Cours d'Informatique pour Tous
2021 –

Jules Svartz
Lycée Masséna

Préambule

Ces notes de cours sont issues du cours d'informatique commune (IPT) subi par les élèves du lycée Masséna des classes de première année MPSI (831), PCSI (833) et de deuxième année MP*, MP, PC*, PC, depuis l'année 2021 en première année et 2022 en deuxième année. Il sera mis à jour régulièrement !

Le cours présenté ici est très détaillé, et tous les points ne sont pas nécessairement abordés en classe : il se veut utile autant pour l'élève qui veut un récapitulatif que pour celui qui souhaite aller plus loin.

Le polycopié se divise en 7 parties, le programme de première année couvrant les quatre premières, le programme de seconde année les autres.

- La première partie est dévolue à l'« initiation ». Elle se subdivise en 4 chapitres :
  - Le chapitre 0 est un chapitre d'introduction à l'informatique, il présente un point de vue historique sur le développement de cette discipline, les principaux éléments constitutifs d'un ordinateur et le rôle du système d'exploitation. Ce chapitre a disparu du nouveau programme, mais il forme une introduction historique importante à la discipline, et sera sûrement traité encore les années à venir. Le cours présenté ici est plutôt présenté en fin d'année, par choix pédagogique : il est en effet plus facile d'expliquer précisément le comportement d'un micro-processeur à des élèves qui savent déja programmer et ont une connaissance du système de numération binaire.
  - Le chapitre 1 présente de manière détaillée les éléments au programme concernant la programmation en Python. Il est en pratique présenté peu à peu en cours et en TP, parallèlement aux chapitres qui suivent.
  - Le chapitre 2 présente la représentation des nombres entiers dans différentes bases, en particulier en binaire. Les algorithmes de changement de base, quoique non au programme, sont un prétexte pour commencer à faire des boucles et utiliser des listes.
  - Le chapitre 3 est normalement au programme au second semestre, mais il me semble intéressant de le traiter tôt dans l'année, en parallèle des travaux pratiques. Il donne les outils pour l'analyse théorique des algorithmes (terminaison / correction / complexité). C'est un chapitre crucial où sont présentés les algorithmes « basiques » au programme de première année : parcours de listes, recherche dichotomique ou de motif dans une chaîne de caractères...
- La deuxième partie (au programme officiel traitée uniquement sous forme de travaux pratiques) présente :
  - au chapitre 4, les techniques gloutonnes en algorithmique. Le chapitre est volontairement court pour laisser des exemples en travaux pratiques.
  - au chapitre 5, la récursivité. On donne quelques éléments de preuve de programmes dans ce contexte.
- La troisième partie donne quelques compléments de Python. Elle se découpe en 3 chapitres :
  - Le chapitre 6 se concentre sur la représentation des nombres. On se restreint maintenant au binaire, et on explique les rudiments de calculs effectués par un processeur, avec des registres de taille fixe. On présente aussi la représentation usuelle des nombres flottants.
  - Le chapitre 7 présente les méthodes pour lire un fichier textuel en Python, ainsi que la création d'un tel fichier. C'est l'occasion de revenir sur les chaînes de caractères !
  - Le chapitre 8 présente les modules usuels en Python, il fera l'objet d'un ou deux TP(s).
- La quatrième partie est l'occasion de mettre en pratique les techniques vues précédemment.
  - On présente d'abord le tri de données, problème essentiel en informatique. On a découpé le traitement des tris en deux parties, chapitre 9 pour les tris quadratiques, ainsi que le tri par comptage, chapitre 10 pour les tris efficaces (tri fusion, tri rapide).
  - Un bref chapitre 11 présente la notion de jeu de tests pour un programme.
  - Enfin, on étudie les bases des graphes dans le chapitre 12. L'accent est porté sur le parcours de graphes, le programme ayant en ligne de mire l'algorithme de Dijkstra et sa variante A^*.

— La partie 5 présente les bases de SQL et de l'algèbre relationnelle.
  — Le chapitre 13 donne le vocabulaire et les premières requêtes SQL, en ce concentrant sur une seule base ;
  — Le chapitre 14 introduit les liens entre relations.
— La partie 6 étend les techniques gloutonnes vues en premières années pour résoudre des problèmes par programmation dynamique.
  — Le chapitre 15 donne une implémentation possible en Python de la structure de dictionnaires (avec des listes Python), via des fonctions de hachage. On explique en particulier comment résoudre les collisions, et comment une structure dynamique permet de justifier la complexité $O(1)$ que l'on attribue aux opérations de dictionnaire en Python.
  — Le chapitre 16 présente les techniques de programmation dynamique proprement dites.
— Enfin, la septième partie donne une introduction à l'apprentissage (supervisé ou non), ainsi qu'à la théorie des jeux.

Table des matières

Première partie

Initiation

Chapitre 0

Ordinateurs, Systèmes d'exploitation et Python

0.1 Qu'est ce qu'un ordinateur ?

Si on tente rapidement de définir ce qu'est un ordinateur au sens large du terme (y compris tablettes, smartphones...), on peut lister les éléments communs suivants :

— un ordinateur reçoit des informations par l'intermédiaire d'un utilisateur ou d'un réseau ;

— un ordinateur émet des informations via le réseau ou un de ses périphériques ;

— un ordinateur a besoin d'une source d'énergie pour fonctionner.

Néanmoins cette première tentative s'avère infructueuse, on peut penser à plusieurs contre-exemples qui satisfont ces trois critères et qui ne sont pas pour autant des ordinateurs :

— un réfrigérateur nécessite une source d'énergie, il reçoit des informations de la part de capteurs (température...), il en émet sous la forme de signaux lumineux électriques ;

— un interrupteur fonctionnant avec la luminosité ambiante reçoit de l'information par le biais de son capteur et transmet de l'information (ouvert-fermé), de plus le capteur peut nécessiter une source d'énergie pour fonctionner ;

— le système ABS d'aide au freinage d'urgence d'une voiture reçoit également de l'information (vitesse...) et en emet sous forme de pression hydraulique sur le système de freinage ;

— plus généralement, tout système électronique embarqué (système électronique et informatique autonome, ayant une tâche précise) vérifie ces trois critères. Mais l'utilisateur ne peut *a priori* pas détourner le système pour lui faire exécuter une autre tâche.

Tout cela montre que la définition de ce qu'est un ordinateur n'est pas une chose si triviale que cela. Tout ceci est lié à une des grandes problématiques du siècle dernier : qu'est ce qu'un calcul ?

0.1.1 Turing et ses machines

Dans les années trente, quatre mathématiciens au moins cherchent à répondre à cette question : Kleene, Church, Turing et Post. Les questions posées commencent à recevoir une réponse et c'est la naissance de la calculabilité, qui vise à définir précisément ce qui est effectivement calculable. Alan Turing explore une autre approche originale, qui va lier la notion de « calcul » à la notion de « machine ». Il imagine une machine qui peut fonctionner sans intervention humaine.

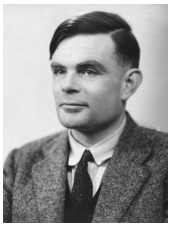


FIGURE 1 – Alan Turing

Cette machine possède les caractéristiques suivantes :

— un ruban : aussi long que nécessaire, sur lequel la machine peut lire des données et en écrire d'autres, le ruban est divisé en cases (voir figure 2) ;

— une tête de lecture, à tout moment positionnée au niveau d'une case du ruban ;

— un ensemble fini d'états : quand la machine lit un symbole, elle réagit en fonction de son état actuel et du symbole lu, en changeant d'état, en modifiant le symbole sur le ruban et en déplaçant la tête de lecture d'un cran sur la droite ou sur la gauche (elle n'est pas forcée d'effectuer toutes ces actions).

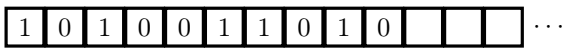


FIGURE 2 – Un ruban d'une machine de Turing

Donnons une brève description d'une machine de Turing testant si un nombre naturel n écrit en binaire est divisible par 3. On suppose que sur le ruban d'entrée (unidirectionnel, infini vers la droite, comme en figure 2) se trouve le mot écrit en binaire de gauche à droite, les bits de poids forts étant au début. Les autres cases du ruban sont vides. La machine doit écrire à la suite du nombre (à la première case vide), le symbole « V » ou « F » (pour vrai ou faux) suivant si le nombre est divisible par 3 ou non. On peut imaginer une machine à 4 états résolvant le problème :

— 3 états correspondant à la congruence modulo 3 de la portion du nombre lue jusqu'ici. Notons les $0, 1, 2$.

— Un état supplémentaire (final) indiquant que le calcul est fini.

Pour compléter la description de notre machine de Turing, il faut donner la *fonction de transition* entre les états, c'est à dire la façon dont la machine réagit suivant son état et le symbole lu. Le tableau suivant donne la congruence modulo 3 de $2a$ et $2a+1$ en fonction de celle de a :

a	0	1	2
$2a$	0	2	1
$2a+1$	1	0	2

Cette table nous donne l'essentiel de la fonction de transition : si la portion des bits lue jusque-là représente a en binaire, lire un 0 mène à la représentation de $2a$, et lire un 1 mène à celle de $2a+1$. Si l'état courant représente la congruence modulo 2 du nombre obtenu en gardant seulement les k premiers bits du ruban, on sait dans quel état passer à la lecture du $(k+1)$-ème. Dans tous les cas, on déplace la tête de lecture d'un cran vers la droite. Le lecteur vérifiera qu'en commençant à l'état 0, la lecture successive des bits du ruban de la figure 2 fait passer successivement par les états $1, 2, 2, 1, 2, 2, 2, 1, 0, 0$. Lorsqu'on atteint la première case vide, il suffit de remplacer le symbole « Vide » par « V » ou « F » suivant si l'on se trouve dans l'état 0 ou non, et de passer en état final. Ici, on écrirait « V » car on est dans l'état 0. Le nombre écrit sur le ruban est en fait 666 en binaire, qui est bien divisible par 3.

Dans l'exemple précédent, on a fait essentiellement de la lecture, mais on peut aussi calculer : pour additionner 1 au nombre naturel écrit en binaire sur le ruban (toujours avec la convention que les bits de poids forts sont au début), il suffit de se déplacer jusqu'à la fin du mot, (caractérisé par une case vide), revenir d'un cran à gauche, et de remplacer ensuite les 1 par des 0 en se déplaçant vers la gauche, jusqu'à retomber sur un 0 ou une case vide, qu'on transforme en 1, voir figure 3.

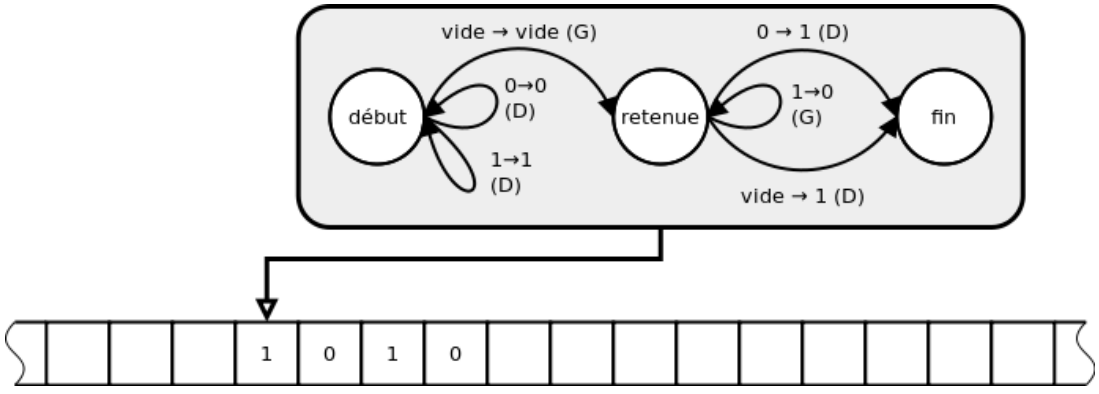


FIGURE 3 – Une machine de Turing permettant d'additionner 1 au nombre écrit sur le ruban

Attention, la « machine de Turing » reste un objet théorique. Cette machine est idéalisée car le ruban est supposé toujours suffisamment long pour permettre le calcul (il est donc virtuellement infini, même si une machine exécutant un calcul qui termine n'utilisera qu'une partie finie du ruban). Avec ce formalisme, un algorithme est simplement une machine de Turing particulière [1].

1. Voir par exemple la page Wikipédia pour une présentation plus complète : `https://fr.wikipedia.org/wiki/Machine_de_Turing`

Une machine de Turing est essentiellement décrite par ses états possibles et sa fonction de transition. Comme le nombre d'états est fini, la description d'une machine de Turing est elle aussi finie, et peut elle même être encodée en binaire (ou avec un alphabet fini quelconque), et écrite sur un ruban. Turing montre qu'il existe (mathématiquement) une machine de Turing *universelle* : elle est capable de prendre sur un ruban la description de *n'importe quelle machine de Turing* et de simuler son exécution sur toute entrée placée à la suite sur le ruban. Un ordinateur est donc la réalisation concrète d'une telle machine de Turing universelle : il est capable d'exécuter un algorithme pour peu qu'il soit traduit dans un langage de programmation idoine.

Nos ordinateurs sont en fait un peu plus complexes afin d'être plus efficaces : se déplacer de case en case étant source d'inefficacité, il vaut mieux permettre de sauter d'une case à une autre case grâce à une adresse (et donc numéroter les cases du ruban). Actuellement, on parle plutôt de mémoire que de ruban, mais du point de vue de la *calculabilité* cela ne change rien.

0.1.2 Prémices aux ordinateurs et modèle de Von Neumann

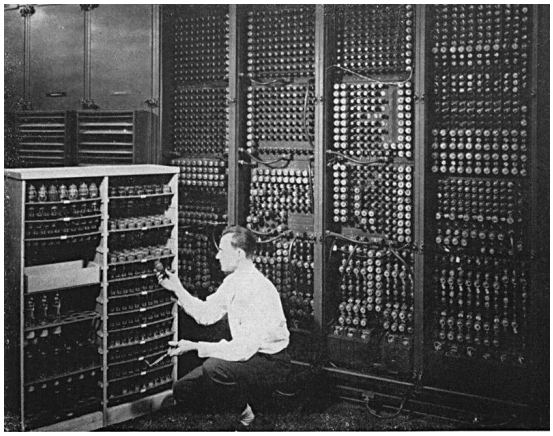


FIGURE 4 – L'ENIAC

La première réalisation concrète d'une machine de Turing date de 1941, c'est le Z3 allemand, qui est électromécanique. La première réalisation entièrement électronique est l'ENIAC, qui date de 1943. L'ensemble fut conçu par John William Mauchly et John Eckert et ne fut pleinement opérationnel qu'en 1946. Von Neumann intégra l'équipe en 1944 et publia un rapport sur la conception de l'EDVAC (un autre ordinateur électronique) en 1945.

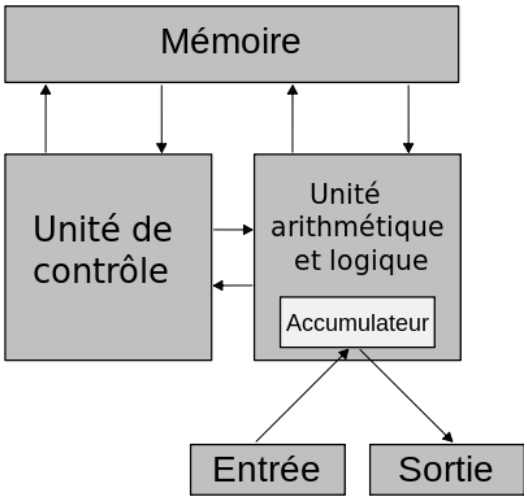


FIGURE 5 – Architecture de Von Neumann

Ce rapport décrit un schéma d'architecture de calculateur, organisé en trois éléments (unité arithmétique, unité de commande et mémoire contenant programme et données). Il décrit aussi des principes de réalisation pour ces éléments, notamment les opérations arithmétiques. Si ce dernier aspect dépend partiellement de la technologie de l'époque (et a

donc vieilli), le modèle d'architecture, qui marque une transition profonde avec les pratiques antérieures, reste d'une étonnante actualité. Ce modèle, auquel reste attaché le nom de Von Neumann, est représenté par le schéma de la figure 5. Il y a schématiquement quatre composants principaux :

— le processeur : qui se décompose en une unité de commande et une unité arithmétique et logique ;
— la mémoire : qui contient des instructions et des données ;
— les périphériques d'entrée-sortie : qui permettent une communication entre l'utilisateur et les machines (via clavier, souris, écran,...) ;
— le bus : qui est le canal de communication entre la mémoire, le processeur et les périphériques.

La première innovation est la séparation nette entre l'unité de commande, qui est chargée d'organiser le flot des instructions, et l'unité arithmétique, chargée de l'exécution proprement dite de ces instructions. La seconde innovation est l'idée du programme enregistré : fini les rubans, cartes à trous... Les instructions et les données sont maintenant enregistrées dans la mémoire [2] selon un codage conventionnel. Un compteur ordinal ou pointeur d'instruction (*program counter* en anglais), contient l'adresse de l'instruction en cours d'exécution ; il est automatiquement incrémenté après exécution de l'instruction, et explicitement modifié par les instructions de branchement (*if, goto, jump...*).

0.1.3 Le rôle de chaque élément.

Le processeur. Le processeur (CPU) est le cerveau de l'ordinateur, il donne des ordres aux périphériques et à la mémoire et est responsable de l'exécution du programme de l'ordinateur. Le processeur dispose d'une toute petite mémoire, typiquement de l'ordre de quelques dizaines ou centaines de mots mémoire (groupes de 64 bits), qu'on appelle des registres. La fonction du registre de données est de contenir les données transitant entre l'unité de traitement et l'extérieur. La fonction de l'accumulateur est principalement de contenir les opérandes ou les résultats des opérations de l'unité arithmétique et logique. Son unité arithmétique et logique permet de réaliser les calculs :

— opérations arithmétiques binaires : addition, multiplication, soustraction, division ;
— opérations logiques, conjonction, disjonction et négation.

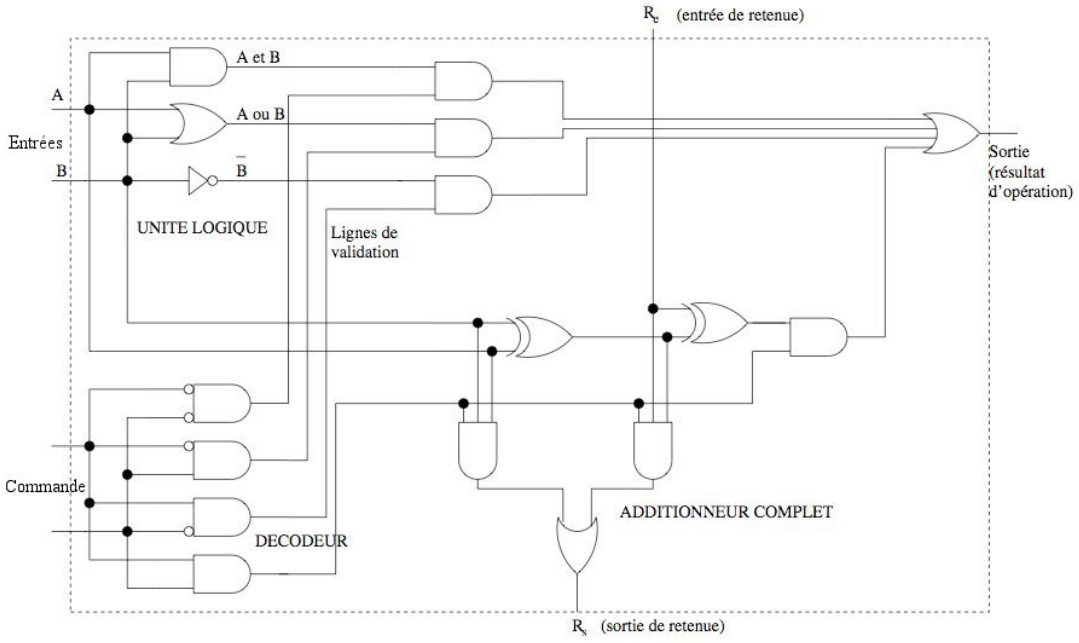


FIGURE 6 – Le circuit (sous forme de portes logiques) d'une unité arithmétique et logique sur des mots de 2 bits

L'unité de contrôle accède à la mémoire via le bus, et peut lire une case mémoire ou y écrire. Cependant cette unité ne contient pas le programme à exécuter : les instructions à exécuter sont codées sous la forme d'une suite de bits stockée en mémoire. Toutes les instructions sont réalisées dans l'ordre, mais les instructions de saut modifient cet ordre.

2. À titre indicatif, la capacité de la mémoire de l'EDVAC était inférieure à 5 ko.

— une instruction de saut conditionnel modifie cet ordre si une certaine condition est vérifiée.

— une instruction de saut inconditionnel modifie cet ordre sans condition.

Les instructions `for` et `while` (qui n'existent pas en langage machine), peuvent être traduites par plusieurs instructions de saut conditionnelles et inconditionnelles.

La mémoire. La mémoire est une suite de chiffes binaires nommés bits, organisés en paquets de huit (les octets) puis en mots mémoire de 64 bits [3]. Un mot en mémoire peut représenter plusieurs choses : une instruction, un entier... La signification du mot dépend de l'utilisation qu'on en fait. La mémoire ne sert qu'à stocker ces mots, elle ne réalise aucune opération et n'effectue aucun calcul. Chaque mot possède une adresse, avec cette adresse on peut lire un mot ou alors écrire un autre mot à la place. Cette adresse est attribuée de manière aléatoire, d'où le nom de *Random access memory (RAM).*

Les périphériques. De manière formelle il s'agit de mémoire supplémentaire dans laquelle le processeur peut écrire pour donner des ordres au périphérique (afficher telle couleur sur tel pixel de l'écran) ou lire (réagir à telle touche tapée sur un clavier).

0.1.4 Avantages et inconvénients

L'architecture de Von Neumann permet une grande souplesse car on peut stocker virtuellement tout type de structure en mémoire, notamment des données ou des programmes. Par contre cette architecture possède trois inconvénients :

— exécution monotâche : les instructions sont exécutées de manière séquentielle : une machine de Von Neumann ne permet donc de ne faire qu'une seule chose à la fois ;

— le bus : ces dernières années, la vitesse des processeurs a considérablement augmenté. Ils sont actuellement si rapides que le processeur passe beaucoup de temps à attendre que les données précédentes soit transférées avant d'en envoyer de nouvelles ;

— faible robustesse : les données et les programmes étant stockés dans la même mémoire, si un bug [4] d'un programme provoque une écriture à un endroit non désiré, cela peut compromettre le fonctionnement de l'ensemble de la machine.

0.1.5 De nos jours

L'architecture de Von Neumann a peu évolué depuis sa création. Néanmoins il existe quelques petites différences. Outre la mémoire RAM, d'autres mémoires sont utilisées :

— la mémoire morte : la RAM nécessite un apport constant d'énergie pour fonctionner, une simple coupure de courant peut mettre en péril tous les calculs et programmes réalisés ; on sait de nos jours construire des mémoires non volatiles permettant un accès en lecture mais pas en écriture (*Read Only Memory ou ROM*) ; elles sont utilisées pour stocker un programme particulier servant au démarrage de la machine (*firmware ou BIOS*) ; cette mémoire ne permet pas de stocker les données utilisateurs ;

— la mémoire de masse : pour stocker ces données utilisateurs on utilise de nos jours des disques durs (pour les ordinateurs) ou une mémoire flash (pour les smartphones).

De plus, certains périphériques peuvent accéder directement à la mémoire sans passer par le processeur, on parle alors de *Direct Memory Access* ou *DMA*, et certains calculs d'affichage sont laissés à un processeur spécialisé possédant une mémoire vive importante présent sur la carte graphique. Les ordinateurs comportent maintenant des processeurs multiples, qu'il s'agisse d'unités séparées ou de « cœurs »multiples à l'intérieur d'une même puce. Cela permet d'obtenir une puissance de calcul plus élevée sans augmenter la puissance d'un processeur individuel qui est limitée par les capacités d'évacuation de la chaleur dans des circuits de plus en plus denses.

0.1.6 Ordres de grandeur

Il est important de connaître grossièrement ce qu'il est possible de calculer et de stocker sur un ordinateur : une fois que l'on a estimé la complexité en temps comme en mémoire d'un algorithme, il est possible d'avoir une idée des entrées sur lesquelles il va pouvoir s'exécuter sans problème, ou au contraire s'il mettra trop de temps ou manquera de mémoire.

3. Ou 32, sur les vieilles machines !

4. En français on dit « bogue ». Mais c'est moche.

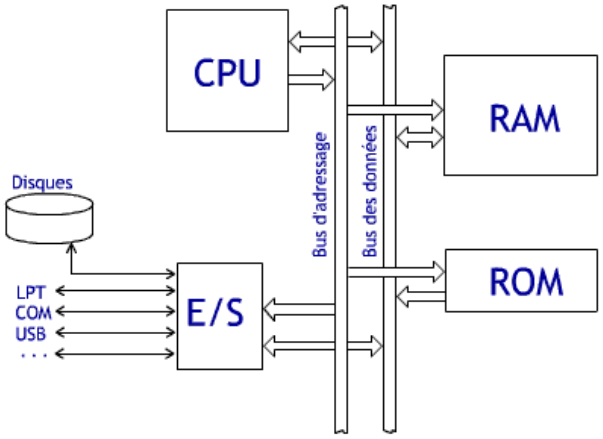


FIGURE 7 – Architecture réelle

Fréquence du micro-processeur. Aujourd'hui, les processeurs ont une fréquence de l'ordre de quelques GHz. On peut garder comme ordre de grandeur qu'un processeur est capable de réaliser environ 10^9 opérations élémentaires par seconde. Pour estimer le temps de calcul d'un programme écrit dans un langage de programmation de haut niveau (comme Python), il est nécessaire de réduire ce nombre : on peut considérer qu'en pratique 10^7 opérations « élémentaires » (opérations arithmétiques sur des petits entiers, modification ou accès à un élément d'une liste...) sont réalisables par seconde.

Mémoire vive. La mémoire vive (RAM) est en quantité de l'ordre du Go (giga-octet) dans un ordinateur actuel. Si, pour réaliser un calcul, un programme nécessite plus de ressources mémoire que la quantité de RAM disponible, on assiste à un phénomène de *« swap »* : on est obligé d'utiliser également le disque dur, d'accès considérablement plus long.

Mémoire de masse. La mémoire de masse est celle du disque dur : elle est peu rapide mais permet un grand stockage de données. La capacité d'un disque dur grand public est de l'ordre du To (tera-octet).

Exemple. Considérons le problème de résolution d'un système linéaire (à coefficients flottants codés sur 64 bits) sur un ordinateur standard. L'algorithme du pivot de Gauss est en $O(n^3)$ pour la résolution, le stockage de la matrice nécessite une mémoire de taille $O(n^2)$ (comme on stocke des flottants 64 bits, il faut compter en fait 8 octets par coefficients). Ainsi :

— On va dépasser la taille de la RAM pour des matrices de taille $n \times n$ avec n de l'ordre de 30000 (on a pris ici une RAM d'un giga-octet).

— L'algorithme du pivot de Gauss, codé en Python, exécuté sur une matrice de taille 10000×10000, mettra plusieurs heures pour s'exécuter (en comptant 10^7 opérations « Python » par seconde et une complexité exactement n^3, l'estimation donne 28 heures).

0.2 Le système d'exploitation

Nous avons vu jusqu'à présent le fonctionnement du matériel constituant l'ordinateur. En particulier, un ordinateur permet l'exécution d'un programme stocké dans la mémoire de masse de l'ordinateur. Or, un utilisateur utilise souvent plusieurs programmes à la fois : il peut par exemple travailler sur Python tout en écoutant de la musique et en téléchargeant une série.

C'est le rôle d'un programme particulier de gérer les programmes que l'on veut utiliser et le stockage des données. Ce programme est chargé en mémoire au démarrage de l'ordinateur et y reste jusqu'à son extinction : c'est le système d'exploitation. Il a plusieurs rôles :

— donner l'impression que l'ordinateur est multitâche ;

— identifier les utilisateurs ;

— gérer le disque dur ;

— contrôler l'accès aux données stockées.

Il existe actuellement beaucoup de systèmes d'exploitations différents mais tous sont issus d'une des deux grandes familles de systèmes d'exploitation :

— Microsoft Windows : en situation de quasi-monopole sur les ordinateurs individuels ;
— Unix : famille dont sont issus Mac Os X, GNU Linux, FreeBSD, Android et qui sont en situation majoritaire sur les serveurs, les smartphones et les super-calculateurs.

0.2.1 Le multitâche

Nous avons vu précédemment qu'un des défauts de l'architecture de Von Neumann est que la machine ainsi réalisée est monotâche. Le système d'exploitation permet de s'affranchir en apparence de cette limite et d'avoir ainsi plusieurs programmes qui s'exécutent en même temps. Pour cela le système d'exploitation stocke en mémoire les différentes instructions à exécuter.

Il lance une première instruction et dès qu'une entrée-sortie se produit ou qu'un certain temps par défaut (de l'ordre de 100 ms) s'est écoulé, le système d'exploitation lance une autre instruction.

Imaginons qu'un utilisateur code en Python tout en écoutant de la musique. Le système d'exploitation commence par exécuter le programme de lecture audio et envoie quelques secondes de son sur le périphérique dédié. Le temps que ces quelques secondes soient passées, le système d'exploitation se met en attente. Au cours de cette attente une lettre est tapée au clavier, le système d'exploitation exécute alors le programme de développement Python (Spyder par exemple) et une lettre est affichée à l'écran. Le système d'exploitation repasse alors en attente, puis le périphérique son indique que les quelques secondes de musique ont été jouées. Le système d'exploitation exécute de nouveau le programme de lecture audio...

0.2.2 Identification des utilisateurs

Tous les systèmes d'exploitation sont multi-utilisateurs : chaque utilisateur dispose d'un identifiant auprès du système et d'un mot de passe. C'est le cas au lycée par exemple : tous les élèves ont un identifiant et un mot de passe. De plus les utilisateurs sont membres de certains groupes : un élève est par exemple membre de groupes comme sa classe. Il est également le seul membre d'un groupe à son nom. Après avoir correctement rentré nom d'utilisateur et mot de passe, le système d'exploitation lance un ensemble de programmes appelé shell. Il existe encore actuellement deux types de shell :

— shell graphique : sur les ordinateurs personnels, un shell graphique se présente sous forme d'une interface graphique, permettant de lancer les programmes que l'utilisateur veut exécuter, par clic ou double-clic ;
— shell textuel (interprète de commandes) : ce type de shell interactif se présente sous la forme d'une ligne de commande. L'utilisateur tape une commande sous la forme d'une ligne de texte qui est ensuite exécutée, puis le shell rend la main à l'utilisateur.

Par exemple, pour ouvrir un fichier `fichier.ods` avec Libre Office, on peut avec le shell graphique cliquer dessus (l'extension `.ods` indique au système que le fichier est à ouvrir avec Libre Office), ou encore taper la ligne de commande `loffice fichier.ods`.

Les shells textuels n'ont pas disparu : sur tous les systèmes Unix, il existe des émulateurs de terminaux qui permettent d'utiliser ces shells textuels. Leur usage demande un apprentissage des commandes mais pour un administrateur système c'est un outil indispensable pour faire exécuter des tâches à un ordinateur. Sur la figure 8, l'utilisateur (moi) crée un nouveau répertoire `dossier_exemple`, puis s'y déplace. La commande `ls` permet de lister le contenu d'un répertoire. On peut créer un fichier avec `touch`, ou écrire dans un fichier avec `echo` et `>`. La commande `cat` permet d'afficher le contenu, la commande `du` (*disk usage*) donne la taille de tous les fichiers du répertoire courant (en kilo-octets par défaut. Ici, il n'y a qu'un petit fichier !). La commande `pwd` (*print working directory*) indique le répertoire courant.

Ces actions peuvent facilement être obtenues à la souris, avec un shell graphique. C'est beaucoup moins facile [5] avec la commande de la figure 9.

5. complètement impossible à ma connaissance, en fait.

```
svartz@svartz-HP:~$ mkdir dossier_exemple
svartz@svartz-HP:~$ cd dossier_exemple/
svartz@svartz-HP:~/dossier_exemple$ ls
svartz@svartz-HP:~/dossier_exemple$ touch fichier
svartz@svartz-HP:~/dossier_exemple$ echo "blablabla" > fichier
svartz@svartz-HP:~/dossier_exemple$ ls
fichier
svartz@svartz-HP:~/dossier_exemple$ cat fichier
blablabla
svartz@svartz-HP:~/dossier_exemple$ du
8       .
svartz@svartz-HP:~/dossier_exemple$ pwd
/home/svartz/dossier_exemple
```

FIGURE 8 – Une suite de commandes simples.

```
svartz@svartz-HP:~$ for f in `find . -type f 2>/dev/null | grep .py$` ; do cat $
f 2>/dev/null | grep -q matplotlib && echo $f ; done | wc -l
1143
```

FIGURE 9 – Une commande complexe en bash (shell linux) : le nombre de fichiers Python sur mon système contenant la chaîne de caractères « matplotlib » est 1143.

0.2.3 Organisation des fichiers

Les données utilisateurs et les programmes sont stockés dans la mémoire de masse, qui est organisée en un système de fichiers qui permet aux utilisateurs et aux programmes de les utiliser, d'en créer de nouveaux, de les modifier...

Le nombre de fichiers stockés sur la mémoire de masse est en général très important. Pour pouvoir les retrouver facilement, ils sont organisés en une structure arborescente de répertoires. Un répertoire est un ensemble de fichiers et de sous-répertoires désignés par des noms. Sous Unix (et donc sous GNU/Linux aussi), tous les fichiers sont regroupés dans une arborescence unique ; le sommet de cette arborescence est un répertoire appelé racine (notée `/`). Cette racine possède plusieurs sous-répertoires dont les principaux sont :

— `home` : qui contient les données de tous les utilisateurs, il y a un sous-répertoire par utilisateur ;

— `bin` : une partie des programmes installés ;

— `mnt` et `media` : c'est ici qu'on retrouve les données stockées sur les autres disques durs, les clés USB, les lecteurs de médias...

Sous Microsoft Windows, il y a une arborescence par périphérique de stockage de masse, chacun d'entre eux étant représenté par une lettre majuscule puis `:\`. Par exemple `C:\` pour le disque dur principal, `D:\` pour le lecteur de DVD, `F:\` pour une clé USB...

A priori, vous savez tous vous déplacer dans cette arborescence au moyen d'un shell graphique. Dans un shell Unix textuel, la commande pour changer de répertoire est `cd` (*change directory*), pour remonter dans le répertoire parent il suffit d'utiliser `cd ..` (deux points représentent le répertoire parent du répertoire courant, qui lui est représenté par un unique point). Ce mécanisme peut être utile en Python lorsqu'on veut changer de répertoire de travail [6].

Sous Unix un répertoire n'est autre qu'un fichier qui ne contient qu'une liste de couples (n, i) où n est le nom du fichier et i son inode. L'inode permet d'avoir accès aux méta-données du fichier, stockées dans la table des inodes :

— la date de création, de dernière modification et de dernière lecture ;

— la taille du fichier ;

— l'emplacement des données sur le disque.

Ainsi on dit souvent que sous Unix tout est fichier (même les périphériques d'entrée/sortie sont représentés par des fichiers).

0.2.4 Droits d'accès

À chaque fichier d'un système est attaché des droits. Au lycée, vous n'avez a priori accès en lecture/écriture qu'à vos fichiers. Ceci est géré par les droits d'accès. Prenons un exemple où on joue avec les droits d'accès (sous Linux), voir figure 10.

Expliquons chaque commande tapée et son effet :

6. avec le module `os` et la commande `os.chdir`, par exemple.

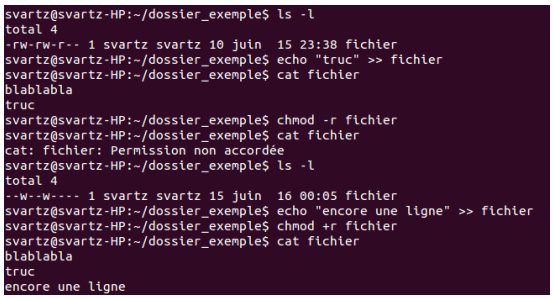


FIGURE 10 – Une suite de commandes modifiant les droits sur un fichier

— `ls -l` nous donne la liste (détaillée) des fichiers présents dans le répertoire courant et des informations. Ici il n'y a qu'un fichier, qui m'appartient, et les droits sont marqués à gauche, sous la forme `-rw-rw-r--`. Concentrons nous sur les deuxième, troisième et quatrième signes : le propriétaire du fichier à les droits en lecture (`r` comme read) et en écriture (`w` comme write). On retrouve trois autres tels paquets de lettres, pour des groupes. Le dernier paquet indique que les autres utilisateurs ont seulement un accès en lecture :

— `echo "truc" >> fichier` écrit à la suite d'un fichier, comme on le vérifie avec `cat` qui lit le fichier ;

— `chmod -r fichier` enlève les droits en lecture : `cat` n'est plus autorisée. On vérifie avec `ls -l` que le droit en lecture (`r`) n'est plus présent ;

— On peut toujours écrire dans le fichier ;

— `chmod +r fichier` remet les droits en lecture : `cat` fonctionne à nouveau !

On peut écrire des fichiers que l'on peut *exécuter*, il faut également gérer les droits d'exécution. Montrons comment écrire un programme autonome en Python (qu'on peut exécuter directement dans le shell). Un exemple simple est le suivant :

```
#!/usr/bin/python3
print("Hello World !")
```

Seule la première ligne diffère par rapport aux scripts Python habituels : il spécifie la localisation du programme qui va pouvoir comprendre le fichier. Ici, il s'agit de Python3. Sur mon système, il est accessible dans le répertoire `/usr/bin/`. Voyons la gestion des droits d'accès sur la figure 11.

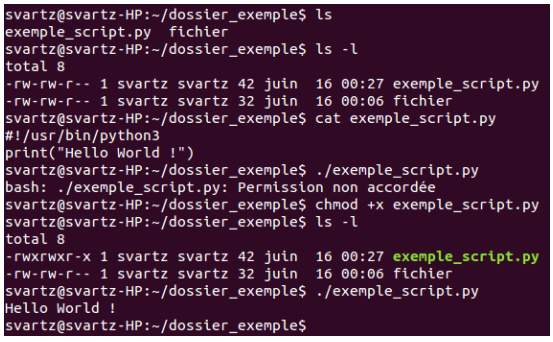


FIGURE 11 – Les droits d'exécution pour un script Python

On vérifie qu'initialement, le fichier `exemple_script.py` a les mêmes droits que `fichier` : on ne peut pas l'exécuter. La commande `chmod +x` le rend exécutable : on voit des `x` dans les droits, et d'ailleurs il apparaît en vert dans l'arborescence. L'exécution nous gratifie du tant attendu « Hello World ! ».

Évidemment, sur un système de fichiers quelconque, seul celui ayant les droits d'administrateur peut gérer les droits sur tous les fichiers du système.

0.3 Le langage Python

Python est le langage de programmation au programme des CPGE scientifiques. Ce langage a été développé par Guido Von Russom à la fin des années 80 et au début des années 90. Celui-ci a nommé le langage en référence à la troupe d'humoristes britanniques des Monthy Python.

Python est un langage *de haut niveau*, c'est-à-dire un langage de programmation orienté vers les problèmes à résoudre, permettant d'écrire facilement des programmes à l'aide de mots usuels (en anglais) et de symboles mathématiques. *A contrario*, un langage de bas niveau se rapproche du langage machine (dit binaire) et permet de programmer à un niveau très avancé, ce qui induit des temps de calculs réduits pour un problème donné par rapport à un langage de haut niveau. La contrepartie dans l'utilisation d'un langage de bas niveau est la longueur du code qui est en général bien plus importante.

C'est un langage de programmation impérative (boucles, tests conditionnels), orientée objets (hors programme en classes préparatoires), permettant aussi l'utilisation de la programmation fonctionnelle. Il est mutli-plateformes, c'est-à-dire qu'il peut être utilisé dans des environnements Unix, Mac-Os ou Windows, ou encore Android et iOS.

Pour travailler en Python, il suffit d'écrire de simples fichiers textes (voir section précédente) et de les interpréter. Cependant, on utilise souvent un *environnement de développement* pour faciliter la programmation. Au lycée, on utilisera au choix Pyzo, Spyder ou Idle.

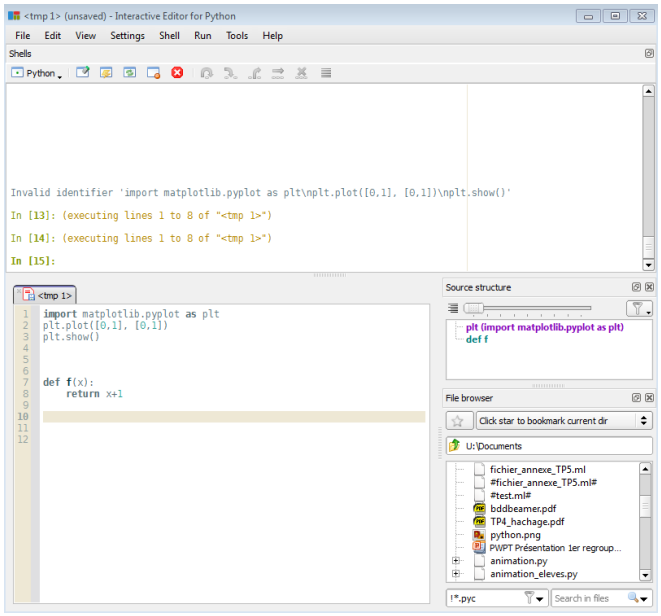


FIGURE 12 – L'environnement Pyzo

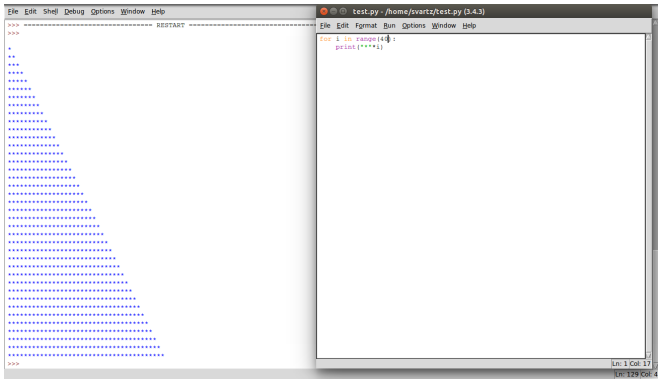


FIGURE 13 – L'environnement Idle

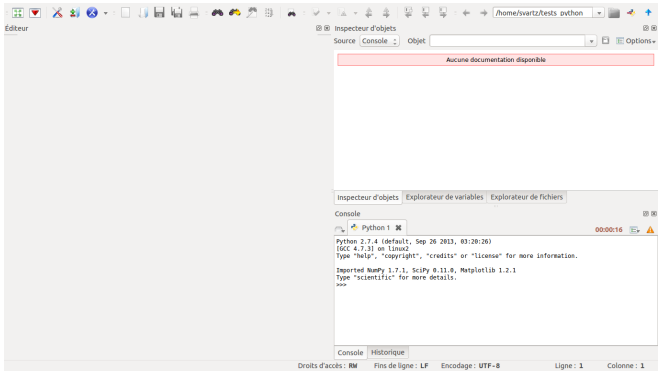


FIGURE 14 – L'environnement Spyder

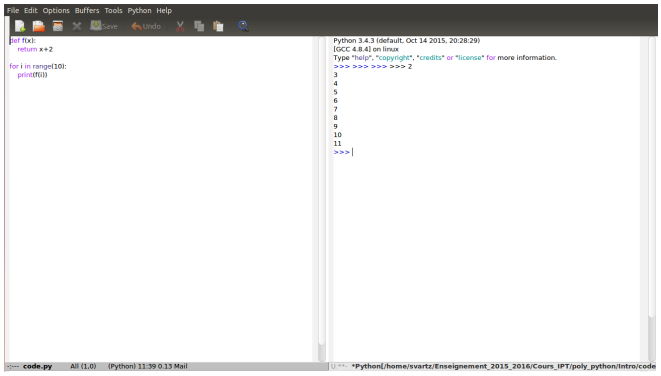


FIGURE 15 – L'environnement Emacs

À titre personnel, j'utilise Emacs, qui me permet aussi de rédiger le document que vous avez sous les yeux. Je ne vous le conseille pas car il est un peu compliqué à prendre en main, à moins que vous vouliez vous orienter vers une carrière d'informaticien.

Chapitre 1

Programmation en Python

Ce chapitre présente de manière détaillée les bases de la programmation Python. Il présente tout ce qui est au programme (et un peu plus), à l'exception de la lecture/écriture dans des fichiers, et les modules usuels, qui feront l'objet d'un chapitre à part. La version de Python utilisée est Python 3 (toutes les versions 3.x).

1.1 Types simples et expressions

1.1.1 Expressions

Une expression est une suite de caractères définissant une valeur. Pour connaître cette valeur, la machine doit *évaluer* l'expression. Voici quelques exemples numériques :

```
>>> 1+4
5
>>> 2.1+7
9.1
>>> 5/2
2.5
>>> 5//2*4.5
9.0
```

Les valeurs possèdent ce qu'on appelle un *type* : par exemple *entier, flottant, booléen, chaîne de caractères, liste, fonction...* Le type détermine les propriétés formelles de la valeur (par exemple, les opérations qu'elle peut subir) et matérielles (par exemple, la façon dont elle est représentée en mémoire et la place qu'elle occupe).
Pour connaître le type d'une expression après évaluation, il suffit de le demander à Python à l'aide de `type` :

```
>>> type(1+4)
<class 'int'>
>>> type(2.1+7)
<class 'float'>
>>> type(5/2)
<class 'float'>
>>> type(4<7)
<class 'bool'>
>>> type("blabla")
<class 'str'>
>>> type([0,1,2])
<class 'list'>
>>> type(lambda x:x+1) #ceci est la fonction qui a x associe x+1
<class 'function'>
```

On retrouve ici des types simples : entier (`int`), flottant (`float`) et booléen (`bool`). Chacun de ces types va faire l'objet d'un traitement particulier dans ce qui suit, de même que les types plus compliqués (chaînes de caractères, listes, fonctions...) un peu plus tard. On ne se préoccupera pas du mot clef `class` qui fait référence au caractère *orienté objet* du langage Python.

Comme dans la plupart des langages de programmation, une expression en Python est soit :

— une constante comme `2` ou `3.5` ;

— un nom de variable comme `x`, `i`, ou `compteur` ;

— le résultat d'une fonction appliquée à une ou plusieurs expressions, comme `PGCD(5,9)` ;

— la composée de plusieurs expressions réunies à l'aide d'opérateurs, comme `not a`, `3**6`, `(6+7)*8`. Les parenthèses servent comme en mathématiques à préciser quels opérateurs doivent être évalués en premier.

Voyons maintenant les types simples en détail.

1.1.2 Entiers

Constantes. Il n'y a pas grand chose à dire sur les entiers. On soulignera simplement qu'en Python, les entiers sont non bornés et permettent donc de faire des calculs exacts, avec des entiers gigantesques.

```
>>> 5**76 # ** est l'exponentiation.
132348898008484427979425390731194056570529937744140625
```

Opérateurs. Les opérateurs sur les entiers sont précisés dans la liste ci-dessous :

opérateur	+	-	*	//	%	**
signification	addition	soustraction	multiplication	division entière	modulo	exponentiation

On notera bien que `//` produit une division entière [1] (quotient dans la division euclidienne). On ne peut pas évaluer `a//b` si `b` est nul, et on fera attention si b est négatif ; en effet le comportement est un peu différent de la définition vue en cours de mathématiques.

Règles de priorités. Certains opérateurs sont évalués avant les autres, dans l'ordre de priorité suivant :

1. Exponentiation.
2. Multiplication, division, modulo.
3. Addition et soustraction.

Sur les opérateurs de même priorité, c'est celui qui est le plus à gauche qui est évalué en premier [2]. Les parenthèses permettent de changer ces priorités.

```
>>> 2+25%3*2**4
18
>>> (2+25%(3*2))**4
81
```

Autres bases. Ce paragraphe peut être ignoré en première lecture, il reprend les idées développées dans le chapitre sur la représentation des nombres. Par défaut, la base utilisée est la base 10 (celle que l'on utilise tous les jours !). Il est possible d'exprimer un entier dans les bases classiques en informatique : la base 2 (binaire), la base 8 (moins utile aujourd'hui) et la base 16 (hexadécimal). Pour cela, on fait précéder la représentation du nombre dans ces bases du préfixe `0b` (binaire), `0o` (octal) ou `0x` (hexadécimal [3]).

```
>>> 0b10011 # le nombre est 1*16+1*2+1*1
19
>>> 0o123 # le nombre est 1*64+2*8+3*1
83
>>> 0x123 # le nombre est 1*256+2*16+3*1
291
>>> 0xab # le nombre est 10*16+11*1
171
```

Inversement, on obtient la représentation en binaire, octal ou hexadécimal sous la forme d'une chaîne de caractères (voir la suite) à l'aide des fonctions `bin`, `oct` et `hex`.

1. En Python 2, `/` utilisé sur des entiers donne le quotient dans la division euclidienne.

2. Une exception notable est l'exponentiation, cohérente avec l'usage en mathématiques. Par exemple $2^{3^2} = 2^9 = 512$ est ce qu'on obtient avec `2**3**2` en Python, alors qu'on devrait obtenir 64 si la première exponentiation était évaluée en premier.

