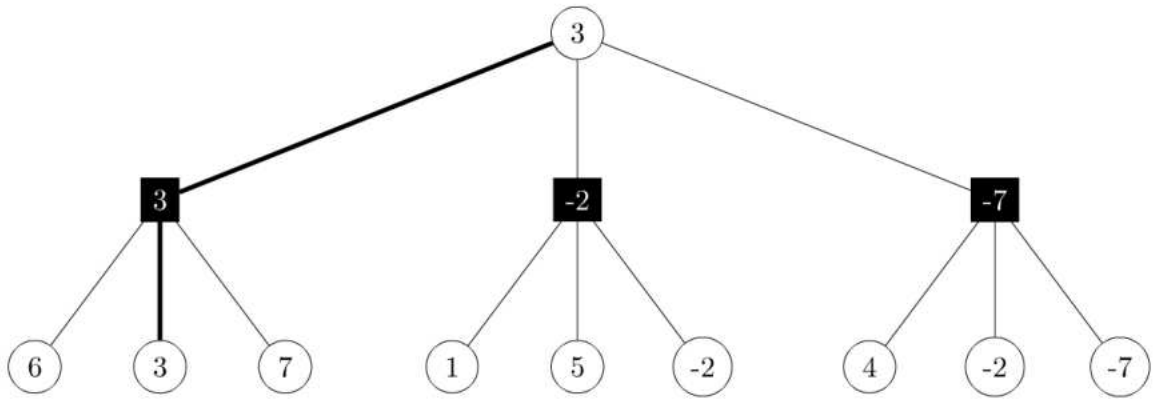
7. Simulation d'une partie humain – humain

Écrire une fonction `simulationpartie(grille, joueur)` qui simule une partie humain contre humain. On part de la situation `(grille, joueur)`, on demande à chaque tour le numéro de la colonne choisie pour le joueur concerné (utiliser `input`) et on affiche la nouvelle grille avec le score calculé par notre fonction d'évaluation.
On s'arrête quand un des joueurs a gagné ou qu'il y a match nul.

Intelligence artificielle

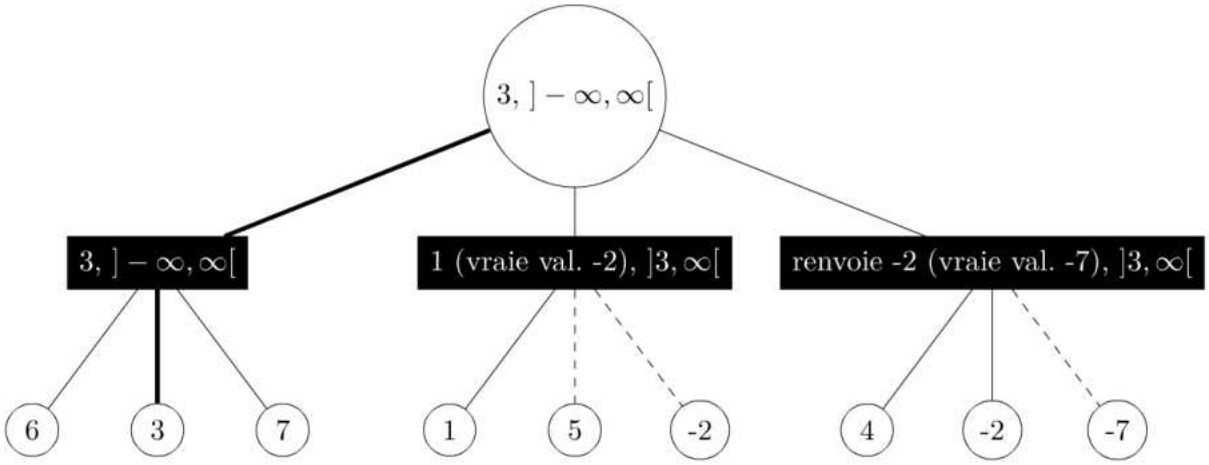
Algorithme minmax Supposons que c'est au joueur `J` de jouer. Celui-ci va regarder tous les coups suivants possibles et choisir la grille de note maximale (resp. pour le joueur `R` celle de note minimale) mais cette évaluation n'est intéressante que si la grille est déjà bien avancée. Il vaut donc mieux étudier plusieurs coups à l'avance et construire ainsi un **arbre** des coups possibles. Dans l'idéal, on construit l'arbre jusqu'aux grilles gagnantes/perdantes/nulles qui en sont les feuilles, puis on remonte les notes en alternant des calculs de **min** ou de **max** suivant la hauteur de l'arbre. Ceci est possible dans le cas d'un jeu comme le morpion par exemple car l'arbre reste de taille raisonnable, mais ce n'est plus envisageable dans le cas du jeu de puissance 4 pour une grille de taille 6×7. On borne donc la hauteur de l'arbre (= on n'étudie que quelques coups d'avance) ; si les grilles des feuilles de cet arbre partiel sont gagnantes/perdantes/nulles, tant mieux car l'évaluation

est simple, sinon il faut donner une note « à la louche ». Dans notre cas, on utilisera la fonction d'évaluation grossière proposée dans l'énoncé.
En résumé, on va écrire un algorithme appelé **minmax** dans une version récursive en construisant la fonction `minmax(G, profondeur)` où `G = (grille, joueur)` qui renvoie la meilleure grille suivante pour le joueur `G[1]` à partir de la grille `G[0]` en étudiant un arbre de racine `G[0]` de profondeur `profondeur`. Les feuilles de cet arbre ont pour « score » la fonction d'évaluation que l'on vous a proposée et les autres nœuds sont calculés récursivement suivant que ce sont des nœuds « max » (joueur `J`) ou des nœuds « min » (joueur `R`). En pratique, la valeur de la variable `profondeur` ne dépassera pas 5 ou 6.
Voici une figure représentant un arbre « minmax »



8. Écrire la fonction `minmax(G, profondeur)` qui renvoie le `tuple (score, coup_suivant)` où `coup_suivant` est le `tuple (grille, joueur)`.
9. Tester cette fonction en écrivant une fonction
`simulationpartieordijoueur(grille, joueur, profondeur)`
qui simule une partie humain – ordinateur ou bien utiliser l'interface graphique proposée dans le fichier `puissance4_gui.py`.

Algorithme $\alpha - \beta$ Vous avez pu constater que dès que la profondeur dépasse 3, l'exploration de l'arbre prend un peu de temps. On peut améliorer le parcours de l'arbre minmax en coupant quelques branches inutiles grâce à l'algorithme appelé $\alpha - \beta$. Il est assez subtil ; voici le cahier des charges de la fonction `alphabeta(G, alpha, beta, profondeur)` : cette fonction doit renvoyer le **score** de la position `G` (= la valeur du nœud) dans l'**arbre** élaboré à la **profondeur** `profondeur` et donner le meilleur coup suivant (ou la position `G` s'il s'agit d'une feuille) **si ce score appartient à** $]\alpha, \beta[$ **sinon, si ce score est** $\geqslant \beta$, **il peut renvoyer une approximation inférieure avec le coup suivant correspondant et, si ce score est** $\leqslant \alpha$, **il renvoie une approximation supérieure toujours avec le coup correspondant.**
On obtiendra le score d'une position `G` en appelant
`alphabeta(G, -INFINI, INFINI, profondeur)` ce qui explique *a posteriori* la valeur `INFINI-1` dans la fonction d'évaluation.
Voici l'arbre précédent modifié par l'algorithme : les traits en pointillés correspondent à des nœuds non parcourus.