3. En hexadécimal, on a besoin de 15 chiffres pour les entiers de 0 à 15. On utilise les lettres de `a` à `f` pour les chiffres de 10 à 15. On a bien $171 = 10 \times 16 + 11$ ou encore (exemple suivant) $200 = 12 \times 16 + 8$.

```
>>> bin(200)
'0b11001000'
>>> oct(200)
'0o310'
>>> hex(200)
'0xc8'
```

1.1.3 Flottants

Constantes. Les flottants sont représentés en mémoire sur 32 ou 64 bits suivant le système (plutôt 64 de nos jours). Sur 64 bits, on a 1 bit de signe, 11 bits d'exposant et 52 bits de mantisse (on étudiera en détail la représentation des flottants dans un cours dédié). On tiendra compte du fait que seul un nombre fini de réels sont représentables en mémoire, ce qui ne permet pas de faire des calculs exacts. En particulier, le plus petit nombre strictement positif représentable exactement en flottant sur 64 bits est 2^{-1074} et le plus grand est légèrement inférieur à 2^{1024}.

Opérateurs. Les opérateurs sur les flottants sont précisés dans la liste ci-dessous (on s'en servira rarement, mais on peut également utiliser le modulo...) :

opérateur	`+`	`-`	`*`	`/`	`**`
signification	addition	soustraction	multiplication	division	exponentiation

Règles de priorités. De même que sur les entiers, certains opérateurs sont évalués avant les autres, dans l'ordre de priorité suivant :

1. Exponentiation.
2. Multiplication et division.
3. Addition et soustraction.

Comme pour les entiers, les opérateurs de même priorité sont évalués de gauche à droite, et les parenthèses permettent de changer ces priorités.

Conversion automatique. On remarque que la plupart des opérateurs sur les entiers et flottants sont les mêmes. Lorsque l'on utilise l'un de ces opérateurs avec des entiers et des flottants, les entiers sont automatiquement convertis en flottants (on pourrait forcer la conversion de l'entier `n` en flottant avec `float(n)`). C'est le cas également pour la division flottante [4] utilisée avec des entiers.

```
>>> 4*3.1
12.4
>>> 3/4
0.75
```

1.1.4 Booléens

Constantes. Les booléens sont essentiels en informatique. Ce type comprend uniquement deux constantes : `True` et `False` (Vrai et Faux) [5] . Ils sont principalement utilisés dans les structures de contrôle (voir section 1.3).

Opérateurs. Les opérateurs sur les booléens sont au nombre de trois. L'un (`not`) est un opérateur unaire (ne prenant qu'un opérande), les deux autres (`and` et `or`) sont des opérateurs binaires (nécessitant deux opérandes). la liste suivante présente les différents opérateurs booléens et leurs *tables de vérité*.

a	b	not a	a or b	a and b
False	False	True	False	False
False	True		True	False
True	False	False	True	False
True	True		True	True

Ce tableau est intuitif : `not` correspond à la négation. Pour que `a and b` soit vrai, il faut que `a` et `b` le soient tous les deux. Pour que `a or b` soit vrai, il suffit que l'un des deux le soit. Attention : le « ou » français peut parfois avoir le sens d'un ou exclusif, comme dans « fromage ou dessert ». Le `or` en informatique est toujours inclusif (si `a` et `b` sont vrais, alors `a or b` aussi).

4. Attention encore, si vous travaillez en Python 2, vous obtiendez une division entière avec une simple barre `/`
5. Et pas `"True"` ou encore `false` !

Règles de priorité. L'ordre de priorité d'évaluation pour les opérations booléennes est le suivant :

1. `not`.
2. `and`.
3. `or`.

De même que pour les entiers et les flottants, on évalue ensuite de gauche à droite les opérateurs de même priorité, et on peut user de parenthèses.

```
>>> False and False or True
True
>>> False and (False or True)
False
```

Opérateurs de comparaisons et booléens. On utilise rarement des booléens tels quels. Leur intérêt réside dans les structures de contrôle conditionnelles que l'on verra en section 1.3. Ces structures font beaucoup usage de l'évaluation d'expressions produisant des booléens, parmi lesquelles on trouve les opérations de comparaisons sur les entiers/flottants :

Opérateur	==	!=	<	<=	>	>=
signification	égal	non égal	inférieur	inférieur ou égal	supérieur	supérieur ou égal

```
>>> 4>=3 or 5==0
True
```

Pour les évaluations des opérateurs binaires, les deux opérandes des opérateurs de comparaisons sont amenés à un type commun avant l'évaluation de la comparaison (flottant dans le cas d'entier et flottant).

La priorité des opérateurs de comparaison est inférieure à celle des opérateurs arithmétiques : ainsi, l'expression `a==b+c` signifie `a==(b+c)`, ce qui est assez logique.

On peut également, comme sur beaucoup d'objets Python, utiliser les opérateurs `==` et `!=` pour l'égalité et la différence de booléens. Notons que si `x` est un booléen, il est parfaitement inutile d'écrire quelque chose comme `x==True` : ce booléen est égal à `x`. De même, on préférera écrire `not x` que `x==False`.

Caractères paresseux des opérateurs `and` et `or`. On notera que dans une expression de la forme `a and b`, où `a` et `b` sont des expressions, si `a` s'évalue en `False`, on n'a pas besoin d'évaluer `b` pour s'apercevoir que `a and b` s'évalue en `False`. De même avec `a or b` si `a` s'évalue en `True`. Python respecte cette logique : si la partie gauche suffit à déterminer si l'expression s'évalue en `True` ou `False`, il n'évalue pas la partie droite (on parle du caractère paresseux des opérateurs).

C'est particulièrement utile lors de l'évaluation d'une expression dont la seconde partie pourrait produire une erreur, mais dont la première sert de garde-fou : `x>0 and log(x)>2` ne produit pas d'erreur, même si `x` est un nombre négatif. En effet, dans ce cas `x>0` s'évalue en `False` et on n'a pas besoin d'évaluer `log(x)>2` qui produirait une erreur, le `log` n'étant pas défini sur les nombres négatifs.

Raccourcis. Plutôt que d'écrire `a<=b and b<=c`, Python comprend très bien `a<=b<=c`. On n'abusera cependant pas de ces raccourcis, pour écrire des choses illisibles comme `a<b>=c`.

1.2 Variables

1.2.1 Identificateurs

Un identificateur est une suite de lettres et chiffres, qui commence par une lettre, et qui n'est pas un mot réservé du langage. Les mots réservés du langage Python sont par exemple `if`, `else`, `def`, `return`, `True`... Le caractère `_` (underscore, ou « tiret du 8 ») est considéré comme une lettre. Ainsi, `i`, `j`, `x`, `x2`, `compteur` et `taille_de_la_liste` sont des identificateurs corrects, contrairement à `4a`, `x{}`, `if` ou encore `taille de la liste`. Les majuscules et minuscules ne sont pas équivalents : `x` et `X` sont des identificateurs distincts. Même si les accents sont autorisés, on veillera à ne pas en mettre dans les identificateurs pour ne pas faire dépendre la bonne exécution d'un programme de l'encodage des caractères.

1.2.2 Variables

Une variable est constituée de l'association d'un identificateur à une valeur. Cette association est créée lors de l'affectation, qui s'écrit sous la forme `variable = expression`. Attention : il ne faut pas confondre = (affectation) avec == (test d'égalité). Le mécanisme de l'affectation est le suivant : l'expression à droite du signe égal est évaluée, puis le résultat de l'évaluation est affecté à la variable. Cela n'a donc **rien** à voir avec le signe = des mathématiques. À la suite d'une telle affectation, chaque apparition de la variable ailleurs que dans la partie gauche d'une autre affectation représente la valeur en question. Cette association entre la variable et la valeur est valable tant qu'il n'y a pas de nouvelle affectation avec cette même variable.

```
>>> x=2+2 #on évalue 2+2, on obtient 4, qu'on affecte à x.
>>> y=x**x+1 #ici, x représente la valeur 4. 4**4+1 vaut 257, qu'on affecte à y.
>>> print(y-1) #print est une fonction d'affichage. y-1 s'évalue en 256, qu'on affiche.
256
>>> x=y/2 #nouvelle affectation de x.
>>> print(x)
128.5
```

Comme on le voit sur ces exemples, en Python :

— contrairement à plusieurs autres langages de programmation, les variables n'ont pas besoin d'être déclarées (c'est-à-dire préalablement annoncées), la première affectation leur tient lieu de déclaration ;

— les variables ne sont pas liées à un type (mais les valeurs auxquelles elles sont associées le sont forcément) : la même variable `x` a été associée à des valeurs de types différents (un `int`, puis un `float`).

Dans la suite, on confondra allègrement l'identificateur, la variable (l'association de l'identificateur à une valeur) et la valeur elle-même. Si un identificateur n'a pas été affecté (en toute rigueur il n'est donc pas un nom de variable) son emploi ailleurs que dans le membre gauche d'une affectation est illégale et provoque une erreur. Par exemple :

```
>>> print(variable_inconnue)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'variable_inconnue' is not defined
```

Seul un identificateur correct peut figurer dans le membre gauche d'une affectation. Une syntaxe de la forme `x+1=y` n'a aucun sens :

```
>>> y=5
>>> x+1=y
  File "<stdin>", line 1
SyntaxError: can't assign to operator
```

L'erreur est explicite : on ne peut pas affecter à l'opérateur +. Un mécanisme qui sert souvent en informatique est l'incrémentation d'une variable : on lui rajoute un certain nombre, souvent 1.

```
>>> x=4
>>> x=x+1 # le signe = n'a rien à voir avec celui des mathématiques !
>>> print(x)
5
```

Il y a un raccourci en Python pour cette opération : `x+=1`. On peut de même écrire `x-=2` ou encore `x/=3`.

Enfin, lors de l'affectation `x=y`, la variable `x` prend la valeur de celle de `y` (l'évaluation de l'expression `y` donne simplement la valeur stockée dans la variable `y`), mais `x` et `y` ne sont pas « liées » pour autant :

```
>>> y=2
>>> x=y
>>> y=5
>>> print(x)
2
```

1.3 Structures de contrôle

1.3.1 Python et l'indentation.

Contrairement à la plupart des autres langages, Python n'utilise pas d'accolades {...} ou de mots clefs de la forme `begin`...`end` pour délimiter des blocs d'instructions : tout est basé sur l'*indentation* : le fait de laisser une marge blanche devant un bloc d'instructions. C'est fort pratique pour écrire un code court, mais il faut faire attention à respecter des règles strictes :

— une séquence d'instructions est faite d'instructions écrites avec la même indentation ;

— dans une structure de contrôle (instruction conditionnelle `if` ou boucles `for` et `while`, qu'on va voir tout de suite) ou dans la définition d'une fonction (voir section 1.5), une séquence d'instructions subordonnées doit avoir une indentation supérieure à celle de la séquence englobante. Toute ligne écrite avec la même indentation que cette séquence englobante marque la fin de la séquence subordonnée.

L'indentation standard est de quatre espaces, ou une tabulation. Les éditeurs intelligents remplacent automatiquement une tabulation par quatre espaces.

Le point-virgule. Pour éviter de revenir à la ligne systématiquement entre deux instructions, on peut séparer les instructions par des points virgules, par exemple `a=2 ; b=4`, qui réalise successivement l'affectation de `a` puis celle de `b`.

1.3.2 Instruction conditionnelle if/else (si/sinon)

La structure générale d'une instruction conditionnelle est la suivante (rappelez-vous, en Python, l'indentation est primordiale).

```
if expression:
    [instructions effectuées si expression s'évalue en True]
elif autre_expression:
    [instructions effectuées si expression s'évalue en False et autre_expression en True]
else:
    [instructions effectuées si expression et autre_expression s'évaluent en False]
```

Les expressions utilisées sont des expressions *booléennes* : leur évaluation doit produire un *booléen* : `True` ou `False`. De telles expressions sont par exemple `a>=4`, ou bien `not a==0 and b>=2` (`a` est non nul et b est supérieur ou égal à 2).

Dans une telle structure conditionnelle, les expressions booléennes sont évaluées les unes après les autres, de haut en bas, jusqu'à ce que l'une d'entre elles s'évalue en `True`. Le bloc d'instructions correspondant (et seulement celui-ci) est alors exécuté, puis on sort de la structure conditionnelle. Le bloc correspondant au `else` est exécuté seulement si toutes les expressions conditionnelles situées au dessus se sont évaluées en `False`. Il est possible de mettre plusieurs `elif` :

Voici un exemple, avec `note` une variable supposée contenir un flottant entre 0 et 20.

```
if note>=16:
    print("Mention Très bien")
elif note>=14:
    print("Mention Bien")
elif note>=12:
    print("Mention Assez bien")
elif note>=10:
    print("Mention Passable")
else:
    print("Raté !")
```

Même si `note` contient un flottant supérieur à 16, on n'affichera à l'écran (effet de la fonction `print`) qu'une seule ligne : la première telle que la condition `note>=...` soit réalisée, ou « Raté ! » si `note` est strictement inférieure à 10.

`elif` et `else` peuvent tous deux être omis. Dans une telle suite d'instructions au plus une (et exactement une si `else` est présent) est exécutée : la première telle que l'expression booléenne associée s'évalue en `True`. Par exemple, dans la séquence suivante, `x` est incrémenté de 1 s'il est supérieur ou égal à 2, et divisé par 2 s'il est strictement inférieur à 0.

```
if x<0:
    x=x/2
elif x>=2:
    x+=1
```

Si `x` appartient à l'intervalle $[0, 2[$, il est inchangé. Il est parfaitement inutile [6] d'écrire quelque chose comme `x=x` dans un bloc `else`.

Prenons un exemple complet un peu plus complexe, la résolution d'une équation polynomiale de degré 2 sur les réels, en supposant que les variables `a`, `b` et `c` contiennent des flottants, avec `a` non nul.

```
Delta=b**2-4*a*c
if Delta<0:
    print("Pas de racines !")
elif Delta>0:
    r=sqrt(Delta)
    x1=(-b-r)/(2*a)
    x2=(-b+r)/(2*a)
    print("Il y a deux racines distinctes, qui sont: ",x1,"et",x2)
else:
    print("Il y a une racine double, qui est: ",-b/(2*a))
```

On laisse le lecteur se reporter au chapitre sur la représentation des nombres pour la pertinence de l'algorithme lorsque le discriminant est calculé comme étant zéro.

Écriture en ligne. Il n'est en fait pas obligatoire de faire un saut de ligne après un `if`, `elif` ou `else`, en particulier s'il n'y a qu'une instruction à écrire. Cela est valable également pour les boucles et les fonctions (voir la suite). Par exemple le code suivant est équivalent à celui vu plus haut :

```
if note>=16: print("Mention Très bien")
elif note>=14: print("Mention Bien")
elif note>=12: print("Mention Assez bien")
elif note>=10: print("Mention Passable")
else: print("Raté !")
```

Personnellement, je trouve cela moins clair qu'avec des sauts de ligne, mais on le voit parfois dans les sujets de concours, vous êtes prévenus !

1.3.3 Boucle conditionnelle while (tant que)

La boucle `while` permet de réaliser une suite d'instructions tant qu'une certaine condition est vraie. La structure est la suivante.

```
while expression:
    [instructions]
```

Le mécanisme est le suivant : on évalue `expression`. Si le résultat est `True`, on effectue toutes les instructions du bloc indenté, puis on recommence l'évaluation de `expression`. Sinon, on passe aux instructions situées après la boucle. Par exemple, la séquence :

```
i=0
while i<10:
    print(i)
    i+=1 #un raccourci pour i=i+1
print("fini !")
```

affiche à l'écran tous les nombres entre 0 et 9, puis « fini ! ». En effet, lorsque `i` atteint 9, on l'affiche à l'écran, puis on incrémente `i` (qui vaut 10 en bas de la boucle). On réévalue ensuite la condition, mais `10<10` s'évalue en `False`, donc on sort de la boucle, et on affiche « fini ! » qui est une expression en dehors du corps de la boucle.

Notez bien que l'expression est évaluée uniquement en haut de la boucle : si elle s'évalue en `True`, on effectue toutes les instructions du corps de boucle, et on réitère l'évaluation. La boucle suivante affiche les entiers de 1 à 10, puis « fini ! ».

6. On le voit souvent dans les copies de concours...

```
i=0
while i<10:
    i+=1
    print(i)
print("fini !")
```

Il se peut très bien qu'à la première évaluation de l'expression, celle-ci soit `False` : dans ce cas on n'effectue jamais le corps de boucle :

```
i=-1
while i>=0:
    print("on n'affichera jamais ça.")
```

Enfin, on fera attention avec les boucles `while`, si on s'y prend mal, on crée un morceau de code qui boucle sans fin :

```
while True: #l'expression s'évalue en True !
    print("ce texte sera affiché, encore et encore !")
```

L'obtention d'une « boucle infinie » est parfois plus subtile :

```
x=0.1
while x!=1:
    x=x+0.1
```

On verra dans le chapitre sur la représentation des nombres qu'en arithmétique flottante, additionner 10 fois 0.1 ne fait pas tout à fait 1 (le résultat est évidemment très proche, mais ne vaut pas exactement 1).

1.3.4 Boucle inconditionnelle for... (pour...)

En informatique, on a très souvent besoin qu'une variable prenne successivement comme valeur tous les entiers entre deux bornes, par exemple de 0 à 100. Évidemment, on peut réaliser ça comme dans la section précédente avec une boucle `while` :

```
i=0
while i<=100:
    [instructions]
    i=i+1
```

Il est intéressant de raccourcir cette écriture, pour ne pas avoir à initialiser manuellement `i` ou écrire l'incrémentation `i+=1`, limiter les erreurs possibles et rendre le code plus lisible. On utilise alors une boucle *inconditionnelle* (i prend toutes les valeurs entières de 0 à 100 sans condition) : la boucle `for`.

Dans certains langages de programmation, celle-ci ne diffère conceptuellement pas d'une boucle `while` et est traduite ainsi au moment de la compilation du programme. Par exemple en C, qui est un langage très populaire :

Une boucle for, en langage C

```
for (i=0; i<=100; i++) {
    [instructions]
}
```

Traduction à la compilation

```
i=0 ;
while (i<=100) {
    [instructions]
    i++ ;
}
```

En Python, la structure de la boucle `for` est légèrement différente, c'est celle-ci :

```
for element in iterable:
   [instructions]
```

L'itérable est quelque chose que l'on peut itérer : en gros, c'est quelque chose qui fournit une séquence de valeurs. La syntaxe `for element in iterable` signifie que la variable `element` doit prendre successivement toutes les valeurs que fournit l'itérable. Pour chacune de ces valeurs, on exécute les instructions du corps de boucle. Bien souvent, on utilisera le constructeur `range` qui fournit des suites (finies) d'entiers. La syntaxe est la suivante, tous les paramètres m, n et p intervenant sont des entiers :

— pour $n \geq 0$, `range(n)` fournit tous les entiers de 0 inclus à n exclus (attention, on s'arête donc à $n-1$!)

— on peut décider de commencer à un autre entier que 0 en précisant un autre paramètre : pour $m \leq n$, `range(m,n)` fournit tous les entiers de m inclus à n exclus.

En précisant un troisième paramètre, on peut faire varier le pas :

— si $p > 0$, `range(m,n,p)` fournit successivement les entiers $m, m+p, m+2p, \ldots$ strictement inférieurs à n ;

— Si $p < 0$, `range(m,n,p)` fournit successivement les entiers $m, m+p, m+2p, \ldots$ strictement supérieurs à n.

Par exemple la boucle :

```
for i in range(0,m,2):
   print(i)
```

affiche à l'écran successivement tous les entiers pairs entre 0 et $m-1$ (la borne m est exclue). La boucle suivante affiche tous les entiers de $n-1$ à 0, (dans l'ordre décroissant) :

```
for i in range(n-1,-1,-1):
   print(i)
```

Petit conseil : apprendre par cœur la syntaxe `range(n-1,-1,-1)`, elle sert souvent. Donnons un exemple un peu plus complet : le calcul de 10!

```
x=1
n=10
for i in range(1,n+1):
    x*=i #un raccourci pour x=x*i
```

On peut vérifier que la variable `x` contient bien $10! = 3628800$ à l'issue de cette boucle.

L'itérable peut également être une liste (dans ce cas `element` prend successivement toutes les valeurs de la liste), ou une chaîne de caractères (dans ce cas, `element` prend successivement comme valeurs tous les caractères de la chaîne), ou encore un tuple (n-uplet)... Ces types seront examinés en section 1.4.

Donnons un petit exemple, montrant que Python peut faire beaucoup de choses (l'exemple n'est pas à retenir). Le module `itertools` permet de faire de la combinatoire. Considérons le triplet `(1, 4, 7)`. On peut facilement produire toutes les permutations possible du triplet avec la fonction `permutations` du module `itertools` :

```
import itertools # pour pouvoir utiliser le module
iterable=itertools.permutations((1,4,7))
for x in iterable:
    print(x)
```

Le code précédent affiche à l'écran :

```
(1, 4, 7)
(1, 7, 4)
(4, 1, 7)
(4, 7, 1)
(7, 1, 4)
(7, 4, 1)
```

1.3.5 Break et Continue

Ces instructions ne sont pas exigibles, mais sont parfois très pratiques (surtout `break`). Dans une boucle (`while` ou `for`), on peut utiliser les commandes `break` et `continue` : `break` sort de la boucle, et `continue` poursuit l'itération en revenant « tout en haut », sans se préoccuper de ce qu'il y a derrière. Voici un exemple un peu stupide qui mélange les deux :

```
u=0
while u<100:
   print(u)
   u+=1 #un raccourci pour u=u+1
   if u==5:
      break
   continue
   print(1/0)
```

Ce code ne produit pas d'erreur et n'affiche que 6 entiers : 0, 1, 2, 3, 4 et 5. Dans le corps de boucle, la partie `print(1/0)` n'est jamais exécutée (ce qui produirait une erreur) puisqu'elle se trouve après le `continue`. Lorsque `u` atteint 5, on rentre dans le `if` et on sort de la boucle avec `break`.

Dans le cas de boucles imbriquées, c'est seulement la boucle interne contenant `break` ou `continue` qui est concernée [7]. On évitera de faire un usage abusif de ces instructions qui peuvent rendre le code difficilement compréhensible, mais on pourra y recourir avec parcimonie.

1.3.6 Boucles imbriquées

Il est tout à fait possible d'imbriquer des boucles, ce que l'on fera par la suite pour (entre autres) résoudre un système linéaire ou trier une liste. Voici un exemple, la recherche de tous les triplets pythagoriciens (triplets (a, b, c) tels que $a^2 + b^2 = c^2$) où les trois composantes sont strictement inférieures à 1000. On cherche ici uniquement ceux qui vérifient $1 \leq a \leq b$. On va donc utiliser deux boucles pour balayer tous les entiers $1 \leq a \leq b < 1000$, et vérifier essentiellement si $\sqrt{a^2 + b^2}$ est un entier strictement inférieur à 1000.

```
N=1000
for a in range(1,N):
    for b in range(a,N):
        c2=a*a+b*b
        if c2>=N**2:
            break
        elif round(c2**0.5)**2==c2:
            print(a,b,round(c2**0.5))
```

On a en fait utilisé `round` qui permet d'arrondir à l'entier le plus proche [8], et vérifié si le carré de l'entier le plus proche de $\sqrt{a^2 + b^2}$ était $a^2 + b^2$ lui même. 878 triplets sont affichés à l'écran.

1.4 Structures de données

1.4.1 Listes

On étudie ici le type `list` en Python, qui est essentiel et nous servira souvent. L'appellation `list` de Python est un peu malheureuse et la traduction en « liste » maladroite : en toute rigueur, il faudrait parler de « tableau redimensionnable inhomogène ». Mais les sujets de concours parlent de listes, donc nous aussi. Voici un exemple de liste : `[True, 4, 5, 3.0]`.

Comme l'exemple le montre, les listes sont des séquences finies d'éléments, possiblement de types différents. La syntaxe consiste à les mettre entre crochets, séparés par des virgules.

Construction de listes. On peut construire une liste de plusieurs manières :

- par la donnée explicite des éléments, entre crochets, séparés par des virgules, comme ci-dessus.
- par concaténation de listes (à l'aide de `+`) : `[1, 2, 3]+[4, 5, 6]` s'évalue en `[1, 2, 3, 4, 5, 6]`.
- `list(iterable)` permet de fabriquer une liste à partir d'un itérable. Par exemple, `list(range(4))` s'évalue en `[0, 1, 2, 3]` et `list("truc")` en `['t', 'r', 'u', 'c']`.
- par compréhension, très pratique avec la structure suivante : `[f(x) for x in iterable if P(x)]`, où `f(x)` est une expression dépendant (ou non) de `x`, et `P(x)` est une expression booléenne (facultative). Par exemple `[x*x for x in range(5) if x%2==0]` s'évalue en la liste `[0, 4, 16]`.
- par *slicing* (tranchage), qu'on va voir bientôt.
- ...

7. Rappel : `if` n'est pas une boucle !
8. Pour éviter tout test d'égalité sur les flottants.

Accès aux éléments. Pour `L` une liste, sa *longueur* (nombre d'éléments, length en anglais) est accessible avec `len(L)`. En notant n cette longueur, les éléments sont indexés par les entiers de 0 à $n-1$. Exemples :

```
>>> L=list(range(1,6)) #range(1,6) fournit les entiers de 1 à 5.
>>> L[2]
3
>>> L[len(L)-1]
5
```

Si on demande l'accès à un caractère d'indice négatif i compris entre -1 et $-n$, où n est la longueur de la liste, celui-ci est considéré comme étant $n+i$:

```
>>> L[-1] # très pratique pour accéder au dernier élément !
5
>>> L[-5]
1
```

L'accès à tout autre indice produit une erreur :

```
>>> L[len(L)]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Retenez bien cette erreur, vous l'aurez souvent !

Modification d'un élément. Étant donnée une liste `L`, on peut modifier l'élément d'indice i de `L` en le remplaçant par l'élément de son choix. La syntaxe est la même que pour une affectation.

```
>>> L=list(range(1,6))
>>> L
[1, 2, 3, 4, 5]
>>> L[2]=10
>>> L
[1, 2, 10, 4, 5]
```

Les règles régissant l'indice i sont les mêmes que précédemment.

Slicing (tranchage). On peut créer une nouvelle liste en extrayant certains éléments d'une liste. Pour extraire les éléments d'indice entre d inclus et $f \geq d$ exclu, on utilise `L[d:f]` : la liste obtenue est donc composée des éléments `L[d], L[d+1],..., L[f-1]`. L'un ou l'autre de ces deux indices peut être omis, et même les deux (dans ce cas, d vaut 0 et f vaut la longueur de la liste). Ce mécanisme est tolérant envers les indices trop grands ou trop petits (attention, les indices négatifs entre -1 et $-n$ sont interprétés comme précédemment), et si $d \geq f$, on obtient la liste vide.

```
>>> L
[1, 2, 10, 4, 5]
>>> L[3:4]
[4]
>>> L[4:3]
[]
>>> L[:4]
[1, 2, 10, 4]
>>> L[:8]
[1, 2, 10, 4, 5]
```

On peut également spécifier un pas, positif ou négatif. L'interprétation est la même que pour les listes et l'itérateur `range`, on ne précisera donc pas ici.

```
>>> L[::2]
[1, 10, 5]
```

Méthodes sur les listes. Python est un langage *orienté objet*. À chaque classe d'objets (comme les listes) peuvent s'appliquer plusieurs *méthodes*, qui modifient l'objet ou renvoient certaines de ces caractéristiques. Au programme en classes préparatoires, on trouve seulement `append` et `pop`, qui permettent d'ajouter ou d'enlever un élément en fin de liste. La syntaxe générale de l'utilisation d'une méthode sur un objet est la suivante : `objet.methode(parametres)`.

Le tableau suivant récapitule les principales méthodes sur les listes. Le but est de présenter ce qu'on peut faire en Python, et de voir que ces opérations ne sont pas toutes triviales pour le processeur. La colonne complexité ne peut être comprise qu'après le chapitre dédié [9]. On note n la longueur de la liste `L`.

méthode	description	complexité
`L.append(x)`	Ajoute `x` à la fin de `L`.	$O(1)$ (amorti)
`L.extend(T)`	Ajoute les éléments de `T` à la fin de `L` (équivalent à `L+=T`).	$O(\text{len}(T))$ (amorti)
`L.insert(i, x)`	Ajoute l'élément `x` en position `i` de `L`, en décalant les suivants vers la droite.	$O(n-i)$ (amorti)
`L.remove(x)`	Supprime de la liste la première occurrence de `x` si `x` est présent, sinon produit une erreur.	$O(n)$.
`L.pop()`	Supprime le dernier élément de `L`, et le renvoie.	$O(1)$
`L.pop(i)`	Supprime l'élément d'indice `i` de `L`, en décalant les suivants vers la gauche. Cette méthode renvoie l'élément supprimé.	$O(n-i)$
`L.index(x)`	Retourne l'indice de la première occurrence de `x` dans `L` si `x` est présent, produit une erreur sinon.	$O(n)$
`L.count(x)`	Retourne le nombre d'occurrences de `x` dans `L`.	$O(n)$
`L.sort()`	Trie la liste `L` dans l'ordre croissant (en place).	$O(n\log n)$
`L.reverse()`	Renverse la liste (en place)	$O(n)$.

Attention, ces méthodes ne renvoient en général rien : elles modifient l'objet. C'est le cas pour `append`, qui permet d'ajouter en fin de liste un nouvel élément. Pour ajouter `x` a la fin de `L`, on écrit simplement `L.append(x)`, et non pas `L=L.append(x)`. En effet, `L.append(x)` est une expression dont l'évaluation produit `None`, c'est-à-dire rien. Écrire `L=L.append(x)` reviendrait à affecter `None` à la variable `L`, ce qui n'est pas *a priori* ce qu'on veut faire ! Voir la section 1.5 pour des précisions sur `None`.

Listes et références. Ce point est important. Vous ferez l'erreur un jour, mais vous vous douterez du problème si vous avez bien compris ce paragraphe. Prenons tout de suite un exemple :

```
>>> T=[0,2,3]
>>> U=T
>>> T[0]=1
>>> T.append(4)
>>> print(U)
[1, 2, 3, 4]
```

Si on modifie `T`, on modifie `U`. Le comportement semble bien différent des variables ! Essayons autre chose :

```
>>> T=[0,2,3]
>>> U=T[:]
>>> T[0]=1
>>> T.append(4)
>>> print(U)
[0, 2, 3]
```

Le comportement est plus sympathique. Il faut savoir que lorsqu'on crée une liste, la variable utilisée est ce qu'on appelle une *référence* (ou un pointeur) vers l'emplacement mémoire où est stockée la liste. L'instruction `U=T` du premier exemple stocke dans la variable `U` la référence stockée dans `T`. Autrement dit, l'emplacement mémoire désigné par `T` et `U` est le même ! Lorsqu'on effectue l'instruction `T[0]=1`, on va modifier directement la mémoire (de même avec `append`), et il est logique que ce changement soit visible lorsqu'on tape `print(U)`, puisque cette action va chercher en mémoire ce qu'indique `U`.

Dans le deuxième exemple, l'instruction `U=T[:]` est différente : on crée une liste dont les éléments sont les mêmes que ceux de `T`, mais ailleurs en mémoire. Autrement dit, les références `T` et `U` pointent vers des endroits différents en mémoire, et donc si on modifie l'une, l'autre n'est pas modifié.

9. Ajoutons que la complexité amortie (hors programme) a le sens suivant : si on fait plein de fois la même opération (par exemple ajouter un élément à la fin d'une liste), le temps d'exécution sera en moyenne celui donné (par exemple, temps constant pour `append`).

Liste de listes. On utilisera couramment des listes qui contiennent des listes, en particulier pour représenter des matrices. L'accès aux éléments se fait de manière similaire :

```
>>> L=[[0,1,2,3],[4,5]] # une liste de deux listes
>>> L[0] # premier élément de L
[0, 1, 2, 3]
>>> L[1][0] # premier élément du deuxième élément de L
4
>>> L[0][2]=6
>>> print(L)
[[0, 1, 6, 3], [4, 5]]
```

Pour terminer, faisons une mise en garde supplémentaire, concernant encore les références :

```
>>> T=[[0,2],[3,4]]
>>> U=T[:]
>>> U[0][0]=1
>>> print(T)
[[1, 2], [3, 4]]
```

Ici, on a pris soin de recopier les éléments de `T`. Mais comme ces éléments sont des références vers `[0,2]` et `[3,4]`, le problème reste le même que précédemment puisque ces listes-là n'ont pas été recopiés. Il aurait fallu écrire :

```
U=[A[:] for A in T]
```

Mais si `T` avait été une liste de listes de listes, le problème se serait encore posé. Faisons deux remarques :

— premièrement, on manipulera rarement des listes de listes de listes ;

— deuxièmement, il existe un module `copy` dont la fonction `deepcopy` permet de copier « en profondeur » un objet.

1.4.2 Tuples

Présentation. Un tuple ressemble beaucoup à une liste, mais on ne peut ni modifier ses éléments, ni lui en ajouter ou en enlever. La structure mathématique associée est celle de n-uplet. On parle de structure immuable (ou statique, ou encore non mutable). En contrepartie de cette rigidité les tuples sont très compacts (ils occupent peu de mémoire) et l'accès à leurs éléments est rapide.

Pour la syntaxe, on crée un tuple en écrivant ses éléments, séparés par des virgules et encadrés par des parenthèses. S'il n'y a pas d'ambiguïté, les parenthèses peuvent être omises (en pratique, dès que le nombre d'éléments du tuple est au moins 2). Un tuple constitué d'un seul élément `a` doit être écrit `a,` ou `(a,)`. Le tuple sans élément se note `()`.

Opérations sur les tuples. Faisons une brève session Python de démonstration, pour vérifier que les opérations sur les listes sont valables pour les tuples :

```
>>> t=4, True, 0.5 ; v=(6,) ; w=((7,8),) # w est un tuple dont le seul élément est un tuple
>>> t+v
(4, True, 0.5, 6)
>>> t[1:]
(True, 0.5)
>>> t+w
(4, True, 0.5, (7, 8))
>>> len(t+w)
4
```

Comme on le voit, un élément d'un tuple peut être un tuple à son tour. Attention, les tuples sont immuables : on ne peut modifier un élément. L'erreur ci-dessous est très explicite.

```
>>> t[0]=3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Déconstruction d'un tuple. On peut *déconstruire* un tuple par affectation simultanée à un tuple de variables de la même taille, pour réaliser une affectation simultanée :

```
>>> couple=(1,2)
>>> (x,y)=couple
>>> print(x) ; print(y)
1
2
```

Notez que les parenthèses autour du tuple de variables sont facultatives et qu'on pourrait (ce qu'on fait en pratique) écrire `x,y=couple`. Si le tuple de variables n'est pas de la même taille que le tuple à déconstruire, on obtient une erreur :

```
>>> a,b,c=couple
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: need more than 2 values to unpack
>>> triplet=1,2,3
>>> x,y=triplet
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 2)
```

L'erreur indique à chaque fois que le tuple à droite n'est pas de la bonne taille, vis-à-vis du tuple de variables.

On peut maintenant expliquer un comportement spécifique à Python mais fort pratique pour échanger les contenus de deux variables :

```
a,b=b,a
```

De même que précédemment, on construit d'abord le tuple qui contient les valeurs de `b` et `a` qu'on déconstruit ensuite pour affecter le contenu aux variables `a` et `b`.

1.4.3 Chaînes de caractères

Une donnée de type chaîne de caractères est une suite de caractères quelconques. Une chaîne de caractères s'indique en écrivant les caractères en question soit entre apostrophes, soit entre guillemets : `'Bonjour'` et `"Bonjour"` sont deux écritures correctes de la même chaîne. Du point de vue structurel, les chaînes sont très proches des tuples. Comme eux, elles sont immuables : on ne peut pas changer un caractère.

Concaténation, accès à des caractères, slicing. Comme pour les tuples, on peut concaténer deux chaînes de caractères à l'aide de `+` pour en produire une troisième, la longueur de la chaîne est donnée par `len`, et l'accès aux caractères est similaire.

```
>>> "C'est une"+" phrase complète."
"C'est une phrase complète."
>>> a="Une chaîne de caractères"
>>> a[0] ; a[5] ; a[-1]
'U'
'h'
's'
>>> a[::2]
'Uecan ecrcèe'
```

Attention, les chaînes de caractères sont *immuables* : une fois créées, on ne peut pas les modifier, ou leur rajouter des éléments. Il faut créer une nouvelle chaîne qu'on peut éventuellement réaffecter à la même variable.

```
>>> a='rateau'
>>> a[0]='b'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> a='b'+a[1:]
>>> a
'bateau'
```

La table des caractères ASCII. Les 128 caractères de la table ASCII standard comprennent les lettres minuscules et majuscules non accentuées, les chiffres de 0 à 9, à peu près tous les caractères d'un clavier QWERTY (pas de lettres accentuées !), et des caractères d'espacement [10]. À chaque caractère est associé un code entre 0 et 127 (donc représentable sur 7 bits). En Python, les fonctions `ord` et `chr` permettent de passer d'un caractère à son numéro et réciproquement.

```
>>> ord('a')
97
>>> chr(97+25)
'z'
>>> ord('0')
48
>>> chr(48+9)
'9'
>>> ord('A')
65
>>> chr(65+25)
'Z'
```

On notera que les lettres minuscules (respectivement les lettres majuscules, les chiffres) sont codés par des numéros contigus, avec `'a'` (respectivement `'A'`, `'0'`) ayant le plus petit numéro. Aujourd'hui, l'ASCII a été remplacé par UNICODE, qui l'inclut. Les fonctions `ord` et `chr` fonctionnent en fait avec d'autres caractères, non ASCII.

```
>>> ord('à')
224
>>> chr(201)
'É'
```

Quelques méthodes sur les chaînes de caractères. Comme les listes, les chaînes de caracactères possèdent leurs propres méthodes, qui ne sont pas exigibles mais peuvent faire l'objet de questions aux concours si la documentation est rappelée. On en donne quelques-unes ici, les deux premières seront celles qui nous serviront le plus. Comme les chaînes de caractères sont immuables, les méthodes ne modifient jamais la chaîne mais renvoie un nouvel objet.

méthode	description
`s.split(c)`	sépare la chaîne `s` autour du caractère `c`. Le résultat est une liste de chaînes de caractères. Par ex. `"une petite chaine".split(" ")` donne `["une", "petite", "chaine"]`.
`s.join(L)`	l'inverse de l'opération précédente. Par ex. `"-".join(["a", "bc"])` donne `"a-bc"`.
`s.count(c)`	compte le nombre d'occurences du caractère `c` dans `s`.
`s.find(c)`	indique la première position du caractère `c` dans `s`, ou -1 s'il n'y est pas.
`s.rjust(n)`	rajoute à la fin de `s` autant d'espaces que nécessaire pour atteindre n caractères. Avec `s.rjust(n,c)`, on remplit avec le caractère `c`. De même avec `ljust` pour remplir au début (right/left).
`s.replace(s1,s2)`	remplace toutes les apparitions de la chaîne `s1` dans `s` par la chaîne `s2`.

Caractères particuliers. Si la chaîne doit contenir un des caractères `'` ou `"` cela fournit un critère pour choisir l'une ou l'autre manière de l'écrire : `'"Oui", dit-il'` ou `"C'est exact"`. Une autre manière d'éviter les problèmes avec le caractère d'encadrement consiste à l'inhiber par l'emploi de \, comme dans la chaîne `'Il ajouta: "c\'est exact !"'`. L'encadrement par de triples guillemets ou de triples apostrophes permet d'indiquer des chaînes qui s'étendent sur plusieurs lignes, et peut servir à ne pas trop réfléchir si la chaîne doit contenir les caractères `'` ou `"`.

```
a="""Cette chaîne s'étend
sur

plusieurs lignes. C'est "beau"."""
```

L'affichage basique d'une telle chaîne montre bien que les fins de ligne, représentées par le signe `\n`, ont été conservées. L'affichage à l'aide de la fonction `print` fournit le résultat attendu :

10. La table complète se trouve par exemple ici : `http://www.table-ascii.com/`.

```
>>> a
'Cette chaîne s'étend\nsur\n\nplusieurs lignes. C'est "beau".'
>>> print(a)
Cette chaîne s'étend
sur

plusieurs lignes. C'est "beau".
```

Notez aussi que `\t` sert à encoder les tabulations. Le backslash étant lui-même un caractère spécial, on le fera précéder d'un autre backslash s'il doit figurer dans la chaîne. Par exemple :

```
>>> s="\\t est une tabulation, \\n est un saut de ligne"
>>> print(s)
\t est une tabulation, \n est un saut de ligne
```

Chaînes de caractères comme commentaires. Pour commenter son code, on peut utiliser le caractère dièse `#` : tout ce qui suit sur la même ligne est ignoré. Lorsqu'on envoie une suite d'instructions de l'éditeur vers la console, une chaîne de caractères sera perçue comme telle, mais seule elle n'a aucune incidence sur le reste du programme, tout comme une expression quelconque comme `10` ou `2+3`. On peut donc utiliser les triples quotes pour commenter facilement un morceau de code s'étendant sur plusieurs lignes.

Conversion. `"42"` est une chaîne de caractères, pas un entier. Ainsi `"42"+5` n'a *aucun* sens[11]. `int`, `float`, `str`,... permettent de convertir un type en un autre.

1.4.4 Dictionnaires

Un dictionnaire est un objet permettant de stocker des couples (clé, valeur), chaque clé ne pouvant être présente que dans un seul couple. Un dictionnaire au sens usuel en est un exemple, où les clés sont les mots de la langue et les valeurs les définitions de ces mots. Un autre exemple serait celui d'un annuaire : à partir d'un couple (nom, prénom) d'une personne (la clé), on souhaite retrouver son numéro de téléphone (la valeur).

Syntaxe. Un dictionnaire s'écrit entre accolades, les couples (clé, valeur) indiqués sous la forme `clé:valeur`, séparés par des virgules. Par exemple :

```
>>> D={'a': 4, 0:5, (1,2):"coucou"}
```

Comme on le voit, il n'y a aucune contrainte d'homogénéité sur les clés ou les valeurs.

Opérations sur un dictionnaire. Le plus souvent, on partira d'un dictionnaire vide pour lui rajouter les éléments un à un. Un dictionnaire vide s'obtient via `dict()` ou simplement `{}` :

```
>>> dict(), {}
({}, {})
```

L'accès à un élément `e` à partir de sa clé `k` dans un dictionnaire `D` se fait via `D[k]`, par exemple avec le dictionnaire précédent :

```
>>> D['a'], D[(1,2)]
(4, 'coucou')
```

Si la clé `k` n'est pas présente, l'opération `D[k]` produit une erreur.

```
>>> D['b']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'b'
```

Tester si une clé `k` est présente se fait via `k in D` (ou `k in D.keys()`) :

11. Par contre, `"42" * 5` vaut `"4242424242"`.

```
>>> 'b' in D, 'a' in D
(False, True)
```

L'ajout d'un nouveau couple `(k,e)` ou la modification de l'élément `e` associé à une clé `k` se fait de la même manière que l'accès, via `D[k]=e`.

```
>>> D['b']=5 ; D[(1,2)]="coucoubis"
>>> D
{'a': 4, 0: 5, (1, 2): 'coucoubis', 'b': 5}
```

Avertissement : on ne peut pas utiliser un objet mutable (comme une liste, ou un dictionnaire, par exemple) comme clé d'un dictionnaire. Il faut se restreindre à des objets immuables (comme les entiers, les flottants, les chaînes de caractères, les tuples de tels éléments, etc...)

Quelques méthodes et fonctions sur les dictionnaires. Voici quelques méthodes, sans exhaustivité, permises sur les dictionnaires. Dans la suite, `D` désigne un dictionnaire.

méthode	description
`len(D)`	renvoie le nombre de couples (clé, élément) stockés dans `D`.
`D.pop(k)`	supprime le couple de clé `k` du dictionnaire, si un tel couple est présent.
`D.keys()`	renvoie les clés présentes dans le dictionnaire. L'objet renvoyé est un itérable, on peut donc utiliser une syntaxe du type `for k in D.keys(): ...`. Écrire simplement `for k in D` est équivalent.
`D.copy()`	renvoie une copie du dictionnaire.

Un mot sur la complexité. Le programme officiel impose d'utiliser en première année les dictionnaires comme des boîtes noires, sans indication sur la manière dont ils sont implémentés. On pourra considérer que toutes les opérations (création, ajout ou modification, suppression) se font en temps constant. Le lecteur voulant aller plus loin pourra consulter l'article Wikipédia sur les tables de hachage [12], qui est la méthode la plus répandue pour implémenter un dictionnaire, et notamment en Python, ou encore consulter le chapitre dédié du polycopié.

1.5 Fonctions

Les fonctions sont d'une importance capitale en informatique, et plus prosaïquement quasiment toutes les questions des sujets de concours demandent d'écrire ou d'examiner des fonctions.

1.5.1 Motivation

Imaginons que l'on veuille calculer $\sum_{k=0}^{n} 2^k$, $\sum_{k=0}^{n} 3^k$,... C'est-à-dire des sommes de la forme $\sum_{k=0}^{n} x^k$, pour plusieurs x et n. On veut donc calculer plusieurs valeurs de la fonction de plusieurs variables $(x, n) \mapsto \sum_{k=0}^{n} x^k$. L'idéal serait de définir cette fonction, ce qu'on peut faire :

```
def f(x,n):
    s=0
    for k in range(n+1):
        s+=x**k
    return s
```

```
>>> f(2, 5)
63
>>> f(3, 4)
121
```

1.5.2 Notions et syntaxe de base

Une fonction en informatique est une séquence d'instructions, dépendant de paramètres d'entrée (appelés *arguments*), et retournant un résultat.

Deux points de vue, souvent complémentaires, permettent de préciser ce qu'est une fonction :

— c'est une séquence d'instructions qui permet de réaliser un calcul précis, que l'on peut utiliser plusieurs fois.

— c'est une brique de base d'un problème plus complexe.

La structure générale d'une déclaration de fonction en Python se fait avec le mot-clef `def` de la façon suivante :

12. `https://fr.wikipedia.org/wiki/Table_de_hachage`

Déclaration d'une fonction
```
def nom_fonction(a_1,a_2,..,a_k): # nom_fonction: nom de la fonction, a_1,..,a_k : arguments
    """ Description de l'action de la fonction """ # spécification de la fonction
    instruction 1
    instruction 2
        ....
    instruction p
# ici, on est hors de la définition de la fonction.
```

— La première ligne `def nom_fonction(a_1,...,a_k)` est l'en-tête de la fonction. Les éléments `a_1,...,a_k` sont des identificateurs appelés *arguments formels* de la fonction et `nom_fonction` est le nom de la fonction. Pour une fonction ne prenant pas d'arguments, on écrit simplement `def fonction():`, les parenthèses étant indispensables. Le nom de la fonction est un identificateur qui suit les mêmes règles que les identificateurs de variables (d'ailleurs la définition d'une fonction n'est rien d'autre qu'une affectation : celle d'un objet de type `function` à une variable).

— La seconde (qui est facultative et peut être sur plusieurs lignes) est une chaîne de caractères appelée *chaîne de documentation* décrivant la fonction : ce que doivent respecter les paramètres passés en entrée, l'action effectuée et la nature du résultat retourné.

— La suite d'instructions est le corps de la fonction.

— Le retour à une indentation au même niveau que `def` marque la fin de la fonction, tout ce qui est à ce niveau ne fait plus partie de la fonction.

Attention, le rôle d'une définition de fonction n'est pas d'exécuter les instructions qui en composent le corps, mais uniquement de mémoriser ces instructions en vue d'une exécution ultérieure (facultative !), provoquée par une expression faisant *appel* à la fonction. Par exemple, définir la fonction qui suit ne provoque pas d'erreur.

```
def fonction_erreur():
    print(1/0)
```

Évidemment, l'appeler en provoque une !

Appel d'une fonction. L'appel de la fonction `nom_fonction` présentée ci-dessus se fait par :

`nom_fonction(e_1,e_2,...,e_k)`,

où `e_1,...,e_k` sont des *expressions*. Elles forment les *arguments effectifs* de l'appel à la fonction : lors de l'appel `e_1` (respectivement `e_2,...,e_k`) est évaluée, puis le résultat est affecté à `a_1` (respectivement `a_2,...,a_k`) juste avant l'exécution du corps de la fonction. Tout se passe comme si l'exécution de la fonction commençait par la suite d'instructions d'affectation

```
a_1=e_1
a_2=e_2
...
a_k=e_k
```

et se poursuivait avec

```
instruction 1
instruction 2
    ...
instruction p
```

L'instruction return. Le corps de la fonction comprend bien souvent une ou plusieurs instructions de la forme `return resultat`, où `resultat` est une expression. Lors du déroulement du corps de la fonction, si une telle instruction est rencontrée, alors l'expression `resultat` est evaluée, l'exécution de la fonction est interrompue et la valeur de `resultat` prend la place de `nom_fonction(e_1,e_2,...,e_k)` là où la fonction a été appelée. Prenons un exemple simple :

```
def incremente(x):
    return x+1
```

Si on exécute `a=incremente(6-2)+9`, alors :

— `6-2` s'évalue en 4.

— `x` prend la valeur 4 dans la fonction.

— `x+1` s'évalue en 5, qui est retourné par la fonction.

— `incremente(6-2)` est donc remplacée par 5.

— `incremente(6-2)+9` s'évalue donc en 14, qui est ensuite affecté à la variable `a`.

Le type « rien ». Si la fonction ne comprend pas de `return` ou qu'aucun `return` n'est rencontré lors de l'exécution, la fonction ne renvoie rien. On dit parfois qu'elle fonctionne uniquement par *effets de bord*[13], et c'est là une différence fondamentale avec les fonctions en mathématiques. Il y a un type pour ça : `NoneType`, qui comporte une unique valeur : `None`. La fonction suivante ne prend aucun paramètre en entrée[14], se contente d'afficher 4 à l'écran, et ne renvoie rien : elle agit par effets de bord.

```
def affiche4():
    print(4)
```

Lors de l'évaluation de `a=affiche4()`, 4 est affiché à l'écran, on sort de la fonction (car on est arrivé en bas !) et `a` prend la valeur `None`. Notez que `return` seul (sans rien derrière) est souvent fort utile pour interrompre une fonction. La fonction en question renvoie alors `None`.