10. Modifier la fonction `minmax` pour concevoir la fonction `alphabeta`, la tester et constater en principe un gain de rapidité.

11. Tester l'exemple suivant :

```
grille = np.array([[0, R, 0, J, R, J, 0],
                   [0, J, 0, R, J, R, 0],
                   [0, R, 0, J, R, J, 0],
                   [0, J, 0, R, 0, R, 0],
                   [0, 0, 0, 0, 0, J, 0],
                   [0, 0, 0, 0, 0, 0, 0]])
joueur = J
```

Que constate-t-on ?

12. Améliorer votre fonction `alphabeta` pour que le meilleur coup suivant soit plus « naturel » humainement parlant...

Corrections des exercices

Corrigé exo 9.0

0. La matrice d'adjacence utilise un espace mémoire de l'ordre de n^2 ; l'espace mémoire nécessaire au stockage de L est de l'ordre de $\sum_{i=0}^{n-1} 1 + len(L[i])$ (il faut ajouter 1 car même le stockage de la liste vide demande un peu d'espace), soit $n + \text{Card}(E)$. L'implémentation sous forme de listes d'adjacence est donc préférable quand le graphe contient peu d'arcs (plus précisément quand le cardinal de E est négligeable devant n^2).

1. Pas de problème particulier : la première est en temps constant, la seconde demande un temps de l'ordre de la longueur de $L[i]$ dans le pire des cas (atteint quand j n'est pas un successeur de i). Ce temps peut donc être de l'ordre de n quand le graphe est dense.

```
def arc_mat(A, i, j):
    return A[i][j] == 1
```

```
def arc_lis(L, i, j):
    return j in L[i]
```

2. Au début du calcul, A est la matrice nulle et un parcours de toutes les listes d'adjacence suffit pour détecter tous les arcs du graphe :

```
def Matrice(L):
    n = len(L)
    A = [[0 for j in range(n)] for i in range(n)]
    for i in range(n):
        for j in L[i]:
            A[i][j] = 1
    return A
```

La fonction réciproque s'écrit facilement en définissant la liste $L[i]$ par compréhension. On remarquera que l'entier $A[i][j]$ peut être directement traité comme un booléen.

```
def Liste(A):
    n = len(A)
    return [[j for j in range(n) if A[i][j]] for i in range(n)]
```

3. Une fois d initialisée à la valeur matrice nulle, nous parcourons une nouvelle fois toutes les listes d'adjacence : dès qu'un arc (i, j) est détecté, nous incrémentons les valeurs $d[i][1]$ et $d[j][0]$ (l'arc sort de i et entre en j).

```
def d(L):
    n = len(L)
    d = [[0, 0] for i in range(n)]
    for i in range(n):
        for j in L[i]:
            d[i][1] += 1
            d[j][0] += 1
    return d
```

Corrigé exo 9.1

0. On raisonne par récurrence, montrons la propriété :
$(\mathscr{P}_n)$: il y a $A^n_{i,j}$ chemins de longueur n allant de i à j pour deux sommets i et j quelconques.
La propriété est vraie pour $n = 1$ par définition de la matrice d'adjacence.
Supposons la propriété vraie pour un $n \geqslant 1$ donné et fixons i et j deux sommets.
On a $A^{n+1}_{i,j} = (A^n A)_{i,j} = \sum_{k=0}^{k=\text{len(A)-1}} A^n_{i,k} A_{k,j}$.
Les chemins de longueur $n+1$ allant de i à j peuvent être vus comme des chemins de longueur n allant de i à un sommet quelconque k (il y en a $A^n_{i,k}$) suivis d'un chemin de longueur 1 allant de k à j (il y en a $A_{k,j}$). Il y a donc $A^n_{i,k}A_{k,j}$ chemins de longueur $n+1$ passant par k après un parcours de longueur n et en sommant sur l'ensemble des sommets k, on trouve bien que $A^{n+1}_{i,j}$ est le nombre total de chemins de longueur $n+1$ allant de i à j. $(\mathscr{P}_{n+1})$ est donc vraie.

1.

```
import numpy as np


def CheminLongueurN(A, i, j, n):
    return np.linalg.matrix_power(A, n)[i, j]
```

Le lecteur ne connaissant pas cette fonction aurait pu écrire à la main :

```
def CheminLongueurN(A, i, j, n):
    B = np.copy(A)
    for k in range(n - 1):
        B = np.dot(B, A)
    return B[i, j]
```

2. On pose $T = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$. On a $M = I_3 + T$, $T^2 = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$ et $T^n = 0_3$ pour $n \geqslant 3$.
Comme I et T commutent, on a d'après la formule du binôme de Newton :

$$M^n = (I+T)^n = \binom{n}{0} I + \binom{n}{1} T^1 + \binom{n}{2} T^2 = I + nT + \frac{n(n-1)}{2} T^2 = \begin{pmatrix} 1 & n & \frac{n(n-1)}{2} \\ 0 & 1 & n \\ 0 & 0 & 1 \end{pmatrix}$$

Il y a donc $\dfrac{100 \times 99}{2} = 4950$ chemins de longueur 100 entre les sommets 0 et 2.

3. Les chemins de longueur 100 font 98 boucles, une transition entre le sommet 0 et le sommet 1 et une transition entre le sommet 1 et le sommet 2. Choisir un chemin de longueur 100 entre 0 et 2 revient donc à choisir deux instants parmi 100 possibles pour effectuer ces transitions : il y a donc $\binom{100}{2} = 4950$ chemins de longueur 100 entre les sommets 1 et 2.

Corrigé exo 9.2

0. La matrice des poids de ce graphe, en prenant comme convention que $a_{i,j} = 0$ si les sommets i et j ne sont pas reliés, s'écrit :

$$\begin{pmatrix} 0 & 1 & 3 & 2 & 4 \\ 1 & 0 & 9 & 0 & 8 \\ 3 & 9 & 0 & 7 & 0 \\ 2 & 0 & 7 & 0 & 0 \\ 4 & 8 & 0 & 0 & 0 \end{pmatrix}$$

ce qui donne en Python :

```
M = [[0, 1, 3, 2, 4], [1, 0, 9, 0, 8], [
    3, 9, 0, 7, 0], [2, 0, 7, 0, 0], [4, 8, 0, 0, 0]]
```

1. Nous allons parcourir L par indices et tester si le nombre en ligne $L[i]$ et en colonne $L[i+1]$ est nul dans la matrice M :

```
def test_chaine(M, L):
    for i in range(len(L) - 1):
        if M[L[i]][L[i + 1]] == 0:
            return False
    return True
```

2. On modifie le code précédent de façon à garder en mémoire la longueur totale du chemin déjà parcouru :

```
def longueur(M, L):
    s = 0
    for i in range(len(L) - 1):
        if M[L[i]][L[i + 1]] == 0:
            return -1
        s = s + M[L[i]][L[i + 1]]
    return s
```

Corrigé exo 9.3

0. On commence par créér une matrice de zéros que l'on remplit à l'aide des éléments de L :

```
def ListeMatrice(n,L):
    M=[[0 for i in range(n)] for j in range(n)]
    for c in L:
        M[c[0]][c[1]]=1
        M[c[1]][c[0]]=0
    return M
```

1. Il suffit de taper dans la console :
`ListeMatrice(4,[[0,2],[0,3],[1,3]])`

Corrigé exo 9.4

0. On peut par exemple utiliser une liste *Liste* : l'insertion se fera « à droite » de la liste à l'aide de la méthode append ; la suppression pourrait se faire en supprimant (et en récupérant) le premier élément de la liste, mais cela serait coûteux en temps de calcul. Il est donc ici plus efficace d'utiliser un pointeur i qui va indiquer la position de la tête de la liste. Nous représenterons donc une file d'attente comme une liste $[Liste, i]$: la tête de la liste est $L[i]$ et on supprime la tête en incrémentant i. La file est vide quand i est égal à la longueur de *Liste* ($L[i]$ n'est alors pas défini).

```
def creer_file_vide():
    return [[], 0]


def est_vide(F):
    return F[1] == len(F[0])


def ajouter(a, F):
    F[0].append(a)


def lire(F):
    a = F[0][F[1]]
    F[1] += 1
    return a
```

1. Dans les deux cas, on initialise les listes π et d aux valeurs $[-2, \ldots, -2]$ et $[-1, \ldots, -1]$ et on applique les méthodes proposées, en mettant à jour les valeurs $\pi[k]$ et $d[k]$ chaque fois qu'un sommet k est découvert :

```
def parcours_largeur(L, i):
    n = len(L)
    d = [-1 for j in range(n)]  # on initialise d et pi
    pi = [-2 for j in range(n)]
    d[i], pi[i] = 0, -1
    D = [i]  # i est le seul sommet à la distance 0 de i
    while(D!= []):  # D contient les éléments j tels que d(i,j) = alpha
        ND = []
        for j in D:  # pour chaque élément j de D
            for k in L[j]:  # pour chaque successeur k de j
                if d[k] == -1:  # qui n'a pas été découvert
                    d[k] = d[j] + 1   # d(i,k) = 1 + alpha = 1 + d(i,j)
                    pi[k] = j   # on retient l'arc (j,k)
                    # on ajoute k à la liste des nouveaux sommets découverts
                    ND.append(k)
        # D contient maintenant les sommets k tels que d(i,k) = alpha + 1
        D = ND
    return d, pi


def parcours_largeur_file(L, i):
    n = len(L)
    d = [-1 for j in range(n)]  # on initialise d et pi
    pi = [-2 for j in range(n)]
    d[i], pi[i] = 0, -1
    F = creer_file_vide()  # i est le seul élément découvert
    # les éléments seront rangés dans la file dans l'ordre de leur distance à i
    ajouter(i, F)
    while(not(est_vide(F))):
        j = lire(F)
        for k in L[j]:
```

```
            if d[k] == -1:
                d[k] = d[j] + 1
                pi[k] = j
                ajouter(k, F)
    return d, pi
```

2. On peut calculer les listes d et π, puis reconstruire le chemin permettant de relier i à j :

```
def plus_court_chemin(L, i, j):
    d, pi = parcours_largeur(L, i)
    # même si d(j) = -1, le chemin cherché contient 1 + d(j) sommets
    # on crée la liste: seule la dernière valeur est correcte
    C = [j for k in range(1 + d[j])]
    for k in range(d[j]):  # on définit le chemin correct
        C[-k - 2] = pi[C[-k - 1]]  # en parcourant C de droite à gauche
    return C
```

Le lecteur pourra également reprendre le code de la fonction `parcours_largeur_file` en arrêtant l'exploration dès que le sommet j est découvert.

Corrigé exo 9.5

0. On traduit la définition du parcours en profondeur (les successeurs sont étudiés dans l'ordre de leur apparition dans les listes d'adjacence) :

```
def parcours_profondeur(L):
    n = len(L)
    p = [-2 for j in range(n)]

    def explorer(i):
        for j in L[i]:  # chaque successeur j de i
            if p[j] == -2:  # qui n'a pas été rencontré
                p[j] = i  # devient le fils de i
                explorer(j)  # et est exploré
    for i in range(n):
        if p[i] == -2:  # dès que l'on rencontre un nouveau sommet
            p[i] = -1  # il devient racine d'un arbre
            explorer(i)  # puis est exploré
    return p
```

1. On commence par initialiser la liste π à la valeur $[-2, \ldots, -2]$ qui traduit le fait qu'aucun sommet n'a été découvert au début du calcul. On va une nouvelle fois étudier chaque sommet : dès que l'on trouve un sommet i qui n'a pas encore été découvert, on lance l'exploration depuis la racine i. L'idée consiste à créer une pile contenant initialement le seul sommet i. Tant que P est non vide, on en extrait la tête j et on ajoute à la pile les successeurs k de j qui n'ont pas encore été découverts. La difficulté, c'est qu'un même sommet pourra être inséré plusieurs fois dans la pile : ce n'est qu'au moment où il sortira de la pile qu'il sera considéré comme « rencontré » et que son père sera connu. Considérons l'exemple très simple où $L = [[1, 2], [\,], [1]]$. On commence donc le calcul avec $P = [0]$; on dépile alors P : le père de 0 est -1 (0 est une racine) ; on ajoute alors les successeurs de 0 dans P, qui devient $P = [1, 2]$. On dépile 2 : son père est 0 (car c'est en étudiant les successeurs de 0 que l'on a inséré 2 dans la pile), puis on insère 1 qui est le seul successeur non encore rencontré de 2. Nous avons alors $P = [1, 1]$ ... on dépile 1 qui n'a toujours pas été rencontré : on sait alors que son père est 2, puisque ce 1 là a été inséré au moment de l'étude des successeurs de 2. 1 n'a pas de nouveau successeur, donc on reste avec $P = [1]$. On termine en dépilant P : 1 a déjà été rencontré et on ne fait rien, ce qui achève

le parcours en profondeur depuis 0. On voit donc qu'il ne faut pas se contenter d'empiler les valeurs des successeurs, mais les couples (k, j) : ainsi, quand on dépilera un couple (k, j), on saura que cette occurence du sommet k a été ajoutée en venant de j, qui sera son père. Dans l'exemple précédent, voici comment vont évoluer la pile P et la liste π :

π	P	
$[-2, -2, -2]$	$[(0, -1)]$	On découvre 0 dont le père -1
$[-1, -2, -2]$	$[(1, 0), (2, 0)]$	On découvre 2 dont le père 0
$[-1, -2, 0]$	$[(1, 0), (1, 2)]$	On découvre 1 dont le père 2
$[-1, 2, 0]$	$[(1, 0)]$	Le sommet 1 a déjà été découvert
$[-1, 2, 0]$	$[]$	Le calcul est terminé

On obtient ainsi la fonction :

```
def parcours_profondeur_iteratif(L):
    n = len(L)
    p = [-2 for j in range(n)]  # liste des pères
    for i in range(n):
        if p[i] == -2:  # dès que l'on rencontre un nouveau sommet i
            # on lance l'exploration depuis i (son père sera -1)
            Pile = [(i, -1)]
            while Pile != []:
                j, pere = Pile.pop()  # c'est au moment où le sommet j est dépilé
                if p[j] == -2:  # et s'il n'a pas encore été découvert
                    p[j] = pere  # que l'on définit son père
                    for k in L[j]:  # puis on ajoute dans la pile les successeurs
                        if p[k] == -2:  # k non encore découverts, qui
                            # ont à cet instant été atteint en venant de j
                            Pile.append((k, j))
    return p
```