Différence entre `print` et `return`. Une erreur classique est de confondre `print` et `return`. `return` est une instruction de sortie de fonction, `print` est une fonction Python, qui affiche l'argument passé en entrée à l'écran et qui ne renvoie rien. Lorsqu'on teste une fonction dans la console, on ne voit pas vraiment la différence mais elle est pourtant significative : une fonction sans `return` ne renvoie rien !

Prenons l'exemple de la fonction suivante, qui calcule un PGCD.

```
def PGCD(a,b):
    """Avec a et b>0, renvoie le PGCD de a et b."""
    while b!=0:
        a,b=b,a%b
    return a
```

Éxecuté dans la console, on ne verrait pas de grande différence entre cette fonction et la même avec `print` à la place de `return`. Si maintenant, on veut utiliser cette fonction pour calculer un PPCM[15], on définit alors la fonction suivante :

```
def PPCM(a,b):
    """Avec b>0, renvoie le PPCM de a et b."""
    return a*b//PGCD(a,b)
```

L'appel `PPCM(6,4)` produit bien 12. Avec `print(a)` au lieu de `return a` dans la fonction `PGCD`, l'expression `PGCD(6,4)` est remplacée par `None`, et l'évaluation `a*b//PGCD(a,b)` produit une erreur :

```
>>> PPCM(6,4)
2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in PPCM
TypeError: unsupported operand type(s) for //: 'int' and 'NoneType'
```

Remarquez que l'erreur est très explicite : l'opérateur `//` ne peut faire d'opération entre un entier (de type `int`, obtenu ici par l'évaluation de `6*4`) et un objet de type `NoneType`, c'est-à-dire `None`. Vous vous posez peut-être la question : mais qu'est-ce que le 2 qui traîne ? Il provient de notre fonction `PGCD`, qui a été appelée par `PPCM`, s'est déroulée sans accroc, et a bravement affiché à l'écran le PGCD de a et b, comme demandé puisqu'on a utilisé `print`.

13. Qui est une mauvaise traduction de l'anglais « side effects »...
14. Oui, c'est possible !
15. suivant la formule bien connue $\text{PGCD}(a,b) \times \text{PPCM}(a,b) = ab$.

Chaîne de documentation. Il est important de préciser ce que fait une fonction lorsqu'on l'écrit. La chaîne de documentation ainsi que l'en-tête, sont accessibles lorsqu'on tape `help(nom_fonction)` :

```
>>> help(PPCM)
Help on function PPCM in module __main__:

PPCM(a, b)
    Avec b>0, renvoie le PPCM de a et b.
```

Et cela marche aussi avec les fonctions Python.

```
>>> from math import *
>>> help(log2)
Help on built-in function log2 in module math:

log2(...)
    log2(x)

    Return the base 2 logarithm of x.
```

Il existe certaines règles qui régissent la rédaction des chaînes de documentation, mais il est inutile de s'embêter avec ça[16]. Mettez une chaîne de caractères après l'en tête qui explique un peu ce que fait votre fonction et ce sera déja très bien. Cette chaîne est bien sûr facultative.

1.5.3 Variables locales et globales

Dans les fonctions `PGCD` et `PPCM` de la sous-section précédente, les variables `a` et `b` ont été utilisées à peu près partout : comme paramètres d'appel des deux fonctions mais également comme variables pour calculer le PGCD puisque par exemple la valeur de retour de `PGCD` est `a`. Pourtant, Python ne se mélange pas les pinceaux et produit le résultat auquel on s'attend, parce que `a` et `b` dans les fonctions sont des variables *locales*. Si elle n'est pas explicitement déclarée globale, toute variable apparaissant dans une fonction comme membre gauche d'une affectation est locale à cette fonction. Cela signifie que sa *portée* est réduite à la fonction, qu'elle est créée à chaque fois que la fonction est appelée et « détruite » à la fin de chaque exécution. Par exemple, supposons que la variable `a` n'ait pas été affectée mais la fonction `PGCD` précédente déclarée :

```
>>> PGCD(8,3)
1
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

On voit bien que `a` est inconnu en dehors de la fonction `PGCD`. Les paramètres des fonctions se comportent comme des variables locales : on peut les modifier mais cette modification est interne à la fonction :

```
>>> a=5 ; b=7
>>> p=PGCD(a,b)
>>> print(a)
5
```

On parle de *passage par valeurs* : lors de l'appel à PGCD, les valeurs de `a` et `b` sont recopiées et les variables `a` et `b` de la fonction ne sont pas les mêmes que les variables `a` et `b` qu'on a définies en dehors de la fonction.

A l'opposé de cela, les variables globales sont des variables créées à l'extérieur de toute fonction. Elles existent depuis le moment de leur création, jusqu'à la fin de l'exécution du programme. Une variable globale peut être utilisée à l'intérieur d'une fonction si elle n'est pas le membre gauche d'une affectation ou le nom d'un paramètre. Par exemple, la fonction suivante ajoute le contenu de la variable (nécessairement globale !) `n` au paramètre `x` et renvoie le résultat de l'addition.

```
def ajoute_n(x):
    return x+n
```

16. Dans les exemples qui précèdent, je ne les ai pas respectées !

Si `n` n'est pas défini au moment de l'appel à `ajoute_n`, on obtient bien sûr une erreur. En général, on réserve l'usage des variables globales aux constantes d'un problème. Si on veut réaliser une simulation en cinétique des gaz, on pourra commencer le script de simulation par `R=8,3144621` (la constante universelle des gaz parfaits), et on pourra librement utiliser `R` dans n'importe quelle fonction [17].

Pour pouvoir affecter à une variable globale dans une fonction, cette variable doit faire l'objet d'une déclaration explicite comme variable globale de la forme `global variable_globale`. Par exemple, imaginons que l'on veuille maintenir un compteur, qui contient le nombre de fois où une certaine fonction a été appelée. On peut donc utiliser une variable globale `nombre_appels`, qu'on incrémentera de 1 dans l'appel de la fonction. La fonction en question commencerait donc par :

```
global nombre_appels
nombre_appels+=1
```

Ainsi, une variable est locale sauf si :

— elle est explicitement déclarée globale ;

— ou bien elle est utilisée sans être affectée.

En général, on n'utilisera pas de variables globales, sauf si la situation le justifie. À l'opposé des fonctions qui fonctionnent par effets de bord, il y a les fonctions pures : elles n'agissent pas sur l'environnement (même par affichage !) et n'en dépendent pas (elles n'ont pas recours à des variables globales). Par exemple, les fonctions `PGCD` et `PPCM` définies plus haut sont pures.

1.5.4 Références vers des objets mutables

Le point abordé maintenant est un peu subtil, mais n'est finalement pas très dur à comprendre. Considérons la fonction suivante :

```
def ajoute_zero(T):
    T.append(0)
```

et regardons l'effet de la fonction sur une liste quelconque :

```
>>> L=[3,4,7]
>>> ajoute_zero(L)
>>> print(L)
[3, 4, 7, 0]
```

On remarque que la fonction a eu un effet *global* sur la liste `L` (un effet de bord !). Lorsqu'on déclare une liste (ici `L`), l'identificateur est en fait une référence (on parle aussi de pointeur) vers l'emplacement mémoire occupé par la liste. Lorsqu'on passe une liste en paramètre d'une fonction, c'est la référence qui est utilisée (en particulier, les éléments de la liste ne sont pas recopiés ailleurs). La *méthode* `append` modifie ce qu'il y a en mémoire, et c'est pour cette raison que l'effet est visible en dehors de la fonction.

Le comportement est sensiblement différent si l'on définit `ajoute_zero` comme ceci :

```
def ajoute_zero(T):
    T=T+[0]
```

Si on exécute la même suite d'instructions que précédemment, on obtient :

```
>>> L=[3,4,7]
>>> ajoute_zero(L)
>>> L
[3, 4, 7]
```

Ici, on commence par créer la liste `T+[0]` en recopiant ailleurs en mémoire les éléments de `T` et en y ajoutant 0. Après l'affectation `T=`, la référence `T` pointe maintenant vers cette nouvelle liste. Cette nouvelle référence est locale à la fonction : la liste `[3,4,7,0]` n'existe que dans la fonction et est donc « détruite » en fin de fonction.

Remarquez qu'avec `T+=[0]`, on obtient le même comportement qu'avec `append`. C'est parce que sur les listes, `+=` est similaire à la méthode `extend` qui s'utilise comme suit : `T.extend(T2)` rajoute à la fin de la liste `T` le contenu de la liste `T2`, la référence à `T` n'étant pas modifiée (voir les méthodes sur les listes).

17. Il est quand même plus raisonnable de l'appeler `constante_gaz_parfaits`, comme ça on est sûr de ne pas se servir par mégarde d'une variable locale du même nom !

1.5.5 Une fonction : un objet comme les autres

Fonctions en paramètres d'autres fonctions. On n'a pas encore parlé du type associé à une fonction. Sans surprise, il s'agit du type `function`. Remarquez que lorsqu'on demande à Python d'afficher une fonction, il renvoie une suite de caractères bizarres :

```
>>> type(PPCM)
<class 'function'>
>>> print(PPCM)
<function PPCM at 0x7fd55b535268>
```

En fait, `0x7fd55b535268` est l'emplacement mémoire occupé par la fonction : il faut bien la stocker quelque part ! Le préfixe `0x` indique que l'emplacement mémoire est codé en hexadécimal.

Les fonctions sont des objets comme les autres en Python. En particulier, il est possible de les passer comme paramètres d'autres fonctions. Généralisons un peu l'exemple vu précédemment : on peut écrire une fonction qui calcule $\sum_{k=0}^{n} g(x^k)$ pour tous paramètres x, n et g :

```
def f(x,n,g):
    s=0
    for k in range(n+1):
        s+=g(x**k)
    return s
```

```
>>> f(5, 5, cos)
0.984965422678516
>>> f(3, 4, lambda x: x)
121
```

Dans ces exemples, `cos` est la fonction cosinus classique, qui se trouve dans le module `math`. Remarquez que `lambda` permet de définir une fonction sans lui donner un nom : la syntaxe est `lambda variable: expression`. On a ici défini la fonction identité, et on retrouve le même résultat qu'avec la version précédente de la fonction f.

Fonctions locales à d'autres fonctions. De la même façon que les variables définies dans une fonction sont locales, on peut définir une fonction locale à une autre. Une variante (un peu stupide) de la fonction précédente est-celle ci :

```
def f(x,n,g):
    def terme(k):
        return g(x**k)
    s=0
    for k in range(n+1):
        s+=terme(k)
    return s
```

Ici, la fonction `terme` est locale à la fonction `f` : elle n'est pas définie en dehors de la fonction `f`. Remarquez que `x` est utilisé comme « variable globale » de la fonction `terme` : c'est normal et tout à fait légitime. Python va chercher en dehors de la fonction ce qu'il ne connaît pas. Même si `x` est également une variable définie en dehors de la fonction, on considère le « plus petit contexte définissant `x` » : ici celui de la fonction `f`.

Arguments optionnels. Dans la définition d'une fonction, on peut déclarer certains arguments comme optionnels en leur donnant une valeur par défaut, utiles s'ils ne sont pas précisés lors de l'appel de la fonction. C'est une possibilité qu'on n'utilisera pas pour nos propres fonctions, mais utile pour comprendre beaucoup de fonctions internes à Python. Prenons un exemple minimaliste, avec l'argument optionnel `y`.

```
def f(x,y=1)
    return x+y
```

```
>>> f(0)
1
>>> f(0,y=4)
4
```

Chapitre 2

Entiers dans différentes bases

Ce court chapitre est prétexte pour écrire quelques boucles `for` et `while`, et commencer à utiliser la structure de liste. Il donne une première idée de comment sont représentés les entiers en interne dans un ordinateur, il sera complété par un chapitre un peu plus poussé où l'on explique la représentation des entiers négatifs et également la représentation des nombres flottants.

2.1 Écriture dans une base

2.1.1 Rappels sur la base 10

Considérons un nombre entier strictement positif, par exemple $N = 432$. Alors, N s'écrit $N = 4 \times 10^2 + 3 \times 10^1 + 2 \times 10^0$. Cette écriture se généralise à tout entier, par le théorème suivant :

Théorème 2.1. *Soit N un entier strictement positif, alors il existe n strictement positif et des entiers $a_0, \dots, a_{n-1}$ tels que :*

- *pour tout i dans $\{0, \dots, n-1\}$, a_i appartient à $\{0, \dots, 9\}$, ce qu'on note $(a_0, \dots, a_{n-1}) \in [\![0, 9]\!]^n$,*
- *$a_{n-1} \neq 0$,*

et $N = a_{n-1} \times 10^{n-1} + a_{n-2} \times 10^{n-2} + \cdots + a_0 \times 10^0 = \sum_{k=0}^{n-1} a_k \times 10^k$. De plus, l'entier n et les entiers (a_i) sont uniques.

2.1.2 Généralisation à une base quelconque

L'écriture précédente se généralise aisément à une base quelconque :

Théorème 2.2. *Soit N un entier strictement positif, et b un entier positif supérieur ou égal à 2. Alors il existe n strictement positif et des entiers $a_0, \dots, a_{n-1}$ tels que :*

- *pour tout i dans $\{0, \dots, n-1\}$, a_i appartient à $\{0, \dots, b-1\}$, ce qu'on note $(a_0, \dots, a_{n-1}) \in [\![0, b-1]\!]^n$,*
- *$a_{n-1} \neq 0$,*

et $N = a_{n-1} \times b^{n-1} + a_{n-2} \times b^{n-2} + \cdots + a_0 \times b^0 = \sum_{k=0}^{n-1} a_k \times b^k$. De plus, l'entier n et les entiers (a_i) sont uniques.

On note un tel entier N dans la base b comme suit : $N = \overline{a_{n-1}a_{n-2}\cdots a_1a_0}^{\,b}$. On va prouver ce théorème dans la suite. Voyons d'abord quelques exemples, par exemple l'écriture de 17 dans toutes les bases entre 2 et 9 :

$$\begin{aligned}
17 &= 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 &&= \overline{10001}^{\,2} \\
&= 1 \times 3^2 + 2 \times 3^1 + 2 \times 2^0 &&= \overline{122}^{\,3} \\
&= 1 \times 4^2 + 0 \times 4^1 + 1 \times 2^0 &&= \overline{101}^{\,4} \\
&= 3 \times 5^1 + 2 \times 3^0 &&= \overline{32}^{\,5} \\
&= 2 \times 6^1 + 5 \times 6^0 &&= \overline{25}^{\,6} \\
&= 2 \times 7^1 + 3 \times 7^0 &&= \overline{23}^{\,7} \\
&= 2 \times 8^1 + 1 \times 8^0 &&= \overline{21}^{\,8} \\
&= 1 \times 9^1 + 8 \times 9^0 &&= \overline{18}^{\,9}
\end{aligned}$$

2.1.3 Généralisation à des bases supérieures à 10. Hexadécimal

Pour représenter des nombres dans une base supérieure à 10, il est nécessaire d'introduire de nouveaux symboles pour exprimer les chiffres entre 10 et $b-1$. En particulier, un exemple important en informatique est la base 16, appelée hexadécimale. Pour représenter les chiffres manquants, on utilise les lettres de A à F :

lettre	A	B	C	D	E	F
signification	10	11	12	13	14	15

Avant de passer à la preuve du théorème 2.2, rappelons un résultat essentiel d'arithmétique : l'existence et l'unicité du reste dans une division euclidienne.

Théorème 2.3. *Soit N et M deux entiers, avec $M>0$. Alors il existe deux entiers q et r tels que :*
- $N = qM + r$,
- $0 \leq r \leq M-1$.

De plus, le couple (q,r) est unique.

Démonstration. L'ensemble $E = \{a \in \mathbb{Z} \mid N - Ma \geq 0\}$ est un sous-ensemble de $\mathbb{Z}$. Il est non vide car tout entier inférieur à $\frac{N}{M}$ est dans E. Il est de plus borné supérieurement car tout entier strictement supérieur à $\frac{N}{M}$ n'est pas dans E. Ainsi, E possède un plus grand élément, qu'on note q. Posons alors $r = N - Mq \geq 0$. Si r était supérieur ou égal à M, alors $(q+1)$ serait dans E ce qui est absurde. Ainsi l'existence du couple (q,r) est démontrée. Pour l'unicité, considérons un autre couple (q',r') satisfaisant les mêmes hypothèses. On a alors $M(q-q') = r'-r$. Or $-(M-1) \leq r'-r \leq M-1$, donc $r'-r$ est un multiple de M strictement supérieur à $-M$ et strictement inférieur à M, donc nul. On en déduit l'unicité de r, puis de q. □

Preuve du théorème 2.2. On montre par récurrence l'existence et l'unicité d'une telle écriture. Fixons $b \geq 2$, et pour $N \geq 1$, posons $P(N)$ la propriété « N admet une écriture comme dans le théorème, et elle est unique ».

Initialisation : $P(1), \dots, P(b-1)$ sont vraies car pour N un des entiers parmi $1, \dots, b-1$, l'écriture $N = \overline{N}^{\,b}$ convient. De plus, si $N = \overline{a_{n-1}a_{n-2}\cdots a_1a_0}^{\,b}$ est une autre écriture, comme $a_{n-1} > 0$, nécessairement $n = 1$ car N est strictement inférieur à b. Ainsi, $N = \overline{N}^{\,b}$ est la seule écriture convenable.

Hérédité : Soit $N \geq b$, et supposons $P(M)$ pour tout entier $M \in [\![1, N-1]\!]$. Soit (q,r) le quotient et le reste dans la division euclidienne de N par b. Puisque $N \geq b$, q est un entier strictement positif, inférieur strictement à N. On peut donc lui appliquer l'hypothèse de récurrence : il existe un entier $p \geq 1$ tel que q s'écrive $\overline{c_{p-1}\cdots c_0}^{\,b}$, avec $c_i \in [\![0, b-1]\!]$ et $c_{p-1} \neq 0$. Alors, en posant $n = p+1$,

$$N = bq + r = b \times \left(\sum_{k=0}^{p-1} c_i b^i\right) + r = \left(\sum_{k=1}^{n-1} c_{i-1} b^i\right) + r = \overline{c_{n-2}c_{n-3}\cdots c_0 r}^{\,b}$$

on voit que le n-uplet $(a_{n-1}, \dots, a_0) = (c_{n-2}, \dots, c_0, r)$ vérifie les conditions du théorème 2.2. De plus cette écriture est unique : le dernier chiffre de N est nécessairement le reste de la division euclidienne de N par b, soit r. Les autres chiffres sont donnés par l'écriture de q, qui est unique par récurrence. Ainsi, par principe de récurrence, $P(N)$ est vraie.

Conclusion : $P(N)$ est vraie pour tout $N \geq 1$: l'existence et l'unicité sont démontrées. □

Même si la preuve est un peu rébarbative, son application donne immédiatement un algorithme de changement de base, qu'on va voir dans la sous-section suivante. Avant ça, un petit point culture.

2.1.4 Histoire

Voici une petite présentation non exhaustive des différentes bases ayant été utilisées. Aujourd'hui, on (l'humanité) utilise la base 10 [1]. Ça n'a pas toujours été le cas : les égyptiens, les mayas, les babyloniens, mésopotamiens et d'autres ont utilisé les bases 20, 24 et 60. Les mésopotamiens n'utilisaient cependant pas 60 symboles différents : chaque « chiffre » était lui même codé avec un certain nombre de chevrons (chacun comptant pour 10) et de clous (chacun comptant pour une unité). En informatique [2], la base 2 (binaire) apparaît naturellement : le 1 et le 0 correspondent à une tension positive (supérieure à un certain seuil) ou une absence de tension (inférieure à ce seuil) en un point d'un circuit électrique. Comme les écritures en binaire sont plutôt longues (on a vu qu'il fallait 5 chiffres pour représenter 17...), l'idée de raccourcir les écritures en utilisant des bases de la forme 2^k a mené à l'hexadécimal, et plus marginalement à l'octal [3] (base 8). On verra qu'il est très facile de passer du binaire à l'hexadécimal (ou à l'octal) et réciproquement.

1. Probablement parce que les humains ont 10 doigts...
2. Il y a 10 types de personnes... Ceux qui savent compter en binaire, et les autres !
3. Pourquoi les informaticiens confondent toujours Halloween et Noël ? Parce que 31 oct = 25 dec...

2.2 Changement de base

L'entier 1345 (en base 10), s'écrit $\overline{10101000001}^{2}$ en binaire et $\overline{541}^{16}$ en hexadécimal. Pour pouvoir passer d'une base à une autre, il est nécessaire de savoir calculer dans la base de départ, ou bien dans la base d'arrivée. On va voir deux algorithmes correspondant à ces deux situations. On va également voir qu'il est facile de passer du binaire à l'hexadécimal, et réciproquement.

2.2.1 Si l'on sait calculer dans la base de départ

Dans ce qui suit, vous pouvez considérer que la base de départ est la base 10 : il nous faut simplement une base dans laquelle on sait faire une division euclidienne. On ne fera pas explicitement mention de cette base. L'algorithme 2.4 reprend l'idée de la preuve du théorème 2.2.

Algorithme 2.4 : Écriture depuis une base où on sait calculer

Entrées : Un entier $N > 0$ et un entier $b \geq 2$.
Sortie : L'écriture de N dans la base b.
$i \leftarrow 0$;
$M \leftarrow N$;
tant que $M \neq 0$ **faire**
  $(q, r) \leftarrow$ quotient et reste de la division euclidienne de M par b;
  $a_i \leftarrow r$;
  $M \leftarrow q$;
  $i \leftarrow i + 1$;
retourner $(a_{i-1}, a_{i-2}, \ldots, a_0)$

En d'autres termes, on effectue des divisions euclidiennes tant que l'on ne tombe pas sur un quotient nul. La suite des restes fournit les chiffres de l'écriture de N dans la base b, du moins significatif au plus significatif (c'est-à-dire dans l'ordre inverse).

On a utilisé ici une boucle *conditonnelle* « Tant que », qui correspond en Python à une boucle `while`. Tant que la condition est vérifiée (ici $M \neq 0$), on exécute le corps de la boucle. Reprenons l'exemple du nombre 1345 que l'on veut convertir en hexadécimal. On effectue les divisions euclidiennes successives :

$$\begin{aligned} 1345 &= 16 \times 84 + \boxed{1} \\ 84 &= 16 \times 5 + \boxed{4} \\ 5 &= 16 \times \boxed{0} + \boxed{5} \end{aligned}$$

Comme le dernier quotient est zéro, on s'arrête. La suite des restes successifs est 1, 4, 5, que l'on inverse. On obtient bien $1345 = \overline{541}^{16}$.

En Python, on stocke la suite des restes dans une liste. À la fin de l'algorithme, on inverse la liste et on la renvoie. Cela donne le code suivant :

```
def base10ab(N,b):
    L=[] #une liste vide
    M=N
    while M!=0:
        r=M%b #reste dans la division euclidienne
        M=M//b #quotient dans la division euclidienne
        L.append(r) #ajouter un élément à la fin d'une liste
    L.reverse() #on retourne la liste
    return L
```

```
>>> base10ab(17,2)
[1, 0, 0, 0, 1]
>>> base10ab(666,3)
[2, 2, 0, 2, 0, 0]
>>> base10ab(1345,16)
[5, 4, 1]
```

Remarque : en pratique, il est très courant de représenter un nombre dans une base b par la donnée de ses chiffres du moins significatif au plus significatif : par exemple $1345 = \overline{541}^{16}$ peut se représenter en hexadécimal par la liste `[1, 4, 5]`. Cette représentation est assez pratique, car à une liste de chiffres `[a_0, a_1, ..., a_p]` est associée le nombre $\sum_{k=0}^{p} a_k b^k$ dans la base b. On appelle les deux représentations possibles *big-endian* (les chiffres les plus significatifs en premier, on commence par « le gros bout ») et *little endian*[4] (on commence par les moins significatifs,

4. Les termes *big endian* et *little endian* ont été popularisés par Dany-Cohen, en référence aux *Voyages de Gulliver*, le roman de Jonathan Swift où il est question d'un décret visant à décider par quel bout on doit commencer à manger un œuf à la coque, le gros ou le petit.

soit le « petit bout »). L'algorithme précédent donne la représentation *big endian*, pour obtenir la représentation en *little endian*, il suffit de ne pas inverser la liste à la fin de l'algorithme. En interne sur un ordinateur, la représentation utilisée (l'« endianness ») dépend du système d'exploitation. Les deux sont utilisées, mais la représentation « little endian » est la plus répandue.

2.2.2 Si l'on sait calculer dans la base d'arrivée

Ici, on suppose que l'on sait faire les opérations + et × dans la base d'arrivée, que l'on pourra voir comme la base 10. Comment évaluer $N = \overline{a_{n-1}\cdots a_0}^{\,b}$ dans cette base ? À partir de l'écriture $N = \sum_{k=0}^{n-1} a_k b^k$, on voit qu'il suffit d'évaluer les puissances de b (jusqu'à b^{n-1}) dans la base d'arrivée, de multiplier b^k par a_k et de sommer. En supposant que 1 et b sont donnés sans calcul, cela nous fait $2n-3$ multiplications : $n-2$ pour calculer les b^k et $n-1$ pour multiplier chaque couple (a_k,b^k), la multiplication $a_0 \times 1$ étant gratuite. On va voir un algorithme classique qui requiert environ moitié moins de multiplications : l'algorithme de Hörner. Celui-ci repose entièrement sur l'identité suivante :

$$N = \sum_{k=0}^{n-1} a_k b^k = a_0 + b \times (a_1 + b \times (a_2 + b \times (a_3 + \cdots + b \times a_{n-1})) \cdots)$$

Algorithme 2.5 : Écriture dans une base dans laquelle on sait calculer

Entrées : Un entier $b \geq 2$ et un entier N donné par la liste de ses chiffres dans la base b : $N = \overline{a_{n-1}\cdots a_0}^{\,b}$
Sortie : L'évaluation de N dans la base ambiante
$M \leftarrow 0$;
pour chaque *i allant de $n-1$ à 0 par pas de -1* **faire**
  $M \leftarrow b \times M + a_i$;
retourner M

Comme on le voit, l'algorithme 2.5 est particulièrement court. On a utilisé ici une boucle *inconditionnelle* : i prend successivement les valeurs $n-1, n-2, \ldots, 0$. La structure en Python correspondante est `for`. Prenons un exemple : convertissons $N = \overline{6ABC}^{\,16}$ en base 10. Cela consiste à évaluer l'expression $12+16\times(11+16\times(10+16\times 6))$. Allons-y :

$$\begin{aligned} N &= 12 + 16 \times (11 + 16 \times (10 + 16 \times 6)) \\ &= 12 + 16 \times (11 + 16 \times 106) \\ &= 12 + 16 \times 1707 \\ N &= 27324 \end{aligned}$$

Voici une fonction Python qui réalise l'algorithme de Hörner. On suppose que `L` est une liste contenant les entiers $a_{n-1}, \ldots, a_0$ dans cet ordre. Il suffit de parcourir la liste dans l'ordre croissant des indices.

```
def baseba10(L,b):
    n=len(L) #longueur de la liste
    N=0
    for i in range(n): #parcours de la liste
        N=b*N+L[i]
    return N
```

```
>>> baseba10([1,0,0,0,1],2)
17
>>> baseba10([5,4,1],16)
1345
>>> baseba10([6,10,11,12],16)
27324
```

2.2.3 Un cas particulier : l'une des bases est une puissance de l'autre

On a dit plus haut qu'il était facile de passer du binaire à l'hexadécimal et réciproquement. En fait, c'est le cas si l'une des bases est b et l'autre b^ℓ, pour un certain $\ell > 1$.

Prenons tout de suite un exemple : $N = 27324$ s'écrit $\overline{6ABC}^{\,16}$ mais aussi $\overline{110101010111100}^{\,2}$. Comme $16 = 2^4$, il suffit d'écrire la correspondance entre les 16 chiffres hexadécimaux et les chaînes de 4 chiffres en binaire (complétés par des zéros à gauche). La correspondance est la suivante :

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

Ainsi, les 16 bits de l'écriture de $\overline{6ABC}^{16}$ en binaire sont bien donnés par les cases correspondant à 6, A, B et C dans ce tableau, en enlevant le 0 inutile à gauche. Réciproquement, pour passer de la base 2 à la base 16, on regroupe les bits par paquets de 4, en commençant par la droite, et en rajoutant éventuellement des zéros à la gauche du nombre, et on utilise le tableau. Par exemple, $\overline{100101}^2$ s'écrit $\overline{25}^{16}$ car $\overline{0101}^2$ correspond à $\overline{5}^{16}$ et $\overline{0010}^2$ à $\overline{2}^{16}$.

Dans le cas général, il suffit d'établir une correspondance entre les paquets de ℓ chiffres dans la base b et les chiffres dans la base $B = b^\ell$. En effet, soit $N = \sum_{k=0}^{n-1} a_k b^k$ un nombre exprimé dans la base b, avec $a_k \in \{0, \dots, b-1\}$. Quite à ajouter des chiffres nuls au début de la représentation en base b, on suppose que n est un multiple de ℓ, il s'écrit donc $n = \ell \times m$. Alors :

$$\begin{array}{rcll} N & = & \sum_{k=0}^{n-1} a_k b^k & \\ & = & \sum_{i=0}^{m-1} \left(\sum_{j=0}^{\ell-1} a_{j+i\ell}\ b^{j+i\ell} \right) & \text{(on découpe par paquets de } \ell \text{ chiffres)} \\ N & = & \sum_{i=0}^{m-1} B^i \sum_{j=0}^{\ell-1} a_{j+i\ell}\ b^j & \text{(car } B = b^\ell\text{)} \end{array}$$

Comme chaque $a_{j+i\ell}$ est entre 0 et $b-1$ (ce sont les chiffres de N dans la base b), chaque somme $A_i = \sum_{j=0}^{\ell-1} a_{j+i\ell} b^j$ est entre 0 et $\sum_{j=0}^{\ell-1}(b-1)b^j = b^\ell - 1 = B - 1$. Autrement dit, les A_i sont des chiffres dans la base $B = b^\ell$. On obtient bien l'écriture de N dans la base B en regroupant les chiffres de N dans la base b par paquets de ℓ à partir de la droite, et en faisant une transcription à l'aide d'une table de la forme :

base b	base $B = b^\ell$
$\overline{0\cdots00}^b$	0
$\overline{0\cdots01}^b$	1
$\vdots$	$\vdots$
$\overline{10\cdots0}^b$	$b^{\ell-1}$
$\vdots$	$\vdots$
$\overline{(b-1)\cdots(b-1)(b-1)}^b$	$b^\ell - 1 = B - 1$

Le premier chiffre A_{m-1} est bien non nul, pour peu qu'on ait rajouté tout juste le nombre de zéros à gauche (éventuellement aucun) nécessaire pour que le nombre de chiffres de N dans la base b devienne un multiple de ℓ.

Réciproquement, si on part d'un nombre dans la base B, il suffit de faire le processus inverse pour retrouver un nombre dans la base b, quite à supprimer les chiffres nuls à gauche obtenus si le premier chiffre de N dans la base B est strictement inférieur à $b^{\ell-1}$.

Chapitre 3

Analyse d'algorithmes

Introduction

Le but de ce chapitre est d'étudier de manière théorique les algorithmes. On va donner les outils permettant de répondre aux trois questions suivantes :

— l'algorithme s'arrête-t-il un jour ?

— est-ce qu'il fait bien ce qu'il est sensé faire ? Autrement dit, est-il correct ?

— combien de temps met-il à s'exécuter ?

Le premier point s'appelle la *terminaison de l'algorithme*, le deuxième sa *correction* et le dernier sa *complexité*. Revoyons la notion d'algorithme en informatique.

Définition 3.1. *Un algorithme est une fonction qui prend des données en argument, effectue une séquence finie non ambiguë d'instructions, et renvoie un résultat.*

Étendons un peu cette définition en donnant une liste de points caractérisant un algorithme, par Donald Knuth [1] :

— finitude : « Un algorithme doit toujours se terminer après un nombre fini d'étapes. »

— définition précise : « Chaque étape d'un algorithme doit être définie précisément, les actions à transposer doivent être spécifiées rigoureusement et sans ambiguïté pour chaque cas. »

— entrées : « des quantités qui lui sont données avant qu'un algorithme ne commence. Ces entrées sont prises dans un ensemble d'objets spécifié. »

— sorties : « des quantités ayant une relation spécifiée avec les entrées. »

— rendement : « [...] toutes les opérations que l'algorithme doit accomplir doivent être suffisamment basiques pour pouvoir être en principe réalisées dans une durée finie par un homme utilisant un papier et un crayon. »

Pour décrire un algorithme, on lui donne en général un nom, on précise quels sont les paramètres (les entrées) et le résultat (les sorties) qu'il est sensé renvoyer. On précise aussi de quelle manière il agit sur son *environnement* : modification de la mémoire, affichage éventuel à l'écran, etc... Tout ceci constitue la *spécification de l'algorithme*. Dans nos algorithmes, outre les opérations d'affectations, d'entrée/sortie et de manipulations des variables, on peut trouver des blocs simples :

— boucles `for` ;

— boucles `while` ;

— blocs conditionnels `if`, `elif`,...,`else`.

Ce découpage en blocs simples est essentiel.

3.1 Terminaison

Pour montrer qu'un algorithme termine quel que soit le jeu de paramètres passé en entrée respectant la spécification, il faut montrer que chaque bloc élémentaire décrit ci-dessus termine ! Or, les boucles `for` et les instructions conditionnelles terminent forcément [2]. Le seul souci pourrait venir d'une boucle `while`.

1. L'un des meilleurs informaticiens de tous les temps ! La liste proposée est tirée de Wikipédia.

2. En fait c'est faux. Par exemple, la boucle `for x in L: L.append(x)` ne s'arrête pas à partir d'une liste non vide. Mais nous allons sagement éviter ce genre de considération, et surtout éviter de modifier un objet sur lequel on itère !

3.1.1 Quelques exemples, exponentiation rapide

Considérons par exemple le code suivant :

```
while n!=0:
   n-=1
```

Si, avant la boucle `while`, la variable `n` contient un entier positif, cette boucle s'arrêtera au bout de n étapes. Par contre, si elle contient un entier strictement négatif, c'est la catastrophe : `n` prendra une infinité de valeurs, toutes strictement négatives.

Prenons un exemple un peu plus intéressant et concret : le calcul de la puissance. Pour `x` un entier (ou un flottant), et n un entier naturel, on peut partir de $y = 1$ et multiplier n fois y par x. C'est l'idée du code suivant.

Algorithme d'exponentiation

```
def expo(x,n):
    """ La fonction prend en entrée un entier (ou flottant) x et un entier naturel n, et retourne x^n"""
    y=1
    for i in range(n):
        y*=x
    return y
```

Une autre idée consiste à utiliser la décomposition en binaire de l'entier n. Prenons un exemple : on souhaite calculer x^{11}. 11 s'écrit en binaire $\overline{1011}^2$. À partir de x et en procédant par élévations au carré successives, il est facile de calculer les x^{2^p} : ici ce sont x, x^2, x^4 et x^8. Comme $11 = \overline{1011}^2$, il suffit de multilplier x^8, x^2 et x pour obtenir x^{11}. L'algorithme de multiplication suivant ce schéma porte le nom d'*algorithme d'exponentiation rapide*. On montrera par la suite qu'il est bien plus efficace que l'algorithme d'exponentiation naïf vu précédemment.

Expliquons son fonctionnement : il suit de près l'algorithme permettant de récupérer les bits d'un entier par division successives par 2. Cet algorithme permet de récupérer les bits 1 par 1, en commençant par les bits de poids faibles. En utilisant une variable annexe que l'on élève au carré à chaque étape, on calcule successivement les x^{2^p}. Il suffit alors de multiplier une variable (`z` dans le code suivant) initialisée à 1 par les x^{2^p} qui conviennent (donnés par les bits de n) pour obtenir x^n. Voici le code Python :

Algorithme d'exponentiation rapide

```
def expo_rapide(x,n):
    """ La fonction prend en entrée un entier (ou flottant) x et un entier naturel n, et retourne x^n"""
    y=x
    z=1
    m=n
    #Inv: z*y^m=x^n
    while m>0:
        #Inv
        q,r=m//2,m%2 # quotient et reste dans la division euclidienne de m par 2.
        if r==1:
            z*=y # on multiplie z par y, le résultat est affecté à z.
        y*=y # on met y au carré, le résultat est affecté à y.
        m=q
        #Inv
    #Inv
    return z
```

La fonction suppose que l'entier n est positif dans sa spécification. Observons maintenant le code : la condition du `while` porte sur m, qui doit être strictement positif pour qu'un tour de boucle s'effectue. Si on supprime tout ce qui n'a pas trait à la modification de la variable m dans le code, on retient :

```
    m=n
    while m>0:
        q=m//2
        m=q
```

Ainsi, les valeurs prises par m sont positives et strictement décroissantes à chaque itération de la boucle (car $\left\lfloor \frac{m}{2} \right\rfloor < m$ pour tout entier strictement positif m) : ainsi, m fini par être nul et la boucle se termine.

3.1.2 Variant de boucle

En général, pour montrer la terminaison d'une boucle on procède ainsi : *on exhibe une quantité, dépendant des paramètres, à valeurs dans* $\mathbb{N}$*, qui décroît strictement à chaque passage dans la boucle.* Puisqu'il n'existe pas de suite infinie strictement décroissante dans $\mathbb{N}$, cela prouve que la boucle se termine ! Cette quantité porte un nom en informatique :

Définition 3.2. *Un* variant de boucle *est une quantité positive, à valeurs dans* $\mathbb{N}$*, dépendant des variables de la boucle, qui décroît strictement à chaque passage dans la boucle.*

Dans l'exemple précédent, le variant de boucle est à peu près évident, et ce sera en général le cas pour nos algorithmes. Prenons un autre exemple, si `L` est une liste, la boucle suivante permet de calculer de manière un peu bête[3] la somme des éléments de `L`.

```
s=0
while L!=[]: #Tant que L est non vide
    s+=L[0]
    L=L[1:] #L[1:] est la liste constituée de tous les éléments de L, sauf le premier.
```

La boucle se termine lorsque la liste `L` est vide. La quantité qui décroît est ici la longueur de la liste `L`. Pour conclure sur cette section, signalons qu'il n'est parfois pas du tout évident de montrer (ou d'infirmer) qu'une boucle termine. Il est conjecturé que la fonction suivante termine quelle que soit l'entier strictement positif passé en argument, mais personne n'a été capable de le prouver[4] ! Remarquez que la fonction en elle-même n'a aucun intérêt, c'est simplement le fait qu'elle termine (ou non) quel que soit le paramètre respectant la spécification qui est intéressant.

```
def syracuse(n):
    """ n entier strictement positif """
    m=n
    while m!=1:
        if m%2==0:
            m=m//2
        else:
            m=3*m+1
    return 1
```

3.2 Correction

Pour montrer qu'un algorithme est correct, il s'agit de montrer que quels que soient les paramètres vérifiant sa spécification, l'action de l'algorithme correspond à ce qui est attendu. Reprenons notre découpage en blocs. Pour montrer la correction de l'algorithme, il s'agit de montrer que chacun des blocs effectue une action bien précise. Pour les blocs conditionnels (`if`, `elif`,...,`else`), il n'y a en général pas grand chose à dire de plus que le bloc lui-même. En revanche, analyser les boucles `for` et `while` est essentiel, car l'action de ces boucles n'est pas forcément évidente en première lecture.

Remarque 3.3. *On parle de* correction partielle *lorsque l'algorithme est correct sur toute entrée vérifiant la spécification,* à condition qu'il termine. *On parle de* correction totale *lorsqu'on a prouvé terminaison et correction partielle sur toute entrée vérifiant la spécification.*

La notion essentielle pour montrer la correction des boucles est celle d'*invariant de boucle*[5].

Définition 3.4. *Un invariant de boucle est une propriété dépendant des variables de l'algorithme, qui est vérifiée à chaque passage dans la boucle.*

La définition précédente est un peu vague, mais on va donner une explication plus précise pour chacune des boucles `for` et `while`, la situation étant légèrement différente. Donnons également une autre définition.

Définition 3.5. *Un algorithme est dit* partiellement correct *s'il est correct dès qu'il termine. Il est dit* totalement correct *si de plus il termine pour toute entrée respectant la spécification.*

3.2.1 Correction des boucles `while`

La boucle `while` (ou boucle `Tant que` en français), est parcourue tant qu'une condition booléenne est vérifiée. Reprenons l'algorithme d'exponentiation rapide, qui consiste principalement en une boucle `while`.

3. En terme d'efficacité, cet algorithme est mauvais : chaque instruction `L=L[1:]` demande de recopier en mémoire tous les éléments de la liste, sauf le premier. Cela a un coût très important !
4. C'est la fameuse conjecture de Syracuse, toujours ouverte.
5. À ne pas confondre avec le *variant de boucle*...

```
    while m>0:
        #Inv
        q,r=m//2,m%2
        if r==1:
            z*=y
        y*=y
        m=q
        #Inv
```

Juste avant la boucle, on a $y = x, z = 1$ et $m = n$. Remarquez qu'ainsi, $z \times y^m = x^n$. Vérifions que si cette propriété est vraie en *haut du corps de boucle* (là où se trouve le premier `#Inv`), alors elle est vérifiée en bas du corps de boucle (là où se trouve le second). Si on se trouve en haut du corps de boucle, ceci signifie que la variable m contient un entier strictement positif. On a deux cas à examiner, suivant la parité de m. Notons z', y' et m' les valeurs des variables z, y, m en bas de la boucle.

— si m est pair, alors $q = \frac{m}{2}$, $r = 0$, $y' = y^2$, et $m' = q = \frac{m}{2}$. $z' = z$ est inchangé. On a alors $z' \times y'^{m'} = z \times (y^2)^{\frac{m}{2}} = z \times y^m$. Puisque $z \times y^m$ valait x^n en haut de la boucle, c'est toujours le cas en bas du corps de boucle.

— si m est impair, alors $q = \frac{m-1}{2}$, $r = 1$. Dans ce cas, z' prend la valeur $z \times y$, $y' = y^2$ et $m' = \frac{m-1}{2}$. On a alors $z' \times y'^{m'} = z \times y \times (y^2)^{\frac{m-1}{2}} = z \times y^m$. De même, puisque $z \times y^m$ valait x^n en haut de la boucle, c'est toujours le cas en bas du corps de boucle.

Ainsi, la propriété $z \times y^m = x^n$ est maintenue à chaque passage dans la boucle, c'est donc un *invariant de boucle*. On en déduit en particulier que cette propriété est vérifiée également *après* la sortie de la boucle. Rappelons que l'on a montré qu'en *sortie de boucle*, m était nul. Or comme l'invariant est vérifié, cela signifie que $z = x^n$. Comme on renvoie z, l'algorithme renvoie bien x^n, et est donc correct !

Formalisons un peu tout ça :

Invariant de boucle dans une boucle while

```
#Inv propriété vraie avant la boucle
while condition:
    # On montre que si Inv est vraie en haut du corps de la boucle
    [actions]
    # alors Inv est vraie en bas du corps de boucle
#On en déduit (en particulier) qu'Inv est vraie après la boucle
```

3.2.2 Correction des boucles `for`

Si la boucle `while` a sensiblement le même comportement quel que soit le langage de programmation, ce n'est pas le cas des boucles `for`. Dans ce cours, bien qu'on adopte la syntaxe Python pour des raisons de facilité de compréhension, on ne traitera que des boucles de la forme

```
for i in range(n):
    [instructions qui ne modifient ni i, ni n]
```

ce qui signifie que `i` prend successivement les valeurs 0, 1, 2, jusqu'à $n-1$. On autorisera aussi `range(m,n)`, ce qui fait que `i` commence à m. Ce qu'on va dire s'étend naturellement aux boucles de la forme `for i in range(m,n,p)` avec un pas `p` différent de 1. Python autorise également la formulation `for x in L`, où `L` est une liste (`L` peut être un *itérateur* quelconque, comme `range(n)`). Dans ce cas on peut également parler d'invariant mais c'est plus complexe : en pratique il est nécessaire de faire appel à l'indice de `x` dans la liste : finalement on se ramène à une boucle de la forme `for i in range(len(L))` et accès aux éléments via `L[i]`.

Le tableau suivant présente une boucle `while` équivalente à une boucle `for`. L'invariant de la boucle `while`, qui a priori dépend de `i` (donc noté Inv_i dans la suite), est identique dans la boucle `for`. Les différences sont les suivantes (pour une boucle sur `range(n)`) :

— on montre que Inv_0 est vraie avant la boucle ;

— on montre de la même façon que si la propriété est vraie en haut du corps de la boucle, elle l'est en bas du corps de boucle. Seulement, puisque le passage `i=i+1` est effectué tout seul par la boucle `for`, on montre dans celle-ci que pour tout $i \in \{0, \dots, n-1\}$, si Inv_i est vraie en haut du corps de la boucle, Inv_{i+1} est vraie en bas du corps de boucle.

— On conclut en sortie de boucle que $\text{Inv}_{n-1+1} = \text{Inv}_n$ est vrai.

Boucle while

```
i=0
#Inv(0)
while i<n:
    #Inv(i)
    [instructions qui ne modifient ni i, ni n]
    i+=1
    #Inv(i)
#Inv(n)
```

Boucle for équivalente

```
#Inv(0)
for i in range(n):
    #Inv(i)
    [instructions qui ne modifient ni i, ni n]
    [les mêmes que dans le while]
    #Inv(i+1)
#Inv(n)
```

3.2.3 D'autres exemples : parcours linéaires de listes

On présente dans cette partie des exemples d'algorithmes consistant à parcourir une seule fois une liste.

- **Somme des éléments d'une liste.** L'exemple suivant présente le calcul de la somme des éléments d'une liste à l'aide d'une boucle `for`. L'invariant est à peu près évident ; rappelons seulement qu'en mathématiques, la somme d'un ensemble vide d'éléments vaut 0 (le neutre pour l'addition).

Algorithme de calcul de la somme des éléments d'une liste

```
def somme(L,x):
    """ La fonction prend en entrée une liste L de flottants ou d'entiers,
    et retourne la somme de ses éléments. """
    s=0
    for i in range(len(L)):
        #Inv(i): s est la somme des i premiers éléments de la liste.
        s+=L[i]
        #Inv(i+1): s est la somme des i+1 premiers éléments de la liste.
    return s
```

Comme notre fonction `somme` est correcte, on en déduit par exemple la correction de la fonction `moyenne` qui suit, prenant en entrée une liste que l'on suppose non vide : il suffit de sommer les éléments et de diviser par le nombre d'éléments.

Calcul de la moyenne des éléments d'une liste

```
def moyenne(L,x):
    """ La fonction prend en entrée une liste non vide L de flottants ou d'entiers,
    et retourne la somme de ses éléments"""
    assert L!=[],"la liste est vide !"
    return somme(L)/len(L)
```

- **Maximum d'une liste.** De même, un algorithme au programme consiste à savoir chercher le maximum d'une liste :

Algorithme de calcul du maximum d'une liste

```
def maximum(L):
    """ La fonction prend en entrée une liste L non vide de flottants ou d'entiers,
    et retourne le maximum de ses éléments. """
    m=L[0]
    for i in range(1,len(L)):
        #Inv(i): m est le plus grand élément de L[0:i].
        if L[i]>m:
            m=L[i]
        #Inv(i+1): m est le plus grand élément de L[0:i+1].
    return m
```

Il faut savoir adapter cet algorithme si l'on recherche le minimum, ou encore l'indice du maximum, etc...

- **Recherche d'un élément dans une liste.** Précisons un point : si dans la boucle se trouve une instruction de sortie (`return` par exemple) : on ne tient plus vraiment compte de l'invariant. Dans l'algorithme qui suit, on cherche si `x` se trouve dans la liste `L`. Si on trouve un indice `i` tel que `L[i]==x`, on sort immédiatement de la fonction en renvoyant `True`, ce qui est correct. Sinon, l'invariant est vérifié en bas de la boucle.

Algorithme de recherche dans une liste

```
def recherche(L,x):
    """ La fonction prend en entrée une liste L et un élément x,
    et retourne True si x est dans L, False sinon."""
    for i in range(len(L)):
        #Inv(i): x ne se trouve pas dans L[0:i].
        if L[i]==x:
            return True
        #Inv(i+1)
    return False
```

Notez que si l'on sort de la boucle, (sans être sorti de la fonction avec `return`), cela signifie que `Inv(len(L))` est vrai : `x` ne se trouve pas dans `L[0:len(L)]=L`. On renvoie alors `False` et la fonction est correcte.

- **Test de croissance.** Pour tester qu'une liste `L` de taille n est croissante, il faut vérifier

$$\forall i \in [\![0, n-2]\!], \quad \texttt{L[i]} \leq \texttt{L[i+1]}$$

En informatique, on ne peut pas tester un « pour tout », par contre il est facile de tester un « il existe ». On cherche donc s'il existe un tel indice tel que `L[i]` $>$ `L[i+1]`, on saura que la liste *n'est pas* croissante. Si à la fin du parcours on n'a trouvé aucun tel indice, on sait que la liste est croissante.