2. Le calcul récursif est très facile à faire puisque le traitement d'un sommet i consiste simplement à lui appliquer la fonction `explorer` : on pourra donc mettre à jour $d[i]$ et $f[i]$ dans le corps de la fonction `explorer`, en utilisant la variable « non locale » t, initialisée à la valeur 1 et incrémentée à chacune de ses utilisations.

```
def parcours_profondeur(L):
    n = len(L)
    p = [-2 for j in range(n)]  # liste des pères
    d = [0 for j in range(n)]  # liste des instants de début de traitement
    f = [0 for j in range(n)]  # liste des instants de fin de traitement
    t = 1

    def explorer(i):
        nonlocal t
        d[i] = t  # début du traitement du sommet i à l'instant t
        t += 1
        for j in L[i]:  # chaque successeur j de i
            if p[j] == -2:  # qui n'a pas été rencontré
                p[j] = i  # devient le fils de i
                explorer(j)  # et est exploré
        f[i] = t  # fin du traitement du sommet i à l'instant t
        t += 1
    for i in range(n):
```

```
        if p[i] == -2:  # dès que l'on rencontre un nouveau sommet
            p[i] = -1  # il devient racine d'un arbre
            explorer(i)  # puis est exploré
    return p, d, f
```

Corrigé exo 9.6

0. Ce graphe est eulérien car 1,2,6,5,2,3,4,5,1,0,7,6,1 (par exemple) est une chaîne eulérienne. Il est possible de trouver une chaîne eulérienne partant de n'importe quel sommet (et ce sera toujours un cycle eulérien).

1. Il suffit de compter le nombre d'éléments non nuls dans $M[i]$:

```
def degre(M, i):
    s = 0
    for j in len(M[i]):
        if M[i][j]!= 0:
            s += 1
    return s
```

2. Un graphe est connexe si, et seulement si, il existe une chaîne entre toute paire de sommets. C'est équivalent à dire qu'il existe une chaîne reliant le sommet d'indice 0 à tout sommet.
En réalisant un parcours du graphe (qu'il soit par largeur ou par profondeur), on obtient cette information. En se servant des fonctions des exercices précédents :

```
def est_connexe(M):
    L = Liste(M)
    P = parcours_profondeur(L, 0)
    return -1 not in P
```

3. Il suffit ici de reprendre les résultats des deux fonctions précédentes :

```
def est_eulerien(M):
    if not est_connexe(M):
        return False
    for i in range(len(M)):
        if degre(M, i)\% 2!= 0:
            return False
    return True
```

Corrigé exo 9.7

0. Une solution est [Caleçon, Chemise, Chaussettes, Pantalon, Chaussures, Veste].

1. *A priori*, trois situations sont envisageables :

- $d[i] < f[i] < d[j] < f[j]$: comme j n'a pas été découvert avant l'exploration de i, ni pendant l'exploration de i, il n'existe pas de chemin de i à j ;
- $d[i] < d[j] < f[i]$: cela signifie que la fonction `explorer` a été appelée sur le sommet j au cours de l'exécution de l'appel `explorer`(i) : cela implique que $f[j] < f[i]$ et ce cas ne peut pas se produire ;
- $d[j] < d[i] < f[i] < f[j]$: l'exploration du sommet i a été faite pendant l'exploration du sommet j, donc il existe un chemin de j à i.

Dans les deux cas, il n'existe pas de chemin de i à j (sinon, dans le second cas, on aurait un cycle en mettant bout à bout un chemin de i à j et un chemin de j à i).

2. La contraposée du résultat précédent s'écrit : s'il existe un chemin de i à j, $f[j] < f[i]$. On obtient donc un ordre topologique en classant les sommets par ordre décroissant de leurs temps de fin de traitement. Il n'est pas nécessaire de calculer la liste f pour ensuite trier les sommets : il suffit de créer une liste `ordre` de longueur n, puis de reprendre le code de la fonction `Parcours_Profondeur` ; à la fin de la fonction `explorer`, au lieu de définir $f[i]$, on copie i dans la liste `ordre` (la liste sera remplie de droite à gauche, puisque nous obtenons les sommets par ordre croissant de leurs temps de fin de traitement). Comme nous n'avons pas besoin de la liste des pères, nous la remplaçons par une liste `deja_vu` qui permet de savoir si un sommet a déjà été rencontré.

```
def ordre_topologique(L):
    n = len(L)
    # au début du calcul, aucun sommet n'a été découvert
    deja_vu = [False for j in range(n)]
    ordre = [0 for j in range(n)]
    t = n - 1  # on va remplir la liste ordre en commençant par la fin

    def explorer(i):
        nonlocal t
        for j in L[i]:  # chaque successeur j de i
            if not(deja_vu[j]):  # qui n'a pas ete vu
                deja_vu[j] = True  # a maintenant été vu
                explorer(j)  # et est exploré
        ordre[t] = i  # on termine l'exploration de i en le copiant dans la liste
        t -= 1  # et on décrémente l'indice d'insertion
    for i in range(n):
        if not(deja_vu[i]):  # dès que l'on rencontre un nouveau sommet
            deja_vu[i] = True  # on dit qu'il a été vu
            explorer(i)  # et on l'explore
    return ordre
```

Corrections des TP

Corrigé TP 9.0

0. Dans un graphe complet, il existe $(n-1)!$ cycles hamiltoniens d'origine donnée (on passe dans un ordre quelconque par les $n-1$ autres sommets entre le départ et l'arrivée). Pour un cycle donné, le calcul de la longueur est linéaire et on réactualise le chemin minimal trouvé. La complexité est en $\mathcal{O}(n!)$, autant dire que c'est rédhibitoire.

1.

```
def genere_perm(L):
    n = len(L)
    if n == 1:
        return [L]
    return [P[:k] + [L[n - 1]] + P[k:] for k in range(n) for P in genere_perm(L[:n - 1])]
```

2.

```
def genere_boucle2(n, depart):
    L = list(range(depart)) + list(range(depart + 1, n))
    perm = genere_perm(L)
    return [[depart] + p + [depart] for p in perm]
```

3.

```
def longueur_chaine(L, T):
    dist = 0
    for i in range(len(L) - 1):
        a = T[L[i], L[i + 1]]
        if a == -1:
            return -1
        dist += a
    return dist

def distances_boucles(LB, T):
    return [longueur_chaine(B, T) for B in LB]
```

4.

```
def meilleure_boucle(LB, T):
    poids = distances_boucles(LB, T)
    i, rec = 0, poids[0]
    for j in range(len(LB)):
        if poids[j] < rec:
            i, rec = j, poids[j]
    return i, rec
```

5.

```
def coord_vers_matrice(LC):
    n = len(LC)
    M = np.zeros((n, n))
    for i in range(n - 1):
        for j in range(i + 1, n):
            M[i, j] = np.sqrt((LC[i][0] - LC[j][0])**2 +
                              (LC[i][1] - LC[j][1])**2)
```

```
            M[j, i] = M[i, j]
    return M
```

6.

```
L = [(0, 0), (1, 1), (2, 4), (1, -3), (0, -5),
     (0, 4), (-1, -5), (-2, 3), (-3, 0)]

M = coord_vers_matrice(L)
LB = genere_boucle(len(L), 0)
indmin, lmin = meilleure_boucle(LB, M)
print(indmin, lmin)

import matplotlib.pyplot as plt
X = [L[i][0] for i in range(9)]
Y = [L[i][1] for i in range(9)]
plt.scatter(X, Y)
CB = [L[i] for i in LB[indmin]]  # meilleur cycle hamiltonien
X1 = [CB[i][0] for i in range(10)]
Y1 = [CB[i][1] for i in range(10)]
plt.plot(X1, Y1)  # , 'bo-')
plt.show()
```