```
def recherche(L,x):
    """ La fonction prend en entrée une liste L et renvoie un booléen
    indiquant si elle est croissante. """
    for i in range(len(L)-1):
        #Inv(i): L[0:i+1] est croissante.
        if L[i+1]<L[i]:
            return False
        #Inv(i+1)
    return True
```

3.2.4 Recherche efficace dans une liste triée : recherche dichotomique

L'algorithme qui suit renvoie également un booléen suivant si `x` est dans `L`, mais suppose également que la liste `L` est triée : on verra dans la section suivante que l'algorithme ainsi obtenu est beaucoup plus rapide !

Algorithme de recherche dichotomique

```
def recherche_dicho(L,x):
    """ La fonction prend en entrée une liste L triée dans l'ordre croissant et un élément x,
    et retourne True si x est dans L, False sinon."""
    g=0
    d=len(L)
    while g<d:
        #Inv: x ne se trouve ni dans L[0:g] ni dans L[d:len(L)].
        m=(g+d)//2
        if L[m]==x:
            return True
        elif L[m]<x:
            g=m+1
        else:
            d=m
        #Inv: x ne se trouve ni dans L[0:g] ni dans L[d:len(L)].
    return False
```

— La terminaison de l'algorithme repose sur celle de la boucle `while` : la quantité $d-g$ est à valeurs dans $\mathbb{N}$ et décroît strictement à chaque itération de la boucle : l'algorithme termine.

— La correction repose elle aussi sur celle de la boucle `while`. On se rend compte facilement qu'elle admet l'invariant indiqué : si `L[m]` est égal à `x`, on renvoie simplement `True` et la fonction est correcte. Sinon, si `L[m]<x`, comme la liste est triée cela signifie que `x` ne peut se trouver qu'à un indice strictement supérieur à `m` et strictement inférieur si `L[m]>x`.

— Après la boucle, comme $g \geq d$ (en fait, $g = d$), l'invariant assure que `x` ne se trouve ni dans `L[0:g]` ni dans `L[d:len(L)]` donc en fait pas dans `L`. On renvoie `False` et la fonction est correcte.

3.3 Complexité

3.3.1 Introduction et tri par sélection

On sait maintenant prouver que nos algorithmes terminent et renvoient le bon résultat. La dernière question est la suivante : quel temps mettent-ils à s'exécuter ? À titre introductif, le tableau qui suit présente le temps en secondes du calcul de 5^n pour différents n (une puissance de 10), avec les algorithmes `expo` et `exo_rapide` présentés plus haut.

Les tests ont été réalisés sur la même machine, et les deux algorithmes calculent la même chose. Comment expliquer la différence d'efficacité entre le premier et le second lorsque n commence à être un peu grand ? Comptons simplement le nombre de multiplications nécessaires à chacun des algorithmes, en fonction de n. L'algorithme `expo` réalise une multiplication à chaque tour de boucle `for`, donc n au total. Pour l'algorithme d'exponentiation rapide, c'est un peu

algorithme \ `n`	10^1	10^2	10^3	10^4	10^5	10^6	10^7
`expo`	1.6×10^{-5}	3.4×10^{-5}	5.0×10^{-4}	0.014	0.97	98	11×10^3
`expo_rapide`	1.3×10^{-5}	1.5×10^{-5}	3.7×10^{-5}	6.0×10^{-4}	0.019	0.71	35

FIGURE 3.1 – Temps de calcul (secondes) de puissances de 5

plus compliqué, et on va simplement donner une majoration. Au pire, l'algorithme effectue deux multiplications à chaque passage de la boucle `while`. Le nombre de tours de boucle effectués correspond au nombre de chiffres de n dans la base 2, qui est de l'ordre de $\frac{\ln(n)}{\ln(2)}$ (ceci est expliqué plus bas), on fait donc de l'ordre de $\frac{2\ln(n)}{\ln(2)}$ multiplications.

Faisons tout de suite deux remarques :

— tout d'abord, l'important ici ne réside pas dans les constantes du résultat : ce qui est essentiel c'est que le calcul de la puissance par l'algorithme naïf fait de l'ordre de n multiplications alors que l'algorithme d'exponentiation rapide en fait seulement de l'ordre de $\ln(n)$. C'est un gain considérable !

— deuxièmement, on remarque que les temps de calcul des algorithmes ne suivent pas vraiment une progression linéaire en n (pour le premier) ou logarithmique (pour le second). Plusieurs facteurs peuvent expliquer la différence, mais le plus important d'entre eux est le suivant : on n'a pas du tout pris en compte le fait que les entiers manipulés étaient de taille variable : 5^{10^7} est un entier de plus de 20 millions de bits (6 millions de chiffres en base 10). Il est évidemment plus difficile de multiplier des entiers de cette taille que des entiers comme 2 et 3.

Prenons un deuxième exemple, le tri d'une liste. On se donne une liste, composés d'entiers ou de flottants (ou plus généralement d'éléments que l'on peut comparer avec `<=`, comme les chaînes de caractères par exemple), et on désire trier la liste, dans l'ordre croissant. Les algorithmes de tri sont au programme de deuxième année, mais voyons quand même l'un des plus simples, le tri par sélection. L'idée consiste à parcourir la liste pour repérer l'élément le plus petit, qu'on vient placer (par un échange) en première position dans la liste. On recommence le procédé à partir de la deuxième case de la liste pour trouver le plus petit élément dans la portion restante, que l'on vient placer en deuxième position, et ainsi de suite. Suivant cette idée, l'algorithme s'obtient facilement à l'aide de deux boucles `for`. Le code Python est donné ci-dessous.

```
def tri_selection(L):
    n=len(L)
    for i in range(0,n-1):
        #Inv(i): L[0:i] est trié et ses éléments sont plus petits que les autres éléments de L.
        imin=i
        minimum=L[i]
        for j in range(i+1,n):
            #Inv2(j): minimum=L[imin] est le plus petit élément de L[i:j].
            u=L[j]
            if u<minimum:
                imin=j
                minimum=u
            #Inv2(j+1)
        if imin!=i:
            L[i],L[imin]=L[imin],L[i]
        #Inv(i+1)
```

On laisse en exercice le soin de vérifier que les invariants de boucle sont corrects, et en déduire que l'algorithme trie bien la liste. Remarquez que la fonction ne renvoie rien : la liste est triée *en place* (et la fonction travaille par effets de bords). Le graphique qui suit montre le temps d'exécution sur des listes de tailles variables (entre 100 et 2000, par pas de 100), pour trier une liste constituée d'entiers tirés aléatoirement dans l'intervalle $[\![0, 10000]\!]$ (les tests sont effectués plusieurs fois, on présente une moyenne).

On remarque que le temps de calcul coïncide (approximativement) avec la fonction polynômiale de degré 2 donnée par $x \mapsto ax^2 + bx + c$, avec $a = 6.01 \times 10^{-8}$, $b = -1.25 \times 10^{-6}$ et $c = 1.72 \times 10^{-3}$. Les constantes (en particulier a), dépendent de la machine utilisée, ici il s'agit de mon ordinateur personnel. Avec un super-calculateur de la NSA, on aurait eu un coefficient dominant beaucoup plus faible. On peut aussi effectuer cet algorithme à la main, sur papier, avec un crayon à papier et une gomme : dans ce cas a risque d'être assez élevé. Ce qui est important, c'est que le temps de calcul semble varier comme un polynôme de degré 2 en n, ce qui est inhérent à l'algorithme et non pas au

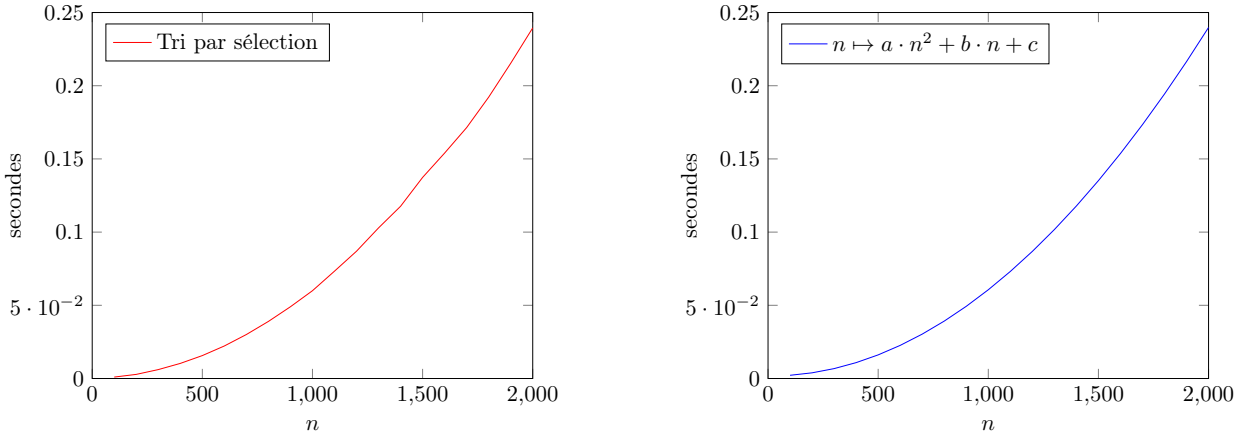


FIGURE 3.2 – Temps pour trier une liste de taille n et approximation par une fonction polynomiale.

langage dans lequel il est écrit, à l'implémentation ou encore à la machine sur laquelle il est exécuté. Pouvions nous prévoir ceci ? La réponse est oui.

Dans la boucle `for` interne, on exécute un nombre constant d'opérations élémentaires (comparaisons, affectations), donc le temps d'exécution de la boucle interne peut-être majoré par une constante c_i. Ces opérations sont effectuées $n-i-1$ fois. Outre cette boucle `for` interne, la boucle externe réalise un nombre constant d'opérations élémentaires, dont le temps total peut-être majoré par une constante c_e. En plus de la boucle `for` externe, l'algorithme réalise également quelques opérations élémentaires (affectation, entrée, sortie), dont le temps est également majoré par une constante c_a. Finalement, le temps d'exécution de l'algorithme est majoré par :

$$c_\text{a} + \sum_{i=0}^{n-2}\left(c_\text{e} + \sum_{j=i+1}^{n-1} c_\text{i}\right) = c_\text{a} + (n-1)c_\text{e} + c_\text{i}\sum_{i=0}^{n-2}(n-1-i) = c_\text{a} + (n-1)c_\text{e} + c_\text{i}\frac{n(n-1)}{2}$$

En développant, on retrouve bien un polynôme de degré 2. On pourrait de même minorer le temps total par une fonction similaire, ce qui explique le comportement quadratique du tri.

3.3.2 Complexité : définitions et méthodes

Qu'est-ce que la complexité ? La complexité d'une fonction sur un ensemble de paramètres est le prix à payer en termes de ressources pour mener à bien le calcul. Différents types de ressources peuvent être évalués :

— Le temps de calcul CPU : nombre d'opérations élémentaires réalisés par le processeur, ce qui est lié directement au temps de calcul.

— La mémoire nécessaire : mémoire RAM ou sur disque dur...

— ...

Définition 3.6. *La complexité est la mesure de l'efficacité d'un programme pour un type de ressources :*

— *complexité temporelle : temps de calcul.*

— *complexité spatiale : espace mémoire.*

En pratique, la complexité temporelle est plus importante que la complexité spatiale. L'étude de la complexité d'une fonction consiste à estimer son coût en ressource en fonctions des entrées. Pour différencier deux entrées entre elles, on compare en général leur taille. Essentiellement pour nous, les entrées seront constituées d'entiers, de flottants ou de listes. Pour les listes, la donnée pertinente est la taille. Pour les entiers, cela dépend du contexte. Pour un entier n, on peut en effet exprimer la complexité d'une fonction dépendant de n en fonction :

— de l'entier n lui-même.

— ou de son nombre de chiffres (sa taille), correspondant à $\log_2(n)$ (car l'entier est représenté en binaire). Notez que la base du log ne compte pas vraiment, on verra qu'on ne tient en général pas compte des constantes multiplicatives.

Le choix dépendra en général du contexte : par exemple pour exprimer la complexité d'une fonction qui renvoie l'écriture en base 2 d'un nombre exprimé en base 10, on se dirigerait plus naturellement vers $\log_2(n)$. Pour calculer toutes les listes de taille n constituées de zéros et de uns, la donnée pertinente est n lui-même. À notre niveau, même si l'on manipule des entiers dont la taille peut varier, on ne tiendra en général pas compte de leur taille.

Coûts. Concentrons-nous d'abord sur la complexité en temps. L'exécution d'un algorithme est une séquence d'opérations nécessitant plus ou moins de temps. Pour mesurer ce temps, on considère certaines opérations comme élémentaires : par exemple faire une opération arithmétique de base (addition, multiplication, soustraction, division...), lire ou modifier un élément d'une liste, ajouter un élément à la fin d'une liste, affecter un entier ou un flottant, etc... Estimer le coût en temps d'une fonction sur une entrée de taille donnée signifie estimer le nombre de ces opérations élémentaires effectuées par la fonction sur l'entrée. La complexité en mémoire consiste à estimer la mémoire nécessaire à une fonction pour son exécution, en plus de celles des entrées.

Complexité dans le pire cas. Considérons le problème de rechercher un élément dans une liste. Que ce soit dans l'algorithme de recherche dans une liste non triée ou de recherche dichotomique dans une liste triée, il se peut que l'on tombe tout de suite sur l'élément : dans ce cas le nombre de d'opérations effecuées par l'algorithme est constant. Pour comparer deux algorithmes, le plus intéressant est en général de comparer ce qu'il se passe *dans le pire cas*, c'est à dire la complexité maximale obtenue sur un jeu de paramètres de taille fixée respectant la spécification. Pour le problème de la recherche dans une liste, cela correspond par exemple au cas où l'élément cherché n'est pas dans la liste. À l'occasion (plutôt en deuxième année), on pourra comparer deux algorithmes vis à vis du *meilleur cas*, et du *cas moyen*, ce dernier requérant une distribution de probabilités sur les entrées possibles. Pour le problème du calcul de x^n en fonction de n (et x), il n'y a ici qu'un seul cas à considérer.

Complexité asymptotique et notations de Landau. Supposons pour simplifier que l'algorithme dont on veut calculer la complexité ne prenne qu'un seul paramètre en entrée. Tout d'abord, lorsque l'on s'intéresse à la complexité $C(n)$ (n est la taille de l'entrée) d'une fonction, c'est bien souvent pour les grandes valeurs de n qu'il est pertinent de connaître $C(n)$, pour comparer vis à vis d'autres fonctions réalisant le même calcul. On cherche donc un comportement asymptotique de n, qu'on rapportera aux fonctions usuelles : logarithmes, puissances, exponentielles... Ensuite, on ne cherchera pas systématiquement un équivalent : celui-ci est souvent difficile à obtenir et n'est pas le plus important. Si deux fonctions nécessitent respectivement environ $9n\ln(n)$ et $\frac{n^2}{2}$ opérations élémentaires, on retiendra que la première nécessite de l'ordre de $n\ln(n)$ opérations, ce qui est bien meilleur que la seconde qui en requiert de l'ordre de n^2. Rappelons les notations de Landau. Soit f et g deux fonctions $\mathbb{N} \to \mathbb{R}_+^*$. On note :

— $f(n) = O(g(n))$, si il existe un entier n_0 tel que $g(n)$ est non nul pour $n \geq n_0$ et $\left(\frac{f(n)}{g(n)}\right)_{n\geq n_0}$ est bornée.

— $f(n) = \Omega(g(n))$, si $g(n) = O(f(n))$.

— $f(n) = \Theta(g(n))$, si $f(n) = O(g(n))$ et $f(n) = \Omega(g(n))$.

Pour exprimer la complexité $C(n)$, on se contentera bien souvent d'un O, qui est d'ailleurs la seule au programme. Par exemple, la recherche dichotomique dans une liste triée de taille n a une complexité (dans le pire cas) de $O(\ln(n))$. Si l'on veut préciser que cette borne est essentiellement optimale, on dira que la complexité est en $\Theta(\ln(n))$. Attention à ne pas dire « la complexité est au moins en $O(\ln(n))$ », ce qui n'aurait aucun sens.

Estimer la complexité d'une fonction. On explique ici comment obtenir une majoration (notation O), ce qui sera notre préoccupation principale lorsque l'on parlera de complexité temporelle.

— Les opérations d'affectations, entrée/sortie de fonctions, opérations arithmétiques élémentaires, opérations booléennes, accès ou modification d'une case d'une liste... sont comptées avec un coût constant (qu'on note $O(1)$.)

— La complexité d'une boucle est égale à la somme des complexités de chaque tour de boucle. On peut en particulier majorer cette complexité par le nombre de tour de boucles multiplié par la complexité maximale d'un tour de boucle. En général cela est suffisant pour estimer correctement la complexité, mais attention toutefois, parfois cela conduit à la surestimer.

— La complexité d'une disjonction conditionnelle `if,elif,...,elif,else` est majorée par le maximum des complexités de chaque cas.

— L'appel à une fonction compte comme le coût de cette fonction sur les paramètres avec lesquels elle est appelée : attention à ne pas oublier ces coûts !

3.3.3 Applications aux algorithmes vus précédemment

Examinons maintenant les complexités des algorithmes vus plus haut. Avant cela, un petit paragraphe sur les logarithmes.

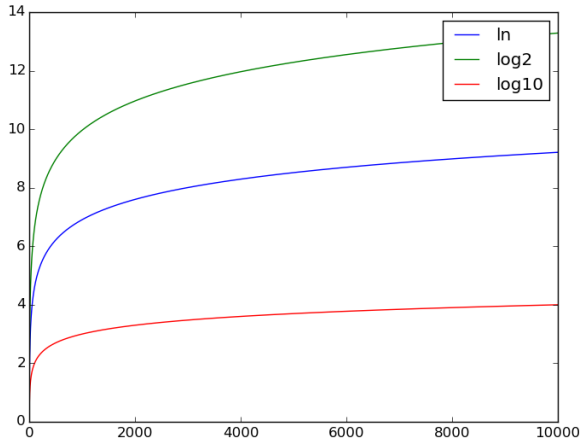


FIGURE 3.3 – Différents logarithmes

Le logarithme en base 2. Vous savez que ln est la fonction réciproque de exp, définie par $\ln(e^x) = x$ pour tout $x > 0$. Pour $a > 1$, on définit la fonction $x \mapsto a^x$ par $a^x = \exp(x \ln(a))$. Sa fonction réciproque est donc $x \mapsto \frac{\ln(x)}{\ln(a)}$, que l'on note usuellement $\log_a$ (prononcer« log en base a »). En chimie, le logarithme en base 10 est utilisé pour définir le pH, en informatique c'est le logarithme en base 2 qui est le plus utilisé. La figure 3.3 présente les logarithmes usuel, en base 2 et en base 10. Une propriété du logarithme en base 2 est la suivante : si x vérifie $2^n \leq x < 2^{n+1} - 1$, alors $n \leq \log_2(x) < n+1$, donc $n = \lfloor \log_2(x) \rfloor$. Autrement dit, un nombre entier strictement positif x nécessite $1 + \lfloor \log_2(x) \rfloor$ bits pour être représenté sur entiers naturels.

Algorithmes linéaires. Les algorithmes de recherche dans une liste ou de calcul de la somme des éléments d'une liste ont une complexité $O(n)$, où n est la taille de la liste. En effet, ces algorithmes sont basés sur une boucle `for` exécutée n fois, qui consiste à parcourir la liste. C'est pareil pour le calcul de la moyenne des éléments d'une liste, puisqu'on ne fait qu'un nombre fini d'opérations élémentaires en plus de l'appel à la fonction `somme`. De même, l'élévation à la puissance de façon naïve a un coût de $O(n)$ opérations élémentaires (en fait exactement n multiplications)

Algorithme du tri par sélection. Il a une complexité $O(n^2)$ (« quadratique en n »).

Algorithme d'exponentiation rapide. Cet algorithme de calcul de x^n effectue autant de tours de boucle que n a de chiffres en binaire, et chaque tour de boucle se fait avec une complexité constante. On en déduit une complexité $O(\log_2(n))$. Remarquez que la base du logarithme importe peu car les logarithmes dans deux bases distinctes ne diffèrent que d'une constante multiplicative, ignorée par la notation O, on peut donc noter la complexité $O(\log n)$ sans préciser la base.

Algorithme de recherche dichotomique dans une liste triée. Là aussi chaque tour de boucle se fait avec une complexité constante, il reste à estimer le nombre de tours de boucle effectués. Notons n la taille de la liste et d_i et g_i les valeurs des variables d et g après i tour de boucle, et $t_i = d_i - g_i$. Initialement, $d_0 = n$ et $g_0 = 0$, donc $t_0 = n$. Distinguons les cas :

— si d prend la valeur m après un tour de boucle supplémentaire, on a $d_{i+1} = \left\lfloor \frac{d_i+g_i}{2} \right\rfloor \leq \frac{d_i+g_i}{2}$ et $g_{i+1} = g_i$, donc $t_{i+1} = d_{i+1} - g_{i+1} \leq \frac{d_i-g_i}{2} = \frac{t_i}{2}$;

— si g prend la valeur $m+1$ après un tour de boucle supplémentaire, on a $g_{i+1} = 1 + \left\lfloor \frac{d_i+g_i}{2} \right\rfloor \geq \frac{d_i+g_i+1}{2}$ (car $\frac{g_i+d_i}{2}$ est un entier ou un demi-entier) et $d_{i+1} = d_i$, donc $t_{i+1} \leq \frac{d_i-g_i-1}{2} \leq \frac{t_i}{2}$;

Ainsi, on vérifie aisément par récurrence que $t_k \leq \frac{n}{2^k}$, et ce terme est donc nul pour $k > \log_2(n)$. Comme la boucle s'arrête lorsque $d - g = 0$, on conclut que le nombre d'opérations effectuées est $O(\log n)$.

3.3.4 Quelques ordres de grandeur

Le tableau suivant présente le n maximal tel qu'un algorithme nécessitant $f(n)$ opérations élémentaires pour s'exécuter sur l'entrée n puisse s'exécuter en moins d'une minute, et ce pour plusieurs fonctions f. On suppose que

l'algorithme exécute exactement 10^9 opérations élémentaires par seconde. Cela donne une indication de ce que peut faire un algorithme de complexité $O(f(n))$.

$f(n)$	$\ln(n)$	$\sqrt{n}$	n	n^2	n^3	2^n	$n!$	n^n
$n_{\max}$	très gros !	3.6×10^{21}	6×10^{10}	244948	3914	35	13	10

Naturellement, plus la fonction f grandit, plus $n_{\max}$ est faible. On pourra remarquer le fossé entre une fonction polynomiale (une fonction de la forme $n \mapsto n^k$) et les fonctions au moins exponentielles (les trois dernières). On parle de complexité linéaire, quadratique, cubique pour des algorithmes de complexité $O(n)$, $O(n^2)$, $O(n^3)$ et de complexité (au moins) exponentielle pour les trois dernières fonctions. Un algorithme de complexité au moins exponentielle en n n'est en pratique utilisable que pour de très petites valeurs de n.

3.4 Recherche d'un motif dans une chaîne de caractères

On s'intéresse au problème de la recherche de motif dans une chaîne. On dit que `m` est un motif de `s` si `m` coïncide avec `s[i:k]` pour deux indices `i` et `k`. Pour tester si `s` possède `m` comme motif, il suffit de vérifier si la proposition suivante est vraie :

il existe un indice `i` entre `0` et `len(s)-len(m)` (inclus), tel que pour tout `j` entre `0` et `len(m)-1`, `s[i+j]` est égal à `m[j]`.

On voit se dessiner une idée d'algorithme : il suffit de faire parcourir à `i` toutes les valeurs entre `0` et `len(s)-len(m)`, et ensuite incrémenter un compteur `j` commençant à `0`, *tant que* `s[i+j]` est égal à `m[j]`. Si `j` atteint `len(m)`, on a trouvé un tel motif, sinon on recommence avec le `i` suivant. Voici le code Python correspondant à cette idée :

Algorithme de recherche de motif dans une chaîne de caractères

```
def recherche_motif(m,s):
    """ La fonction prend en entrée deux chaînes de caractères m et s,
    et retourne True si m apparaît comme sous-chaîne de s, False sinon."""
    lm=len(m)
    ls=len(s)
    for i in range(0,ls-lm+1):
        #Inv(i): m n'apparaît pas comme motif de la sous-chaîne s[0:i+lm-1]
        j=0
        #Inv2: Les sous-chaînes m[0:j] et s[i:i+j] sont égales
        while j<lm and m[j]==s[i+j]:
            #Inv2
            j+=1
        if j==lm:
            return True
        #Inv(i+1)
    return False
```

Montrons que l'algorithme est correct. On notera ℓ_m et ℓ_s les longueurs des chaînes m et s.

Tout d'abord, on n'essaiera jamais d'accéder à un élément d'une chaîne au delà de sa longueur : en effet, on essaie d'accéder à `m[j]` que si $j < \ell_m$, en vertu du caractère *paresseux* du `and` : si $j < \ell_m$ est faux (donc $j \geq \ell_m$), alors on n'a pas besoin d'évaluer `m[j]==s[i+j]`. De même, on n'accède à `s[i+j]` qu'avec $i < \ell_s - \ell_m + 1$ et $j < \ell_m$ donc $i + j < \ell_s$.

L'algorithme termine bien, car la boucle `while` interne termine à chaque fois : $\ell_m - j$ est une quantité qui reste positive et décroît strictement à chaque passage dans la boucle.

L'invariant proposé pour la boucle `while` interne est correct : il est vrai avant la boucle puisque les chaînes `m[0:0]` et `s[i:i]` sont vides. Si on est encore dans la boucle cela signifie que les deux chaînes considérées coïncident sur un caractère supplémentaire.

Pour la boucle externe, l'invariant proposé est correct car il repose sur l'invariant de la boucle `while`. En sortie de boucle `while`, l'invariant est vrai : si $j = \ell_m$ alors `m` est un motif de `s` (en effet, `m=s[i:i+lm]`), et on renvoie `True`, donc la sortie de la fonction est correcte. Sinon, on est sorti de la boucle `while` parce que `m[j]` était différent de `s[i+j]` et `m` et `s[i:i+lm]` ne coïncide pas, et l'invariant est vrai en bas de la boucle.

Ainsi, si l'on atteint le bas de la boucle `for`, cela signifie que `m` n'apparaît pas comme motif de `s[0:ls-lm+1+lm-1]` qui est égal à `s[0:ls]` donc à `s`. On renvoie `False` et la fonction est correcte.

Examinons le coût de la fonction : la boucle `while` interne réalise au plus ℓ_m tours de boucle à chaque étape. La boucle `for` fait exactement $(\ell_s - \ell_m + 1)$ étapes (si $\ell_s \geq \ell_m$, sinon 0), on en déduit une complexité totale de $O(1 + \ell_m(\ell_s - \ell_m + 1))$ (le 1 devant est fait pour que la formule soit vraie même si $\ell_m = 0$, dans ce cas on ne fait pas grand chose mais le coût n'est pas nul). On peut majorer cette complexité par $O(1 + \ell_m \ell_s)$.

Remarque : Une remarque pour terminer. En Python, il est tout à fait possible de calculer la somme des éléments d'une liste, faire une recherche dans une liste, trier une liste, chercher un motif... à l'aide des commandes suivantes. Il faut avoir à l'esprit que ce ne sont pas opérations élémentaires, et ne pas oublier qu'elles cachent un travail important pour le processeur [6] !

`sum(L)`	calcul de la somme des éléments de la liste `L`.
`max(L)`, `min(L)`	calcul du max/min de la liste `L`
`L.index(x)`	calcul du premier indice `i` tel que `L[i]` est égal à `x` s'il existe, produit une erreur sinon.
`x**n`	calcul de la puissance par exponentiation rapide si `x` est un entier et `n` un entier positif.
`x in L`	recherche de `x` dans `L`.
`L.sort()`	tri de la liste `L`, avec un tri plus efficace que le tri par sélection !
`m in s`	recherche de `m` comme motif de `s` (avec un algorithme plus efficace que celui ci-dessus).

On peut utiliser également une fonction Python pour la recherche dichotomique, avec une légère modification d'une fonction importée du module `bisect`.

Algorithme de recherche dichotomique (Python)

```
import bisect
def cherche_dicho_Python(L,x):
    i=bisect.bisect_left(L,x)
    return i<len(L) and L[i]==x
```

6. La commande `x in L` est une recherche linéaire dans la liste. Le lecteur intéressé pourra vérifier que notre algorithme de recherche dichotomique est bien plus efficace sur une liste triée que l'instruction Python `x in L`, sur des listes assez grosses (j'obtiens un facteur 10000 dans le temps d'exécution sur une liste de taille 10^7). Il ne faut pas croire qu'une instruction d'une ligne s'exécute rapidement !

Deuxième partie

Techniques algorithmiques

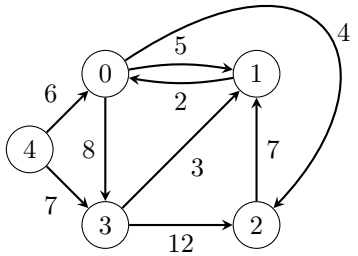
Chapitre 4

Algorithmes gloutons

4.1 Introduction

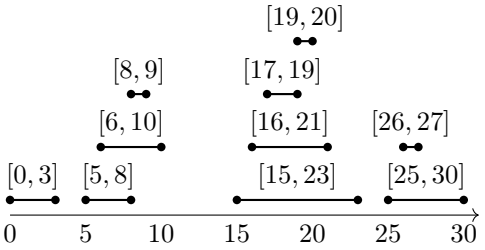
Dans ce chapitre, on cherche à résoudre des problèmes *d'optimisation*. Formellement, sur un univers $\mathcal{U}$, on cherche à minimiser (ou maximiser, la situation est symétrique) une certaine fonction, souvent à valeurs dans les entiers. Concrètement, on se donne $f : \mathcal{U} \to \mathbb{Z}$, et on cherche à déterminer x tel que $f(x) = \min_{y\in\mathcal{U}}\{f(y)\}$ ou $f(x) = \max_{y\in\mathcal{U}}\{f(y)\}$, si cette quantité existe bien. Citons quelques exemples concrets :

— Dans le graphe suivant, quel est le plus petit poids d'un chemin [1] entre le sommet 3 et le sommet 2 ? Cette question sera résolue dans le chapitre sur les graphes.



— On appelle sous-séquence d'une chaîne de caractères s une chaîne de caractère m dont les caractères sont présents dans s, dans l'ordre (avec possiblement d'autres caractères intercalés). Le problème de la plus longue sous-séquence commune (PLSSC) est de trouver une sous-séquence commune à deux chaînes de caractères données, qui soit de longueur maximale. Par exemple, une PLSSC de « loutre » et « troupe » est « tre ». « oue » convient également. On résoudra ce problème en deuxième année.

— Le problème suivant sera résolu dans ce chapitre. On se donne un ensemble d'activités (voir figure suivante), chacune pouvant être effectuée sur une certaine plage horaire (un intervalle de temps). Trouver un ensemble d'activités disjointes (c'est-à-dire que les intervalles ne se recouvrent pas) de cardinal maximal.



Usuellement, l'espace de recherche (l'univers $\mathcal{U}$) est grand, ce qui rend une recherche exhaustive inefficace. Par exemple, pour le problème précédent avec n intervalles, l'univers $\mathcal{U}$ est constitué des 2^n parties de $\mathcal{U}$ (et la fonction f à maximiser est le cardinal), une recherche exhaustive serait donc de complexité exponentielle en la taille de l'entrée (les n intervalles). On cherche donc une stratégie plus rapide.

1. Naturellement le poids d'un chemin est la somme des poids des arcs qui composent ce chemin.

4.2 Principe des algorithmes gloutons

Le principe d'un algorithme glouton est de construire une solution au problème en faisant une succession de choix *localement* optimaux. Chaque choix effectué ramène le problème à un problème plus simple, jusqu'à ce que celui-ci soit trivial.

Par exemple, pour le problème précédent, on peut appliquer l'algorithme suivant :

Algorithme 4.1 : Algorithme glouton pour la sélection d'activités

Entrée : Un ensemble $\mathcal{I}$ d'intervalles
Sortie : Une partie de $\mathcal{I}$ constituée d'intervalles disjoints
$\mathcal{S} \leftarrow \emptyset$;
pour *tout intervalle* I *de* $\mathcal{I}$ **faire**
  si I *n'intersecte aucun des éléments de* $\mathcal{S}$ **alors**
    Rajouter I à $\mathcal{S}$
Renvoyer $\mathcal{S}$

Le choix localement optimal ici est d'ajouter un intervalle I donné dès qu'il est compatible avec ceux précédemment choisis. L'algorithme précédent ne fournit pas nécessairement une solution maximale en nombre d'intervalles, tout dépend dans quel ordre les éléments de $\mathcal{I}$ sont examinés. Par exemple, si $\mathcal{I} = \{[0,5],[1,2],[3,4]\}$, l'algorithme renvoie $\{[0,5]\}$ si cet intervalle est examiné en premier, et $\{[1,2],[3,4]\}$ sinon (cette partie, de cardinal 2, est optimale). On va voir à la section suivante comment améliorer l'algorithme pour qu'il renvoie toujours une solution optimale.

Pour résumer :

— les algorithmes gloutons effectuent un choix localement optimal pour se ramener à un problème plus simple ;

— ces algorithmes sont rapides ;

— *a priori*, la solution obtenue n'est pas *globalement optimale*. Néanmoins, pour certains problèmes, on peut prouver l'optimalité du choix glouton.

4.3 Maximisation d'activités

On reprend le problème précédent. Pour simplifier à la fois les preuves théoriques et l'implémentation, on suppose que les intervalles sont tous de la forme $[g,d[$ (fermés à gauche, ouverts à droite, avec $g < d$). A priori l'algorithme 4.1 ne produit pas une solution optimale, néanmoins c'est le cas si les intervalles sont *triés par date de fin croissante*.

4.3.1 Un algorithme optimal

Proposition 4.2. *Si, dans l'ensemble d'entrée $\mathcal{I}$, les intervalles $[g_i,d_i[$ sont triés par dates de fin d_i croissantes, alors l'algorithme 4.1 renvoie un sous-ensemble d'intervalles disjoints de $\mathcal{I}$ de cardinal maximal.*

Démonstration. Notons $\mathcal{I} = \{I_0,\dots,I_{n-1}\}$, avec $I_k = [g_k,d_k[$ pour tout k, vérifiant $d_0 \leq d_1 \leq \cdots \leq d_{n-1}$. Montrons la propriété par récurrence forte sur n.

— Si $n = 0$ (il n'y a aucun intervalle !) l'algorithme renvoie l'ensemble vide, ce qui est bien optimal.

— Soit $n > 0$, et supposons l'optimalité de l'algorithme démontré sur des ensembles de cardinal strictement inférieur. Soit $\mathcal{S}$ le sous-ensemble renvoyé par l'algorithme (les intervalles sont considérés par valeurs croissantes de k), Notons que $I_0 \in \mathcal{S}$. Soit $\mathcal{T}$ un sous-ensemble d'intervalles disjoints de $\mathcal{I}$ de cardinal maximal. On peut supposer que $I_0 \in \mathcal{T}$: en effet, supposons que $I_0 \notin \mathcal{T}$. Avec $I = [g,d[$ l'élément de $\mathcal{T}$ dont la borne droite d est la plus petite, tous les autres intervalles de $\mathcal{T}$ démarrent après d, et sont donc compatibles avec I_0, ainsi $\mathcal{T}\backslash\{I\}\cup\{I_0\}$ est également un sous ensemble d'intervalles disjoints de $\mathcal{I}$ de cardinal maximal. Il existe donc une solution optimale contenant I_0. Supprimons de $\mathcal{I}$ l'intervalle I_0 et tous ceux qui l'intersectent pour obtenir $\mathcal{I}'$. Par hypothèse de récurrence, appliqué à cet ensemble l'algorithme glouton fournit une solution optimale, qui est $\mathcal{S}\backslash\{I_0\}$. Or, $\mathcal{T}\backslash\{I_0\}$ est un sous-ensemble d'intervalles disjoints inclus dans $\mathcal{I}'$, donc $|\mathcal{S}\backslash\{I_0\}| \geq |\mathcal{T}\backslash\{I_0\}|$. Il y a donc égalité, et $\mathcal{S}$ est bien de taille maximale.

□

4.3.2 Implémentation

Le code suivant applique l'algorithme 4.1, en triant au préalable les intervalles par valeurs croissantes des bornes droites. On suppose qu'un intervalle $[g, d[$ est stocké comme le couple `(g,d)` ou la liste `[g,d]`. Le test *I n'intersecte aucun des éléments de S* du pseudo-code est particulièrement simple ici, puisque les intervalles sont considérés par valeurs de d croissante : il suffit de vérifier que I débute après le dernier intervalle de S (ou que S est vide).

```
def maxi_intervalles(L):
    L_trie = L[:]
    L_trie.sort(key = lambda x:x[1])
    S = []
    for I in L_trie:
        if S == [] or I[0]>=S[-1][1]:
            S.append(I)
    return S
```

Rappels :

— la syntaxe `L.sort()` permet de trier `L` en place (c'est-à-dire que `L` est modifiée).

— on peut préciser une fonction de tri : `L.sort(key = f)` trie `L` par valeurs de `f` croissante.

— `lambda variable:expression` est la syntaxe pour déclarer une fonction sans lui donner de nom. Ici `lambda x:x[1]` est la fonction qui a un couple (ou un tuple plus grand, une liste, etc...) associe son deuxième élément (d'indice 1).

— pour ne pas modifier `L`, on effectue une copie au préalable.

Remarque : la complexité de l'algorithme est dominée par le tri, qui s'effectue en $O(n \log n)$ avec n le nombre d'intervalles.

4.4 Rendu de monnaie

4.4.1 Énoncé du problème

On se donne un ensemble $\mathcal{P} = \{p_0, \dots, p_{n-1}\}$ de valeurs de pièces, avec $p_0 = 1 < p_1 < \cdots < p_{n-1}$, ces valeurs étant entières. On se donne également une somme d'argent $s \in \mathbb{N}$ que l'on souhaite décomposer à l'aide des p_i, de manière à minimiser le nombre de pièces. En résumé :

But du problème. Trouver $(m_0, \dots, m_{n-1}) \in \mathbb{N}^n$, avec $\sum_{i=0}^{n-1} m_i p_i = s$, tel que $\sum m_i$ soit le plus petit possible.

Remarquons que comme $p_0 = 1$, le problème admet toujours une solution (et le nombre de pièces total est inférieur à s).

4.4.2 Choix glouton

Le choix glouton est assez naturel : on commence par la pièce p_{n-1}, de plus grande valeur que l'on utilise le plus possible : $\left\lfloor \frac{s}{p_{n-1}} \right\rfloor$ fois. On recommence avec $s - \left\lfloor \frac{s}{p_{n-1}} \right\rfloor p_{n-1}$ et les pièces de valeurs inférieures. Le code Python suivant fournit la liste des valeurs de pièces à utiliser.

```
def rendu_monnaie(s,P):
    """ P[0]=1, P trié dans l'ordre croissant."""
    i=len(P)-1
    L=[]
    while s>0:
        if P[i]<=s:
            L.append(P[i])
            s-=P[i]
        else:
            i-=1
    return L
```

4.4.3 Optimalité

L'algorithme précédent est-il optimal, c'est-à-dire, fournit-il tout le temps un nombre minimal de pièces ? En général non. Par exemple, si $\mathcal{P} = \{1, 3, 4\}$ et $s = 6$, l'algorithme proposera la décomposition $6 = 4 + 1 + 1$, alors que $6 = 3 + 3$ utilise moins de pièces. On verra en deuxième année un algorithme un peu plus couteux fournissant toujours une solution optimale.

Remarque 4.3.
— *Un système de pièces $\mathcal{P}$ pour lequel l'algorithme glouton fournit toujours une réponse optimale est dit* canonique. *Une description complète des systèmes canoniques est toujours un problème ouvert (bien que l'on connaisse des algorithmes rapides permettant de tester si un système est canonique, voir l'article Wikipédia*[2] *pour les curieux.)*

— *Tous les systèmes monétaires actuels sont canoniques. On pourra, à titre d'exercice, montrer que le système européen* $(1, 2, 5, 10, 20, 50, 100, 200, 500)$ *est canonique. Ça n'a pas toujours été le cas : un contre-exemple historique est l'ancien système monétaire anglais (qui n'a été réformé qu'en 1971 !), qui comportait notamment les valeurs* $(1, 3, 6, 12, 24, 30)$. *On peut vérifier que* 48 *n'est pas décomposé optimalement par l'algorithme glouton.*

— *Autre exercice : pour $p \geq 2$ et $n \geq 0$, montrer que le système $(1, p, p^2, \ldots, p^n)$ est canonique.*

2. `https://fr.wikipedia.org/wiki/Probl%C3%A8me_du_rendu_de_monnaie`

Chapitre 5

Récursivité

5.1 Principes de la récursivité

5.1.1 Définition

La définition d'une fonction récursive est plutôt simple : c'est une fonction qui s'appelle elle-même. Prenons un exemple classique : le calcul de la factorielle. On définit, pour $n \geq 0$, $n! = \prod_{i=1}^{n} i$. Informatiquement, on peut donc définir la factorielle comme :

```
def fact(n):
   assert n>=0, "n doit etre positif"
   f=1
   for i in range(1,n+1):
      f=f*i
   return f
```

Une autre définition mathématique classique de la factorielle se fait par *récurrence* :

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{sinon.} \end{cases}$$

En reprenant quasiment mot pour mot cette dernière définition, on obtient la fonction Python suivante.

```
def fact_rec(n):
   assert n>=0, "n doit etre positif"
   if n==0:
      return 1
   else:
      return n*fact_rec(n-1)
```

qui fonctionne tout aussi bien ! On distingue clairement deux cas dans cette fonction :

— le cas $n = 0$, appelé cas terminal ;

— le cas $n > 0$, qui produit un appel *récursif* à la fonction `fact_rec`.

5.1.2 La pile d'exécution

Ce paragraphe est plutôt culturel, mais nécessaire si on veut comprendre vraiment pourquoi la fonction précédente se déroule sans accroc.

Notion de pile. En informatique, la pile [1] est une structure de données de base, devant supporter les opérations suivantes :

— création d'une pile vide

— test si une pile est vide ;

— ajout d'un élément dans la pile (au *sommet*, voir la suite) ;

— accès au sommet d'une pile non vide ;

1. On reparlera de cette structure au moment de parcourir des graphes.

— suppression du sommet d'une pile non vide.

Les éléments sont stockés dans la pile suivant un principe *LIFO*, pour « Last In, First Out ». Autrement dit, dans une pile non vide, le prochain élément à sortir est le dernier à avoir été inséré. Visuellement, on peut voir ça comme une pile d'assiettes que l'on regarderait par au dessus : on peut rajouter un sommet en haut de la pile, ou retirer le sommet de la pile, s'il reste des éléments. Le sommet est le seul élément accessible. En Python, on peut implémenter cette struture avec une simple liste, en se restreignant à `append`, `pop` et accès au dernier élément. Le sommet de la pile est donc le dernier élément de la liste dans ce cas.

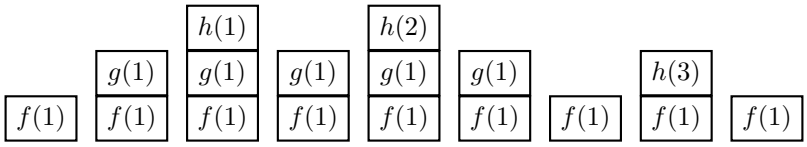
Mécanisme des appels de fonctions. En informatique, la pile d'exécution (ou pile d'appels, *call stack* en anglais) est une structure de données de type pile, qui sert à enregistrer des informations au sujet des fonctions actives dans un programme. Une fonction active est une fonction dont l'exécution n'est pas encore terminée.

L'utilisation principale de la pile d'appels est de garder la trace de l'endroit où chaque fonction active doit retourner à la fin de son exécution. En pratique, lorsqu'une fonction est appelée par un programme, son adresse de retour (adresse de l'instruction qui suit l'appel) est empilée sur la pile d'appels. En plus d'emmagasiner des adresses de retour, la pile d'exécution stocke aussi d'autres valeurs, comme les variables locales de la fonction, les paramètres de la fonction, etc...

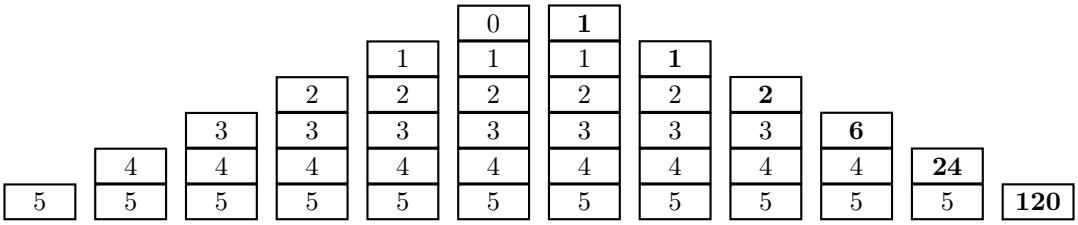
En particulier, lors d'appels *imbriqués* c'est à dire lorsqu'une fonction f appelle une fonction g, ce qui est relatif à l'appel de la fonction g est placé juste au dessus de ce qui est relatif à la fonction f. Lorsque g termine son exécution, ce qui est relatif à l'exécution de g est *dépilé*. Comme l'adresse de retour est contenue dans la pile d'appel, l'exécution de f peut reprendre juste après l'endroit où g a été appelée. Voici un exemple :

```
def h(x): return x**2
def g(x): return h(x)+h(2*x)
def f(x): return g(x)+h(3*x)
print(f(1))
```

Voici les états successifs de la pile d'appels lors de l'évaluation de $f(1)$ (en dessous se trouve le programme principal et l'appel à `print`, non représentés).



Appels récursifs. Avec ce mécanisme, rien n'interdit à une fonction de s'appeler elle-même : les appels successifs à une fonction récursive f s'empilent dans la pile d'appels, voici l'évolution de la pile lors de l'appel `fact_rec(4)` (on n'inscrit que les arguments de `fact_rec`, et en gras la valeur de retour).



On voit que le nombre *d'appels imbriqués* réalisés par une fonction récursive peut être important : il faut stocker ces appels, ce qui est coûteux en mémoire. En Python, il est impossible de dépasser un certain nombre d'appels récursifs, comme on va le voir un peu plus loin.

Un rappel sur les variables dans les fonctions. On rappelle que, sauf mention du contraire, les affectations dans un appel de fonction sont locales à cet appel, ce qui vaut également pour les affectations aux paramètres d'entrée. Dans l'explication sur le mécanisme des appels, on a réalisé plusieurs appels de fonctions : à chaque fois, l'affectation d'une valeur au paramètre `x` est locale à l'appel en question, ce qui évite de se mélanger les pinceaux ! De même, dans l'exécution de `fact`, chaque affectation de `n` est locale à l'appel en question.

5.1.3 D'autres exemples

Algorithme d'Euclide. Soient a et b deux entiers naturels, avec $a > 0$. Si $b \neq 0$, on a la relation $\text{PGCD}(a, b) = \text{PGCD}(b, r)$, où r est le reste dans la division euclidienne de a par b. L'algorithme usuel de calcul du PGCD consiste donc à faire des divisions euclidiennes :

```
def PGCD(a,b):
    while b>0:
        a,b=b,a%b
    return a
```

La version récursive repose sur le même principe :

```
def PGCD(a,b):
    if b==0:
        return a
    else:
        return PGCD(b,a%b)
```

Calcul de puissances. On a déja vu deux méthodes pour calculer x^n, un algorithme d'exponentiation naïf (faisant usage d'une boucle `for`, calculant toutes les puissances de x entre 1 et n), et un algorithme d'exponentiation rapide (basé sur la décomposition en binaire de n). Les deux admettent des équivalents récursifs.