7.

```
f = open('villes250.txt', 'r')
villes = []
for ligne in f:
    x, y = ligne.split(',')
    villes.append((float(x), float(y)))
f.close()
```

8.

```
M = coord_vers_matrice(villes)
```

9.

```
import random as rd


def cree_initiale(N, nbvilles=250):
    Pop = []
    for i in range(N):
        a = list(range(1, nbvilles))
        np.random.shuffle(a)
        Pop.append([0] + a + [0])
    return Pop
```

10.

```
def meilleur_individu(population, M):
    i, p = meilleure_boucle(population, M)
    return population[i]   # on oublie p
```

11.

```
import random as rd

def mutation(individu, probamut=p):
    x = rd.random()
```

```
    if x < probamut:  # on mute
        newind = individu[:]
        n = len(individu)
        a, b = rd.sample(list(range(n)), 2)
        a, b = min(a, b), max(a, b)
        T = newind[a:b]
        T.reverse()
        return newind[:a] + T + newind[b:]
    return individu
```

12.

```
def crossover(p1, p2):
    indice = rd.randint(1, len(p1) - 1)
    fils1, fils2 = p1[:indice], p2[:indice]
    for j in range(len(p1)):
        if p2[j] not in fils1[:indice]:
            fils1.append(p2[j])
        if p1[j] not in fils2[:indice]:
            fils2.append(p1[j])
    return fils1, fils2
```

13.

```
def genereroulette(population, M):
    ch = min([longueur_chaine(i, M) for i in population])
    S = [1 / (longueur_chaine(i, M) - ch * .99)**3 for i in population]
    somme = sum(S)
    S = [i / somme for i in S]
    R = [S[0]]
    for i in range(len(S) - 1):
        R.append(R[i] + S[i + 1])
    return R
```

14.

```
def indiceroulette(R):
    indice, a = 0, rd.random()
    while indice < len(R):
        if a <= R[indice]:
            return indice
        indice += 1
    return indice - 1  # normalement on ne passe pas par là
```

15.

```
def generation_suivante(population, probamut, M):
    newgen = []
    R = genereroulette(population, M)
    for a in range(len(population) // 2):
        i, j = indiceroulette(R), indiceroulette(R)
        fils1, fils2 = crossover(population[i], population[j])
        newgen.extend([fils1, fils2])
    for i in range(len(newgen)):
        newgen[i] = mutation(newgen[i], probamut)
    return newgen
```

16.

```
def generationssuccessives(N, nbiter, probamut, M):
    nvilles, Pop = len(M), cree_initiale(N)
    L = [meilleur_individu(Pop, M)]
    for i in range(nbiter):
```

```
        Pop = generation_suivante(Pop, probamut, M)
        L.append(meilleur_individu(Pop, M))
    print(i, longueur_chaine(L[-1], M))
    return L

L = generationssuccessives(100, 500, .06, M)
```

Corrigé TP 9.1

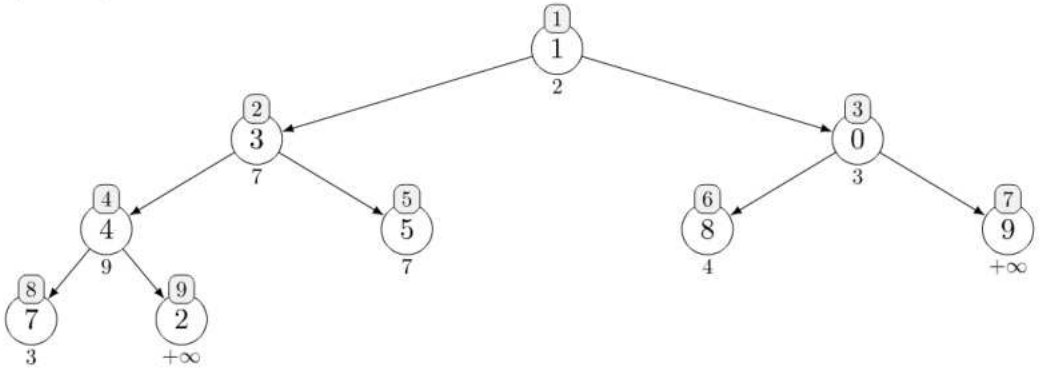
0. Les fils gauche et droit portent respectivement les numéros $2x$ et $2x+1$. Le nœud x possède donc un fils gauche (resp. un fils droit) si et seulement $2x \leqslant k$ (resp. $2x+1 \leqslant k$).

1. Le père du nœud x (quand $2 \leqslant x \leqslant k$) a pour numéro le quotient entier de x par 2.

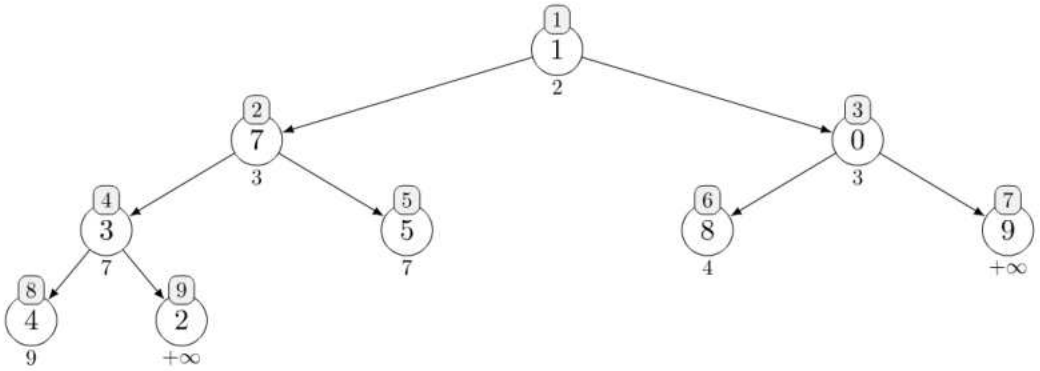
2. En posant $i = T[x]$ et $j = T[y]$, il faut échanger dans T les contenus des cases x et y et échanger dans *Position* les contenus des cases i et j. Comme $Position[i] = x$ et $Position[j] = y$, cela donne :

```
def echange(T, Position, x, y):
    Position[T[y]], Position[T[x]], T[x], T[y] = x, y, T[y], T[x]
```

3. En diminuant $d[j]$, il est possible que la condition de tas ne soit plus vérifiée entre le nœud contenant j et son éventuel père. Il faudra, si c'est le cas, faire un échange avec ce père, puis continuer à remonter dans l'arbre tant que la structure de tas est contrariée. Dans l'exemple proposé, nous partons de l'arbre :



puis nous échangeons les nœuds 8 et 4 (car $3 < 9$), puis les nœuds 4 et 2 (car $3 < 7$) et la structure est rétablie car $3 \geqslant 2$:

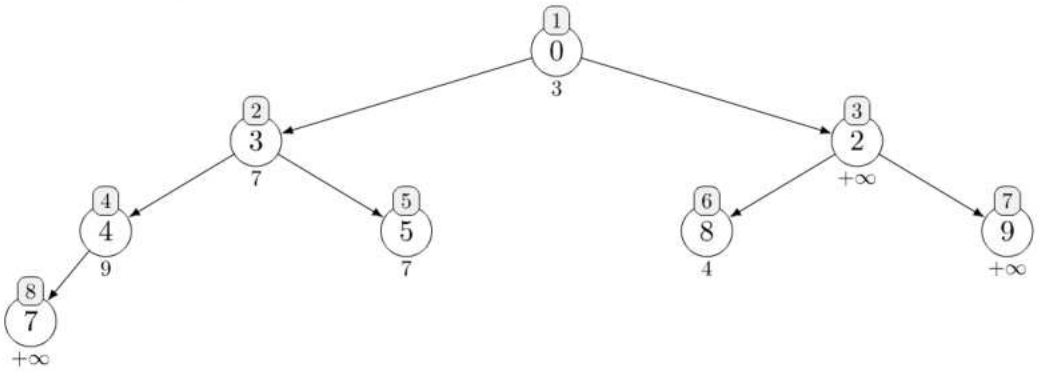


4. On note x le numéro du nœud contenant j et y le numéro de son père : tant que $x > 1$ (c'est-à-dire que x possède un père) et qu'il y a un problème entre les nœuds x et y, on échange x et y, et on remonte d'un niveau dans l'arbre :

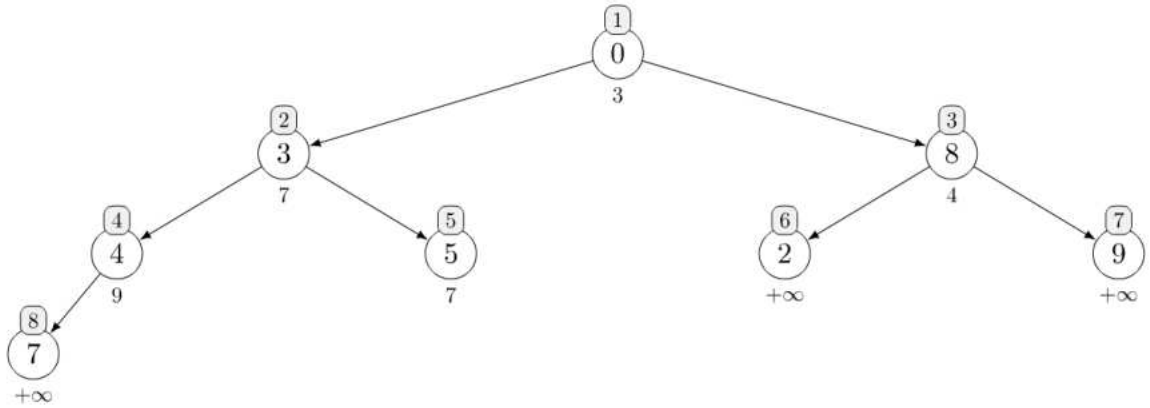
```
def remettre_en_forme(T, Position, d, j):
    x = Position[j]
    y = x // 2
    while x > 1 and d[T[x]] < d[T[y]]:
        echange(T, Position, x, y)
        x, y = y, y // 2
```

Le temps de calcul dans le pire des cas est de l'ordre de la hauteur de l'arbre, c'est-à-dire de l'ordre de $\ln(k)$.

5. Comme $d[0] < d[3] < d[2]$, nous allons échanger les contenus des nœuds 1 et 3 (ce qui échangera leurs contenus 2 et 0) pour supprimer le problème qui existe à la racine et obtenir :



Il y a maintenant un problème au niveau du nœud 3, et nous allons échanger les contenus des nœuds 3 et 6 pour achever la remise en forme du tas :



6. On obtient directement :

```
def Match_a_Trois(a, b, c):
    if a <= b and a <= c:
        return 0
    elif b <= c:
        return 1
    else:
        return 2
```

7. Le seul problème est la descente de l'élément dans l'arbre. Nous utilisons pour cela la fonction auxiliaire récursive `descendre` qui prend en paramètre un nœud x, qui est supposé être le seul nœud du tas où la propriété de tas peut être bafouée. Nous noterons i, j_1, j_2 les contenus du nœud x, de son fils gauche (éventuel) $2x$ et de son fils droit (éventuel) $2x+1$. Seuls deux cas nous obligent à modifier le tas :

- x possède un fils unique (i.e. $2x = T[0]$) et $d[i] > d[j_1]$; il suffit alors d'échanger les nœuds x et $2x$;
- x possède deux fils et $d[i]$ n'est pas plus petit que $d[j_1]$ et que $d[j_2]$: ceci se produit quand le résultat m renvoyé par la fonction `Match_a_Trois` appliqué à $(d[i], d[j_1], d[j_2])$ est différent de -1 ; il faut alors échanger les nœuds x et $2x+m$ et appliquer récursivement la fonction `descendre` au nœud $2x+m$.

```
def extraire(T, Position, d):
    echange(T, Position, 1, T[0])
    T[0] -= 1  # on supprime le dernier noeud

    def descendre(x):
        if 2 * x == T[0] and d[T[x]] > d[T[0]]:
            # premier cas: x a un seul fils (qui est le dernier noeud du tas)
            # et il y a un problème de structure
            echange(T, Position, x, T[0])
        elif 2 * x < T[0]:
            # deuxième cas: x a deux fils
            m = Match_a_Trois(d[T[x]], d[T[2 * x]], d[T[2 * x + 1]])
            if m != -1:
                # et il y a un problème de structure
                echange(T, Position, x, 2 * x + m)
                descendre(2 * x + m)
    descendre(1)
    return T[T[0] + 1]  # ne pas oublier que l'on a diminué la taille du tas
```

8. Au début du calcul, le tas contient les n sommets et la seule contrainte est de placer I en première position du tas, puisque l'on a $d[I] = 0$ et $d[i] = +\infty$ pour tous les autres sommets. On peut donc créer les listes :

$$T = [n, 0, 1, \ldots, n-1],\ Position = [1, 2, \ldots, n-1],\ d = [+\infty, \ldots, +\infty] \text{ et } \pi = [-2, \ldots, -2]$$

puis modifier $\pi[I]$ et $d[I]$ et échanger les nœuds 1 et $I+1$.

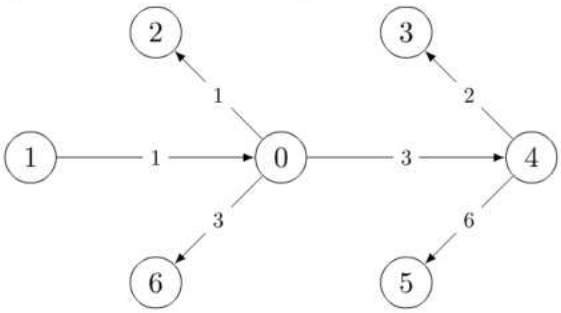
9. Il ne reste qu'à recoller les morceaux :

```
def Dijkstra(L, I):
    n = len(L)
    T, Position = [i - 1 for i in range(n + 1)], [i + 1 for i in range(n)]
    d, Pi = [Inf for i in range(n)], [-2 for i in range(n)]
    T[0], d[I], Pi[I] = n, 0, -1
    echange(T, Position, 1, I + 1)
    while(T[0] != 0):
        i = extraire(T, Position, d)
        if d[i] == Inf:
            return d, Pi  # Plus aucun sommet n'est accessible depuis I
        else:
            for j, delta in L[i]:
                if d[i] + delta < d[j]:  # on a trouvé un chemin plus court
                    d[j] = d[i] + delta  # pour arriver en j
                    Pi[j] = i  # en venant de i
                    remettre_en_forme(T, Position, d, j)
    return d, Pi
```