```
def expo(x,n):
    if n==0:
        return 1
    else:
        return x*expo(x,n-1)
```

```
def expo_rapide(x,n):
    if n==0:
        return 1
    else:
        y=expo_rapide(x,n//2)
        if n%2==0:
            return y*y
        else:
            return y*y*x
```

On notera que l'expontentiation rapide est plus simple à écrire en récursif, elle est basée sur l'égalité suivante :

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ (x^{\frac{n}{2}})^2 & \text{si } n \text{ est pair, non nul} \\ x \times (x^{\frac{n-1}{2}})^2 & \text{si } n \text{ est impair} \end{cases}$$

(Rappel : en Python, `n//2` vaut $\frac{n}{2}$ si n est pair, $\frac{n-1}{2}$ sinon).

5.1.4 Limites de la récursivité

L'usage de la récursivité présente deux inconvénients : il faut faire attention à ce que les appels récursifs ne se chevauchent pas, et prendre garde à ne pas faire un trop grand nombre d'appels récursifs imbriqués.

Chevauchement des appels récursifs. Considérons l'exemple suivant : soit $(F_n)_{n\in\mathbb{N}}$ la suite définie par

$$F_n = \begin{cases} 1 & \text{si } n = 0 \text{ ou } 1. \\ F_{n-2} + F_{n-1} & \text{sinon} \end{cases}$$

Vous aurez probablement reconnu la fameuse suite de Fibonacci. Une transcription récursive en Python s'obtient aisément [2] :

```
def fib_rec(n):
   assert n>=0
   if n==0 or n==1:
      return 1
   return fib_rec(n-1)+fib_rec(n-2)
```

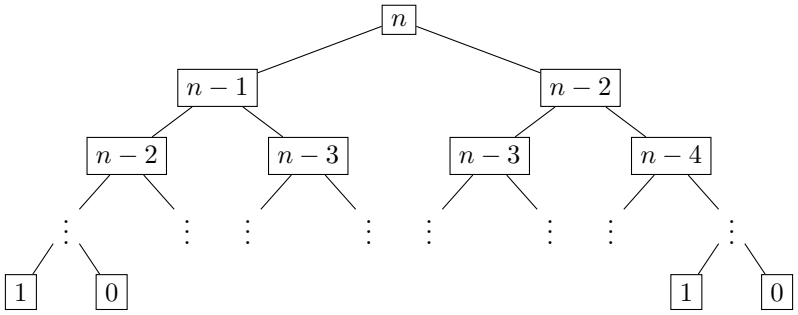


FIGURE 5.1 – Un arbre d'appels récursifs

Le problème de l'algorithme précédent est le nombre d'appels récursifs effectués, et donc la complexité. On peut montrer que $F_n \underset{n\to+\infty}{\sim} C \cdot \left(\frac{1+\sqrt{5}}{2}\right)^n$, où C est une constante. La fonction précédente calcule F_n en additionnant uniquement des zéros et des uns (les cas de base) : il s'ensuit que le nombre d'opérations pour le calcul de F_n est minoré par F_n (on peut montrer qu'elle est bien en $O(F_n)$).

Il vaut mieux utiliser une méthode itérative dans ce cas, comme la suivante [3] :

```
def fib(n):
    a,b=1,1
    for i in range(n-1):
        a,b=b,a+b
    return b
```

On peut observer que la version itérative est moins claire que la version récursive. Elle a l'avantage de s'effectuer en temps linéaire [4] en n.

En conclusion, il faut faire attention à ne pas faire des appels récursifs qui se recoupent, c'est-à-dire calculent la même chose, sous peine de voir la complexité exploser !

Taille de la pile (Python). Supposons que l'on souhaite calculer $n!$, pour n relativement grand [5], disons 10000. A priori, les deux versions (itérative et récursive) font l'affaire. Or l'appel à `fact_rec(10000)` produit un message d'erreur dont l'intitulé est `RuntimeError: maximum recursion depth exceeded`. En d'autres termes, le nombre *maximal* d'appels de fonctions imbriqués a été atteint. En Python, ce niveau est fixé à 1000. Cette valeur arbitraire peut-être augmentée [6] mais elle est là pour éviter un dépassement de capacité de la pile d'appels : le fameux « stack overflow » [7].

Temps d'exécution : itératif contre récursif. Stocker systématiquement l'état d'une fonction avant chaque appel récursif dans la pile d'appels n'est pas gratuit, en temps comme en mémoire. Si la formulation itérative s'obtient facilement, il est en général préférable de l'utiliser.

5.1.5 Avantage de la récursivité

On a déja vu un avantage des fonctions récursives : leur formulation est plus simple que leur équivalent itératif. Montrons un exemple de problème pour lequel une solution récursive est très adaptée, mais pour lequel une solution itérative n'est pas facile à trouver : le problème des tours de Hanoï.

On dispose de n disques troués en leur centre, numérotés de 1 à n, de diamètres croissants. On se donne également 3 piquets (numérotés de A à C). Initialement, tous les disques sont enfilés sur le premier piquet, le plus grand étant à la base, le plus petit au sommet, comme sur la figure 5.2.

Le but du jeu est d'amener les disques sur le troisième piquet, en suivant les règles suivantes :

— déplacer les disques un à un d'un piquet à un autre ;

2. Notez que l'on effectue *deux* appels récursifs ici, pour $n > 1$.

3. On verra en deuxième année un moyen pour calculer F_n récursivement en gardant en mémoire (on dit *mémoïzer*) les F_i déja calculés, à l'aide d'un dictionnaire, ce qui permet une complexité en $O(n)$ pour le calcul de F_n.

4. Une autre remarque : cette fonction permet de calculer tous les termes de la suite. On peut faire mieux si on cherche uniquement le n-ième, voir TD !

5. Rappelons à toute fin utile que les entiers ne sont pas bornés en Python.

6. À l'aide de la fonction `setrecursionlimit` du module `sys`.

7. Sur mon ordinateur personnel, le calcul de $n!$ avec la méthode récursive ne passe plus à partir de 20000 environ.

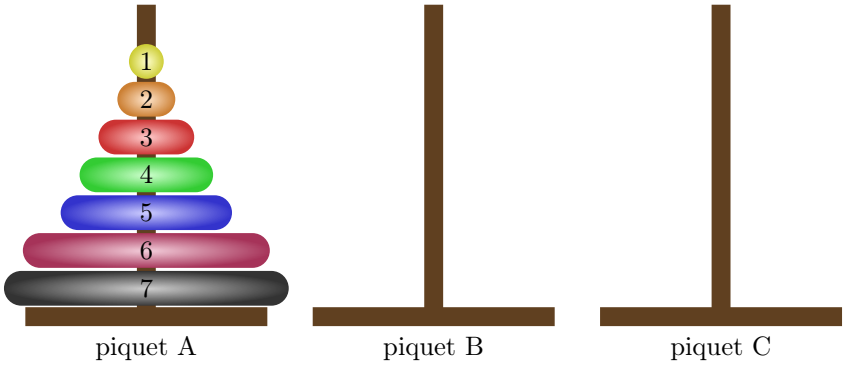


FIGURE 5.2 – Le jeu de Hanoï : comment déplacer les 7 disques du piquet A au piquet C, en suivant les règles ?

— un disque ne doit jamais être posé sur un disque de diamètre inférieur.

La figure 5.3 montre les 7 mouvements à effectuer pour résoudre le jeu avec seulement 3 disques. Il est facile de voir qu'il faut 127 mouvements pour le jeu à 7 disques (voir la suite).

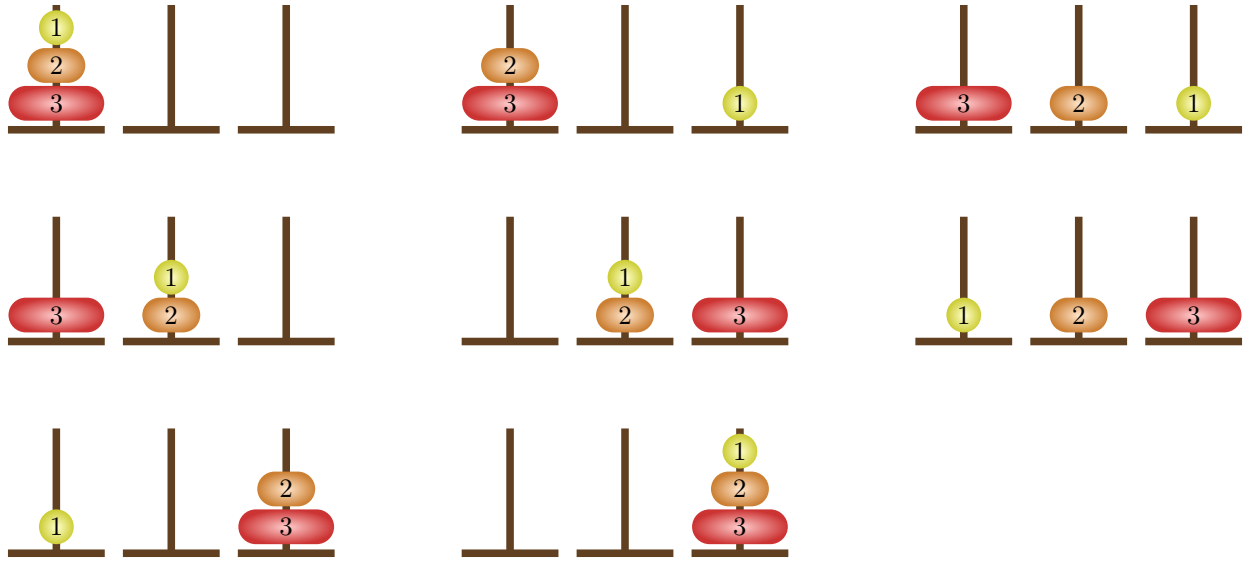


FIGURE 5.3 – Résolution du jeu de Hanoi pour $n = 3$

On cherche à donner les mouvements de disques à effectuer pour résoudre le jeu. De manière itérative, il n'est pas évident à résoudre, mais il est très facile de le faire lorsqu'on pense à la récursivité. Soient i, j et k trois caractères tels que $\{i, j, k\} = \{A, B, C\}$, et $n \in \mathbb{N}$. Pour faire passer n disques du piquet i au piquet j :

— il n'y a rien à faire si $n = 0$;

— pour $n \geq 1$, il suffit de faire passer les $n-1$ disques numérotés de 1 à $n-1$ du piquet i au piquet k, de déplacer ensuite le disque n du piquet i au piquet j, puis de refaire passer les $n-1$ disques du piquet k au piquet j. Le fait de travailler avec les disques les plus petits permet de ne pas violer la deuxième règle.

Écrivons donc une fonction Python qui imprime à l'écran la suite des mouvements à effectuer pour résoudre le jeu. Un mouvement est décrit comme `i -> j`, ce qui signifie faire passer le disque supérieur du piquet i au piquet j :

```
def deplacement(i,j):
    print(i, " -> ", j)
```

La discussion précédente invite à écrire une fonction `hanoi(n)` résolvant le jeu à n disques, qui fait un unique appel à une fonction récursive interne `aux(n,i,j,k)` qui doit donner la suite des mouvements permettant de faire passer n disques du piquet i au piquet j, avec $\{i, j\} \subset \{A, B, C\}$. Pour des raisons de commodité, il est pratique d'indiquer le dernière lettre parmi $\{A, B, C\}$ dans une variable (k) :

```
def hanoi(n):
    """ problème de Hanoi: déplacer une pile de n disques du piquet 1 au piquet 3 """
    def aux(n,i,j,k):
        """ deplacer n disques du piquet i au piquet j, le piquet restant étant k. {i,j,k}={1,2,3}"""
        if n!=0:
            aux(n-1,i,k,j)
            deplacement(i,j)
            aux(n-1,k,j,i)
    aux(n,"A","C","B")
```

Testons avec $n = 3$:

```
>>> hanoi(3)
A  ->  C
A  ->  B
C  ->  B
A  ->  C
B  ->  A
B  ->  C
A  ->  C
```

On retrouve les mouvements de la figure 5.3. Il est facile de montrer par récurrence que le nombre de mouvements produits est $2^n - 1$: c'est optimal [8].

Comme on le voit sur cet exemple, les fonctions où plusieurs appels récursifs sont nécessaires ne sont pas vraiment faciles à traduire de façon itérative (à moins d'utiliser une pile pour essentiellement réécrire la récursivité...) : c'est un avantage de l'emploi de fonctions récursives.

5.2 Terminaison, correction et complexité d'une fonction récursive

Montrer la terminaison d'une fonction récursive signifie démontrer que le processus récursif s'arrête pour tous paramètres. Démontrer sa correction signifie prouver que la fonction a bien le comportement attendu. Donner sa complexité signifie estimer son coût en ressources (complexité temporelle voire complexité spatiale).

L'analyse d'un programme récursif est en général (en tout cas à notre niveau) un peu plus simple que pour un programme itératif. Voyons ça dans le détail.

5.2.1 Terminaison

Pour montrer la terminaison d'une fonction récursive, il suffit d'exhiber une quantité, dépendant des paramètres de la fonction, à valeurs dans $\mathbb{N}$, dont les valeurs décroissent strictement au cours des appels récursifs successifs. Pour la fonction factorielle, il suffit de prendre le paramètre n lui-même. Considérons un exemple où la fonction en question est (un peu) moins évidente : la recherche dichotomique dans une liste triée, qui est explicitement au programme (au moins dans sa version itérative).

```
def dicho_rec(L,x):
    """L liste triée dans l'ordre croissant, x un élément. On renvoie True si x est dans L, False sinon"""
    n=len(L)
    if n==0:
        return False
    m=n//2
    if L[m]==x:
        return(True)
    elif L[m]<x:
        return dicho_rec(L[m+1:],x) #la partie à droite de L[m].
    else:
        return dicho_rec(L[:m],x) #la partie à gauche.
```

La fonction `dicho_rec(L,x)` retourne un booléen caractérisant le fait que `x` est dans `L` ou non. L'idée de la recherche dichotomique est simple :

— Si la liste est vide, `x` n'y est pas ;

8. La fonction a aussi une complexité en $O(2^n)$, ce qui est exponentiel... Mais là on ne peut pas faire mieux !

— Sinon, on regarde l'élément situé au milieu de la liste (d'indice $\lfloor n/2 \rfloor$ avec n la taille de la liste). Si c'est `x`, on a terminé, sinon le fait que la liste soit triée nous permet de chercher `x` uniquement dans la partie droite (éléments d'indices au moins $m+1$, si `x > L[m]`), ou gauche (éléments d'indices au plus $m-1$, si `x < L[m]`).

La terminaison de la fonction `dicho_rec` est alors facile à montrer : une quantité à valeurs dans $\mathbb{N}$, dépendant des paramètres de la fonction, qui décroît strictement à chaque appel récursif est la longueur de la liste `L`. Ainsi, la fonction termine.

Attention aux coûts cachés ! Une remarque : en terme de complexité, la fonction précédente est très mauvaise, car l'extraction d'une partie de la liste (`L[:m]` ou `L[m+1:]`) est de complexité linéaire en la taille de la partie extraite. En fait, pour une liste de taille n, on peut montrer que la complexité de la recherche d'un élément avec `dicho_rec` est $O(n)$, ce qui n'est pas meilleur que la recherche classique dans une liste non triée. Voir la section complexité pour une meilleure complexité, utilisant une fonction auxiliaire pour éviter le recopiage de listes.

5.2.2 Correction

Pour prouver la correction d'une fonction récursive, on procède en général par récurrence : on prouve d'abord que la sortie de la fonction est correcte pour les cas *terminaux*, ce qui correspond à l'initialisation de la récurrence, et on montre ensuite que les appels récursifs se ramènent à des instances plus petites (dans le même sens que la terminaison), ce qui constitue l'hérédité. La plupart du temps, la correction est facile à prouver. Par exemple pour la fonction `dicho_rec` définie plus haut, on considère la proposition suivante :

Si `L` est une liste triée dans l'ordre croissant et `x` un élément comparable à ceux de `L`, alors `dicho_rec(L,x)` retourne `True` si et seulement si `x` est dans `L`, False sinon.

— Initialisation : si la longueur du tableau est 1, alors la sortie de la fonction est correcte.

— Hérédité : si `L` est de longueur au moins 2, un et un seul des cas suivants se produit :

  — `L[m]` est égal à `x`, auquel cas la sortie de la fonction est correcte.

  — `L[m]` est inférieur strictement à `x`, auquel cas `x` ne peut se trouver qu'après `m` puisque `L` est trié dans l'ordre croissant. Le tableau `L[m:]` correspond aux éléments placés après `m` (inclus), triés également dans l'ordre croissant. Par hypothèse de récurrence, la sortie de la fonction `dicho_rec` sur l'instance `(L[m:],x)` est correcte, donc également sur `(L,x)`.

  — On procède de même,si `L[m]` est strictement supérieur à `x` avec le tableau `L[:m]`.

— Par principe de récurrence, la fonction `dicho_rec` est correcte.

On montre de même la terminaison et la correction des fonctions récursives introduites plus haut dans le chapitre.

5.2.3 Complexité des fonctions récursives

Comme pour les fonctions itératives, on se concentrera sur la complexité temporelle. Pour la complexité spatiale, elle n'est en général pas trop dur à estimer en suivant les mêmes principes, la différence résidant dans la pile d'appels récursifs dont le coût est un peu caché. La complexité temporelle s'estime de la même manière qu'avec des fonctions itératives, à la différence qu'on introduit des relations de récurrences qui suivent les appels récursifs, qu'il faut ensuite résoudre.

La factorielle. On a déja dit que $n!$ se calculait en complexité linéaire avec la fonction `fact_rec`. Vérifions-le en notant $C(n)$ la complexité associée. Si n est nul, il n'y a rien à faire d'autre que de retourner 1, ce qui se fait en temps constant $O(1)$. Sinon, il faut multiplier le résultat de `fact_rec(n-1)` par n. Outre l'appel récursif (complexité $C(n-1)$), il n'y a donc qu'un coût constant [9].On a donc

$$C(n) = \begin{cases} O(1) & \text{si } n = 0 \\ C(n-1) + O(1) & \text{sinon} \end{cases}$$

On vérifie immédiatement que $C(n) = O(n)$ est solution de cette récurrence, car $C(n) = \sum_{k=1}^{n} [C(k) - C(k-1)] + O(1) = n \times O(1) = O(n)$. (Remarque : on abuse ici de la notation O, il faudrait préciser que la constante cachée ne dépend pas de n pour que le raisonnement soit valable. Mais cet abus est fait couramment en informatique).

9. Ce qui n'est pas tout à fait vrai, car les opérations sont plus complexes lorsque la taille des entiers augmentent. On compte ici les opérations arithmétiques, pas les opérations binaires.

Tours de Hanoï. Pour le problème des tours de Hanoï, en notant $C(n)$ la complexité requise pour le calcul des mouvements nécessaires au déplacement de n disques, on établit facilement que $C(n) = 2C(n-1) + O(1)$. Il s'ensuit que $C(\frac{n}{2^n}) = C(\frac{n-1}{2^{n-1}}) + O(\frac{1}{2^n})$. En sommant les termes de la suite associée, on en déduit $C(\frac{n}{2^n}) = O(\sum_{k=0}^{n} \frac{1}{2^k}) = O(1)$. D'où $C(n) = O(2^n)$.

Récurrences usuelles. En se souvenant que (voir le cours de mathématiques) $\sum_{k=0}^{n} n^\alpha = O(n^{\alpha+1})$ pour $\alpha \geq 0$, où que $\sum_{k=0}^{n} q^k = O(q^n)$ pour $q > 1$, on peut résoudre des récurrences similaires aux deux précédentes.

Recherche dichotomique récursive. Donnons maintenant une implémentation correcte (en terme de complexité) de la recherche dichotomique récursive dans une liste triée. On reprend l'idée du code précédent, mais en utilisant une fonction auxiliaire interne pour éviter les copies. Elle prend en argument deux entiers délimitant la portion dans laquelle peut se trouver l'élément cherché.

```
def dicho_rec(L,x):
    def aux(g,d):
        """ renvoie True si x est dans L[g:d], False sinon. """
        if g>=d:
            return False
        m=(g+d)//2
        if L[m]==x:
            return True
        elif L[m]>x:
            return aux(g,m)
        else:
            return aux(m+1,d)
    return aux(0,len(L))
```

En notant $p = d - g$, on s'aperçoit que la fonction `aux` a une complexité vérifiant [10] $C(p) = C(\lfloor p/2 \rfloor) + O(1)$. Faisons le calcul pour p une puissance de 2, qui s'écrit donc $p = 2^k$, on a pour $k \geq 1$: $C(2^k) = C(2^{k-1}) + O(1)$. On peut alors télescoper cette somme. Pour tout $k \geq 1$:

$$C(2^k) = \sum_{i=1}^{k} \underbrace{C(2^i) - C(2^{i-1})}_{O(1)} + \underbrace{C(1)}_{O(1)} = O(k)$$

Or $k = \log_2(2^k)$, on a donc $C(p) = O(\log p)$ pour p une puissance de 2. On admet le résultat pour p quelconque.

Conclusion sur la récursivité

On a vu dans ce cours les avantages et les inconvénients de la récursivité.

- **Inconvénients :**
  - Il faut faire attention à ne pas faire trop d'appels récursifs imbriqués.
  - Il faut faire attention à ne pas faire plusieurs fois les mêmes calculs, sous peine de voir la complexité exploser.
  - Lorsque deux solutions sont équivalentes, l'une en récursif, l'autre en itératif, il est en général préférable d'utiliser la version itérative.
- **Avantages :**
  - Dans l'ensemble, il est plus facile de prouver un algorithme récursif que son équivalent itératif.
  - L'écriture d'un programme récursif est souvent plus claire (plus mathématique), que son équivalent itératif.
  - Parfois, et c'est là où la récursivité est vraiment importante, il n'est pas du tout aisé de transformer un algorithme récursif en algorithme itératif. C'est en général le cas des algorithmes « diviser pour régner », dont on verra un exemple avec le tri fusion, ou la construction de fractales (en TP). D'une manière générale, lorsqu'une fonction récursive fait plusieurs appels récursifs, elle n'est pas immédiate à traduire en itératif.

10. Pas tout à fait, c'est même « un peu moins », car la portion à droite de l'indice m a pour taille $\lceil p/2 \rceil - 1 \leq \lfloor p/2 \rfloor$. Mais le résultat est le même.

Troisième partie

Compléments de Python

Chapitre 6

Représentation des entiers relatifs, représentation des flottants

On a déja vu comment représenter des entiers naturels dans différentes bases, notamment en binaire. On va maintenant expliquer ce qu'il se passe plus précisément dans les registres d'un processeur (les nombres sont en binaire, mais le nombre de bits est fixe), comment la représentation choisie s'étend aux entiers relatifs, et enfin comment représenter les nombres flottants.

6.1 Représentation des entiers relatifs en binaire, additions

On se concentre maintenant sur les entiers en base 2. D'après le théorème 2.2, un entier strictement positif N s'écrit donc de manière unique $N = \sum_{k=0}^{n-1} b_k 2^k$, avec $n \geq 1$, $b_{n-1} = 1$ et $a_i \in \{0, 1\}$ pour $i < n-1$. Les entiers (b_k) sont appelés les bits de N.

6.1.1 Entiers naturels de taille fixée et additions

Entier naturel de taille fixée. En pratique en informatique, les entiers sont stockés dans des emplacements mémoire ayant une taille fixée : aujourd'hui les *registres* d'un microprocesseur ont une taille de 32 ou 64 bits. On suppose donc maintenant que nos entiers ont une taille n fixée (par exemple $n = 64$), et on note toujours $N = \sum_{k=0}^{n-1} b_k 2^k$, mais on ne suppose plus que b_{n-1} soit égal à 1. Ainsi, le plus petit nombre que l'on peut représenter est $\underbrace{\overline{00\cdots0}}_{n}^{2} = 0$ et le plus grand est $\underbrace{\overline{11\cdots1}}_{n}^{2} = \sum_{k=0}^{n-1} 2^k = \frac{2^n-1}{2-1} = 2^n - 1$. Le bit b_0 est appelé *bit de poids faible* et le bit b_{n-1} est le *bit de poids fort*.

Additions. L'addition sur entiers naturels se fait comme sur les entiers en base 10 : il suffit de savoir comment additionner deux chiffres et propager les retenues. C'est particulièrement facile en binaire, puisqu'il n'y a que 2 chiffres ! La table d'addition est la suivante :

+	0	1
0	0	1
1	1	10

Le 10 signifie que le résultat fait 0 et qu'il faut ajouter un bit de retenue. L'addition se fait de droite à gauche, comme à l'école primaire. Par exemple sur 6 bits (les 1 en exposants sont des retenues) :

$$\begin{array}{cccccccc} & 1 & 0^1 & 0^1 & 1 & 0 & 1 \\ + & 0 & 0 & 1 & 1 & 1 & 0 \\ \hline & 1 & 1 & 0 & 0 & 1 & 1 \end{array}$$

En repassant en base 10, on vérifie que $37 + 14$ vaut bien 51.

Dépassement de capacité sur entiers naturels. Il n'est pas exclu que la dernière addition génère une retenue (on parle de retenue sortante). Dans ce cas, le résultat de l'addition des deux entiers de n bits ne tient pas sur n bits : on parle alors de dépassement de capacité. Prenons un exemple sur 8 bits :

$$\begin{array}{cccccccccc} & 1 & 0 & 0^1 & 1^1 & 0^1 & 1 & 0 & 1 \\ + & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ \hline \boxed{1} & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \end{array}$$

Le 1 encadré correspond à la retenue issue de l'addition des deux derniers bits. Il faudrait donc 9 bits pour représenter la somme. En effet, 149+142 vaut 291, qui dépasse $255 = 2^8 - 1$, la valeur maximale représentable sur 8 bits. C'est exactement un tel dépassement de capacité (oui, sur 8 bits !) qui a causé le crash d'Ariane 5 en 1996 [1].

En interne. En pratique, un additionneur n bits correspond au chaînage de n additioneurs 1 bit de la forme de la figure 6.1.

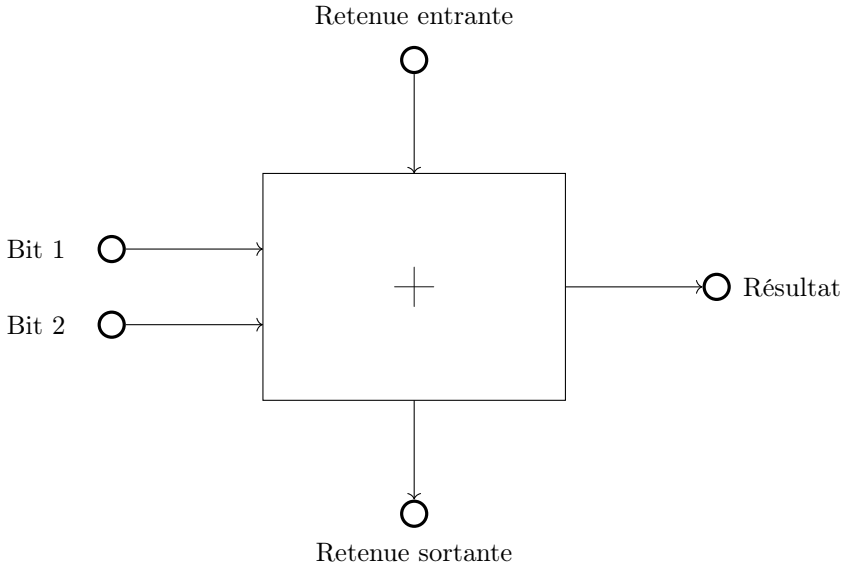


FIGURE 6.1 – Additionneur 1 bit

Le chaînage se fait en connectant les retenues entrantes et sortantes des additionneurs successifs (on positionne la retenue entrante initiale à 0). Les valeurs du résultat et de la retenue sortante dans un additionneur 1 bit en fonction des deux bits d'entrée et de la retenue entrante sont données dans la tableau suivant.

Bit 1	0	0	0	0	1	1	1	1
Bit 2	0	0	1	1	0	0	1	1
Retenue entrante	0	1	0	1	0	1	0	1
Résultat	0	1	1	0	1	0	0	1
Retenue sortante	0	0	0	1	0	1	1	1

6.1.2 Entiers relatifs

On va voir une représentation des entiers relatifs qui permet d'additionner en faisant exactement la même chose qu'avec des entiers naturels !

Représentation par valeur absolue. Une première idée pour représenter des entiers relatifs sur n bits est d'utiliser le premier bit comme bit de signe, les autres bits étant dévolus à la représentation de la valeur absolue de l'entier. Avec la convention que le bit de signe 1 est utilisé pour les nombres négatifs et 0 pour les nombres positifs, on obtient par exemple sur 6 bits :

$$\overline{011010}^2 = 26 \qquad \text{et} \qquad \overline{100001}^2 = -1.$$

1. En fait, l'accélération horizontale était codée comme un entier sur 8 bits, comme sur les versions précédentes des fusées Ariane. Seulement, Ariane 5 étant beaucoup plus puissante, cette accélération pouvait atteindre la valeur 300, qui nécessite 9 bits sur entiers naturels. Le dépassement de capacité a produit une valeur aberrante, qui a mené le logiciel à ordonner la destruction de la fusée.
Source : `https://fr.wikipedia.org/wiki/Vol_501_d%27Ariane_5`.

De cette façon, on représente l'ensemble des entiers de l'intervalle $[\![-(2^{n-1}-1), 2^{n-1}-1]\!]$, avec deux zéros. Le zéro « positif » est $00\cdots0$, et le zéro « négatif » est $100\cdots0$. Le problème de cette représentation est qu'elle ne permet pas d'effectuer facilement les additions.

Représentation en complément à 2. C'est la représentation des nombres relatifs la plus utilisée. Pour $a_{n-1}\cdots a_0$ une séquence de n bits, on considère que ce nombre représente :

$$a_{n-1}\cdots a_0 = \begin{cases} \sum_{k=0}^{n-2} a_k 2^k & \text{si } a_{n-1}=0 \\ -2^{n-1} + \sum_{k=0}^{n-2} a_k 2^k & \text{si } a_{n-1}=1 \end{cases}$$

Puisque la séquence $\sum_{k=0}^{n-2} a_k 2^k$ peut décrire tous les entiers naturels entre 0 et $2^{n-1}-1$, la représentation en complément à 2 permet de représenter tous les nombres de l'intervalle $[\![-2^{n-1}, 2^{n-1}-1]\!]$, de manière unique. Notez qu'il est facile de voir si un nombre est positif ou strictement négatif dans cette représentation : il suffit de regarder le bit de poids fort. S'il est nul, le nombre est positif, sinon, il est strictement négatif. En reformulant, on représente $N \in [\![-2^{n-1}, 2^{n-1}-1]\!]$ par :

$$\begin{cases} \text{la représentation de } N \text{ sur } n \text{ bits en entier naturel, si } N \geq 0 \\ \text{la représentation de } 2^n + N = 2^n - |N| \text{ sur } n \text{ bits en entier naturel, si } N < 0. \end{cases}$$

Donnons tout de suite la table des entiers sur 4 bits pour clarifier les choses. Ici $n=4$, donc on peut représenter les nombres de -8 à 7.

suite de bits	0000	0001	0010	0011	0100	0101	0110	0111
signification sur entiers naturels	0	1	2	3	4	5	6	7
signification sur entiers relatifs	0	1	2	3	4	5	6	7
suite de bits	1000	1001	1010	1011	1100	1101	1110	1111
signification sur entiers naturels	8	9	10	11	12	13	14	15
signification sur entiers relatifs	-8	-7	-6	-5	-4	-3	-2	-1

On observe bien que dans le cas où la suite de bits commence par un 1, on déduit sa signification sur entiers relatifs en complément à 2 de celle sur entiers naturels par l'opération $x \mapsto -2^4 + x$. Cette représentation devrait s'appeler *complément à* 2^n, mais est rentrée dans le langage courant sans référence au nombre de chiffres de l'entier représenté.

Remarque 6.1 (Entier naturel ou relatif ?)**.** *Une même suite de bits $a_{n-1}a_{n-2}\cdots a_0$ peut donc avoir deux significations différentes. Savoir si elle représente un entier naturel ou un entier relatif*[2] *ne dépend pas de la mémoire (qui se contente de stocker des bits), ni du processeur (qui se contente d'effectuer des opérations), mais du programme qui manipule cette suite de bits.*

Addition d'entiers relatifs. Montrons sur quelques exemples que, s'il n'y a pas dépassement de capacité, le résultat de l'addition faite avec cette représentation comme avec des entiers naturels donne le bon résultat.

Pour chacun des couples (3,4), (-1,6) et (-2,-3), représentable sur 4 bits en tant qu'entiers relatifs, le résultat de l'addition appartient à l'intervalle $[\![-8, 7]\!]$, il n'y a donc pas dépassement de capacité de l'addition sur entiers relatifs avec 4 bits disponibles.

$$\begin{array}{cccccc} & 0 & 0 & 1 & 1 \\ + & 0 & 1 & 0 & 0 \\ \hline \boxed{0} & 0 & 1 & 1 & 1 \end{array} \qquad \begin{array}{cccccc} & 1^1 & 1^1 & 1 & 1 \\ + & 0 & 1 & 1 & 0 \\ \hline \boxed{1} & 0 & 1 & 0 & 1 \end{array} \qquad \begin{array}{cccccc} & 1^1 & 1 & 1 & 0 \\ + & 1 & 1 & 0 & 1 \\ \hline \boxed{1} & 1 & 0 & 1 & 1 \end{array}$$

Le chiffre encadré contient l'éventuelle retenue sortante. On voit que le résultat est correct, à condition de ne pas tenir compte de cette retenue sortante. Montrons que le résultat est en effet correct :

Théorème 6.2. *Soit n un entier strictement positif, et N et M deux nombres entiers appartenant à l'intervalle $[\![-2^{n-1}, 2^{n-1}-1]\!]$. Si $N+M$ appartient lui aussi à cet intervalle, alors la représentation sur entiers relatifs en complément à 2 de l'entier $N+M$ se déduit de celles de N et de M par addition sur entiers naturels, en ignorant l'éventuel bit de retenue sortante.*

Démonstration. On va distinguer les cas suivants si N et M sont positifs ou strictement négatifs.

2. ou autre chose !

— Supposons que N et M sont tous deux positifs ou nuls. Alors leur représentation sur entiers relatifs en complément à 2 sur n bits correspond à celle sur entiers naturels, et le résultat de l'addition est celui de $N+M$ comme entier naturel sur n bits. Comme le résultat est supposé appartenir à $[\![-2^{n-1}, 2^{n-1}-1]\!]$, le bit de poids fort de la somme est zéro, et correspond bien à la représentation sur entiers relatifs en complément à 2 sur n bits de $N+M$.

— Supposons $N \geq 0$ et $M < 0$. Remarquez que dans ce cas, la somme $N+M$ appartient forcément à l'intervalle $[\![-2^{n-1}, 2^{n-1}-1]\!]$: il ne peut y avoir dépassement de capacité en additionnant deux nombres de signes contraires. L'addition sur entier naturels correspond à l'entier $2^n + N + M$. Deux cas sont à distinguer :

  — Si $N+M \geq 0$, alors 2^n correspond à la retenue sortante : si on l'ignore on obtient bien $N+M$ comme entier positif en représentation sur entiers relatifs en complément à 2.

  — Si $N+M < 0$, alors 2^n+N+M correspond précisément à la représentation sur entiers relatifs en complément à 2 de $N+M$.

  Dans les deux cas, le résultat est correct.

— Si $N < 0$ et $M \geq 0$, le résultat est correct : il suffit de reprendre le raisonnement précédent en échangeant N et M.

— Enfin, si N et M sont tous deux strictement négatifs, l'addition correspond à l'addition sur entiers naturels de $2^n + N + 2^n + M \geq 2^n$. Il y a donc nécessairement une retenue sortante et l'ignorer revient à considérer l'entier naturel $2^n + N + M$, qui correspond bien à la représentation sur entiers relatifs en complément à 2 de $N+M$ puisque $N+M$ est strictement négatif

□

Comme on l'a dit dans la preuve, le dépassement de capacité (c'est-à-dire que $N+M$ n'appartient pas à l'intervalle $[\![-2^{n-1}, 2^{n-1}-1]\!]$) ne peut se produire que si N et M sont de même signe. Voici deux exemples de dépassement sur 4 bits donnés par les couples (5,6) et (-8,-1) :

$$\begin{array}{cccccc} & & 0^1 & 1 & 0 & 1 \\ + & & 0 & 1 & 1 & 0 \\ \hline & \boxed{0} & 1 & 0 & 1 & 1 \end{array} \qquad \begin{array}{cccccc} & & 1 & 0 & 0 & 0 \\ + & & 1 & 1 & 1 & 1 \\ \hline & \boxed{1} & 0 & 1 & 1 & 1 \end{array}$$

Remarque 6.3. *On aurait pu affiner le théorème précédent en montrant de plus qu'il y a dépassement de capacité, si et seulement si les deux dernières retenues (la retenue sortante et la retenue sur le bit de poids fort) sont différentes. Vous pouvez vérifier sur les exemples. En pratique, c'est comme cela que fonctionne l'additionneur d'une unité arithmétique et logique d'un processeur : l'additionneur est réalisé en connectant des additionneurs 1 bits, de plus on peut détecter les dépassements de capacité sur entiers naturels (la retenue sortante vaut 1) et sur entiers relatifs (la retenue sortante est différente de la dernière retenue). Le programme qui a lancé l'opération peut récupérer ces informations pour éventuellement prendre en compte le dépassement de capacité, par exemple pour avertir l'utilisateur.*

6.1.3 En pratique

Dans un ordinateur, on utilise maintenant des registres de 32 ou 64 bits, ce qui autorise la représentation d'entiers relatifs dans les intervalles $[\![-2^{31}, 2^{31}-1]\!]$ ou $[\![-2^{63}, 2^{63}-1]\!]$. Si le résultat d'un calcul ne rentre pas dans l'intervalle, le résultat est erroné.

Par exemple, dans un langage de bas niveau comme le C, le type `int` correspond à des entiers relatifs codés sur 32 bits. Le calcul et l'affichage des puissances de 3 successives produit le résultat (tronqué) suivant :

```
3^18=387420489
3^19=1162261467
3^20=-808182895
```

Que se passe-t-il ? On a l'encadrement suivant : $3^{19} < 2^{31}-1 < 3^{20}$. Le calcul de 3^{20} produit donc un dépassement de capacité, ce qui explique le résultat aberrant [3]. Fort heureusement, il est possible de calculer avec des entiers plus longs en C, mais voila la preuve qu'il faut faire attention : on n'a pas eu droit à un message d'erreur !

Et en Python ? En Python, les entiers sont *non bornés*. Par exemple, Python n'a aucun mal à calculer et afficher [4] correctement 4444^{4444}. Les longs entiers sont en fait codés par paquets de bits de longeur fixée, et il peut y avoir un nombre potentiellement infini de paquets (limité par la mémoire, naturellement). Tout ceci se fait de manière transparente pour l'utilisateur, on n'aura donc pas à se soucier des dépassements de capacité lorsqu'on manipulera des entiers.

3. Pas si aberrant que ça : le résultat obtenu pour 3^{20} est exactement $3^{20} - 2^{32}$!
4. Ce que je ne ferai pas ici, il y a quand même plus de 16000 chiffres !

Exemple Numpy. Les entiers sont non bornés en Python. Néanmoins, le module Numpy utilise des entiers bornés pour des raisons d'efficacité. Les entiers naturels codés sur 8 bits sont utilisés par exemple pour représenter les valeurs des pixels d'une image, et ont donc un type propre : `uint8` (pour *unsigned integer 8*, entier positif codé sur 8 bits) :

```
>>> import numpy as np #importation de Numpy sous le nom np
>>> a=np.uint8(149) ; b=np.uint8(142) # 2 entiers naturels sur 8 bits
>>> a+b
<stdin>:1: RuntimeWarning: overflow encountered in ubyte_scalars
35
>>> 35+256 == 149+142 #la retenue sortante correspondrait à 256.
True
```

Le module Numpy possède également des types pour des entiers (naturels ou relatifs) codés sur 16, 32 ou 64 bits. Par exemple avec des entiers signés sur 32 bits, on retrouve le comportement ci-dessus :

```
>>> np.int32(3**20) #entier relatif sur 32 bits.
-808182895
```

Sur vos calculatrices. Dépendant de votre modèle, le nombre de bits maximal est différent, et bien qu'assez élevé, est limité. Vous pouvez faire une boucle, calculant par exemple les puissances de 3 jusqu'à 10000 pour voir si vous obtenez une erreur ou un résultat faux.

6.2 Représentation des nombres réels

Voyons maintenant comment exprimer des nombres réels en machine, ce que l'on fera dès qu'on manipulera des quantités physiques (résultats d'une mesure, par exemple). Prenons quelques constantes physiques célèbres [5] :

- La vitesse de la lumière dans le vide : $c_0 = 2.99792458 \times 10^8$ m.s^{-1}.
- Charge élémentaire : $e = 1.602176565 \times 10^{-19}$ A.s.
- Constante gravitationnelle : $G = 6.67384 \times 10^{-11}$ m^3.kg^{-1}.s^{-2}.
- Nombre d'Avogadro : $N_A = 6.02214129 \times 10^{23}$ mol^{-1}.
- Constante des gaz parfaits : $R_0 = 8.3144621$ J.K^{-1}.mol^{-1}.

Certaines sont *exactes*, comme la vitesse de la lumière dans le vide (c'est ainsi qu'on définit le mètre aujourd'hui), d'autres ont été *mesurées*. Lorsqu'on fait un calcul en physique, les résultats des mesures ne sont connus qu'avec une certaine précision. L'important est donc de pouvoir représenter des réels d'ordres de grandeur très différents, en gardant une précision suffisante pour chacun. La représentation *scientifique* utilisée ci-dessus s'y prête bien : on garde un certain nombre de *chiffres significatifs*, et on peut représenter des nombres très petits (en valeur absolue) ou très grands en jouant sur *l'exposant* dans la puissance de 10, qui peut être négatif ou positif.

6.2.1 Représentations des nombres dyadiques en binaire

Faisons une petite parenthèse sur les nombres dyadiques. Vous connaissez les nombres décimaux, par exemple 12.34, 3.14159 et -5.2. Ceux-ci sont les nombres réels ayant un « nombre fini de chiffres après la virgule [6] ». Par exemple, $\frac{1}{3} = 0.333\cdots$ ou encore $\pi = 3.14159\cdots$ n'en font pas partie. De même que l'écriture des entiers, l'écriture des nombres à virgule se généralise à toute base.

Les nombres dyadiques ne sont rien d'autres que ceux qui s'écrivent comme une somme finie de puissances de 2, ces puissances pouvant être à exposants positifs ou négatifs. On généralise l'écriture des entiers en base 2 à celle des nombres dyadiques. Par exemple, le nombre $\overline{101.001}^2$, s'interprète comme $\underbrace{2^2+2^0}_{\text{partie entière}} + \underbrace{0\times 2^{-1}+0\times 2^{-2}+1\times 2^{-3}}_{\text{partie dyadique non entière}}$ (soit 5.125).

De la même manière qu'un nombre décimal en notation scientifique en base 10 s'écrit sous la forme d'un signe, multiplié par un nombre décimal de l'intervalle $[1, 10[$, multiplié par une puissance de 10, en base 2, un nombre dyadique non nul s'écrit comme un signe, multiplié par un nombre à virgule de l'intervalle $[1, 2[$, multiplié par une puissance de 2. C'est sous cette forme que sont représentés les nombres en machine.

5. Les chiffres sont tirés de Wikipédia. On choisit d'utiliser la syntaxe anglo-saxonne dans ce cours, la virgule étant notée par un point.
6. Tout cela sera précisé en cours de mathématiques.

Définition 6.4. *Dans l'écriture signe× nombre à virgule de* $[1,2[\times 2^{exposant}$*, le nombre à virgule s'appelle* la mantisse.

Dans la représentation des nombres en mémoire, un bit est réservé au signe, et on notera m le nombre de bits réservés à la mantisse et e le nombre de bits réservé à l'exposant. En mémoire, on a donc $1+e+m$ bits consécutifs, comme ceci :

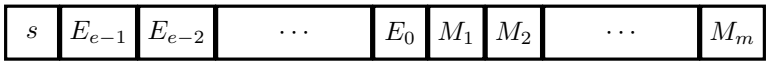


FIGURE 6.2 – La représentation des nombres flottants.

Avec un nombre de bits fixés, il n'est pas possible de représenter tous les réels, mais seulement un nombre fini d'entre eux. Ceux-ci sont appelés *nombres flottants*, ce sont tous des dyadiques. Il y a trois cas à distinguer :

— lorsque les bits $E_0,\dots,E_{e-1}$ ne sont ni tous nuls ni tous égaux à 1, on parle de flottant *normalisé* : c'est donc le plus courant ;

— lorsque $E_0=\cdots=E_{e-1}=0$, on parle de flottant *dénormalisé* ;

— le cas $E_0=\cdots=E_{e-1}=1$ est utilisé pour représenter les infinis et les NAN (voir la suite).

6.2.2 Nombres flottants normalisés

On reprend la suite de bits de la figure 6.2, dans le cas où les bits E_i ne sont ni tous nuls, ni tous égaux à un. L'interprétation est la suivante : cette suite de bits représente le nombre

$$x=\mathrm{S}\times\mathrm{M}\times 2^{\mathrm{E}-\mathrm{D}}$$

où :

— $\mathrm{S}=(-1)^s\in\{\pm 1\}$ est le *signe* de x, représenté par le bit s, avec la convention 0 pour un nombre positif et 1 pour un nombre négatif.

— M est la *mantisse*. C'est, pour un flottant normalisé, un nombre appartenant à l'intervalle $[1,2[$. La partie entière (1) est implicite et non représentée, si bien que les m bits de mantisse s'interprètent en $M=\overline{1.M_1\cdots M_m}^{\,2}=1+\sum_{k=1}^{m}M_i\times 2^{-i}$.

— $\mathrm{E}-\mathrm{D}$ est l'*exposant*. Les e bits s'interprètent comme l'entier naturel $\mathrm{E}=\overline{E_{e-1}\cdots E_0}^{\,2}=\sum_{k=0}^{e-1}E_k\times 2^k$, appelé *exposant décalé*. Puisque les E_i ne sont ni tous nuls ni tous égaux à 1, l'exposant décalé E est un entier de l'intervalle $[\![1,2^e-2]\!]$. Le décalage D ne dépend que du nombre de bits e, et a pour valeur $\mathrm{D}=2^{e-1}-1$. Ainsi $E-D\in[\![-2^{e-1}+2,2^{e-1}-1]\!]$.

Classiquement, on utilise une représentation des flottants en simple précision (32 bits) ou double précision (64 bits). Maintenant que les processeurs ont tous 64 bits, c'est plutôt la double précision qui s'impose. Notons qu'on trouve également des représentations avec plus de bits, pour une plus grande précision. Même si les entiers e et m changent, le principe est toujours le même.

On donne dans le tableau suivant le nombres de bits dévolus à la mantisse et à l'exposant décalé dans les représentations sur 32 et 64 bits :

format	signe	taille e de E	décalage D	taille m de la mantisse	signification
32 bits	1 bit	8 bits	$2^{8-1}-1=127$	23 bits	$(-1)^s\times\underbrace{\overline{1.M_1\cdots M_{23}}^{\,2}}_{\text{mantisse}}\times 2^{\mathrm{E}-127}$
64 bits	1 bit	11 bits	$2^{11-1}-1=1023$	52 bits	$(-1)^s\times\underbrace{\overline{1.M_1\cdots M_{52}}^{\,2}}_{\text{mantisse}}\times 2^{\mathrm{E}-1023}$

Donnons comme exemple la représentation du nombre 21.625 sur 32 bits. Ce nombre est un dyadique, qui s'écrit :

$$1\times 2^4+0\times 2^3+1\times 2^2+0\times 2^1+0\times 2^0+1\times 2^{-1}+0\times 2^{-2}+1\times 2^{-3}$$

Autrement dit, $21.625=\overline{10101.101}^{\,2}=\overline{1.0101101}^{\,2}\times 2^4$.

— Le signe est positif, le bit correspondant est donc 0.

— Les 23 bits de la mantisse sont obtenus en complétant 0101101 avec des zéros.

— L'exposant décalé E est obtenu en rajoutant à 4 le décalage (127) et en convertissant ce nombre en binaire. Ainsi $E = 131 = \overline{10000011}^2$.

Par suite, $21,625$ est représenté sur 32 bits comme 01000001101011010000000000000000.

Inversement, considérons le nombre représenté par 10010011100100000000000000000000.

— Son bit de signe est 1 : c'est un nombre négatif.

— Son exposant décalé est 00100111 soit 39. En retirant le décalage, on a donc $\mathrm{E} - \mathrm{D} = -88$.

— Sa mantisse est représentée par $0010\cdots$ soit $\overline{1.001}^2 = 1 + 2^{-3} = 1.125$.

On obtient donc le nombre -1.125×2^{-88}, soit environ $-3.6350710512584224 \times 10^{-27}$.

Rappelons que E ne peut avoir tous ses bits à 0 ou tous ses bits à 1 pour un flottant normalisé. C'est un bon exercice de calculer le plus petit / plus grand nombre flottant normalisé pour les deux représentations (32 bits et 64 bits). Voyons maintenant ce qui se passe lorsque l'exposant décalé E a tous ses bits à 0 ou à 1.

6.2.3 Exceptions

Cette sous-section n'est pas au programme, mais est intéressante quand même. Lorsque l'exposant décalé E n'est constitué que de 0 ou que de 1 (donc est égal à 0 ou $2^e - 1$, où e est son nombre de bits), l'interprétation du nombre n'est pas la même que ci-dessus.

$E = 0$: nombre dénormalisé. Si E est nul, avec la représentation normalisée on aurait un nombre de la forme $S \times \overbrace{\overline{1.\cdots}^2}^{\text{mantisse}} \times 2^{-\mathrm{D}}$. Autrement dit, le plus petit nombre positif représentable serait $2^{-\mathrm{D}}$, obtenu avec des bits de mantisse tous nuls. Il est plus intéressant de se rapprocher de zéro [7]. Ainsi, si l'exposant décalé E est nul, on ne suppose plus que la mantisse est $\overline{1.M_1\cdots M_m}^2$, mais au contraire $\overline{0.M_1\cdots M_m}^2$. En faisant ainsi, on crée par contre un fossé entre le plus petit nombre normalisé positif ($2^{1-\mathrm{D}}$), et le plus grand nombre que l'on peut obtenir avec exposant nul : $\overline{0.1111\cdots}^2 \times 2^{-\mathrm{D}}$, qui est très proche de $2^{-\mathrm{D}}$. Pour compenser ceci, on suppose que le décalage pour un nombre dénormalisé est donné par $\mathrm{D}' = \mathrm{D} - 1 = 2^{e-1} - 2$ au lieu de $\mathrm{D} = 2^{e-1} - 1$. L'interprétation d'un nombre dénormalisé est donc, puisque E est nul :

$$(-1)^{\mathrm{S}} \times \overline{0.M_1\cdots M_m}^2 \times 2^{1-\mathrm{D}}$$

Un cas particulier (compatible avec ce que l'on vient de dire) : si tous les bits M_i sont nuls, on représente zéro. Il y a donc deux zéros, l'un positif et l'autre négatif.

$E = 2^e - 1$: infinis et NAN. Les nombres ayant un exposant décalé E égal à $2^e - 1$ sont utilisés pour représenter les infinis et les NAN. NAN signifie « not a number », et est utilisé pour les calculs produisant des erreurs, par exemple le calcul de $\sqrt{-1}$. Les infinis sont utilisés pour exprimer le fait qu'un calcul dépasse le plus grand nombre représentable par valeur positive (on obtient alors $+\infty$), ou le plus petit par valeur négative (on obtient alors $-\infty$).

La règle est la suivante : si les bits représentant la mantisse sont nuls, c'est un infini ($+\infty$ ou $-\infty$ suivant le bit de signe), sinon, c'est un NAN.

Exemple en Python. Les lignes suivantes produisent les infinis et un NAN :

```
a=1. #La virgule est nécessaire pour travailler avec des flottants et non des entiers.
for i in range(1000): #3**1000 est trop grand pour être représentable !
   a*=3
b=-a
c=a+b
```

En effet, si on affiche `a`, `b` et `c`, on obtient :

```
>>> print(a,b,c)
inf -inf nan
```

Il est en fait assez dur de les obtenir en Python, en général on aura une erreur. C'est le cas lorsqu'on calcule directement $3.^{1000}$ comme `3.**1000` ou encore qu'on essaie de calculer $\sqrt{-1}$. En C ou avec le module Numpy, par contre, on les obtient facilement :

7. Et ce serait mauvais que le nombre zéro ne soit pas représentable en flottant !