Le tout premier exemple du TP donne :

```
>>>  L = [[(2, 1), (6, 3), (4, 3)], [(0, 1), (2, 3), (6, 6)], [(1, 2), (3, 5)], [
    (0, 8)], [(3, 2), (5, 6)], [(0, 3), (4, 5)], [(5, 8)]]
>>> Dijkstra(L, 1)
([1, 0, 2, 6, 4, 10, 4], [1, -1, 0, 4, 0, 4, 0])
```

d'où une arborescence de plus courts chemins depuis le sommet 1 :



10. Nous appliquons l'algorithme de Dijkstra depuis le sommet I : si $\pi[J] = -2$, il n'existe pas de chemin de I à J ; sinon, on construit la liste cherchée en insérant dans une liste initialement vide les sommets $i_p = I$, $i_{p-1} = \pi(i_p)$, $i_{p-2} = \pi(i_{p-1})$, et ainsi de suite jusqu'à ce que le père soit égal à -1 :

```
def Plus_court_chemin(L, I, J):
    d, Pi = Dijkstra(L, I)
    if Pi[J] == -2:  # on ne peut atteindre J depuis I
        return []
    else:
        L = [J]
        i = Pi[J]  # i est le père du dernier sommet ajouté au chemin
        while(i != -1):  # le calcul n'est pas terminé
            L.append(i)  # on ajoute i au chemin
            i = Pi[i]  # et on remonte vers le père de i
        return L
```

Il est bien sûr possible d'accélérer un peu le calcul (quand il existe un chemin de I à J) en reprenant le code de la fonction `Dijkstra` et en arrêtant le calcul dès que J sort du tas.

11. Pour chaque couple (i, j) avec $0 \leqslant i < j \leqslant n-1$, on ajoute des arêtes de même poids aléatoire a entre i et j et entre j et i. Pour un tel graphe de taille 5000, il faut environ 11 secondes pour calculer un plus court chemin. On peut vérifier que le temps de calcul (mesuré avec la fonction `time` du module `time`) semble être de l'ordre de n^2 :

```
import numpy.random as rd

def Graphe_alea(n):
    L = [[] for i in range(n)]
    for i in range(n - 1):
        for j in range(i + 1, n):
            a = rd.random()
            L[i].append((j, a))
            L[j].append((i, a))
    return L
```

```
>>> L = []
>>> for n in [100, 200, 400, 700, 1000, 1500, 2000, 2500]:
    G = Graphe_alea(n)
    t = time.time()
    Plus_court_chemin(G, 0, 1)
    L.append(round((time.time() - t) / (n**2), 11)
>>> L
[5.3191e-07, 4.5015e-07, 4.3908e-07, 4.2416e-07,
4.3659e-07, 4.0021e-07, 3.8904e-07, 4.141e-07]
```

Corrigé TP 9.2

Un corrigé complet prendrait trop de place dans cet ouvrage (voir le corrigé complet sur la page dédiée à cet ouvrage sur le site de Dunod). Voici quelques codes, pour les questions les plus difficiles :

0.

```
def affichegrille(grille, joueur=None):
    ''' affiche la grille de puissance 4
    '''
    dico = {V: ' ', J: 'X', R: 'O'}

    lignes, cols = grille.shape
    s = []
    for i in range(lignes):
        s.append(' | '.join([dico[x] for x in grille[i, :]]))
    s.append(' | '.join(['-']*cols))
    s.append(' | '.join([str(a) for a in range(1, cols+1)]))
    s = ['| ' + x + ' |' for x in s]
    L = [str(a) for a in range(1, lignes+1)] + [' ', ' ']

    s = [i + ' ' + x for i, x in zip(L, s)]
    s.reverse()
    s = '\n'.join(s)

    if joueur == J:
        s += "\nC'est au joueur Jaune de jouer (J = X)"
    elif joueur == R:
        s += "\nC'est au joueur Rouge de jouer (R = O)"
    else:
        pass  # s += '\nCas spécial'
    print(s)
```

1.

```
def colpleine(grille, col):
    return not (grille[:, col] == V).any()  # True si pas de V
```

2.

```
def listeind(lignes, cols):
    A = np.arange(lignes * cols, dtype=int).reshape((lignes, cols))
    L = []
    # horizontaux
    for l in A:
        ajouteind(l)
    # verticaux
    for l in np.transpose(A):
        ajouteind(l)
    # diagonaux
    for d in range(4-lignes, cols-3):
        ajouteind(A.diagonal(d))
    # diagonaux inverses
    Ainv = A[:, ::-1]
    for d in range(4-lignes, cols-3):
        ajouteind(Ainv.diagonal(d))
    return L
```

3.

```
def testfin(grille):
    ''' teste si un joueur a gagné et renvoie le numéro du joueur, 0 sinon
    bonus: renvoie -1 si toutes les colonnes sont pleines
    '''
    T = creepaquets(grille)
    for l in T:
        if (l == J).all():  # façon élégante d'écrire == [J, J, J, J]
            return J
        if (l == R).all():
            return R
```

```
    # petit rajout: si c'est fini, grille pleine
    if not (grille == V).any():
        return -1
    return 0  # pas de joueur gagnant
```

8.

```
def minmax(G, profondeur):
    ''' g = (grille, joueur)
    algorithme minmax
    '''
    grille, joueur = G
    test = testfin(grille)
    if test!= 0 or profondeur <= 0:  # si on a une feuille ou bien on a atteint la profondeur limite
        return evaluegrille(grille, joueur), G

    L = coupsuivants(grille, joueur)
    meilleurscore, meilleurcoup = minmax(L[0], profondeur - 1) # améliorable...

    if joueur == J:
        for g, j in L[1:]:
            score, _ = minmax((g, j), profondeur - 1)
            if score >= meilleurscore:
                meilleurscore = score
                meilleurcoup = (g, j)
            if meilleurscore == INFINI-1:
                break
    else:
        for g, j in L[1:]:
            score, _ = minmax((g, j), profondeur - 1)
            if score <= meilleurscore:
                meilleurscore = score
                meilleurcoup = (g, j)
            if meilleurscore == - (INFINI-1):
                break

    return meilleurscore, meilleurcoup
```

10.

```
def alphabeta(G, alpha, beta, profondeur):
    ''' G = (grille, joueur)
    algorithme minmax, version alpha-beta
    amélioration possible: compter la profondeur et privilégier le
    coup de profondeur minimale (gagner au plus vite)
    rappel: renvoie la valeur exacte si on est dans ]alpha, beta[
    (avec ici, en prime, une grille correspondante)
    une approx inférieure si >= beta
    une approx supérieure si <= alpha
    '''
    grille, joueur = G
    test = testfin(grille)
    if test!= 0 or profondeur <= 0:  # si on a une feuille ou que
        # la profondeur est atteinte
        return evaluegrille(grille, joueur), G

    L = coupsuivants(grille, joueur)
    meilleurcoup = L[0]  # par défaut on prend le premier choix...
    if joueur == J:   # max à prendre
        for g, j in L:
            score, _ = alphabeta((g, j), alpha, beta, profondeur - 1)
            if score >= beta:  # on a une approx. inférieure
                # donc on s'arrête, ça vaut au moins score mais on ne
                # distingue pas les profondeurs atteintes...
                alpha = score
                meilleurcoup = (g, j)
```

```
                return alpha, meilleurcoup
            if score > alpha:
                alpha = score  # c'est là que l'on gagne en comlexité et que l'on coupe des branches !
                meilleurcoup = (g, j)

        return alpha, meilleurcoup
    else:    # min à prendre
        for g, j in L:
            score, _ = alphabeta((g, j), alpha, beta, profondeur - 1)
            if score <= alpha:
                beta = score
                meilleurcoup = (g, j)
                return beta, meilleurcoup
            if score < beta:
                beta = score
                meilleurcoup = (g, j)

        return beta, meilleurcoup
```

12.

```
def alphabeta2(G, alpha, beta, profondeur):
    grille, joueur = G
    test = testfin(grille)
    if test!= 0 or profondeur <= 0:     # si on a une feuille ou que
        # la profondeur est atteinte
        return evaluegrille(grille, joueur), G, profondeur

    L = coupsuivants(grille, joueur)
    meilleurcoup = L[0]  # par défaut on prend le premier choix...
    meilleurprof = - INFINI
    if joueur == J:   # max à prendre
        for g, j in L:
            score, _, prof = alphabeta2((g, j), alpha, beta, profondeur - 1)
            if score >= beta:
                alpha = score
                meilleurcoup = (g, j)
                meilleurprof = prof
                return alpha, meilleurcoup, meilleurprof
            if score > alpha:  # on améliore le alpha
                alpha = score
                meilleurcoup = (g, j)
                meilleurprof = prof
            elif (score == alpha and meilleurprof < prof):  # on cherche la profondeur maximale = coup
                # le plus proche
                alpha = score
                meilleurcoup = (g, j)
                meilleurprof = prof
        return alpha, meilleurcoup, meilleurprof
    else:    # min à prendre
        for g, j in L:
            score, _, prof = alphabeta2((g, j), alpha, beta, profondeur - 1)
            if score <= alpha:
                beta = score
                meilleurcoup = (g, j)
                meilleurprof = prof
                return beta, meilleurcoup, meilleurprof
            if score < beta:  # on améliore le beta
                beta = score
                meilleurcoup = (g, j)
                meilleurprof = prof
            elif (score == beta and meilleurprof < prof):
                beta = score
                meilleurcoup = (g, j)
                meilleurprof = prof

        return beta, meilleurcoup, meilleurprof
```

Bibliographie

[ALS+07] Aho, Alfred Vaino and Monica S. Lam and Sethi, Ravi and Ullman, Jeffrey David and Deschamp, Philippe and Lorho, Bernard and Sagot, Benoît and Thomasset, François and Aho, Alfred Vaino. *Compilateurs : principes, techniques et outils*. Pearson Education, 2007. Seconde édition.

[ARRa] Arrêté du 17 novembre 2003 portant approbation du règlement technique fixant les conditions d'agrément des machines à voter. NOR : INTX0306924A, JORF n°274 du 27 novembre 2003.

[ARRb] Arrêté du 3 août 2016 modifiant l'arrêté du 16 décembre 2011 relatif à l'application de l'article R. 111-14 du code de la construction et de l'habitation. NOR : LHAL1519497A, JORF n°0183 du 7 août 2016.

[Ben16] Bentley, Jon. *Programming pearls*. Addison-Wesley Professional, 2016.

[BFP+73] Blum, Manuel and Floyd, Robert W and Pratt, Vaughan and Rivest, Ronald L and Tarjan, Robert E. Time bounds for selection. *Journal of computer and system sciences*, 7(4) : 448–461, 1973.

[Car07] Carrasco, Valérie. Le pacte civil de solidarité : une forme d'union qui se banalise. *Infostat Justice*, 97(4), 2007.

[EEA07] Ehrenfest, Paul and Ehrenfest-Afanassjewa, Tatjana. *Über zwei bekannte Einwände gegen das Boltzmannsche H-Theorem*. Hirzel, 1907.

[GAO92] Patriot Missile Defense : Software Problem Led to System Failure at Dhahran, Saudi Arabia. Technical Report IMTEC-92-26, GAO U.S. Government Accountability Office, 1992.

[GRD+04] Gumel, Abba B and Ruan, Shigui and Day, Troy and Watmough, James and Brauer, Fred and Van den Driessche, P and Gabrielson, Dave and Bowman, Chris and Alexander, Murray E and Ardal, Sten and others. Modelling strategies for controlling SARS outbreaks. *Proceedings of the Royal Society of London B : Biological Sciences*, 271(1554) : 2223–2232, 2004.

[Gr 11] Grémy, Jean-Paul. Les intentions de vote recueillies par les sondages avant le premier tour des élections présidentielles de 2002. (halshs-00576045), March 2011.

[HLS03] Hyman, James M and Li, Jia and Stanley, E Ann. Modeling the impact of random screening and contact tracing in reducing the spread of HIV. *Mathematical biosciences*, 181(1) : 17–54, 2003.

[Hoa62] Hoare, Charles AR. Quicksort. *The Computer Journal*, 5(1) : 10–16, 1962.

[IDQ10] ID quantique white paper, random number generation using quantum physics. Technical report, IDQ, 2010.

[IEC] Quantities and units – part 13 : Information science and technology. *IEC 80000-13 :2008*.

[IEE] IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*.

[INT12] *Intel® Digital Random Number Generator Generator (DRNG)*. 2012. Revision 1.1.

[INT16] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 2016. Order Number : 248966-035.

[Kac47] Kac, Mark. Random walk and the theory of brownian motion. *The American Mathematical Monthly*, 54(7) : 369–391, 1947.

[KC05] Karine Chemla, Guo Shuchun. *Les neuf chapitres, Le classique mathématique de la Chine ancienne et ses commentaires*. Dunod, 2005.

[Knu13] Knuth, Donald. *The Art of Computing, Vol. II : Seminumerical Algorithms, Third edition*. Addison-Wesley, 2013.

[KZVH00] Kribs-Zaleta, Christopher M and Velasco-Hernandez, Jorge X. A simple vaccination model with multiple endemic states. *Mathematical biosciences*, 164(2) : 183–201, 2000.

[LM02] Li, Jianquan and Ma, Zhien. Qualitative analyses of SIS epidemic model with vaccination and varying total population size. *Mathematical and Computer Modelling*, 35(11) : 1235–1243, 2002.

[NF 98] NF EN ISO 11562. Geometrical Product Specifications (GPS) – Surface texture : Profile method – Metrological characteristics of phase correct filters, 1998.

[NFC] Norme NF C15-100.

[NW70] Needleman, Saul B and Wunsch, Christian D. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3) : 443–453, 1970.

[Rag82] Raggett, Graham F. Modelling the Eyam plague. *Bull. Inst. Math. and its Applic*, 18(221-226) : 530, 1982.

[San09] Santoro, Renaud. *Vers des générateurs de nombres aléatoires uniformes et gaussiens à très haut débit.* PhD thesis, Université Rennes 1, 2009.

[Sch06] Schelling, Thomas C. *Micromotives and macrobehavior*. WW Norton & Company, 2006.

[WD16] Whittles, Lilith K. and Didelot, Xavier. Epidemiological analysis of the Eyam plague outbreak of 1665–1666. *Proceedings of the Royal Society of London B : Biological Sciences*, 283(1830), 2016.

Index