```
>>> np.sqrt(-1.0)  #fonction racine carrée (sqrt) du module Numpy
nan
>>> a=np.exp(1000.) #fonction exponentielle du module Numpy
>>> a
inf
```

6.3 Arrondis

En général, un calcul faisant intervenir deux nombres flottants sur n bits ne donne pas un nombre représentable exactement sur n bits. Il suffit par exemple de prendre un nombre décimal non dyadique, comme $1/10 = 0.1$. Celui ci s'écrit $\overline{1.100110011001100\cdots}^2 \times 2^{-4}$. (La périodicité du développement n'est pas un hasard, c'est le cas pour tous les rationnels). La mantisse n'ayant qu'un nombre fini de bits, il est nécessaire de couper ce développement infini. Ainsi, la représentation par un flottant de 0.1 ne sera qu'une approximation. Elle est obtenue en prenant le flottant le plus proche [8]

Le fait que les réels ne soient représentés qu'approximativement fait que les égalités mathématiques ne tiennent plus avec des flottants. Voici trois exemples de ce qu'on peut obtenir en Python (sur 64 bits) :

```
>>> a=0.1
>>> b=0
>>> for i in range(10):
...     b=b+a
...
>>> b==1
False
```

```
>>> a=2.**1000
>>> a==a+1
True
```

```
>>> a, b, c=1, 2**-53, 1
>>> a+b-c==a-c+b
False
```

Pour le premier exemple, 0.1 n'est représenté en mémoire que sous forme arrondie. La boucle à pour objet de calculer 10×0.1 en faisant 10 additions. Les erreurs d'approximation se cumulent, et au final on obtient un résultat très proche de 1, mais qui n'est pas 1 (j'obtiens $1 - 2^{-53}$).

Pour le deuxième, l'explication est la suivante : sur 64 bits, il y a 52 bits de mantisse. Le plus petit flottant strictement supérieur à 2^{1000} est donc $(1 + 2^{-52}) \times 2^{1000}$. Ainsi, $2^{1000} + 1$ est indiscernable de 2^{1000}. Plus exactement le résultat de l'addition $2^{1000} + 1$ est arrondi au flottant le plus proche, à savoir 2^{1000} lui même. D'où l'égalité `a==a+1` qui peut paraître choquante !

Pour le dernier exemple, les opérations `+` et `-` ayant même priorité, elles sont évaluées de gauche à droite. Or ici $1+2^{-53}$ est arrondi au flottant le plus proche, à savoir 1 lui-même. Donc dans l'exemple, `a+b-c` vaut exactement zéro. Par contre, `a-c+b` vaut `b`, car `a` et `c` sont tous deux égaux (à 1).

Il ne faut surtout pas croire à la lecture de ces exemples que les résultats obtenus via des opérations sur les nombres à virgule sont complètement faux en informatique. Néanmoins, il faut être conscient que dans le monde des flottants, les égalités mathématiques ne sont plus vérifiées « qu'à ε près », à cause des erreurs d'arrondi et de l'impossibilité de représenter de manière exacte les réels. La leçon à retenir des exemples et la suivante : sauf cas particuliers bien précis,

Le test d'égalité entre deux flottants n'est, en général, pas pertinent !

On se contentera d'un test de la forme $|a - b| < \epsilon$, où ϵ est un « petit » flottant, dépendant du problème. Par exemple, pour tester si un réel x est une racine d'un polynôme P, on se contentera par exemple de $|P(x)| \leq 2^{-20}$. Si on veut un test qui fonctionne avec des quantités très petites ou très grandes mais non nulles, prendre $\frac{|a-b|}{|a|+|b|} < \varepsilon$.

Pour conclure, considérons le code Python suivant, qui résout une équation du second degré en donnant les racines réelles.

```
from math import sqrt

def trinome(a,b,c):
    assert a!=0, "ce n'est pas un trinôme du second degré!"
    Delta=b**2-4*a*c
    if Delta<0:
        print("Pas de racines !")
    elif Delta>0:
```

8. Le lecteur voulant des précisions sur les règles d'approximation pourra se reporter à l'adresse : `http://fr.wikipedia.org/wiki/IEEE_754`.

```
        r=sqrt(Delta)
        x1=(-b-r)/(2*a)
        x2=(-b+r)/(2*a)
        print("Il y a deux racines distinctes, qui sont: ", x1, "et", x2)
    else:
        print("Il y a une racine double, qui est: ", -b/(2*a))
```

Prenons un premier exemple :

```
>>> trinome(1,-1,-1)
Il y a deux racines distinctes, qui sont:  -0.6180339887498949 et 1.618033988749895
```

On obtient des valeurs approchées très correctes de $\frac{1\pm\sqrt{5}}{2}$. Cherchons maintenant les racines du polynôme $x^2+2^{-600}\,x$. On travaille sur 64 bits, ainsi les coefficients et les racines (0 et -2^{-600}) sont tous représentables de manière exacte par un flottant.

```
>>> trinome(1,2**(-600),0)
Il y a une racine double, qui est:  -1.204959932551442e-181
```

L'explication est simple : le discriminant du trinôme, qui est 2^{-1200}, n'est pas représentable sur 64 bits. Il est arrondi à zéro, ce qui explique le déroulement : le discriminant étant trop petit pour être représentable, le programme conclut à l'existence d'une racine double alors qu'il y a deux racines distinctes, très proches. Voici un dernier exemple avec le polynôme $(x-0.1)^2 = x^2 - 0.2x + 0.01$:

```
>>> trinome(1,-0.2,0.01)
Il y a deux racines distinctes, qui sont:  0.09999999868291098 et 0.10000000131708903
```

Ici « l'erreur » est légèrement différente : les coefficients du polynôme ne sont pas représentables exactement, et le discriminant du polynôme « flottant » est non nul, ce n'est pas du à une erreur d'arrondi dans l'opération. Le programme renvoie donc deux racines distinctes, assez proches de 0.1.

Ce code fonctionne très bien dans la plupart des situations, seulement il faut garder à l'esprit que les coefficients sont représentés à un petit ε près, de même que le résultat du calcul des racines.

Chapitre 7

Lecture et écriture dans des fichiers

Ce chapitre est consacré aux entrées/sorties. Il s'agit de récupérer des informations depuis le clavier ou un fichier, et d'afficher des choses à l'écran ou dans un fichier.

7.1 print et input

La fonction `print`. On a déja vu la fonction `print` pour afficher des choses à l'écran :

```
>>> x=6 ; y=7.5 ; s="une chaîne"
>>> print(x,y,s) #affichage séparé par des espaces.
6 7.5 une chaîne
```

Jetons un coup d'œil à la documentation de la fonction `print` :

```
>>> help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

Remarquons que par défaut, les objets affichés sont séparés par des espaces (argument optionnel `sep` valant par défaut `' '`), et l'affichage se termine par un retour chariot (argument optionnel `end` valant par défaut `'\n'`), mais ceci peut être modifié :

```
>>> print("mot1","mot2","mot3",sep=" xxxxxx ", end=" fin ! \n")
mot1 xxxxxx mot2 xxxxxx mot3 fin !
```

Le champ `file` sert à spécifier la destination : c'est `sys.stdout` par défaut. `stdout` signifie *standard output*, c'est-à-dire la sortie standard, qui est par défaut l'écran. `flush` est un peu compliqué à expliquer et pas vraiment utile pour nous, mais disons que par défaut, Python n'envoie pas directement les éléments à imprimer sur la sortie standard, mais les « stocke » temporairement. Cela fait gagner du temps.

La fonction `input`. À l'inverse de `print`, `input` permet de récupérer quelque chose depuis *l'entrée standard*, qui est par défaut, le clavier :

```
>>> help(input)
Help on built-in function input in module builtins:

input(...)
    input([prompt]) -> string
```

```
Read a string from standard input.  The trailing newline is stripped.
If the user hits EOF (Unix: Ctl-D, Windows: Ctl-Z+Return), raise EOFError.
On Unix, GNU readline is used if enabled.  The prompt string, if given,
is printed without a trailing newline before reading.
```

Cette fonction prend en paramètre optionnel une chaîne de caractères (*prompt string* dans la documentation), l'affiche avant de lire une chaîne de caractères depuis l'entrée standard (standard input, par défaut le clavier). En général, on affecte cette chaîne lue à une variable. Dans l'exemple ci-dessous, `une chaîne` a été entrée avec mes petites mains au clavier. La saisie s'arête lorsqu'on appuie sur la touche Entrée.

```
>>> s=input("Entrez-moi quelque chose : ")
Entrez-moi quelque chose : une chaîne
>>> print(s)
une chaîne
```

Les fonctions de conversions de type permettent de convertir une chaîne de caractères en le type qui nous intéresse. Une conversion de type se fait sous la forme `t(x)` où `t` est le type voulu et `x` l'objet à convertir. Lorsqu'on utilise `input` ou lorsqu'on lit des informations dans un fichier, on convertit souvent des chaînes de caractères en d'autres types.

```
>>> n=int(input("Entrez-moi un entier : "))
Entrez-moi un entier : 42
>>> (n+12)//5
10
```

7.2 Fonctions pour les fichiers

En pratique, on utilise assez peu les fonctions `print` et `input` depuis un clavier ou vers l'écran : si on veut traiter une grande quantité de données (par exemple pour faire une étude statistique pour un TIPE...), ces données sont en général stockées dans des fichiers ad-hoc, qu'on veut pourvoir lire pour ensuite en manipuler le contenu. Il est assez maladroit de copier le contenu du fichier dans un script Python, il vaut mieux séparer les données du script qui les exploite. En particulier, même si les données changent (à la suite d'une nouvelle série de mesures, par exemple), le script Python reste identique. Après manipulation, on peut ensuite vouloir réécrire nos données transformées vers un autre fichier. Cette sous-section décrit la manipulation des entrées/sorties vers des fichiers, qui existent en dehors de notre programme Python.

Du point de vue du programmeur, un fichier ouvert *en lecture* doit être vu comme un tube (pipe en anglais) par lequel arrivent des données extérieures chaque fois que le programme les demande, de même qu'il faut voir un fichier ouvert *en écriture* comme un tube par lequel s'en vont les données que le programme envoie. Remarquez qu'ainsi, le clavier ou l'écran ne sont que des tubes particuliers.

Pour les fonctions Python, un fichier est une séquence de caractères : une fois qu'un fichier a été ouvert par un programme, celui-ci maintient un marqueur (fictif) à la position courante, qui indique à tout moment où sera lu/écrit le prochain octet. Toute opération de lecture ou d'écriture fait bouger ce pointeur vers l'avant.

fonction	description
`f=open(nom_du_fichier,'r')`	Ouvre le fichier `nom_de_fichier` (donné sous la forme d'une chaîne de caractères indiquant son emplacement) en lecture (`r` comme read). Le fichier doit exister et seule la lecture est autorisée.
`f=open(nom_du_fichier,'w')`	Ouvre le fichier `nom_de_fichier` en écriture (`w` comme write). Si le fichier n'existe pas, il est créé, sinon il est écrasé (vidé avant utilisation).
`f=open(nom_du_fichier,'a')`	Ouvre le fichier `nom_de_fichier` en ajout (`a` comme append). Identique au mode `'w'`, sauf que si le fichier existe, il n'est pas écrasé et ce qu'on écrit est ajouté à partir de la fin du fichier.
`f.close()`	Sur un fichier ouvert comme précédemment, le ferme. Cette ligne est impérative pour les fichiers ouverts en écriture, puisque le fichier n'est réellement écrit complètement qu'à la fermeture (c.f comportement de `flush`).
`f.read()`	Lit tout le fichier d'un coup et le renvoie sous forme de chaîne de caractères (à ne réserver qu'aux fichiers de taille raisonnable).
`f.readlines()`	Pareil que précédemment, mais le résultat est une liste de chaînes de caractères, chaque élément correspondant à une ligne. Attention, le saut de ligne `\n` est présent à la fin de chaque chaîne.
`f.readline()`	Lit une unique ligne du fichier et la renvoie sous forme de chaîne (avec `\n` au bout). Le curseur de lecture (virtuel!) est placé en début de ligne suivante. En pratique, on sait que l'on est arrivé en fin de fichier lorsqu'un appel à cette méthode renvoie une chaîne de caractères vide.
`for ligne in f`	On peut itérer sur les lignes d'un fichier, ceci est équivalent à `for ligne in f.readlines()`.
`f.write(s)`	Écrit la chaîne `s` à la suite du fichier.
`f.writelines(T)`	Écrit l'ensemble des éléments de `T` dans le fichier `f` comme des lignes successives. `T` est une liste, une séquence, un tuple... bref, un itérable.

Le tableau ci-dessus résume les principales fonctions pour le traitement des fichiers. `f` désigne un tube, qu'on pourra considérer comme étant un fichier ouvert en lecture ou en écriture.

Mieux vaut un petit exemple qu'un long discours : le script suivant `moyenne.py` prend un fichier `notes_eleves` en lecture et un fichier `notes_eleves_triees` en écriture. On suppose que le fichier `notes_eleves` est composé de lignes de la forme `nom; note`, où `nom` est une chaîne de caractères donnant le nom de l'élève et `note` est une note (supposée entière dans le script).

Le script moyenne.py

```
f=open('notes_eleves','r')
f2=open('notes_eleves_triees','w')
total=0
T=[]
lignes=f.readlines()
nombre=len(lignes)
for ligne in lignes:
    c=ligne.split(';')
    T.append((int(c[1]),c[0]))
    total+=int(c[1])
T.sort()
T.reverse()
print("Le nombre d'élèves est de ",nombre,", avec une moyenne de ",total/nombre,".",sep="")
for u in T:
    f2.write(u[1]+";"+str(u[0])+"\n")
f2.close()
```

Par exemple, le fichier `notes_eleves` peut être :

Le fichier notes_eleves

```
Eddard "Ned" Stark;7
Lady Catelyn Stark;10
Sansa Stark;8
Arya Stark;15
Bran Stark;6
Jon Snow;11
```

Le script procède ainsi :

— il récupère toutes les lignes du fichier `notes_eleves` dans une liste (`lignes`) ;
— traite dans la boucle chaque ligne en la découpant en deux, car `split` est une méthode sur les chaînes de caractères, retournant une liste correspondant au découpage de la chaîne suivant l'argument. Ici `ligne.split(';')` sépare chaque ligne en deux parties.
— le couple `(note,nom)` (avec `note` convertie en entier) est ajouté à la liste `T` ;
— les notes sont ajoutées dans la variable `total` ;
— on trie la liste `T` dans l'ordre croissant (qui correspond aux notes croissantes), puis on l'inverse ;
— en fin de script, on affiche à l'écran une unique ligne donnant le nombre d'élèves et la moyenne, et on écrit dans le fichier `notes_eleves_triees` les lignes `nom;note`, en suivant l'ordre de `T` (qui est triée par note décroissante).

Par exemple, l'exécution du script sur le fichier `notes_eleves` précédent affiche à l'écran :

```
>>> Le nombre d'eleves est de 6, avec une moyenne de 9.5.
```

et le fichier `notes_eleves_triees` (créé ou écrasé par le script) est :

Le fichier notes_eleves_triees

```
Arya Stark;15
Jon Snow;11
Lady Catelyn Stark;10
Sansa Stark;8
Eddard "Ned" Stark;7
Bran Stark;6
```

qui est bien trié par ordre décroissant de notes.

Chapitre 8

Modules usuels

On détaille dans ce chapitre les modules à connaître en Python. Bien que cette connaissance ne soit pas exigible, il ne faut pas les découvrir aux concours. En particulier, lors de l'épreuve Maths 2 de Centrale. Ils pourront aussi être utilisés dans un TIPE.

8.1 Module `math`

Ce module contient les fonctions mathématiques usuelles, ainsi que les constantes e et π. Il est parfois importé directement par l'éditeur, c'est le cas de Spyder par défaut, par exemple. Il n'y a pas grand chose à en dire, mais voyons rapidement comment importer un module et obtenir de l'aide.

8.1.1 Importation du module

Importation d'une fonction particulière. `from math import sin,cos,e` par exemple, permet d'importer spécifiquement les fonctions `sin`, `cos` et la constante `e`, sous ces noms là.

Importation de toutes les fonctions. `from math import *` est similaire à l'importation précédente, mais le joker `*` est utilisé à la place des noms explicites des fonctions.

Importation du module. `import math` importe le module, les fonctions sont alors accessibles par `math.sin`, `math.cos`, par exemple.

Importation du module et alias. C'est la méthode qu'on utilisera la plupart du temps pour importer un module. Elle est similaire à la précédente, mais on abrège le nom du module avec un alias. Pour le module `math`, on pourrait écrire `import math as m`, les fonctions sont alors accessibles par `m.sin`, `m.cos`...

8.1.2 De l'aide !

La documentation Python est assez bien fournie, pour y accéder il suffit d'écrire `help(nom_du_module)`. Ceci fonctionne également avec les alias. Par exemple, `import math as m ; help(m)` fournira de l'aide sur toutes les fonctions du module. Bien sûr, il est possible d'obtenir de l'aide sur une fonction spécifique :

```
>>> help(m.sqrt)
Help on built-in function sqrt in module math:

sqrt(...)
    sqrt(x)

    Return the square root of x.
```

N'hésitez pas à lire l'aide des fonctions présentées ci-après si vous les utilisez, elles sont très peu détaillées dans ce chapitre.

8.2 Module numpy

On importera ce module avec l'alias `np` : `import numpy as np`.

Construction de tableaux numpy.

— `np.zeros(n)` : crée un vecteur (ligne) à n composantes nulles.

— `np.zeros((n,m))` : crée une matrice à n lignes et m colonnes remplie de zéros ; marche aussi avec des dimensions supplémentaires.

— `np.zeros(..., dtype="uint8")` : on peut préciser le type des données. Par défaut ce sont des flottants 64 bits, ici `uint8` signifie « entier non-signé sur 8 bits » (donc entre 0 et 255). Ce format est utile pour les images.

— `np.ones(...)` : même chose que précédemment, avec des 1 à la place des 0.

— `np.eyes(n)` : construit la matrice identité de taille $n \times n$.

— `np.linspace(a,b,n)` : crée un tableau à n éléments régulièrement espacés entre a et b (inclus).

— `np.arange(a,b,h)` : crée un tableau contenant les éléments $a, a+h, a+2h...$ strictement inférieurs à b. Similaire à `range` mais avec des flottants.

Récupération de données. Avec `M` un tel tableau Numpy :

— `M.ndim` : nombre de dimensions de `M` : 1 pour un tableau « linéaire », 2 pour une matrice, etc...

— `M.shape` : tuple donnant les dimensions de `M`. Par exemple `(4,3)` pour une matrice 4×3.

— `M.size` : nombre total d'éléments : 12 pour une matrice 4×3.

— `M.min()`, `M.max()`, `M.sum()` parlent d'elles-même.

— argument optionnel : « l'axe » sur lequel on calcule min, et max... Par exemple si `M` est une matrice 4×3, `M.sum(0)` calcule la somme des colonnes sous la forme d'un tableau à 4 éléments, `M.max(1)` donne le maximum sur chaque ligne sous la forme d'un tableau à 3 éléments.

Listes à tableaux

— `np.array(L)` : convertir une liste en tableau Numpy. Si `L` est une liste de listes, on obtient une matrice, etc... Là aussi on peut préciser le type.

— `M.tolist()` : opération inverse.

Cas des matrices. Avec `M` et `N` deux matrices :

— `M[i,j]`, `M[i][j]` : élément en case (i, j).

— `M[i]`, `M[i,:]` : ligne d'indice i.

— `M[:,i]`, colonne d'indice i.

— `M[i:i+k,j:j+l]` : bloc de taille (k, ℓ) démarrant à la case (i, j).

— `M.copy()` : copie de la matrice.

— `M+N`, `M-N`, `M*N`, `M/N` : opérations terme à terme si `M` et `N` de même taille.

— `c*M` : multiplication de `M` par `c`.

— `M+c` : addition de `c` à tous les éléments de `M`.

— `M.dot(N)`, `np.dot(M,N)` : produit matriciel de `M` par `N` (si dimensions compatibles).

— `M.transpose()`, `np.transpose(M)` : copie transposée de `M`.

— `M.trace()`, `np.trace(M)` : trace de `M`.

— `np.concatenate((M,N),axis=0)` : concatène `M` et `N` verticalement. `axis=1` pour une concaténation horizontale.

Fonctions vectorielles. Numpy possède des versions « vectorielles » de la plupart des fonctions usuelles, par exemple `np.exp`. Appliquer une telle fonction f à un tableau Numpy `M` crée un nouveau tableau de même taille, les coefficients étant les $f(x)$ pour tout x de `M`. On peut créer une fonction vectorielle à partir d'une fonction f quelconque à l'aide de `np.vectorize(f)`.

Algèbre linéaire. Le sous-module `numpy.linalg` (importé comme `import numpy.linalg as alg`) permet de faire de l'algèbre linéaire.

- `alg.det(M)` : déterminant de `M`.
- `alg.inv(M)` : inverse de `M`.
- `alg.matrix_rank(M)` : rang de `M`. Attention toutefois, bien souvent les coefficients seront des flottants, le résultat est à prendre avec des pincettes.
- `alg.matrix_power(M,n)` : calcule M^n pour $n \in \mathbb{N}$.
- `alg.solve(M,Y)` : résolution de $MX = Y$.
- `alg.eigvals(M)` : valeurs propres de `M` (sous forme de tableau Numpy).
- `alg.eig(M)` : valeurs et vecteurs propres de `M`. Les vecteurs propres sont donnés sous la forme d'une matrice de passage. Avec `T,P=alg.eig(M)`, on a $M = PDP^{-1}$, avec D la matrice diagonale dont les coefficients diagonaux sont donnés par `T`, si M est diagonalisable. Attention : avec des flottants, les égalités ne sont vraies qu'à ε près...

Polynômes. Le sous-module `polynomial` permet de travailler avec des polynômes. On utilisera principalement la fonction `Polynomial` qui permet de construire un polynôme à partir de la liste de ses coefficients : `from numpy.polynomial import Polynomial as pol`.

- `pol(C)` : construit un polynôme à partir de la liste de ses coefficients. Ils sont donnés par degré croissant, par exemple `pol([1,0,2,3])` consruit le polynôme $1 + 2X^2 + 3X^3$;
- les opérations `+`, `*` et `-` peuvent être utilisées entre polynômes. Les constantes sont automatiquement converties en polynômes. `//` et `%` donnent quotient et reste dans une division euclidienne. Enfin `/` permet de diviser un polynôme par une constante et `**` permet de calculer une puissance d'un polynôme.

Avec `P` un polynôme :

- `P.coef` : donne la liste des coefficients de `P`.
- `P.degree` : son degré.
- `P.roots()` : calcule ses racines complexes (`j` est utilisé pour le $\imath$ mathématique, comme en physique).
- `P.deriv()` : polynôme dérivé.
- `P.integ(n)` : intégrer n fois. On peut préciser les constantes d'intégration (par défaut nulles), utilisées à chaque étape, par exemple `P.integ(3,[0,1,2])` intègrera successivement `P` trois fois, d'abord avec la constante 0, puis la constante 1, puis la constante 2.
- `P(x)` : avec `x` un nombre, permet de calculer $P(x)$. Les polynômes sont également des fonctions vectorielles, on peut les appliquer à un tableau Numpy.

La fonction `poly`. Cette fonction est un peu complémentaire du module précédent.

- `np.poly([`a_0, a_1, a_{n-1}`])` renvoie les coefficients du polynôme unitaire de degré $n+1$ s'annulant en les a_i, avec multiplicités. Attention, les coefficients sont ordonnés par degré décroissant, ainsi le premier coefficient du tableau renvoyé est 1.
- `np.poly(M)`, avec M une matrice carrée, renvoie les coefficients du polynôme caractéristique $\chi(M) = \det(XI_n - M)$ (là aussi, c'est un polynôme unitaire, et les coefficients sont donnés par degré décroissant).

```
>>> np.poly([0, 0, 1, 2]) #polynôme X^2(X-1)(X-2)
array([ 1, -3,  2,  0,  0])
>>> np.poly([[1, 1], [0, 1]]) #polynôme caractéristique
array([ 1., -2.,  1.])
```

Nombres aléatoires et probabilités Le sous-module `random` permet de générer des nombres aléatoires, et plus généralement de traiter de probabilités. On l'importe ici comme `import numpy.random as rd`.

- `rd.randint(a,b)` retourne un entier aléatoire entre a (inclus) et b (exclus), en suivant la loi uniforme sur $[\![a, b[\![$.
- `rd.random()` : un flottant suivant la loi uniforme sur $[0, 1[$.
- `rd.binomial(n,p)` : un entier suivant la loi $\mathcal{B}_{n,p}$.
- `rd.geometric(p)` : entier suivant la loi géométrique de paramètre p.
- `rd.poisson(x)` : entier suivant la loi de Poisson de paramètre x.

Pour toutes ces fonctions, on peut préciser un paramètre supplémentaire (`size`) pour avoir un tableau Numpy constitué de nombres suivant la loi, de format donné par `size`. Par exemple :

```
>>> rd.geometric(0.5,8)
array([1, 1, 2, 3, 4, 2, 1, 1])
>>> rd.randint(0,2,(3,4))
array([[1, 1, 0, 1],
       [0, 1, 0, 1],
       [0, 1, 0, 0]])
```

8.3 Module `matplotlib`.

On utilisera principalement le sous-module `pyplot`, qui sert à tracer des courbes. On l'importera comme suit : `import matplotlib.pyplot as plt`.

8.3.1 Options `pyplot`

- `plt.figure('titre')` : crée une nouvelle fenêtre de tracé (vide).
- `plt.plot(X,Y)` : relie les points (x_i, y_i) par lignes brisées, les deux listes (ou tableaux Numpy) `X` et `Y` doivent avoir même taille. On peut préciser en option :
  - la couleur : `b`, `g`, `r`, `c`, `m`, `y`, et `k` pour **b**lue, **g**reen, **r**ed, **c**yan, **m**agenta, **y**ellow et blac**k**.
  - le style du tracé : `-` pour un trait plein, `--` pour des pointillés, `-.` pour une ligne en pointillés qui alterne avec des petits points, etc...
  - les marques sur les points (x, y) : `o` pour un cercle, `v` pour un triangle vers le bas, `*` pour une étoile, `x` pour une croix, etc...

  On peut également préciser une étiquette (label). Par exemple, `plt.plot(X,Y,'y--x', label="bidule")` tracera une courbe jaune, en pointillés, avec des croix sur les points (x, y), de nom « bidule ».
- `plt.xlim(xmin, xmax)` : fixer les bornes de l'axes des abscisses dans la figure. De même avec `plt.ylim`.
- `plt.axis([xmin, xmax, ymin, ymax])` : même chose que précédemment.
- `plt.axis('off')` : pas d'axes. `plt.axis('equal')` : axes égaux (un cercle est un cercle !)
- `plt.xlabel("nom")` : donner un nom à l'axe des abscisses. De même avec `ylabel`.
- `plt.legend(loc="upper right")` : rajout et positionnement de la légende (les étiquettes des courbes). Voir l'aide pour les options.
- `plt.show()` : affichage de la fenêtre. Tant qu'on fait des `plt.plot`, ils sont ajoutés à la même figure, ce qui permet de tracer plusieurs courbes sur le même graphique.
- `plt.savefig("figure.png")` : sauvegarde la figure, l'extension (ici PNG) peut être précisée.

8.3.2 Quelques exemples

Un tracé basique. Deux courbes tracées sur le même graphique (voir figure 8.1) :

```
X=np.linspace(0,np.pi,100)
Y=[np.sin(x)**2 for x in X]
Z=np.exp(-X**2) #facile de travailler avec des tableaux Numpy !
plt.plot(X,Y,label="f1")
plt.plot(X,Z,label="f2")
plt.legend(loc="upper right")
plt.show()
```

Des lignes de niveaux. On va voir comment afficher des lignes de niveau avec la fonction `contour`. On souhaite avoir les lignes de niveaux de la fonction $f(x, y) = x^2 - 2y^2$. Voir la figure 8.2

```
delta=0.025
x=np.arange(-5, 5, delta)
y=np.arange(-5, 5, delta)
X, Y=np.meshgrid(x,y) # très pratique: on crée une grille dont les points sont donnés par les (x_i,y_j)
                      # pour x_i dans x et y_j dans y. X et Y sont deux matrices donnant les
```

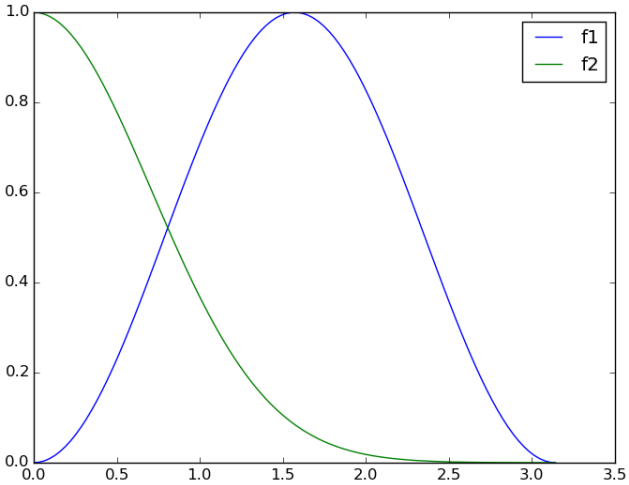


FIGURE 8.1 – Un tracé facile

```
                    # abscisses/ordonnés des points de la grille.

f=np.vectorize(lambda x: x*x)
F=f(X) ;  G=2*f(Y) ; H=F-G
valeurs=[0, 1, 2, 3, 5, 8, 12, 20]
couleurs=["maroon", "pink", "red", "orange", "yellow", "green", "black", "blue"]
plt.contour(X, Y, H, valeurs, colors=couleurs) #x^2-y^2=valeur
plt.show()
```

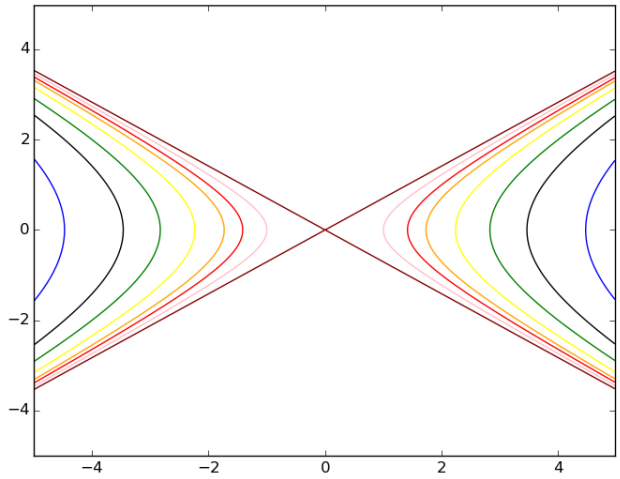


FIGURE 8.2 – Des lignes de niveau

Un petit exemple en 3D : la caténoïde. On va tracer une figure en 3D. La caténoïde de la figure 8.3 a pour équation :

$$u \in \mathbb{R}, \quad v \in [0, 2\pi[, \qquad \begin{cases} x &= \text{ch}(u)\cos(v) \\ y &= \text{ch}(u)\sin(v) \\ z &= u \end{cases}$$

Pour tracer une surface paramétrée comme précédemment, on utilise la méthode `plot_surface` : il suffit de

construire des matrices Numpy `x`, `y` et `z` contenant les valeurs prises sur les 3 axes en les points du paramétrage. On restreint ici u à $[-2,2]$:

```
from mpl_toolkits.mplot3d import Axes3D #importation d'un module de tracé 3D.
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d') #111 signifie juste un seul tracé.
u = np.linspace(-2, 2, 100)
v = np.linspace(0,2*np.pi, 100)
x = np.outer(np.cosh(u), np.cos(v)) #np.outer est le produit extérieur. np.outer(a,b) est
                    # transposée(a) * b, avec a et b des tableaux Numpy vus comme des vecteurs colonnes.
y = np.outer(np.cosh(u), np.sin(v))
z = np.outer(u,np.ones(np.size(v)))
ax.plot_surface(x, y, z)
plt.show()
```

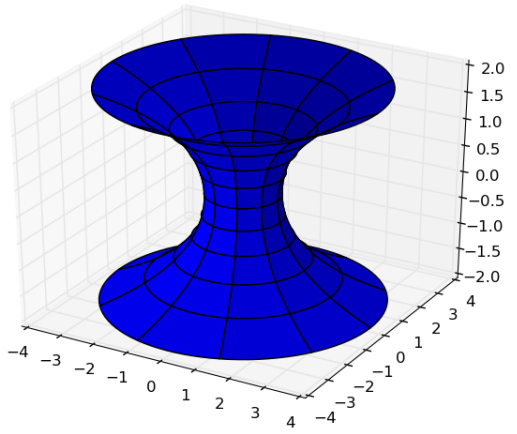


FIGURE 8.3 – Une caténoïde

Catalogue. On trouvera de nombreux exemples à l'adresse `http://matplotlib.org/examples/`.

8.4 Module `scipy`

Le module `scipy` est le module de calcul scientifique. Trois points au programme : résolution d'équations numériques, intégration de fonctions et résolution d'équations différentielles.

8.4.1 Résolution d'équations numériques

On utilise le sous-module `optimize` de scipy : `import scipy.optimize as sco`. La fonction `fsolve` est tout indiquée pour résoudre une équation de la forme $f(x) = 0$, avec f une fonction éventuellement vectorielle. Le résultat est un tableau Numpy contenant une solution de l'équation. Outre la fonction f, elle prend en paramètre un x_0 au voisinage duquel on cherche un zéro de f.

— `sco.fsove(f,x0)` : résout l'équation $f(x) = 0$ au voisinage de x_0. La méthode utilisée est proche de la méthode de la sécante [1] ou un analogue en dimension supérieure.

— on peut préciser `sco.fsove(f,x0, fprime=...)` pour donner aussi la dérivée sous forme formelle. Ceci permet une résolution plus efficace (via la méthode de Newton ou analogue en dimension supérieure).

Par exemple, avec l'intersection de l'hyperbole $x^2 - y^2 = 1$ et de l'ellipse $2x^2 + y^2 = 3$ (qui est constituée de 4 points) :

1. `https://fr.wikipedia.org/wiki/M%C3%A9thode_de_la_s%C3%A9cante`

```
>>> f=lambda u: np.array([u[0]**2-u[1]**2-1, 2*u[0]**2+u[1]**2-4])
>>> sco.fsolve(f,np.array([-1,2]))
array([-1.29099445,  0.81649658])
```

8.4.2 Intégration de fonctions

On utilise le sous-module `integrate` de scipy : `import scipy.integrate as sci`. La fonction que l'on va utiliser est `quad`. Il suffit d'indiquer la fonction à intégrer et les bornes :

```
>>> def f(x): return x**2-1
>>> sci.quad(f, 0, 5)
(36.66666666666667, 4.219579007134704e-13)
```

Attention, `quad` renvoie un couple : le premier élément est la valeur approchée de l'intégrale, le deuxième une estimation du terme d'erreur. Remarque : utiliser `np.inf` pour l'infini, qui peut-être utilisé (ainsi que `-np.inf`) dans les bornes. Éviter cependant les intégrales *semi-convergentes*[2] : la méthode ne fonctionne pas bien, d'ailleurs Python l'indique et on obtient un grand terme d'erreur.

8.4.3 Intégration d'équations différentielles

L'intégration d'équations différentielles se fait avec le même sous-module, et la fonction `odeint` (pour *ordinary differential equations integration*). Attention, Python ne résout que des équations de la forme $x'(t) = f(x(t), t)$, où x peut être une fonction à valeurs vectorielles. On transformera donc une équation (scalaire par exemple) d'ordre au moins 2 en équation (vectorielle) d'ordre 1. Elle s'utilise ainsi : `sci.odeint(f,x0,T)` où :

— `f` est la fonction de l'équation différentielle ;

— `x0` est un scalaire ou un vecteur Numpy ;

— `T` est un tableau Numpy de dimension 1 (tableau des temps).

En notant t_0 le premier élément de `T`, il n'y a *a priori* qu'une seule solution au système différentiel avec la condition $x(t_0) = x_0$. La fonction renvoie un tableau Numpy contenant les valeurs (approchées) de la solution aux temps donnés dans le tableau `T`.

Par exemple, avec l'équation de Lokta-Volterra :

$$\left\{\begin{array}{rcrcl} x'(t) & = & x(t)(7-2y(t)) & \text{et} & x_0 = 1 \\ y'(t) & = & -y(t)(1-4x(t)) & \text{et} & y_0 = 5 \end{array}\right.$$

On transforme d'abord ce système en un système d'ordre un en introduisant le vecteur $X(t) = \binom{x(t)}{y(t)}$ de sorte que le système se reformule en

$$X'(t) = f(X(t), t), \quad X_0 = \binom{1}{5}, \quad \text{et} \quad f\left(\binom{u}{v}, t\right) = \begin{pmatrix} u(7-2v) \\ -v(1-4u) \end{pmatrix}$$

Remarquez que la fonction $f(X, t)$ ne dépend pas de t, un tel système est dit autonome. Mais `odeint` prend en entrée une fonction de la forme $t \mapsto f(X(t), t)$. Résolvons cette équation sur l'intervalle $[0, 30]$:

```
def f(X,t):
    x,y=X[0],X[1]
    return np.array([x*(7-2*y),-y*(1-4*x)])

X0=np.array([1,5])
T=np.linspace(0,30,10000)
X=sci.odeint(f,X0,T)
plt.plot(T,[u[0] for u in X],label="x")
plt.plot(T,[u[1] for u in X],label="y")
plt.legend()
plt.show()
```

qui donne la courbe de gauche dans la figure 8.4. Le tracé du portrait de phase avec le code suivant

2. comme le classique $\int_1^{+\infty} \frac{\sin x}{x}\,\mathrm{d}x$

```
plt.plot([u[0] for u in X],[u[1] for u in X])
plt.xlabel("x") ; plt.ylabel("y")
plt.show()
```

donne la courbe de droite.

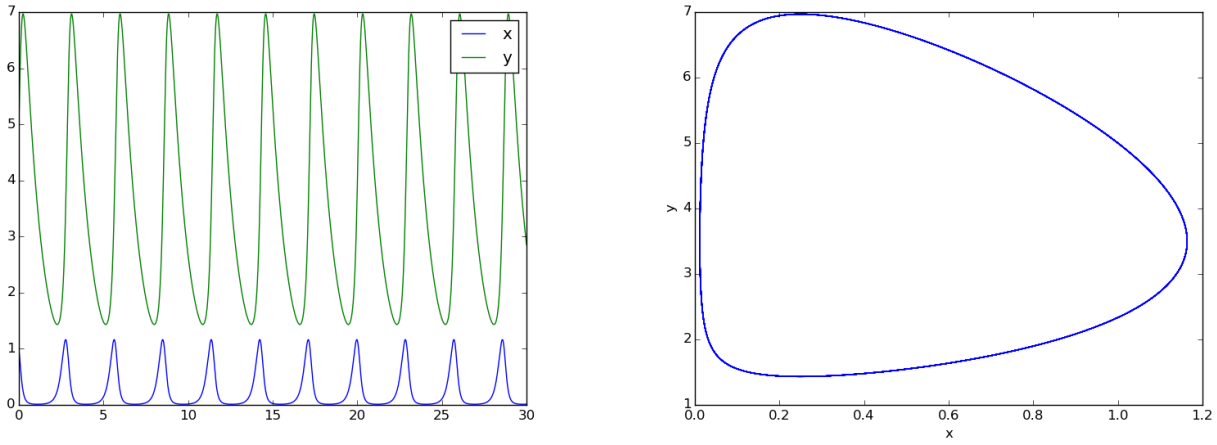


FIGURE 8.4 – Résolution de l'équation de Lokta-Volterra. (a) x et y en fonction du temps. (b) x en abscisse, y en ordonnée.

8.5 Quelques autres modules

`copy`. Ce module permet de faire des copies. Il contient notamment la fonction `copy` (copie simple) et `deepcopy` (copie en profondeur). Par exemple une liste « double » comme `[[0,1],[2,3]]` devra être copiée avec `deepcopy` si on veut copier également les listes `[0,1]` et `[2,3]`.

`fractions`. Un module qui permet d'utiliser des fractions. Pour construire la fraction p/q, on écrira simplement `fractions.Fraction(p,q)`. Une fois que l'on travaille avec des fractions, on peut utiliser les opérateurs usuels `+`, `*`,...

`time`. Le module `time` permet de mesurer des temps d'exécutions. Il contient la fonction `time`, qui mesure le temps absolu en secondes depuis le 1er janvier 1970, mais on lui préferera `clock` pour mesurer le temps CPU (donné par l'horloge du processeur). Pour mesurer le temps d'exécution d'un script, la suite d'instructions `t=time.clock() ; script() ; t=time.clock()-t` permet de stocker dans la variable `t` le temps d'exécution du script.

`random`. Un module pour générer des nombres aléatoires, un peu redondant avec `numpy.random`. `random.random()` fournit par exemple un flottant aléatoire de $[0,1]$, et `random.randint(a,b)` un entier aléatoire de $[\![a,b]\!]$. Attention, contrairement à Numpy, la borne b est ici incluse.

`itertools`. Ce module permet de générer facilement des objets combinatoires, par exemple des permutations :

```
>>> from itertools import permutations
>>> for x in permutations([0,1,2]): print(x)
>>> (0, 1, 2)
(0, 2, 1)
(1, 0, 2)
(1, 2, 0)
(2, 0, 1)
(2, 1, 0)
```

`PIL.` Un module pour traiter des images. Une documentation se trouve ici [3]. Montrons en exemple comment afficher la composante rouge d'une image au format PNG, via manipulation de tableaux Numpy : à une image est associée un tableau Numpy de dimension $n \times m \times 3$, où n est le nombre de lignes, m le nombre de colonnes, et chaque pixel est constitué de 3 composantes rouge, verte et bleue :

```
im=Image.open("image.png") # localisation de l'image
t=np.array(im) #conversion en tableau Numpy
n,m,_=t.shape #format n*m*3
for i in range(n):
    for j in range(m):
        t[i][j][1]=0 ; t[i][j][2]=0 #composantes verte et bleue mises à zéro
Image.fromarray(t).show() #conversion en image et affichage
```

`tkinter.` Ce module permet de réaliser des interfaces graphiques, et peut éventuellement être utile pour un TIPE (et pour plus tard...). Voir la documentation ici [4].

3. `http://effbot.org/imagingbook/pil-index.htm`
4. `http://tkinter.fdex.eu/`

Quatrième partie

Algorithmique avancée

Chapitre 9

Algorithmes de tris naifs

9.1 Introduction : le problème du tri

Dans ce chapitre ainsi que dans le prochain sur les tris, on étudie les algorithmes de tris par comparaisons. Un algorithme de tri est un algorithme qui permet d'organiser une liste homogène `L` d'éléments selon un ordre fixé. Les éléments à trier font donc partie d'un ensemble E muni d'une relation d'ordre total noté $\leq$, c'est-à-dire vérifiant des hypothèses de

— Transitivité : $\forall x, y, z \in E \quad x \leq y \quad \text{et} \quad y \leq z \quad \Longrightarrow \quad x \leq z.$

— Réflexivité : $\forall x \in E \quad x \leq x.$

— Antisymétrie : $\forall x, y \in E \quad x \leq y \quad \text{et} \quad y \leq x \quad \Longrightarrow \quad x = y.$

— Totalité de l'ordre : $\forall x, y \in E \quad x \leq y$ ou $y \leq x$

Parfois, on se passera de l'hypothèse d'antisymétrie (on parle de pré-ordre). Voici quelques exemples :

Exemple 9.1. — *Les ensembles classiques* $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}\cdots$ *sont munis de l'ordre standard* $\leq$.

— $\mathbb{R}^2$ *peut-être muni de l'ordre lexicographique* $\preceq_{lex}$, *défini par :*

$$(x, y) \preceq_{Lex} (x', y') \qquad \Longleftrightarrow \qquad x < x' \qquad ou \qquad x = x' \quad et \quad y < y'$$

Cet ordre est bien total, et se généralise à $\mathbb{R}^n$, *ou encore à des produits cartésiens* $(E_1, \leq_1) \times \cdots \times (E_n, \leq_n)$ *d'ensembles ordonnés.*

— $\mathbb{R}^2$ *peut aussi être muni du pré-ordre* $\preceq_1$ *défini par :*

$$(x, y) \prec_1 (x', y') \qquad \Longleftrightarrow \qquad x \leq x'$$

Ce n'est pas un ordre car $(0, 0) \preceq_1 (0, 1)$ *et* $(0, 1) \preceq_1 (0, 0)$.

— *L'ensemble des chaînes de caractères peut-être muni de l'ordre lexicographique (celui du dictionnaire).*

Exemple 9.2. *En Python, ce sont les ordres ci-dessus qui sont utilisés pour comparer des n-uplets ou des chaînes de caractères avec* `<=`.

Un algorithme générique de tri est le suivant :

Algorithme 9.3 : Tri générique

Entrées : Une liste `L` dont les éléments sont à valeurs dans un ensemble ordonné.
Sortie : La même liste `L`, triée dans l'ordre croissant pour l'ordre.
Séquence d'instructions;

A priori, l'algorithme de tri ne retourne rien : en sortie de fonction, la liste passée en entrée est triée. Par la suite, on ne se préoccupera pas de la relation d'ordre utilisée, celle-ci sera vue comme une « boite noire », comparant deux éléments quelconques de E.

Complexité. Pour comparer deux algorithmes de tri entre eux, on comptera deux types d'opérations distinctes :

— Le nombe de comparaison effectuée entre deux éléments de E.

— Le nombre d'affectations.

On supposera que ces deux opérations s'effectuent en temps constant, et pour comparer deux algorithmes de tris, on comparera principalement leur *complexité temporelle*. Différents types de complexité sont pertinentes. Fixons un algorithme de tri $\mathcal{A}$.

— La complexité dans le pire des cas permet de fixer une borne supérieure du nombre d'opérations qui seront nécessaires pour trier une liste de n éléments. Elle est définie par :

$$C_{\mathcal{A},\text{pire}}(n) = \max_{L \text{ liste de taille } n} C(\mathcal{A}, L)$$

où $C(\mathcal{A}, L)$ est le nombre d'opérations élémentaires effectuées par l'algorithme $\mathcal{A}$ pour trier la liste L. La complexité dans le pire cas est la seule dont l'étude est au programme.

— La complexité en moyenne est le nombre d'opérations élémentaires effectuées en moyenne pour trier une liste de n éléments. L'étude de la complexité en moyenne est en générale difficile et requiert une probabilité sur E^n. En pratique, on suppose que les éléments de la liste sont distincts, et que les $n!$ permutations décrivant les positions relatives des éléments dans la liste d'entrée sont équiprobables. En résumé, tout se passe comme ci l'on considérait uniquement des listes de la forme $[\sigma(0), \dots, \sigma(n-1)]$ où σ est une permutation de $\mathfrak{S}_n$ (ensembles des bijections de $[\![0, n-1]\!]$). La complexité moyenne de l'algorithme de tri $\mathcal{A}$ sur les listes de taille n est alors donnée par la formule :

$$C_{\mathcal{A},\text{moy}}(n) = \frac{1}{n!} \sum_{\sigma \in \mathfrak{S}_n} C(\mathcal{A}, \sigma)$$

où $C(\mathcal{A}, \sigma)$ est le nombre d'opérations nécessaires pour trier la liste $[\sigma(0), \dots, \sigma(n-1)]$ avec l'algorithme $\mathcal{A}$. On calculera en particulier dans le chapitre idoine la complexité en moyenne du tri rapide, qui est bien meilleure que sa complexité dans le pire cas.

— La complexité dans le meilleur des cas n'est pas la plus pertinente, mais permet de distinguer deux algorithmes égaux par ailleurs. À l'opposé de la complexité dans le pire cas, elle consiste à regarder les situations favorables à l'algorithme $\mathcal{A}$:

$$C_{\mathcal{A},\text{meilleur}}(n) = \min_{T \text{ liste de taille } n} C(\mathcal{A}, T)$$

Propriétés intéressantes des tris. Outre la complexité temporelle, certaines propriétés sont appréciables, parmi lesquelles :

— **le caractère *en place*** : si l'algorithme ne requiert qu'un espace en mémoire constant (pour quelques variables) en plus de la liste d'entrée pour le trier, on dit que le tri s'effectue en place. Tous les tris de ce chapitre s'effectuent en place. Dans les tris qui seront étudiés plus tard, on verra que le tri par fusion ne trie pas en place.

— **le caractère *stable*** : l'algorithme est dit stable si les positions relatives de deux éléments égaux ne sont pas modifiées par l'algorithme : c'est-à-dire que si deux éléments x et y égaux se trouvent aux positions i_x et i_y de la liste avant l'algorithme, avec $i_x < i_y$, alors c'est également le cas de leurs positions après l'algorithme. Le caractère stable peut-être utile lorsqu'on utilise des pré-ordres. Par exemple, la liste

$$L = [(4,1), (2,2), (2,3), (4,5)]$$

est triée avec le pré-ordre consistant à regarder uniquement le deuxième élément de chaque couple. Si l'on trie ensuite L suivant le pré-ordre obtenu en regardant simplement le premier élément de chaque couple (le préordre $\preceq_1$ dont on a parlé plus haut) avec un tri stable, on obtient la liste

$$[(2,2), (2,3), (4,1), (4,5)]$$

qui est triée suivant l'ordre lexicographique. En effet, lorsque deux éléments ont même première composante, le plus petit pour l'ordre lexicographique est situé à gauche. Un tri non stable produirait par exemple la liste

$$[(2,3), (2,2), (4,5), (4,1)]$$

qui ne vérifie plus la propriété d'être triée suivant l'ordre lexicographique.

Hypothèses sur l'entrée. Sous certaines hypothèses sur les éléments des listes à trier (entiers relatifs dont on connait les bornes, réels supposés équirépartis sur l'intervalle $[0,1]$...) il est possible de proposer des algorithmes de tris tenant compte de ces hypothèses qui sont plus rapides que ceux proposés dans ce chapitre. Cependant, ce ne sont pas des *tris par comparaison* dont les seules opérations autorisées sont la comparaison d'éléments et l'affectation dans la liste ou éventuellement dans une liste annexe.

Conventions. Puisque les algorithmes que l'on va écrire sont simples, on donne directement le code Python (du pseudo-code serait une simple paraphrase). Dans les preuves de correction, on adoptera une syntaxe proche de Python : rappelons que si `L` est une liste, `i` et `j` deux indices vérifiant $0 \leq i \leq j \leq n$ où n est la longueur de la liste, alors `L[i:j]` est une liste constituée des éléments de `L` entre les indices `i` (inclus) et `j` (exclus), c'est-à-dire `L[i],...,L[j-1]`. En particulier, cette liste est vide si $i = j$.

Dans les codes Python qui suivent, on a choisi de ne jamais faire d'échanges d'éléments de la liste de la forme `L[i],L[j]=L[j],L[i]` si les indices `i` et `j` sont égaux. Bien que cet échange ne pose pas de problème à Python, cela nuit à la fois à la compréhension de l'algorithme et à l'adaptabilité du code dans d'autres langages.

Tri en Python. On rappelle que les listes dans ce cours correspondent aux objets de type `list` en Python[1]. Pour trier une liste `L` avec Python, on utilise la *méthode* `sort` (syntaxe : `L.sort()`). Si on ne veut pas modifier la liste et obtenir une copie triée, on utilise la fonction *sorted* qui renvoie une liste correspondant aux éléments de `L`, triés. C'est équivalent à copier la liste et à appliquer ensuite la méthode `sort`.

Structure du chapitre. On décrit dans ce chapitre uniquement les tris naïfs les plus classiques. Ils sont efficaces sur de petites listes (de taille au plus 50), et sont en complexité $O(n^2)$, avec n la taille de la liste. On leur préférera l'un des algorithmes du chapitre sur les tris efficaces lorsque l'on doit travailler avec des listes plus grandes. On commence par décrire le tri par sélection, puis le tri à bulles et enfin le tri par insertion. On verra enfin à titre d'exemle un tri efficace qui n'est pas un tri par comparaisons : le tri par comptage.

9.2 Tri par sélection

Ce tri particulièrement simple est peut-être celui auquel on pense en premier lorsqu'on écrit un algorithme de tri.

Idée du tri. L'idée est simple : supposons que la liste de taille n est déja en partie triée avec ses k premiers éléments à leur place définitive. On *sélectionne* le plus petit des $n-k$ éléments restants, qu'on amène en position $k+1$. la liste a alors ses $k+1$ premiers éléments à leur position définitive. Itérer ce procédé $n-1$ fois suffit pour trier la liste.

Code Python. On donne maintenant la procédure en Python. Dans tout ce chapitre, la liste sera appelée `L`, associée au type `list` en Python.

Le tri par sélection

```
def tri_selection(L):
    n=len(L)
    for i in range(n-1):
        #Inv(i): L[0:i] triée, ses éléments sont plus petits que les autres éléments de L.
        imin=i
        for j in range(i+1,n):
            #Inv2(j): imin est l'indice du plus petit élément de L[i:j]
            if L[j]<L[imin]:
                imin=j
            #Inv2(j+1)
        if i!=imin:
            L[i],L[imin]=L[imin],L[i]
        #Inv(i+1)
```

Terminaison de l'algorithme. L'algorithme de tri par sélection est constitué de deux boucles `for` imbriquées, il termine donc !

1. Les listes Python sont en fait des tableaux redimensionnables. En particulier pour les options info, les algorithmes que l'on va écrire se traduisent facilement en Caml en des algorithmes travaillant sur des vecteurs.

Preuve de l'algorithme. La boucle `for` interne a pour effet de positionner la variable `imin` à l'indice de l'élément minimal de la liste entre les indices `i` et `n`. Ainsi, un passage dans la boucle `for` externe positionne l'élément minimal de la liste entre les indices `i` et `n` en position `i`. Cette boucle `for` principale possède l'invariant suivant :

Inv_i : Les éléments de la liste entre les indices `0` (inclus) et `i` (non inclus) sont triés dans l'ordre croissant et plus petits que les autres éléments de la liste.

— Tout d'abord, Inv_0 est vrai : en effet, la sous-liste `L[0:0]` est vide.

— Clairement, si Inv_i est vrai en haut de la boucle (début de la ligne 4), Inv_{i+1} est vrai en bas de la boucle (fin de la ligne 9) : en effet, on positionne le plus petit élément de `L[i:n]` en position `i`.

Le compteur de boucle `i` prend toutes les valeurs entre 0 et $n-2$. Par suite, l'invariant $\text{Inv}_{n-2+1} = \text{Inv}_{n-1}$ est vérifié en sortie de boucle, ce qui implique que la sous-liste `L[0:n-1]` est triée en sortie de boucle, et ses éléments sont plus petits que l'autre élément de la liste, à savoir `L[n-1]`. Ainsi, la liste est entièrement triée en sortie de fonction, et le tri est correct.

En toute rigueur, il faudrait exhiber un invariant de boucle pour la boucle `for` interne (justifiant au passage la discussion précédente), celui-ci est plutôt évident : `imin` est l'indice du plus petit élément de `L[i:j]`.

Complexité. On compte séparémment le nombre de comparaisons et le nombre d'*échanges* (correspondant à 2 affectations).

— Le nombre de comparaisons ne dépend pas de la liste : on en fait exactement $n-i-1$ dans la boucle `for` interne, et donc

$$\sum_{i=0}^{n-2}(n-i-1) = \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2}$$

en tout.

— Le nombre d'échanges `L[i],L[imin]=L[imin],L[i]` est dans le meilleur des cas de 0 : cela correspond au cas où la liste est déja triée dans l'ordre croissant, auquel cas l'élément minimal de la sous-liste `L[i:n]` est déja en position `i`, pour tout `i`. Dans le pire cas, on fait un échange pour chaque passage dans la boucle `for` externe, c'est-à-dire $n-1$. Cela correspond par exemple à la liste suivant :

2	3	4	$\cdots$	$n-1$	n	1

Propriétés. Le tri par sélection s'effectue en place, mais n'est pas stable. En effet, la liste `[(2,0),(2,1),(1,2)]` est triée dans l'ordre croissant pour le pré-ordre obtenu en ne regardant que le deuxième élément de chaque couple (vis à vis de l'ordre classique sur $\mathbb{N}$). Si on trie la liste suivant le pré-ordre obtenu en ne regardant que le premier élément de chaque couple avec le tri par sélection, le premier et le dernier élément de la liste sont permutés et on obtient la liste `[(1,2),(2,1),(2,0)]`. Les positions relatives des deux éléments égaux (pour ce pré-ordre) que sont `(2,0)` et `(2,1)` ont été permutées.

9.3 Tri à bulles

Ce tri est un des plus populaires, mais l'un des moins efficaces. Le code est donné dans ce cours mais sa terminaison, sa correction, sa complexité sont laissées en exercice. On pourra montrer que ce tri est stable.

Idée du tri. L'algorithme du tri à bulles parcourt la liste, et compare les couples d'éléments successifs. Lorsque deux éléments successifs ne sont pas dans l'ordre croissant, ils sont échangés. Si au moins un échange a eu lieu pendant le parcours, l'algorithme procède à un nouveau parcours. S'il n'y a pas eu d'échange pendant un parcours, cela signifie que la liste est triée et l'algorithme s'arrête. A chaque nouveau parcours, on peut en fait s'arrêter un élément plus tôt, d'où le `p-=1` dans le code suivant.

Code Python.

Le tri à bulles

```
def tri_bulles(L):
    p=len(L)
    pasfini=True
    while pasfini:
        pasfini=False
        for i in range(p-1):
            if L[i]>L[i+1]:
                L[i],L[i+1]=L[i+1],L[i]
                pasfini=True
        p-=1
```

Terminaison de l'algorithme. p est strictement décroissant à chaque passage dans la boucle `while`. De plus, si $p \leq 1$, la boucle `for` ne fait rien, car `range(0)` est(un itérateur) vide. Ainsi, `pasfini` reste à `False` et la boucle `while` termine.

Preuve de l'algorithme. La boucle `while` admet l'invariant suivant :

`L[p:len(L)]` est triée, ses éléments sont plus grands que les autres éléments de la liste. De plus, `pasfini=False` ou `L[0:p]` est triée.

Le ou de l'invariant est inclusif : il se peut que `pasfini=False` et que `L[0:p]` soit triée.

Complexité. Voici les complexités dans les meilleur et pire cas.

- La complexité dans le meilleur cas est linéaire, contrairement au tri par sélection : en effet, si la liste est déja triée on effectue seulement un parcours, soit $n-1$ comparaisons et aucune affectation.
- La complexité dans le pire cas est quadratique : on effectue au plus n parcours, avec égalité notamment si le plus petit élément de la liste se trouve à la fin. Le cas le pire en nombre d'échanges se produit de plus lorsque la liste est triée dans l'ordre décroissant. On fait n parcours de la boucle `while`, et à chaque fois $p-1$ échanges et comparaisons. Le nombre total d'échanges et de comparaisons est donc de $\sum_{i=1}^{n}(i-1) = \frac{n(n-1)}{2}$.

Propriétés. L'algorithme du tri à bulles s'exécute en place, et est stable.

9.4 Tri par insertion

Idée du tri : On suppose que la sous-liste `L[0:i-1]` est triée. On s'intéresse à l'élément `L[i]`, que l'on va faire descendre à la bonne position de sorte que la sous-liste `L[0:i]` soit triée. On pourrait faire des échanges à la manière du tri à bulles, mais pour diminuer le nombre d'affectations, on stocke la valeur de `L[i]` dans une variable `x`, et on déplace successivement les éléments `L[i-1]`, `L[i-2]`... d'un cran vers la droite. On s'arrête lorsque l'élément `L[j-1]` considéré vérifie `L[j-1]` $\leq$ `L[i]`, on assigne alors `x` en position `j`.

Le tri par insertion

```
def tri_insertion(L):
    n=len(L)
    for i in range(1,n):
        #Inv(i): L[0:i] est trié
        j=i
        x=L[i]
        while j>0 and L[j-1]>x:
            #Inv: Pour tout k vérifiant j<k<=i, L[k]>x
            L[j]=L[j-1]
            j-=1
            #Inv: Pour tout k vérifiant j<k<=i, L[k]>x
        L[j]=x
        #Inv(i+1): L[0:i+1] est trié
```

Terminaison de l'algorithme. L'algorithme de tri par insertion est constitué d'une boucle `while` dans une boucle `for`. Il faut donc montrer que pour tout $i \in \{1, \dots, n-1\}$, la boucle `while` termine, ce qui est à peu près évident : la variable `j` est initialisée à `i` juste avant la boucle, la condition de continuation du `while` comporte notamment la condition `j>0` et `j` est décrémenté à chaque tour de boucle. Notons que les indices de la liste considérés ne produisent jamais d'erreurs (d'accès en dehors de la liste). Remarquez que si `j=0` dans la condition du `while`, la condition `j>0`

n'est pas vérifiée et on n'a pas besoin d'évaluer `L[j-1]>x` (qui irait chercher le dernier élément de la liste, ce qui est spécifique à Python) pour s'apercevoir que la condition `j>0 and L[j-1]>x` est fausse. Ceci est dû au comportement *paresseux* de l'opérateur logique `and`.

Preuve de l'algorithme. La boucle `while` a pour effet de déplacer d'un cran vers la droite tous les éléments d'indice strictement inférieur à `i` qui sont strictement supérieurs à `x`. Elle admet pour invariant :

Inv : Pour tout k tel que $j < k \leq i$, `L[k]>x`.

En sortie de boucle `while`, la condition `j>0` and `L[j-1]>x` est fausse, ainsi la boucle `for` possède l'invariant suivant :

Inv_i : la liste `L[0:i]` est triée.

En effet :

— la liste `L[0:1]`, constituée de l'unique élément `L[0]`, est triée. Donc Inv_1 est vrai.

— Si, pour $i \in \{1, \dots, n-1\}$, Inv_i est vrai en haut de la boucle, alors Inv_{i+1} est vrai en bas de la boucle. En effet, après l'exécution de la boucle `while`, les éléments de `L[j+1:i+1]` sont strictement supérieurs à `x` et ceux de `L[0:j]` sont inférieurs (avec `j` éventuellement nul). Ainsi, placer `x` en position `j` dans `L` mène à la liste triée `L[0:i+1]`

Par suite, après l'exécution de la boucle `for` la liste `L[0:n]` est triée, et la fonction est correcte.

Complexité. On compte séparémment le nombre de comparaisons et le nombre d'affectations. Le comportement est très différent dans le meilleur des cas et dans le pire cas.

- Dans le meilleur des cas, la condition de la boucle `while` n'est jamais vérifiée (car `L[j-1]<=x`) : c'est le cas si la liste est triée. On fait dans ce cas 1 comparaison et 2 affectations à chaque tour de la boucle `for`, c'est-à-dire $n-1$ et $2(n-1)$ en tout. Ainsi la complexité dans le meilleur cas est linéaire.
- Dans le pire cas, la condition `while` devient fausse à chaque tour de la boucle `for` car `j=0`. Cela correspond à une liste triée en sens inverse.
  Dans ce cas, la boucle `while` a été exécutée j fois, pour un total de j comparaisons et j affectations. A cela s'ajoute 2 affectations à chaque tour de la boucle `for`, on obtient donc :

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \text{ comparaisons} \qquad \text{et} \qquad \sum_{i=1}^{n-1} (i+2) = \frac{(n+4)(n-1)}{2} \text{ affectations}$$

Propriétés. La tri est à la fois stable et en place. Pour la stabilité, il est impératif que la condition dans le `while` soit `L[j-1]>x`, on perdrait la stabilité avec `L[j-1]>=x`.

9.5 Conclusion sur les tris par comparaisons quadratiques

On donne dans le tableau qui suit un équivalent des complexités des trois tris pour trier une liste de taille n, dans les cas le pire, le meilleur et en moyenne (voir la feuille d'exercice pour les complexités en moyenne). On distingue le nombre de comparaisons et le nombre d'affectations. Un échange entre deux éléments de la liste compte comme deux affectations.

Parmi les trois tris quadratiques évoqués, le tri par insertion est certainement le meilleur. On peut montrer qu'en moyenne il fait environ $n^2/4$ comparaisons et affectations pour trier une liste de taille n. Il possède de plus un très bon comportement vis à vis des listes « presques triés » (voir la feuille d'exercice). Le tri par sélection est intéressant si les affectations sont plus coûteuses que les comparaisons puisqu'il fait seulement $n-1$ échanges dans le pire cas. Le tri à bulles possède lui aussi un bon comportement vis à vis de certaines listes presque triées (le nombre d'échanges effectués dépend en fait du nombre d'*inversions* dans la liste), mais on lui préférera le tri par insertion. La figure 9.1 présente une comparaison des trois tris sur des listes de petites tailles, tirées au hasard. Ce graphique est cohérent avec la colonne « Cas moyen » du tableau ci-dessus. On s'aperçoit qu'avec Python, il y a un léger avantage au tri par sélection sur des entrées moyennes.

9.6 Un tri qui n'est pas un tri par comparaisons : le tri par comptage

Dans cette section, on montre un algorithme de tri qui n'est pas un tri par comparaisons : l'algorithme fait l'hypothèse que les éléments de la liste à trier sont des entiers naturels, bornés par une certaine constante $k > 0$. Sous cette hypothèse, il atteint un temps d'exécution en $O(n+k)$ pour trier une liste de taille n, ce qui est meilleur que les algorithmes de ce chapitre si $k = o(n^2)$ et même meilleur que les tris efficaces (à suivre) si $k = o(n \log n)$.

Tri	Cas	Meilleur	Pire	Moyen
Sélection	Comparaisons	$n^2/2$	$n^2/2$	$n^2/2$
	Affectations	0	$2n$	$2n$
Bulles	Comparaisons	n	$n^2/2$	$n^2/2$
	Affectations	0	n^2	$n^2/2$
Insertion	Comparaisons	n	$n^2/2$	$n^2/4$
	Affectations	$2n$	$n^2/2$	$n^2/4$

TABLE 9.1 – Équivalent du nombre de comparaisons et d'affectations pour trier une liste de taille n avec l'un des tris quadratiques, dans les meilleur, pire et moyen cas.

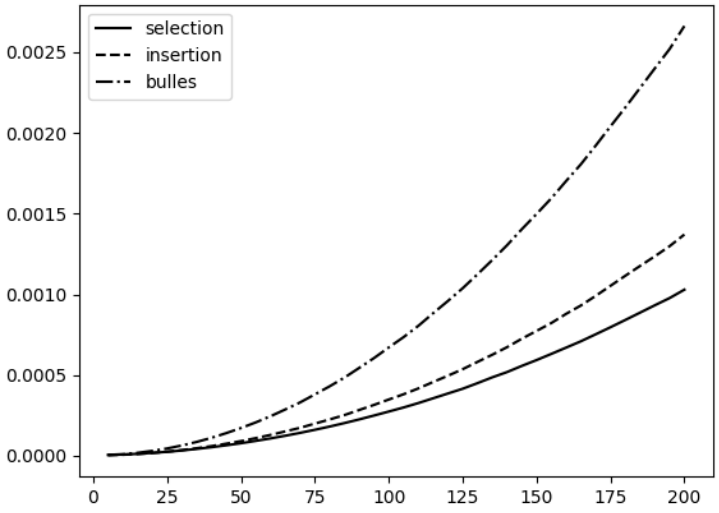


FIGURE 9.1 – Temps CPU pour trier une liste de taille n constituée d'entiers tirés aléatoirement entre 0 et 10000 (moyenne sur 1000 tests), avec $5 \leq n \leq 200$.

Idée de l'algorithme. On suppose que la liste à trier est constituée d'entiers de l'intervalle $[\![0, k[\![$. L'algorithme fonctionne suivant un principe simple : il suffit de parcourir une fois la liste et de compter le nombre d'éléments de la liste égaux à 0, 1, ..., $k-1$. Pour ce faire on utilise une liste de taille k. On peut alors facilement procéder à une réécriture de la liste initiale, de sorte qu'en sortie elle soit constituée des mêmes éléments, mais triés dans l'ordre croissant.

Code Python. L'algorithme prend en entrée la liste `L` à trier, ainsi qu'un entier k tel que tous les éléments de la liste soient des entiers de l'intervalle $[\![0, k[\![$. On procède en deux étapes : d'abord compter les éléments de chaque type, ensuite réécrire la liste `L`.

```
def tri_comptage(L,k):
    C=[0]*k
    for i in range(len(L)):
        C[L[i]]=C[L[i]]+1
    p=0
    for i in range(k):
        for j in range(C[i]):
            L[p]=i
            p+=1
```

Terminaison et correction. L'algorithme termine car il est constitué de boucles `for`. Si la liste à trier est bien constituée d'éléments de $[\![0, k[\![$, la première boucle `for` ne produit pas d'erreur, et après cette boucle la somme des

éléments de `C` vaut exactement la taille n de la liste `L`. Ainsi, les deux boucles `for` imbriquées suivantes ne produisent pas non plus d'erreurs car on écrit n fois dans `L`, aux indices $0, 1, \dots, n-1$. La correction est alors évidente : après la première boucle `for`, un élément $i \in [\![0, k[\![$ apparaît exactement `C[i]` fois dans la liste `L`, et on écrit dans `L` chaque entier i exactement `C[i]` fois avec les deux boucles imbriquées.

Complexité. Remarquons que la *complexité en espace* de l'algorithme est en $O(k)$: on utilise en effet la liste `C` de taille k pour trier `L`. La création de la liste `C` se fait en temps $O(k)$. Le remplissage de `C` se fait avec la première boucle `for` en temps $O(n)$. La boucle `for j` interne a une complexité $O(1+$`C[i]`$)$: en effet, même si `C[i]` est nul, il faut quand même incrémenter i. Ainsi, les deux boucles `for` imbriquées ont une complexité totale $O(\sum_{i=0}^{k-1}(1+$`C[i]`$)) = O(n+k)$. On a bien une complexité totale $O(n+k)$.

Remarques. • Une « erreur » classique d'implémentation de l'algorithme consiste à parcourir k fois la liste `L` pour remplir la liste `C` (pour $i \in [\![0, k[\![$, on teste au i-ème parcours si chacun des éléments de `L` est égal à i). Ceci mène à une complexité $O(nk)$, ce qui est maladroit.

• Ce tri n'est pas un tri par comparaisons, puisqu'on ne compare pas les éléments entre eux.

• Il existe bien d'autres tris qui ne sont pas des tris par comparaisons : le tri par base trie lui aussi des entiers naturels bornés par une certaine constante B, mais n'utilise qu'un espace de taille $O(\log B)$ pour une complexité temporelle $O(n \log B)$. On utilise en général la base 2, l'algorithme peut alors être vu comme l'application successive de $O(\log_2 B)$ algorithmes de tri par comptage avec $k = 2$.

• Un autre tri classique est le tri par baquets : il est efficace pour trier des réels supposés bien répartis dans un intervalle semi-ouvert $[a, b[$. Voir la feuille d'exercices !

Chapitre 10

Algorithmes de tris efficaces

On aborde dans ce chapitre les tris efficaces pour trier des listes de grande taille. On en présente deux : le tri par fusion et le tri rapide. Tous deux sont basés sur la stratégie « diviser pour régner », et sont plus efficaces que le tri par insertion dès que le nombre d'éléments à trier dépasse environ 50. Le tri par fusion a une complexité quasi-linéaire (en $O(n\log(n))$) dans le pire cas, ce qui n'est pas le cas du tri rapide qui est quadratique. Mais contrairement aux tris de la section précédente, le tri rapide a une complexité quasi-linéaire en moyenne. Il a l'avantage de s'effectuer en place (contrairement au tri fusion), ce qui en fait le tri le plus efficace en pratique. Un autre tri qui cumule les deux avantages (quasi-linéaire dans le pire cas, en place) est le tri par tas, qui pourra faire l'objet d'un problème.

Plan du chapitre. On donne brièvement les principes de la stratégie « Diviser pour régner », avant de l'appliquer aux deux tris (tri fusion et tri rapide). On détaille les complexités de ces tris, en montrant en particulier que le tri fusion (dans le pire cas) et le tri rapide (en moyenne) ont une complexité $O(n\log n)$. Cette dernière démonstration sera seulement heuristique.

10.1 Introduction à la stratégie « Diviser pour régner »

On rappelle ici les principes de la stratégie « diviser pour régner » :

— Diviser le problème principal en sous-problèmes *disjoints* de tailles inférieures (division).

— Traiter récursivement les sous-problèmes.

— Recomposer la solution du problème principal à partir des solutions des sous-problèmes (règne).

Concrètement, pour trier une liste à l'aide de cette stratégie, on va se ramener au tri de deux listes de taille inférieure. Le tri fusion et le tri rapide diffèrent conceptuellement. Pour le tri fusion, l'étape de division est immédiate : il s'agit juste de couper la liste en deux. Une fois les parties gauche et droite de la liste triées, il faut *fusionner* ces deux parties triées en une liste triée. Pour le tri rapide, on commence par *partitionner* la liste autour d'un élément appelé *pivot* : les éléments à gauche du pivot sont plus petits, ceux à droite sont plus grands. On trie récursivement ces deux parties gauche et droite, et il n'y a rien à faire pour l'étape de règne : la liste obtenue est triée.

10.2 Tri Fusion

Contrairement aux tris du chapitre sur les tris naïfs, le tri fusion proposé dans ce chapitre ne trie pas la liste initiale mais retourne une copie triée : en effet, il est très difficile d'écrire un tri fusion qui utilise un espace mémoire constant en plus de la liste à trier. Si on veut, il est immédiat d'écrire une version qui modifie la liste initiale (il suffit d'utiliser le tri de ce chapitre comme une fonction intermédiaire qui calcule une copie triée, puis faire une recopie des éléments dans la liste initiale).

10.2.1 La fonction de fusion

Comme on l'a dit, la partie difficile à écrire du tri fusion est la *fusion* de deux listes triées. La fonction qui suit prend en entrée deux listes `L1` et `L2`, supposées triées dans l'ordre croissant, et retourne une liste `L` dont les éléments sont ceux de `L1+L2`, mais triés dans l'ordre croissant.

Principe. On va parcourir les deux listes de gauche à droite au moyen de deux indices i_1 et i_2. À chaque étape, le plus petit élément parmi `L1[i1]` et `L2[i2]` est ajouté à la fin de la liste `L` (initialement vide), et l'indice correspondant est incrémenté. Une petite difficulté se produit lorsqu'on a terminé la lecture d'une des deux listes, il faut alors faire attention à ne pas tenter d'accéder à des éléments en dehors des listes : par exemple si l'on a terminé `L1`, l'indice i_1 vaut alors `len(L1)` et l'accès `L1[i1]` produirait une erreur.

Illustration. Le schéma suivant détaille l'algorithme de fusion sur deux listes triées.

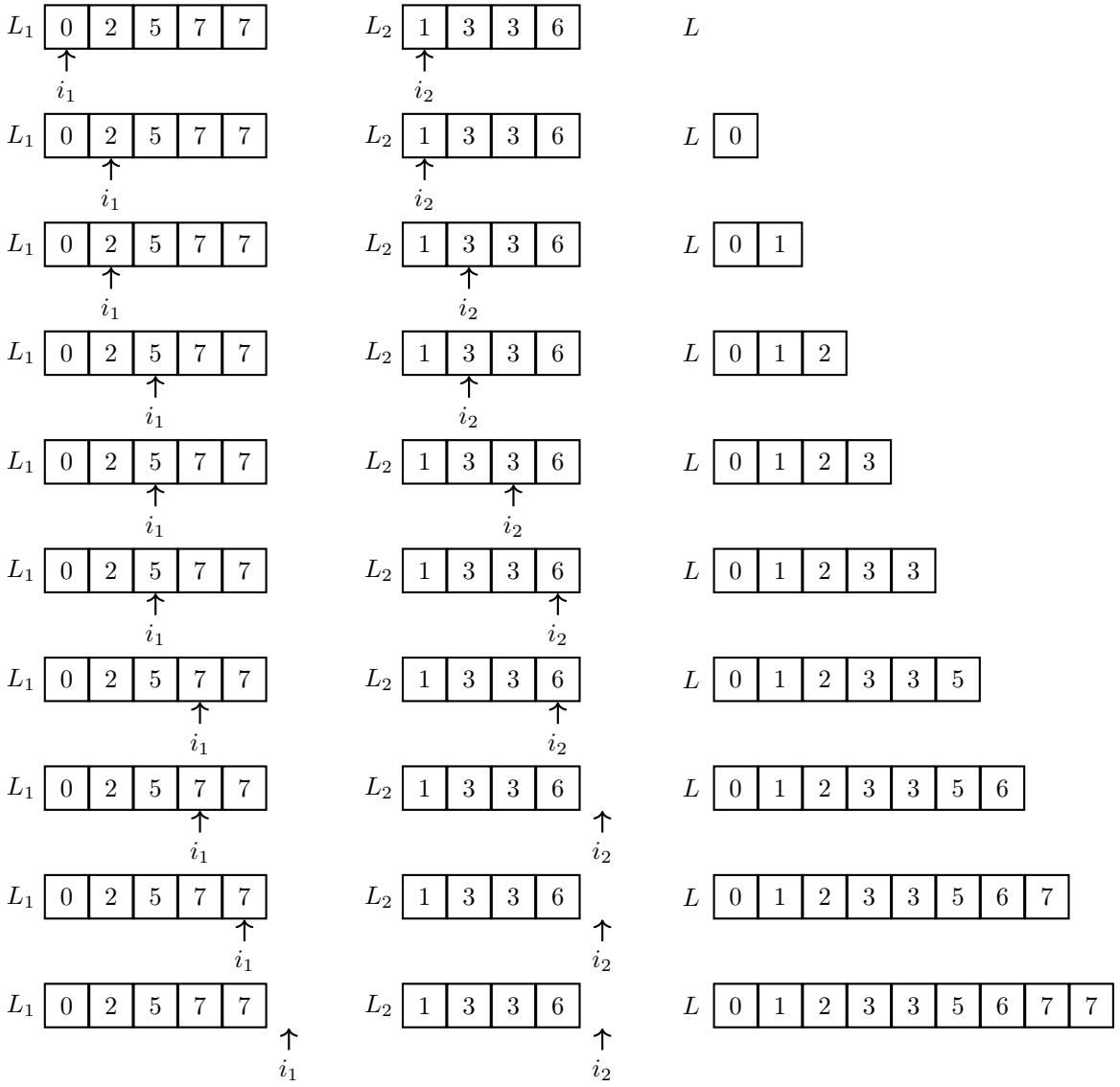


FIGURE 10.1 – Illustration de la fusion de deux listes triées

Code Python. Il s'agit simplement d'une boucle `for`, qui est exécutée $n_1 + n_2$ fois, avec n_1 et n_2 les tailles des listes `L1` et `L2`. La condition du `if` est naturelle : on prend l'élément `L1[i1]` si l'une des deux conditions suivantes est vérifiée :

— on a déja pris tous les éléments de la liste `L2` (auquel cas `i2` $=$ `n2`) ;

— ou alors la condition précédente et fausse, et de plus on n'a pas encore pris tous les éléments de la liste `L1` (auquel cas `i1` $<$ `n1`) et `L1[i1]` $\leq$ `L2[i2]`.

Comme `and` est évalué avant `or` et que ces opérateurs là sont paresseux, le code s'effectue sans erreur : en effet, si `i2` $=$ `n2`, ce qui est à droite du `or` n'est pas évalué, et sinon, `L1[i1]<=L2[i2]` n'est évalué que si `i1` $<$ `n1`.

```
def fusion(L1,L2):
    L=[]
```

```
    i1=0 ; i2=0
    n1=len(L1) ; n2=len(L2)
    for i in range(n1+n2):
        #Inv(i): L est triée dans l'ordre croissant et ses éléments sont ceux de L1[0:i1] et L2[0:i2].
        if i2==n2 or i1<n1 and L1[i1]<=L2[i2]:
            L.append(L1[i1])
            i1+=1
        else:
            L.append(L2[i2])
            i2+=1
        #Inv(i+1)
    return L
```

Terminaison et correction. La terminaison est évidente car il n'y a qu'une boucle `for`. La fonction s'effectue sans erreur, car on n'accède pas à des éléments situés en dehors des listes `L1` et `L2`. Un invariant de la boucle `for` est le suivant :

Inv(i) : `L` est croissante, ses éléments sont ceux de `L1[0:i1]` et `L2[0:i2]`, qui sont les plus petits de `L1+L2`.

En effet, cette propriété est vérifiée avant la boucle (`L`, `L1[0:i1]` et `L2[0:i2]` sont toutes vides), et est conservée à chaque passage de boucle : comme `L1` et `L2` sont triées, on rajoute à `L` le plus petit élément parmi ceux de `L1[i1:]` et `L2[i2:]`. Ainsi, on en déduit en particulier qu'après la boucle, `L` contient bien les éléments de `L1` et `L2` dans l'ordre croissant.

Complexité. Il est clair que la complexité est en $O(n_1 + n_2)$ affectations et comparaisons, avec n_1 et n_2 les tailles des listes à trier.

10.2.2 Le tri en lui-même

Idée. Si la liste contient 0 ou 1 élément, elle est triée et on retourne une copie de cette liste (`L[:]` dans le code suivant). Sinon, on la coupe en 2 (en notant $m = \lfloor \frac{n}{2} \rfloor$ où n est la taille de la liste, `L[:m]` contient les éléments de `L` d'indice strictement inférieur à m et `L[m:]` ceux d'indice supérieur ou égal à m), on trie récursivement les deux parties, et on fusionne les deux listes obtenues.

Code Python. L'écriture est très simple. Attention à ne pas oublier le cas de base !

```
def tri_fusion(L):
    n=len(L)
    m=len(L)//2
    if n<=1:
        return L[:]
    else:
        return fusion(tri_fusion(L[:m]),tri_fusion(L[m:]))
```

Terminaison. La terminaison de la fonction `tri_fusion(L)` est immédiate : si la taille de la liste est au plus 1, on renvoie une copie, sinon on sépare la liste en deux (m vérifie $0 < m < n$, avec n la taille de la liste `L`, donc les deux listes sont de tailles strictement inférieures à n), on appelle récursivement la fonction `tri_fusion` sur les deux listes, et on fusionne le résultat.

Correction. Comme pour beaucoup de fonctions récursives, la correction est facile à établir par récurrence forte (sur la taille de la liste). Soit pour $n \in \mathbb{N}$ la proposition $\mathcal{P}(n)$:

$\mathcal{P}(n)$: `tri_fusion(L)` retourne une copie triée dans l'ordre croissant de la liste `L` si celle-ci est de taille n.

— Si $n = 0$ ou 1, c'est immédiat.

— Sinon, avec $m = \lfloor \frac{n}{2} \rfloor$, comme $m < n$ et $n - m < n$, par hypothèse de récurrence, les listes `tri_fusion(L[:m])` et `tri_fusion(L[m:])` sont bien des copies triées de `L[:m]` et `L[m:]`. Ainsi $\mathcal{P}(n)$ est vraie car la fonction `fusion` est correcte.

— Par principe de récurrence, `tri_fusion(L)` est correcte.

Complexité La fonction de fusion a une complexité $O(n)$ pour fusionner deux listes de taille n. Construire les deux listes `L[:m]` et `L[m:]` prend également un temps $O(n)$. Ainsi, la complexité du tri vérifie pour $n \geq 2$:

$$C(n) = C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C\left(\left\lceil \frac{n}{2} \right\rceil\right) + O(n)$$

car la liste `L[:m]` est de taille $\left\lfloor \frac{n}{2} \right\rfloor$ et la liste `L[m:]` de taille $n - \left\lfloor \frac{n}{2} \right\rfloor = \left\lceil \frac{n}{2} \right\rceil$ car $n/2$ est un entier ou un demi-entier.

Résolvons cette récurrence pour les puissances de 2 (c'est plus simple). On a donc $C(2^p) = 2C(2^{p-1}) + O(2^p)$ pour tout $p \geq 1$. En divisant par 2^p, on obtient $\frac{C(2^p)}{2^p} = \frac{C(2^{p-1})}{2^{p-1}} + O(1)$, relation que l'on peut télescoper :

$$\frac{C(2^p)}{2^p} = \underbrace{C(1)}_{O(1)} + \sum_{k=1}^{p} \underbrace{\left[\frac{C(2^k)}{2^k} - \frac{C(2^{k-1})}{2^{k-1}}\right]}_{O(1)} = O(p)$$

D'où $C(n) = O(n \log n)$ car $p = \log_2(n)$ ici. Ce résultat s'étend à n quelconque, ce que l'on admettra ici.Remarquez que dans ce cas là, l'arbre des appels récursifs est équilibré : tous les niveaux de l'arbre contribuent pour $O(n)$ à la complexité totale :

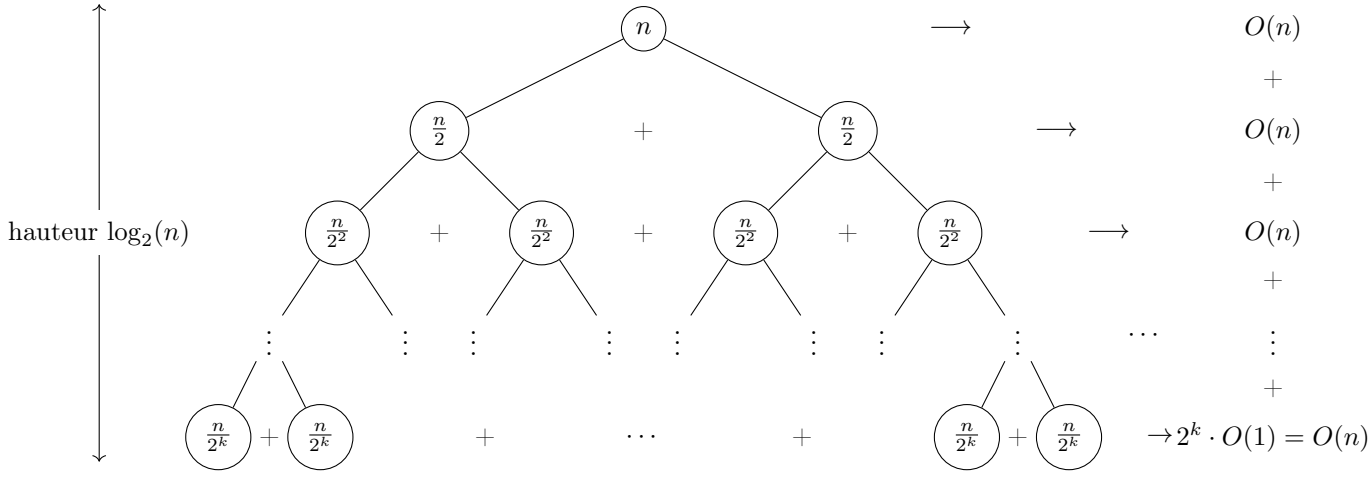


FIGURE 10.2 – Complexité dans le tri fusion sur une liste de taille $n = 2^k$.

Propriétés. Le tri ne s'effectue pas en place, en effet on renvoie une copie triée de la liste, (ou au mieux on utilise une liste de même taille que la liste à trier pour effectuer les fusions, voir feuille d'exerices), par contre, il est stable.

10.3 Tri Rapide (QuickSort)

Le tri rapide a été inventé par Hoare en 1961. Bien implémenté, il est probablement le tri le plus efficace en pratique. Bien que sa complexité dans le pire cas soit quadratique, il possède une très bonne complexité en moyenne. Il a de plus l'avantage de s'effectuer en place, contrairement au tri fusion. Tout comme ce dernier, il repose sur la stratégie diviser pour régner.

Idée. Contrairement au tri fusion, on ne coupe pas arbitrairement en deux. Supposons que l'on veuille trier la séquence constituée des éléments 4,1,8,3,2,5,7,6. On choisit le premier élément (c'est arbitraire) comme *pivot*, et on sépare les autres éléments en deux ensembles : $\{1, 2, 3\}$ et $\{5, 6, 7, 8\}$. Les éléments du premier ensemble sont plus petits que le pivot, ceux du deuxième plus grand. Il suffit alors de trier récursivement les deux ensembles pour obtenir une liste triée, en intercalant le pivot au milieu. Bien sûr, en pratique, on travaille avec des listes et pas avec des ensembles : peu importe comment sont répartis les éléments à gauche et à droite du pivot : l'important est d'avoir partagé la liste en 3 : les éléments plus petits que le pivot, le pivot, et les éléments plus grands.

10.3.1 Fonction de partition

On pourrait écrire une fonction `partition(L,x)` partitionnant les éléments de `L` en deux listes formées des éléments plus petits et plus grands que `x`. On en déduirait alors un tri récursif similaire au tri fusion de la section précédente, qui renverrait une nouvelle liste [1]. Mais l'avantage du tri rapide sur le tri fusion est qu'il peut s'écrire facilement « en place », c'est à dire sans recourir à des listes intermédiaires. Pour cela, on va travailler sur des portions de la liste initiale.

Partition d'une portion de liste. La fonction a écrire prend en entrée une liste `L`, et deux indices g et d délimitant la sous-liste de `L` à partionner, qui est `L[g:d]`. On suppose que $d-g \geq 2$, de sorte qu'il y ait au moins deux éléments dans la portion. L'idée est alors la suivante :

— On choisit (arbitrairement) le pivot comme étant le premier élément de la portion, à savoir `L[g]`.
— On parcourt le reste de la portion (de l'indice $g+1$ inclus à l'indice d exclus) en gardant une séparation de la portion déja parcourue en deux morceaux : celui des éléments strictement inférieurs au pivot, et celui des morceaux supérieurs ou égaux (le pivot reste à l'indice g)
— Une fois la portion parcourue entièrement, il suffit d'amener le pivot à la jonction des deux morceaux, pour avoir partionné la portion en trois zones : les éléments strictement inférieurs au pivot, le pivot, et les éléments supérieurs.
— Dans l'optique de trier la liste, il faudra faire des appels récursifs sur les deux zones à gauche et à droite du pivot : on renvoie donc l'indice où se trouve le pivot à la fin de la partition.

La partie délicate à écrire est celle du parcours de la portion. Lorsqu'on examine un nouvel élément, il faut distinguer les cas suivant si celui-ci est supérieur ou égal au pivot (auquel cas on ne fait rien), ou strictement inférieur : dans ce cas il faut le permuter avec l'élément supérieur ou égal au pivot situé le plus à gauche. Ceci est résumé dans le dessin suivant :

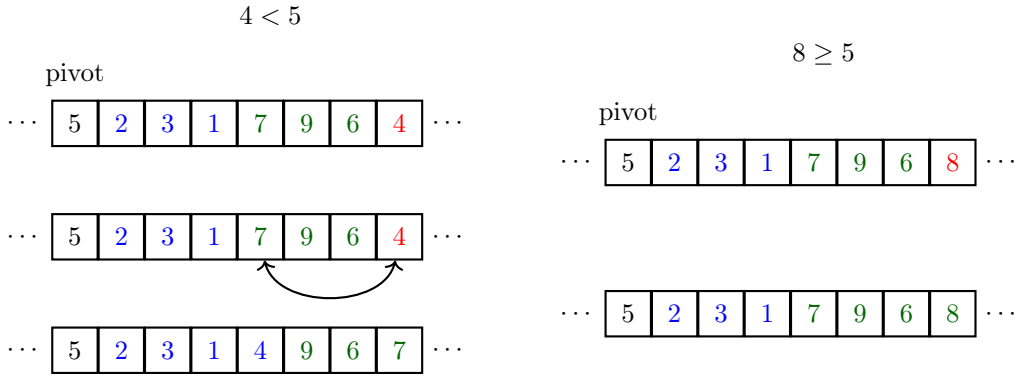


FIGURE 10.3 – Le pivot est 5. L'élément auquel on s'intéresse (en rouge) est soit strictement inférieur au pivot, auquel cas il est permuté avec l'élément supérieur ou égal le plus à gauche, soit supérieur, auquel cas on ne fait rien.

Code Python. La discussion précédente invite à repérer par un indice m la position de l'élément le plus à droite parmi ceux qui sont strictement inférieurs au pivot. À l'examen de `L[i]`, si `L[i]` $<$ `pivot` on incrémente `m` de 1, puis on fait l'échange entre `L[m]` et `L[i]` (on ne décide de le faire que si `m` $<$ `i`, ce qui signifie que la portion des éléments supérieurs ou égaux au pivot est non vide). En fin de fonction il suffit de permuter `L[m]` et `L[g]` pour amener le pivot en position `m` (de même, on ne le fait que si `m` $>$ `g`, ce qui signifie que la portion des éléments strictement inférieurs au pivot est non vide).

L'algorithme de partition

```
def partition(L,g,d): #g inclus, d exclus.
    pivot=L[g]
    m=g
    for i in range(g+1,d):
        #Inv(i): L[g+1:m+1] a ses éléments < pivot, ceux de L[m+1:i] sont >= pivot, avec i>=m+1.
        if L[i]<pivot:
            m+=1
```

1. On laisse l'écriture d'un tel tri en exercice.

```
            if i>m:
                L[i],L[m]=L[m],L[i]
        #Inv(i+1)
    if m>g:
        L[m],L[g]=L[g],L[m]
    return m
```

Terminaison et correction. Il est clair que cette fonction termine. Montrons la correction. L'invariant de la boucle `for` est :

Inv_i : Les éléments de la liste `L[g+1:m+1]` sont strictement inférieurs au pivot `L[0]`, les éléments de la liste `L[m+1:i]` sont supérieurs, avec $i \geq m+1$.

Montrons le :

- Inv_{g+1} est vrai en début de boucle : en effet, les deux sous-listes `L[g+1:m+1]` et `L[m+1:g+1]` sont vides dans ce cas, car $g+1 = m+1$.
- Pour tout $i \in \{g+1, d-1\}$, si Inv_i est vrai en haut de la boucle, alors Inv_{i+1} est vrai en bas de la boucle : en effet, si `L[i]>=pivot`, on ne fait rien. Si `L[i]<pivot`, on incrémente m puis on échange `L[i]` et `L[m]` (seulement si i est différent de m, car sinon il n'y a rien à faire). Donc Inv_{i+1} est vrai en fin de boucle.

Par suite, $\text{Inv}_{d-1+1} = \text{Inv}_d$ est vrai en fin de boucle : les éléments de `L[g+1:m+1]` sont strictement inférieurs au pivot, et les éléments de `L[m+1:d]` sont supérieurs. Comme le dernier `if` échange `L[g]` et `L[m]` (ce qui est nécessaire seulement si $m > g$), la fonction se termine avec `L` partitionnée au niveau du pivot. On renvoie m, qui est maintenant l'indice du pivot.

Complexité. Il est clair que la complexité de la fonction `partition(L,g,d)` est $O(d-g)$ (taille de la portion).

10.3.2 Le tri en lui-même

Idée du tri rapide. Si l'on a une portion délimitée par les indices g (inclus) et d (exclus) à trier, alors :

- si $g \geq d-1$, la portion possède 0 ou 1 élément et est donc triée : il n'y a rien à faire.
- sinon, on réalise une partition avec ces indices. On obtient alors l'indice m où l'on a mis le pivot, les éléments de `L[g:m]` sont strictement inférieurs à `L[g]`, ceux de `L[m+1:d]` sont supérieurs à `L[g]` : il suffit de recommencer sur les portions délimitées par (g, m) et $(m+1, d)$.

Code Python. On en déduit immédiatement un algorithme récursif, faisant usage d'une fonction auxiliaire.

Le tri rapide

```
def tri_rapide(L):
    def aux(g,d):
        if g<d-1: #sinon il n'y a rien a faire.
            m=partition(L,g,d)
            aux(g,m)
            aux(m+1,d)
    aux(0,len(L))
```

La fonction `tri_rapide` se contentant d'appeler `aux` entre les indices `0` et `len(L)`. Les preuves de terminaison et de correction sont très semblables à celles du tri fusion.

Complexité : pire cas. Supposons la liste triée dans l'ordre croissant. Alors le premier appel à partition réalise une très mauvaise partition : la liste est laissée dans le même état, et le pivot renvoyé est celui en position 0. Le premier appel récursif `aux(0,0)` ne fera strictement rien, alors que le second sera `tri_rapide(1,len(L))`, ce qui revient à recommencer le même procédé avec la liste `L[1:]`. Clairement, le tri est mauvais dans ce cas (pas meilleur que le tri par sélection) car quadratique en le nombre de comparaisons.

Complexité : meilleur cas (heuristique). En revanche, si chaque appel à `partition` sépare les éléments en deux parties égales à un élément près (plus le pivot), on se convainc facilement que la complexité est en $O(n\log(n))$ car la récurrence vérifiée par le tri est, à peu de chose près, celle du tri fusion.

Complexité : cas moyen (heuristique). Considérons une situation où les partitions effectuées ne sont pas mauvaises, mais pas non plus très bonnes, par exemple avec toujours un cinquième des éléments d'un côté et quatre cinquièmes de l'autre. L'arbre des appels récursifs (voir figure 10.4) est tordu par rapport à celui du tri fusion, car les portions de droite sont plus grande. Néanmoins, la portion de droite étant de taille $\frac{4^k \cdot n}{5^k}$, la hauteur reste en $O(\log n)$, car $\left(\frac{4}{5}\right)^k n \leq 1$ pour $k \geq \log_{5/4}(n)$. Chaque niveau étant en $O(n)$, on en déduit que la complexité reste en $O(n \log n)$. Même en prenant un ratio plus défavorable comme $1/100, 99/100$, la conclusion aurait été la même : ce n'est que dans le pire cas que la complexité est quadratique.

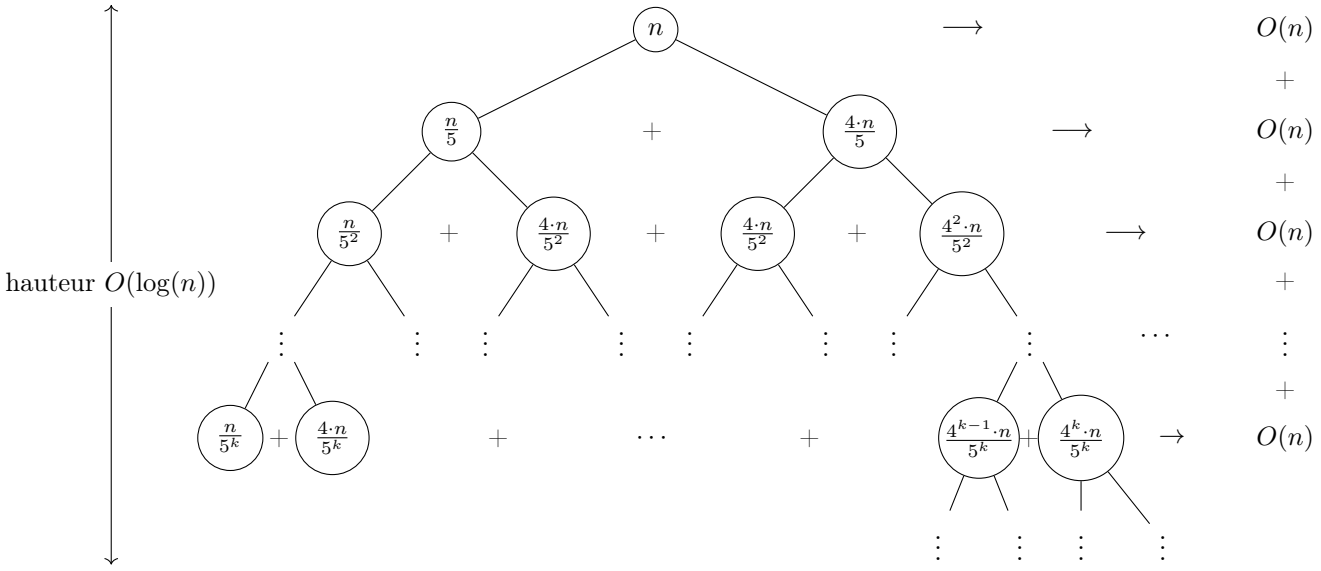


FIGURE 10.4 – Complexité dans le tri rapide, partitions aux ratios $1/5, 4/5$.

Complexité moyenne (preuve). Ce paragraphe est hors programme. On suppose que les éléments de la liste à trier sont tous distincts et que les positions relatives des éléments sont équiprobables. Puisque le tri fonctionne par comparaisons, on peut supposer que la liste à trier est constituée des éléments de l'ensemble $\{0, 1, \ldots, n-1\}$, et on veut compter le nombre de comparaisons effectuées en moyenne, c'est-à-dire :

$$C_{\text{comp,moy}}(n) = \frac{1}{n!} \sum_{\sigma \in \mathfrak{S}_n} C_{\text{comp}}([\sigma(0), \ldots, \sigma(n-1)])$$

Pour donner une estimation de cette complexité, on va raisonner en termes probabilistes, chaque permutation σ ayant probabilité $1/n!$. Remarquons que deux éléments $i < j$ ne sont comparés qu'au plus une fois par l'algorithme, et c'est le cas si et seulement si l'un des deux est un pivot au moment où l'on applique la fonction `partition` sur une sous-liste contenant à la fois i et j. Il est intéressant de remarquer que, lorsque l'on applique la fonction `partition` sur une sous-liste contenant à la fois i et j, avec $i < j$:

— soit le pivot est i ou j et dans ce cas i et j sont comparés.
— soit le pivot est un élément k vérifiant $i < k < j$ et i et j ne seront *jamais* comparés, parce qu'après la fonction de partition k est à sa place finale dans la liste et l'algorithme est rappelé sur deux portions ne contenant pas à la fois i et j.
— soit le pivot est un élément k vérifiant $k < i$ ou $k > j$, dans ce cas, la fonction de `partition` est rappelée sur une sous-liste plus petite contenant encore i et j.

Considérons pour $i < j$ l'ensemble $U_{i,j} = \{i, i+1, i+2, \ldots, j-1, j\}$. La discussion précédente montre que :

> Les éléments i et j sont comparés si et seulement si i ou j est le premier élément de $U_{i,j}$ à être choisi comme pivot dans le déroulement de l'algorithme.

Or, tout élément de $U_{i,j}$ a même probabilité d'être choisi en premier, à savoir $\frac{1}{|U_{i,j}|} = \frac{1}{j-i+1}$. Ainsi, en notant X la variable aléatoire donnant le nombre de comparaisons effectuées par le tri, on a $X = \sum_{i<j} X_{i,j}$ avec $X_{i,j}$ la variable

aléatoire prenant la valeur 1 si i et j sont comparés, et 0 sinon. $X_{i,j}$ suit donc une loi de Bernoulli de paramètre $\frac{2}{j-i+1}$. L'espérance d'une variable aléatoire étant une forme linéaire, on a :

$$\mathbb{E}(X) = \sum_{i<j} \mathbb{E}(X_{i,j}) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} \frac{2}{j-i+1}$$

Donnons un équivalent de cette somme lorsque n est grand :

$$\begin{aligned} \mathbb{E}(X) &= 2\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} \frac{1}{j-i+1} \\ &= 2\sum_{i=0}^{n-2} \sum_{k=2}^{n-i} \frac{1}{k} = 2\sum_{i=0}^{n-2} \sum_{k=2}^{i+2} \frac{1}{k} \\ 2\sum_{i=0}^{n-2}(\ln(i+3)-\ln(2)) \leq \mathbb{E}(X) &\leq 2\sum_{i=0}^{n-2}\ln(i+2) \end{aligned}$$

Les deux inégalités étant obtenues par encadrement de $\frac{1}{k}$ entre $\int_k^{k+1} \frac{dx}{x}$ et $\int_{k-1}^{k} \frac{dx}{x}$. Les deux sommes obtenues ont pour même terme général un équivalent de $\ln(i)$ lorsque i tend vers l'infini. Par une comparaison série intégrale ($t \mapsto \ln(t)$ est croissante et tend vers l'infini), on en déduit que :
$\mathbb{E}(X) \underset{n\to+\infty}{\sim} 2n\ln(n) = 2\ln(2)n\log_2(n) = O(n\ log n)$. Le nombre de comparaisons moyen étant en $O(n\log n)$, et la complexité de `partition` étant linéaire en le nombre de comparaisons effectuées, on voit que la somme des complexités des appels à partition est $O(n\log n)$, qui est la complexité moyenne du tri.

Propriétés. L'avantage du tri rapide sur le tri fusion est qu'il s'effectue en place. Par contre, il n'est pas stable.

Optimisations. une liste triée est un pire cas pour l'algorithme du tri rapide. Un moyen d'y remédier est de mélanger la liste de manière aléatoire. En fait, plutôt que de faire ce pré-traitement, il est plus judicieux de modifier l'algorithme de partition pour que le pivot de la liste `L[g:d]` choisi ne soit pas `L[g]`, mais un élément pris au hasard dans `L[g:d]`. On peut ainsi ajouter les lignes suivantes au début de l'algorithme `partition` :

```
pos_pivot=randint(g,d-1)
if pos_pivot>g:
    L[g],L[pos_pivot]=L[pos_pivot],L[g]
```

où `randint`, importée du module `random`, prend en entrée deux entiers a et b et retourne un élément aléatoire de $[\![a,b]\!]$.

Une deuxième optimisation possible est la suivante : si l'algorithme (même dans sa version aléatoire), est proche du pire cas, on risque sur de grands listes de dépasser la limite des 1000 appels récursifs imbriqués de Python. Pour y remédier, il est possible de transformer l'appel récursif sur la plus grande des sous-listes en boucle `while`. Cette transformation est laissée en exercice.

Enfin, la fonction de partition proposée dans ce cours fait beaucoup d'échanges, il est possible de l'améliorer un peu.

10.4 Conclusion et comparaison des tris

Le tableau suivant récapitule les résultats que nous avons montré sur les deux tris efficaces de ce chapitre :

Tri	Pire Cas	Meilleur Cas	Cas moyen
Tri fusion	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$
Tri rapide	$O(n^2)$	$O(n\log n)$	$O(n\log n)$

TABLE 10.1 – Complexité des tris efficaces (notation O)

Les deux graphiques suivants présentent les temps d'exécutions des tris de ce chapitre ainsi que des tris naïfs sur des listes aléatoires de taille variant entre 10 et 1000. Comme on le voit, il vaut mieux utiliser un tri efficace dès que la taille de la liste à trier atteint quelques dizaines. Pour des listes de taille moyenne (plus de 500 éléments) le temps d'exécution des deux tris rapides est bien inférieur à celui des tris quadratiques. Notons que sur ces listes aléatoires, le tri rapide est incontestablement le meilleur, alors que si l'on somme les opérations élémentaires effectuées, on trouve un avantage au tri fusion : $2n\log_2(n) = \frac{2}{\ln(2)}\ln(n)$, et $\frac{2}{\ln(2)} \simeq 2.89 < 4$. L'avantage est dû au fait qu'il s'exécute en place.

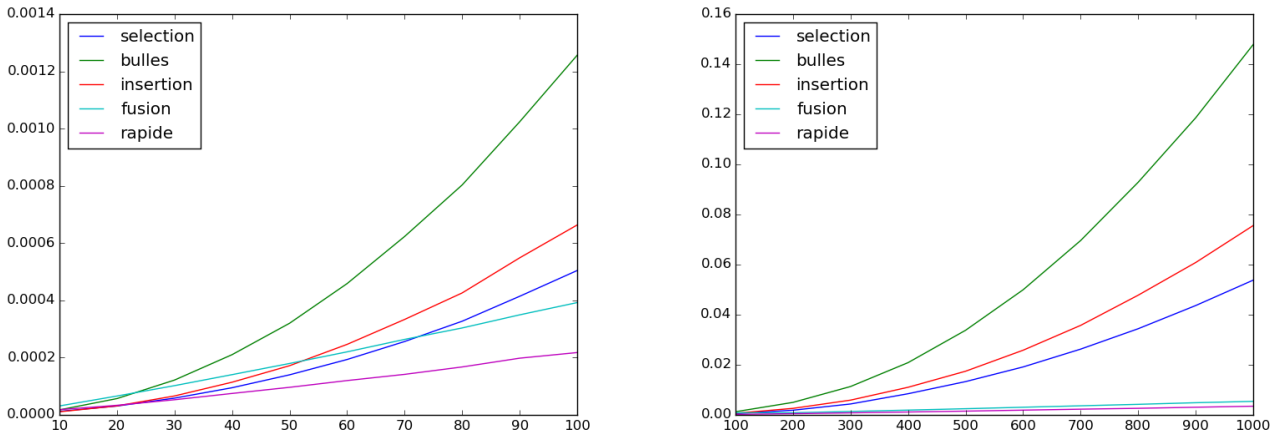


FIGURE 10.5 – Comparaisons des tris pour des tailles de 10 à 100 à gauche, et de 100 à 1000 à droite. Le temps est en secondes, on a pris une moyenne sur 100 listes différentes.

Chapitre 11

Validation des entrées et jeux de tests pour un programme

11.1 Introduction

Ce court chapitre présente une brève introduction à la notion de validation des entrées d'un programme, et aux tests. Il est un peu déconnecté du reste et ne semble pas à l'auteur de ces lignes très en adéquation avec l'idée d'une introduction à l'algorithmique : l'optique de la formation en tronc commun de classes préparatoires étant de former des étudiants capables de s'aider de l'outil informatique pour un usage varié, éventuellement dans d'autres disciplines. En particulier, lorsqu'on programme pour soi, le but n'est pas de noyer le code dans une vérification intensive (et inutiles !) des types, ainsi qu'une gestion d'exceptions qui ne se produiront pas. De plus, il me paraît contre-productif d'accorder à ce niveau une importance démesurée aux tests, il vaut mieux chercher à écrire un programme correct d'un seul jet en s'appuyant sur des invariants de boucle (en programmation itérative) ou des propriétés de récurrence (pour la programmation récrusive).

Néanmoins, à titre culturel, il est bon de savoir qu'un programmeur doit, en particulier dans un domaine sensible comme la sécurité, savoir ce que fait son programme lorsque les entrées ont la forme attendue, *mais également lorsqu'elles ne l'ont pas*. De même, un vrai travail de développement logiciel se fait en équipe, et la phase de tests, devant idéalement être réalisée par quelqu'un qui n'a pas travaillé directement sur le logiciel, est une tâche ingrate mais essentielle avant le déploiement. On discutera donc de commment former un petit jeu de tests pour nos programmes.

11.2 Validation des entrées. Assertions

11.2.1 Assertion

Commençons par un exemple simple :

```
def fact(n):
    """ Renvoie la factorielle de n """
    if n==0:
        return 1
    else:
        return n*fact(n-1)
```

Cette fonction est tout à fait correcte lorsqu'on lui passe un entier positif en paramètre. Néanmoins, `fact(-1)` est un appel qui ne termine pas[1]. Une fonction plus robuste vis à vis de l'entrée (et plus claire dans sa chaîne de documentation) serait :

```
def fact(n):
    """
    Entrée : n entier positif.
    Renvoie la factorielle de n.
    """
    assert n>=0, "n doit etre positif"
    if n==0:
        return 1
```

1. En pratique, le programme s'arrête parce que la capacité de la pile d'appels de fonction est dépassée.

```
    else:
        return n*fact(n-1)
```

L'appel précédent produisent maintenant (on a enlevé quelques lignes) :

```
>>> fact(-1)
[..]
AssertionError: n doit etre positif
```

Et le code de la fonction suivant les instructions `assert` n'est pas exécuté.

Fonctionnement de `assert`. On utilise cette instruction sous la forme `assert condition`. À l'exécution, `condition` est évaluée, le résultat doit être un booléen [2]. Si ce booléen est `False`, l'exécution du programme est immédiatement arrêtée et une erreur d'assertion est indiquée. L'instruction peut être suivie d'une chaîne de caractères (optionnelle, et techniquement c'est hors programme !) sous la forme `assert condition, chaine`. La chaîne sera affichée en complément de l'erreur d'assertion, comme ci-dessus.

Un mot sur `try, except` (hors programme). Dans la pratique, on pourrait vouloir que l'exécution du code se poursuive même si l'entrée est non valide. Par exemple dans un jeu de « Plus ou Moins » où on impose au joueur de rentrer des nombres entre 1 et 1000, on peut utiliser le code suivant dans une boucle `while` pour s'assurer que le joueur rentre bien un tel entier.

```
s=input('Tentative ?')
try:
    n=int(s)
    assert n>0 and n<=1000
except:
    print('Merci de rentrer un nombre entre 1 et 1000 !')
```

Le principe général est le suivant : le code dans le bloc `try` est exécuté. Si celui-ci produit une erreur (erreur d'assertion, division par zéro, dépassement d'indice dans une liste, échec d'une fonction...), l'exécution du bloc s'arrête et c'est le code du bloc `except` est exécuté. Si l'exécution du bloc `try` se fait sans erreur, le bloc `except` est ignoré et l'exécution se poursuit après celui-ci.

Une remarque sur les exceptions. Cette remarque est parfaitement facultative, et n'est là que dans l'espoir de fournir un cours complet et sans éléments glissés sous le tapis. En Python, `assert` n'est pas utilisable si une exécution optimisée est requise, notamment en cas de compilation préalable en C : dans ce cas un `assert` est ignoré, ce qui est problématique si le but est effectivement de produire une erreur ! La manière correcte de procéder dans ce cas est de lever une exception via `raise`. voir par exemple : `https://docs.python.org/fr/3.5/tutorial/errors.html` pour apprendre à lever et rattraper des exceptions.

11.2.2 Type d'un élément

Reprenons l'exemple précédent. Si l'on effectue `fact(3.4)` ou encore `fact("coucou")`, on aura une erreur :

— dans le premier cas, le programme s'exécute jusqu'à l'appel récursif sur `-0.4`, on obtient alors l'erreur d'assertion précédente (qu'un utilisateur pourrait ne pas comprendre car il a passé `3.4` en paramètre !

— dans le second cas, on obtient une erreur dans l'évaluation de `n>=0` car un entier et une chaîne ne sont pas comparables.

Ces deux cas peuvent être évités en vérifiant au préalable que l'argument est bien un entier. Les types usuels ont déja été vus : `int`, `float`, `bool`, `str`, `list`, `dict`... Voici deux expressions booléennes équivalentes permettant de tester que `n` est un entier :

— `type(n) == int` ;

— `isintance(n, int) == int`.

La fonction `fact` peut donc être complétée ainsi :

2. Si ce n'est pas le cas, il sera converti, sachant que quelque chose de vide ou nul est converti en `False`, le reste en `True`. Mais n'utilisez pas de conversions implicites qui rendent les choses peu claires : même si `assert L` et `assert L != []` sont équivalents, préférez la deuxième écriture.

```
def fact(n):
    """
    Entrée : n entier positif.
    Renvoie la factorielle de n.
    """
    assert isinstance(n,int), "n doit être un entier"
    assert n>=0, "n doit etre positif"
    if n==0:
        return 1
    else:
        return n*fact(n-1)
```

Remarque 11.1. *On pourrait multiplier les tests ainsi, et même vérifier que le nombre d'arguments passés par l'utilisateur est bien le bon, mais c'est sans fin et ne me semble pas d'un grand intérêt à notre niveau. En particulier, en DS, devant votre copie de concours, ou lorsque vous codez pour votre TIPE ou en TP, ne passez pas votre temps à vérifier que les entrées sont correctes sans que l'on vous le demande.*

11.3 Tests d'un programme

Les tests sont essentiels en informatique, lors du déploiement d'un logiciel. En effet, un *bug* peut coûter très cher ! Par exemple, en 1996, le premier lancement de la fusée Ariane 5 s'est soldé par un échec. La cause se situe dans le système de navigation, qui était le même que pour Ariane 4, réputé fiable. Or celui-ci était en partie inadapté pour Ariane 5, plus puissante. Le Centre national d'études spatiales a voulu économiser le test du logiciel pour économiser 800000 francs, qui aurait détecté le problème. La destruction de la fusée a causé la perte des satellites qu'elle transportait, pour un coût de plusieurs centaines de millions de dollars [3].

11.3.1 Plusieurs niveaux de tests

Cette section se veut une brève introduction culturelle aux tests. Tout d'abord, notons que les tests logiciels se font en général à plusieurs niveaux. On distingue potentiellement quatre niveaux de tests :

— tests unitaires : menés par le développeur lui-même, pour valider ses propres fonctions. Il s'agit d'élaborer de petits jeux de tests pour chaque fonction, permettant de recouvrir un maximum de situations possibles et en valider le fonctionnement. Dans la suite, on va principalement se concentrer sur les tests unitaires, les autres ne sont détaillés qu'à titre culturel.

— tests d'intégration : menés par un testeur dédié mais proche du développeur. Il s'agit de tester l'interaction des fonctions entre elles. On parle de « tests en boîte blanche » : il s'agit de regarder précisément les relations entre les fonctions pour vérifier leur coopération.

— tests de qualification : menés normalement par une équipe distincte de l'équipe du développement. Là le logiciel est vu comme une « boîte noire » : on teste le logiciel dans son ensemble sans se préoccuper des composants internes.

— tests d'acceptation : ce sont les tests par les premiers utilisateurs du logiciel.

11.3.2 Des tests de natures différentes

Voici une liste non exhaustive de tests qui peuvent être réalisés.

— tests fonctionnels : des tests dans des conditions d'utilisation normale. Pour un test unitaire, il s'agirait de tests où les entrées sont conformes à la spécification. *Remarque :* pour une variable pouvant, d'après la spécification, prendre des valeurs dans un intervalle fini d'entiers $[\![a, b]\!]$, on porte une attention particulière aux limites : les entiers $a, a+1, b-1$ et b.

— tests de robustesse : il s'agit par contre de voir comment réagit le logiciel dans des conditions dégradées ou aux limites. Pour un test unitaire, cela pourrait être lorsque les entrées ne sont pas conformes à la spécification. Dans ce cas, que se passe-t-il ? *Remarque :* pour une variable pouvant, d'après la spécification, prendre des valeurs dans un intervalle fini d'entiers $[\![a, b]\!]$, on utilise couramment deux tests de robustesse : l'entier $a-1$ et l'entier $b+1$.

— tests de performance : il s'agit d'évaluer la qualité du logiciel en terme de consommation de ressources (temps de calcul, quantité de mémoire, flux...)

3. L'histoire mieux racontée ici : `https://fr.wikipedia.org/wiki/Vol_501_d%27Ariane_5`

— ...

À notre niveau, on va se contenter de tests fonctionnels et de tests de robustesse. Les algorithmes au programme en classes préparatoires sont suffisamment simples pour qu'une analyse de complexité soit possible, ce qui limite l'intérêt d'un test de performance. Toutefois, des tests de performance auraient du sens pour des algorithmes plus complexes.

11.3.3 Un exemple complet : classification de triangles

Le problème. Cet exemple est inspiré d'un problème dans *The Art of Software Testing* de G. J. Meyers. On se pose la question suivante :

> Écrire une fonction `nature_triangle(L)` prenant en entrée une liste de 3 entiers strictement positifs, décrivant les longueurs des côtés d'un triangle non plat, et renvoyant un caractère indiquant sa nature : `'E'` (pour équilatéral), `'I'` (pour isocèle), ou `'S'` (pour scalène, c'est-à-dire que les longueurs des côtés sont distinctes deux à deux).

Voici les tests qu'il serait nécessaire d'effectuer, selon l'auteur.

1. Tests fonctionnels : il s'agit d'effectuer au moins 6 tests.
   - cas équilatéral valide (comme `[2, 2, 2]`)
   - cas isocèle valide, avec les trois permutations possibles (comme `[2, 2, 3]`, `[2, 3, 2]` et `[3, 2, 2]`).
   - cas scalène valide (comme `[2, 3, 4]`).
2. Tests de robustesse : on prête une attention particulière aux limites du domaine de la spécification (triangle plat, entrée nulle...). La liste suivante contient déja 12 tests !
   - cas où l'entrée n'est pas une liste ;
   - cas où l'entrée ne contient pas 3 éléments ;
   - cas où l'une des entrées de la liste n'est pas un entier ;
   - cas où l'une des entrées de la liste est strictement négative ;
   - cas où l'une des entrées de la liste est nulle ;
   - cas où toutes les entrées de la liste sont nulles ;
   - cas où le triangle est plat (c'est-à-dire que deux des entrées ont une somme égale à la troisième, avec les trois permutations possibles) ;
   - cas où deux des entrées ont une somme strictement inférieure à la troisième (avec les trois permutations possibles) ;

Tests fonctionnels. Les tests fonctionnels peuvent être décrits par des assertions, l'ensemble formant un petit *jeu de tests* : c'est très utile de disposer d'un tel jeu, qui reste pertinent pour tester plusieurs implémentations possibles. Voic un tel exemple de jeu de tests :

```
assert nature_triangle([2, 2, 2]) == 'E' #équilatéral
assert nature_triangle([2, 2, 3]) == 'I' #isocèle 1
assert nature_triangle([2, 3, 2]) == 'I' #isocèle 2
assert nature_triangle([3, 2, 2]) == 'I' #isocèle 3
assert nature_triangle([3, 2, 4]) == 'S' #scalène
```

On pourrait rajouter des tests où l'une des longueurs est 1 : c'est la limite du domaine de chaque élément de la liste.

Tests de robustesse. Le comportement attendu n'est pas spécifié lorsqu'on exécute une entrée non valide, utiliser des assertions n'a donc pas lieu d'être pour des tests de robustesse. On peut néanmoins élaborer un jeu de tests, et se renseigner sur ce que produit la fonction sur ces cas là. On aimerait voir des erreurs explicites pour chaque cas !

Implémentation de la fonction. Voici une implémentation de la fonction, passant les tests fonctionnels et renvoyant une erreur d'assertion sur chaque test de robustesse que l'on pourrait écrire.

```
def nature_triangle(L):
    assert type(L)==list, "l'entrée n'est pas une liste"
    assert len(L)==3, "la liste doit contenir 3 éléments"
    for x in L:
        assert type(x)==int, "chaque élément de la liste doit être entier"
        assert x>0, "chaque élément de la liste doit être strictement positif"
```

```
    a,b,c=L
    assert a+b>c and a+c>b and b+c>a, "les entrées ne forment pas un triangle non plat"
    if a==b and b==c:
        return 'E'
    elif a==b or a==c or b==c:
        return 'I'
    else:
        return 'S'
```

Chapitre 12

Introduction aux graphes

12.1 Introduction

L'intérêt pour les graphes remonte au 18ème siècle d'un point de vue mathématiques. L'exemple historique est le problème qu'Euler s'était posé dans la ville de Königsberg (aujourd'hui Kaliningrad). Peut-on partir d'un point de la ville, faire une promenade en passant par tous les ponts une seule fois ? La réponse est non. On peut montrer relativement facilement qu'un parcours de graphe *eulérien* (qui passe par toutes les arêtes, une seule fois) est possible si et seulement si le nombre de sommets de degré impair est 0 ou 2, ce qui n'est pas le cas ici.

FIGURE 12.1 – Modélisation du problème des ponts de Königsberg par des graphes. Les images proviennent de Wikipédia

Aujourd'hui, d'un point de vue pratique, les graphes sont présents partout :

- réseaux routiers ;
- réseaux de distribution d'eau, électricité ;
- Web et liens entre les pages ;
- Facebook et autres réseaux sociaux ;
- ...

Dans ce cours, on va s'intéresser aux questions algorithmiques, comme les suivantes.

- Comment parcourir un graphe ?
- Quelle est la plus courte distance entre 2 sommets d'un graphe ?

12.2 Vocabulaire des graphes et propriétés mathématiques

12.2.1 Graphes non orientés

Définition 12.1. *Un graphe est un couple (V, E) tel que :*

- *V est un ensemble fini non vide, ses éléments sont appelés* sommets *ou* nœuds.
- *E est un sous-ensemble de $\mathcal{P}_2(V)$, les parties à 2 éléments distincts de V. Les éléments de E sont appelés les* arêtes *du graphe.*

On utilise la notation anglo-saxonne : V pour vertices (sommets) et E pour edges (arêtes).

Remarque 12.2. *Puisque $E \subseteq \mathcal{P}_2(V)$, un graphe à n sommets a au plus $\binom{n}{2} = \frac{n(n-1)}{2}$ arêtes.*

Remarque 12.3. *Les graphes au programme ne présentent pas de multi-arêtes (plusieurs arêtes possibles entre deux sommets donnés). Le graphe associé au problème des ponts de Königsberg présente des multi-arêtes. Le programme officiel mentionne la définition de* boucle *(des arêtes qui relieraient un sommet à lui-même), notion qui n'est d'aucune utilité dans les applications visées : je choisis de simplifier les définitions.*

Incidence et degré. On dit :

— que deux sommets v et w sont *adjacents* (ou *voisins*) s'ils sont reliés par une arête (c'est-à-dire $\{v, w\} \in E$) ;

— qu'une arête est *incidente* aux sommets qu'elle relie.

Le nombre de sommets $|V|$ du graphe s'appelle l'*ordre du graphe*, que l'on notera en général n dans ce cours. Le *degré* d'un sommet v, noté $d(v)$ est le nombre d'arêtes qui lui sont incidentes.

Chemins et cycles. Un chemin de longueur p dans le graphe est une suite de $p+1$ sommets $v_0, v_1, \dots, v_p$, telle que $\{v_i, v_{i+1}\} \in E$ pour tout $0 \leq i \leq p-1$.

Définition 12.4. *Un cycle de G est un chemin $v_0, v_1, \dots, v_p$, avec $p \geq 1$, composé d'arêtes distinctes, tel que $v_0 = v_p$.* [1]

Définition 12.5. *Un graphe est dit acyclique s'il ne possède pas de cycle.*

Connexité.

Définition 12.6. *Un graphe non orienté $G = (V, E)$ est dit* connexe *si, pour deux sommets quelconques u et v de V, il existe un chemin de u à v dans G.*

Remarque 12.7. *On peut vérifier que la relation « être relié à ... par un chemin » est une relation d'équivalence sur les sommets d'un graphe non orienté. On parle de composantes connexes pour les classes d'équivalence pour cette relation.*

Exemple 12.8. *En figure 12.2 est illustré un graphe à trois composantes connexes.*

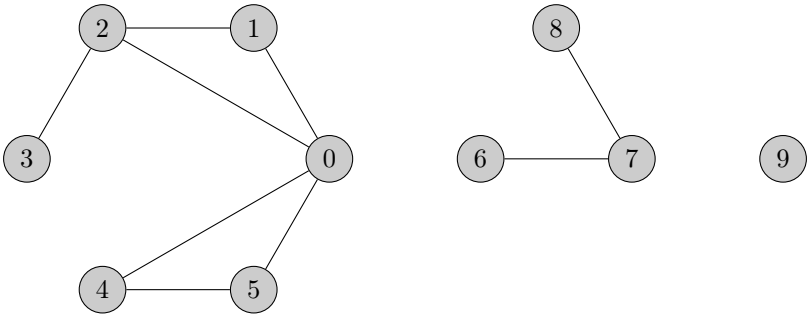


FIGURE 12.2 – Un graphe non orienté qui n'est pas connexe

Exemple 12.9. *Voici deux graphes « classiques » sur $[\![0, n-1]\!]$ (voir figure 12.3) :*

— *le cycle C_n, dont les arêtes sont $\{i, i+1\}$ pour tout $i \in [\![0, n-2]\!]$ ainsi que $\{0, n-1\}$;*

— *la clique K_n, constituée des $\binom{n}{2}$ arêtes possibles.*

Remarque 12.10. *Les graphes connexes acycliques sont appelés des* arbres.

1. Ce n'est pas forcément standard. Certains auteurs se contentent de $v_0 = v_p$. Mais pour parler de graphes acyliques comme dans la suite, il faut éliminer les cycles « triviaux », comme par exemple un chemin parcouru dans les deux sens, ce qui est assez pénible à écrire convenablement. Une autre définition possible est celle d'un chemin de longueur au moins 3 constitué de sommets distincts, sauf les extrémités. La définition proposée ici permet d'englober les boucles.

FIGURE 12.3 – Le cycle C_6 et la clique K_6

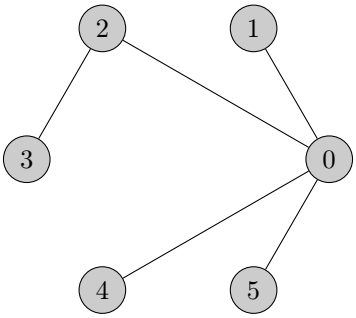


FIGURE 12.4 – Un exemple d'arbre (au sens des graphes non orientés)

12.2.2 Graphes orientés

Les graphes orientés sont des graphes où les arêtes ont un sens. La plupart des définitions s'adaptent à ce cadre, à quelques modifications près.

Définition 12.11. *Un graphe orienté est un couple $G = (V, E)$ où E est un ensemble de couples d'éléments distincts de V. Les éléments de E sont alors appelés* arcs *(au lieu d'arêtes).*

Remarque 12.12. *Un graphe d'ordre n possède donc au plus $n(n-1)$ arcs. Là encore, j'ai choisis d'interdire les boucles (arcs d'un sommet vers lui-même).*

Les arcs sont alors représentés comme des flèches entre deux sommets.

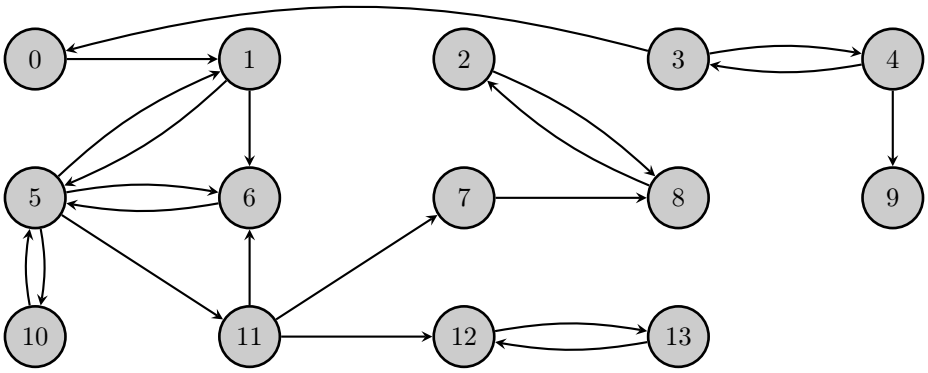


FIGURE 12.5 – Un graphe orienté

Degré entant et sortant. Puisque les arcs ont une orientation, on ne parle plus du degré d'un sommet v, mais de son degré entrant (nombre d'arcs de la forme (u, v)) et de son degré sortant (nombre d'arcs de la forme (v, u)). On notera $d_+(v)$ et $d_-(v)$ les degrés sortant et entrant d'un sommet.

Chemin et circuit. Dans un graphe orienté, une suite $v_0, v_1, \ldots, v_p$ telle que $(v_i, v_{i+1}) \in E$ est également appelée un chemin. On appelle circuit un chemin non réduit à un sommet, dont les sommets aux extrémités sont les mêmes.

Exemples. Quelques exemples de graphes orientés :

— réseaux routiers (il y a des routes à sens unique) ;

— (Humanité, $\{(h_1, h_2) \mid h_1$ connaît $h_2\}$) : nous sommes nombreux à connaître Alain Chabat, mais *a priori* il ne nous connaît pas !

— graphe divisoriel sur $[\![0, n-1]\!]$: $i \to j$ si $i|j$ et $i \neq j$.

12.3 Implémentation des graphes

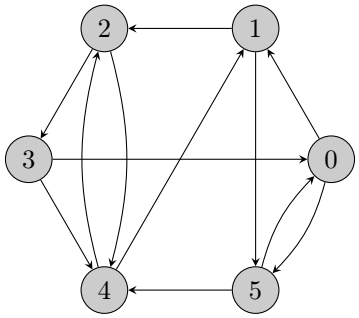
Du point de vue des applications, on s'intéressera à des graphes « statiques », c'est-à-dire que l'ensemble des sommets est fixé. Quite à procéder à un renommage, on supposera que l'ensemble des sommets est $[\![0, n-1]\!]$. Une petite remarque avant de débuter : il est un peu plus difficile de manipuler des graphes non orientés que des graphes orientés, car il faut faire attention à maintenir la non orientation. Comme il n'est pas facile de manipuler des ensembles à deux éléments, l'information « $\{u, v\}$ est une arête du graphe » est dupliquée sous la forme « u est voisin de v, v est voisin de u ». Il y a essentiellement deux manières d'implémenter un graphe :

— via des *listes d'adjacence* : on stocke pour chaque sommet la liste de ses voisins. Cette représentation est économique en mémoire, et adaptée aux graphes ayant peu d'arcs / arêtes (on dit qu'ils sont « creux »). On pourrait également utiliser des dictionnaires à la place des listes.

— via une *matrice d'adjacence* : on stocke à la case (i, j) d'une matrice $n \times n$ la présence ou non d'un arc entre les sommets i et j. On peut utiliser au choix des entiers (0 ou 1) ou des booléens. Cette représentation est bien adaptée aux graphes « denses » (ayant beaucoup d'arcs / arêtes).

12.3.1 Implémentation « creuse »

Pour un graphe à n sommets dont les sommets sont $[\![0, n-1]\!]$, on utilise une liste `G` de taille n. L'élément `G[i]` est une liste, contenant les voisins de `i`. En général, on n'impose pas que les voisins de `i` soient ordonnés dans `G[i]`.



Par exemple, le graphe orienté de la figure précédente peut être implémenté comme suit.

```
G = [[5, 1], [2, 5], [3], [0, 4], [2, 1], [0, 4]]
```

Pour représenter un graphe non orienté, on l'implémente simplement comme un graphe orienté, en dupliquant l'information : si l'arête $\{i, j\}$ est présente, i se trouve dans la liste d'adjacence de j et réciproquement.

Exercice 1. Écrire une fonction `desoriente(G)`, supprimant l'orientation d'un graphe orienté. Le résultat est un graphe non orienté où i et j sont voisins si et seulement si on avait i voisin de j ou j voisin de i dans `G`. Vous ne modifierez pas le graphe passé en paramètre mais en renverrez un nouveau.

Complexités. Cette représentation est économique, car elle nécessite un espace mémoire en $O(|V|+|E|)$. Parcourir les voisins d'un sommet se fait en temps linéaire en son degré, par contre il faut parcourir une liste d'adjacence pour tester l'existence d'un arc ou une arête. Pour cette opération, utiliser des dictionnaires serait préférable.

12.3.2 Représentation « dense »

Une manière de représenter un graphe dont les sommets sont $[\![0, n-1]\!]$ est d'utiliser une matrice pour indiquer l'existence d'un arc : la matrice d'adjacence.

Définition 12.13. *La matrice d'un adjacence d'un graphe* $(\llbracket 0, n-1 \rrbracket, E)$ *est la matrice* $M = (m_{i,j})_{0 \leq i,j \leq n-1}$ *définie par* $m_{i,j} = \begin{cases} 1 & si\ (i,j) \in E. \\ 0 & sinon. \end{cases}$

Par exemple, la matrice du graphe précédent est :

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

La diagonale de la matrice est constituée de zéros (si on avait des boucles dans nos graphes, il pourrait y avoir des uns). Pour un graphe non orienté, la matrice est symétrique. Voici, en Python, la matrice précédente (implémentée comme une liste de listes).

```
M=[[0, 1, 0, 0, 0, 1], [0, 0, 1, 0, 0, 1], [0, 0, 0, 1, 1, 0], [1, 0, 0, 0, 1, 0], [0, 1, 1, 0, 0, 0],
[1, 0, 0, 0, 1, 0]]
```

Exercice 2. De même que précédemment, écrire une fonction de désorientation d'un graphe, encodé via la représentation dense. Créer un nouveau graphe pour ne pas modifier l'entrée.

Complexités. Cette représentation n'est pas très économique en mémoire pour les graphes ayant peu d'arcs, car elle nécessite toujours un espace en $O(|V|^2)$. Parcourir les voisins d'un sommet se fait en temps $O(|V|)$ quel que soit le degré du sommet, par contre tester l'existence d'un arc ou une arête se fait en temps constant.

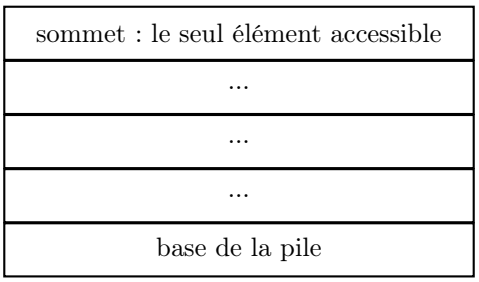
Exercice 3. Écrire des fonctions `creux_a_dense` et `dense_a_creux` permettant de passer d'une représentation à une autre. Vos fonctions ne modifieront pas l'entrée mais renverront de nouveaux objets.

12.4 Intermède : structure de pile et de file

Dans l'optique de parcourir des graphes, il nous faut définir deux structures abstraites très utilisées en informatique : la pile et la file. Le mot *abstrait* signifie que l'implémentation concrète joue un rôle secondaire : ce qui importe est la description de la structure et des opérations que l'on peut lui appliquer. Bien sûr, lorsqu'on souhaite réaliser une implémentation, on visera à avoir la meilleure complexité possible.

12.4.1 Structure de pile

Description de la structure. La structure abstraite de pile permet de stocker des éléments en suivant le principe *LIFO* : *last in, first out*. Ce qui signifie que les premiers éléments à sortir sont ceux qui ont été insérés en dernier. Visuellement, on peut se représenter une pile comme une pile du langage courant. L'élément inséré en dernier se trouve tout en haut, et s'appelle le *sommet* de la pile.



Dans le cahier des charges de la structure, les opérations qui doivent être permises sont les suivantes :

- Création d'une pile vide ;
- Test si une pile est vide ;
- Rajout d'un élément au sommet d'une pile ;
- Accès au sommet d'une pile non vide ;
- Suppression (et renvoi) du sommet d'une pile non vide.

Une implémentation possible. Via les méthodes `append` et `pop` (sans argument), une liste `L` se comporte comme une pile, le dernier élément de la liste étant le sommet.

Complexité. Les opérations `append` et `pop` (sans argument) s'effectuant en temps constant, la réalisation via une liste est très bonne.

12.4.2 Structure de file

Description de la structure. La structure abstraite de file est semblable à celle d'une file, mais suit le principe *FIFO* : *first in, first out*. Les opérations qui doivent être permises sont les suivantes :

— Création d'une file vide ;

— Test si une file est vide ;

— Rajout d'un élément dans la file ;

— Suppression (et renvoi) du premier élément inséré dans la file.

Implémentation avec une liste ? Avec une liste `L`, la méthode `pop` peut prendre également un indice en argument, dans ce cas l'élément correspondant à cet indice est supprimé (et renvoyé). On pourrait donc implémenter une structure de file dans une liste `L` via `L.append(x)` pour l'ajout (à la fin de la liste), et `L.pop(0)` pour la suppression (on supprime le premier élément de la liste). Malheureusement, cette idée est mauvaise du point de vue de la complexité : en interne, `L.pop(0)` nécessite de décaler dans la mémoire tous les éléments qui suivent le premier d'un cran, la complexité de cette opération est donc linéaire en la taille de la liste.

Remarque 12.14. *Il est en fait posible d'implémenter facilement une structure de file avec des listes, mais il faut en utiliser deux pour garantir une bonne efficacité des opérations.*

12.4.3 Sous-module `deque`

Présentation. Le module `collections` propose notamment la structure `deque`, pour *double ended queue*, c'est-à-dire en français *file à deux bouts*. Concrètement, c'est une structure de stockage linéaire des éléments (comme une liste !), mais qui fournit la possibilité d'effectuer l'insertion ou la suppression des deux côtés en temps constant.

Opérations. On ne discutera pas de la manière dont est réalisée cette structure en interne, voici simplement quelques exemples d'utilisation. La création se fait via `deque()`, et les opérations d'ajout sont `append` (à droite) et `appendleft` (à gauche). De même pour la suppression avec `pop` et `popleft`. On peut tester si une file à deux bouts est vide en regardant sa longueur.

```
>>> from collections import deque # importation
>>> f = deque() # une file à deux bouts vide
>>> for i in range(5): f.append(i) #ajout à droite
...
>>> f
deque([0, 1, 2, 3, 4])
>>> f.pop() #suppression à droite
4
>>> f.appendleft(5) ; f #ajout à gauche
deque([5, 0, 1, 2, 3])
>>> f.popleft() #suppression à gauche
5
>>> f
deque([0, 1, 2, 3])
>>> len(f)
4
```

Piles et files avec `deque`. On peut utiliser cette structure pour obtenir une pile ou une file : si on se restreint aux opérations `append` et `pop`, on obtient une pile, si on utilise `append` et `popleft` (ou, de manière symétrique, `appendleft` et `pop`), on obtient une file. À chaque fois les opérations se font en temps constant, c'est donc une bonne manière de réaliser les structures.

12.5 Parcours de graphes donnés par liste d'adjacence

Les parcours de graphes sont à la base de nombreux algorithmes sur les graphes. Ils vont nous permettre de calculer des plus courts chemins, tester la connexité, tester l'existence d'un cycle (circuit) dans un graphe, etc... On suppose que le graphe est donné par listes d'adjacence, et que l'ensemble de ses sommets est $[\![0, n-1]\!]$.

12.5.1 Parcours générique de graphe depuis un sommet source

On se donne un sommet source s_0, depuis lequel on veut explorer le graphe. On va donc suivre les arcs ou les arêtes et découvrir de nouveaux sommets. On suppose donnée une structure (de données) `a_traiter` dans laquelle on stocke les sommets à traiter : lorsqu'on traite un nouveau sommet, on ajoute à `a_traiter` la liste de ses sommets qui n'ont pas déja été traités. Pour savoir si un sommet a déja été rencontré, on utilise un tableau de booléens.

Algorithme 12.15 : Parcours générique de graphe

Entrée : Un graphe G donné par listes d'adjacence, un sommet de départ s_0
`a_traiter` $\leftarrow \{s_0\}$;
`B` $\leftarrow$ [`Faux`, ..., `Faux`] ;
`B`$[s_0] \leftarrow$ `Vrai`;
tant que *`a_traiter` est non vide* **faire**
  $s \leftarrow$ sortir un élément de `a_traiter`;
  pour *tout voisin s' de s tel que `B`$[s']$ est `Faux`* **faire**
    `a_traiter` $\leftarrow$ `a_traiter` $\cup \{s'\}$;
    `B`$[s'] \leftarrow$ `Vrai`

Si les opérations d'ajout et de retrait d'un élément dans `a_traiter` se font en temps constant, la complexité du parcours générique est linéaire en $O(|V| + |E|)$ (en fait plutôt que $|E|$ c'est même le nombre d'arcs/arêtes du sous-graphe induit par l'ensemble des sommets accessibles depuis s_0). En effet, chaque arc (resp. arête) est exploré au plus une fois (resp. 2 fois). L'algorithme précédent ne renvoie rien, mais on peut l'adapter pour obtenir des informations sur le graphe : ceci dépend de la structure de données utilisée. Il y a deux choix naturels :

— une file mène à un parcours dit *en largeur*

— une pile mène à un parcours ressemblant au parcours *en profondeur* vu plus loin.

Un tel parcours permet d'explorer uniquement des sommets accessibles depuis un sommet s_0 donné (un sommet s est dit accessible depuis s_0 s'il existe au moins un chemin de s_0 à s). Vérifions qu'il les explore tous.

Proposition 12.16. *Un parcours de graphe avec l'algorithme 12.15 lancé en s_0 visite tous les sommets accessibles depuis s_0.*

Démonstration. Posons $\delta(s) = \inf\{$ longueur d'un chemin de s_0 à $s\} \in \mathbb{N} \cup \{+\infty\}$, défini pour tout sommet du graphe. Naturellement, $\delta(s) < +\infty$ si s est accessible depuis s_0. Raisonnons par l'absurde et supposons qu'il existe au moins un sommet s accessible depuis s_0 mais non visité. Considérons un de ces sommets s tel que $\delta(s)$ est minimal. $s \neq s_0$ car s_0 est visité. Considérons un plus court chemin de s_0 à s, noté $s_0, s_1, \ldots, s_k = s$, avec $k = \delta(s)$. Clairement $\delta(s_i) \leq i$ pour tout $i \in [\![0, k]\!]$, mais il y a en fait égalité. En effet, s'il existait j tel que $\delta(s_j) < j$, on pourrait trouver un chemin de longueur strictement inférieur à $\delta(s)$ de s_0 à s (ce qui est absurde) en complétant un chemin de longueur $\delta(s_j)$ de s_0 à s_j avec $s_{j+1}, \ldots, s_k$. Ainsi $\delta(s_{k-1}) < \delta(s)$, et s_{k-1} est visité par l'algorithme, donc s l'est aussi. □

12.5.2 Parcours en largeur et plus courts chemins

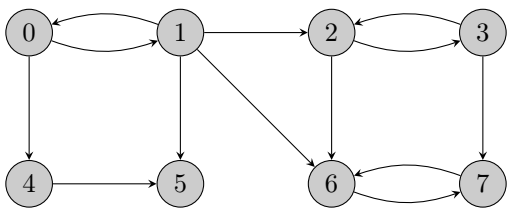


FIGURE 12.6 – Un exemple de graphe à parcourir

En partant du sommet 0, tous les sommets sont accessibles. En supposant les listes d'adjacences données dans l'ordre croissant, voici l'ordre dans lequel sont traités les sommets :

a_traiter	0	1, 4	4, 2, 5, 6	2, 5, 6	5, 6, 3	6, 3	3, 7	7	∅
B	0	0, 1, 4	0, 1, 2, 4, 5, 6	idem	0, 1, 2, 3, 4, 5, 6	idem	tous	tous	tous

Une application du parcours en largeur est le calcul de plus courts chemins depuis l'origine s_0 du parcours. Il suffit de rajouter une liste `dist` de distances à la source s_0. Lorsqu'on découvre un nouveau sommet t à partir d'un sommet s, `dist[t]` prend la valeur `dist[s]+1`.

Avant de montrer que cette approche est correcte, donnons un code complet en Python.

```
def parcours_largeur(G,s):
    """ graphe non pondéré, distances depuis l'origine s. On utilise une structure de file
    fournie par deque. """
    n=len(G)
    inf=float('inf')
    dist=[inf]*n
    dist[s]=0
    F=deque()
    F.append(s)
    while len(F)>0:
        x=F.popleft()
        for y in G[x]:
            if dist[y]==inf:
                dist[y]=dist[x]+1
                F.append(y)
    return dist
```

Dans le code précédent, on a utilisé le flottant infini comme initialisation. Tous les sommets à distance finie de s_0 auront après le parcours une valeur associée dans `dist` qui est positive : c'est la distance d'un plus court chemin entre s_0 et ce sommet.

Proposition 12.17. *À la fin de l'algorithme, si s est un sommet accessible depuis s_0,* `dist[s]` *contient la distance entre s_0 et s.*

Démonstration. Tout d'abord, les sommets sont insérés dans la file avec une distance au sommet s_0 croissante : en effet, ceci se montre facilement par récurrence sur la distance à s_0, en considérant la propriété $\mathcal{P}(d)$: « les sommets à distance d sont insérés dans la file avant tous ceux à distance au moins $d+1$ ». $\mathcal{P}(0)$ est vraie, et si $\mathcal{P}(d)$ est vraie, un sommet à distance $d+2$ est inséré à partir d'un sommet à distance (au moins) $d+1$, donc après traitement de tous les sommets à distance d et donc insertion des sommets à distance $d+1$, donc $\mathcal{P}(d)$ implique $\mathcal{P}(d+1)$.

Considérons maintenant le chemin donné par les prédécesseurs dans le parcours (si y est découvert pour la première fois dans la liste d'adjacence de x, on dit que x est le prédecesseur de y), entre s_0 et un sommet $s \neq s_0$, qu'on note $s_0 \to s_1 \to \cdots \to s_k = s$. Considérons de plus un plus court chemin de s_0 à s, qu'on écrit $s_0 \rightsquigarrow t \to s$. t a été inséré après s_{k-1} dans la file (car s est découvert par s_{k-1}), donc $\delta(s_0, s_{k-1}) \leq \delta(s_0, t)$ d'après la propriété précédente. Ainsi $\delta(s_0, s) = \delta(s_0, t) + 1 \geq \delta(s_0, s_{k-1}) + 1$, ce qui prouve que le chemin $s_0 \to s_1 \to \cdots \to s_k = s$ est un plus court chemin de s_0 à s. □

Remarque 12.18. *Après le parcours, on peut construire facilement des plus courts chemins en utilisant une autre liste* `pred`*, stockant pour chaque sommet différent de s_0 son prédecesseur dans le parcours. Si s est accessible depuis s_0, on peut construire un plus court chemin en remontant le chemin à l'envers depuis s en suivant les valeurs données par* `pred`*.*

12.5.3 Parcours en profondeur

Le parcours en profondeur d'un graphe consiste à s'enfoncer le plus possible dans le graphe avant de remonter vers des sommets déja vus, dont les voisins n'ont pas tous été découverts. On pourrait écrire un algorithme utilisant une pile à la place d'une file pour `a_traiter`, mais on découvrirait tous les voisins d'un sommet d'un coup, ce qui n'est pas un vrai parcours en profondeur (et il est important de respecter ceci dans les applications). Deux solutions :

— modifier légèrement l'algorithme générique : marquer un sommet comme traité une fois qu'il a été dépilé pour la première fois, pas dès qu'il est découvert. Ceci implique qu'un sommet peut se trouver plusieurs fois dans la pile (mais ne sera traité qu'une fois). L'inconvénient est une complexité spatiale $O(|V| + |E|)$ pour la pile, au lieu de $O(|V|)$;

— utiliser une formulation récursive (solution choisie !)

De plus, dans les applications du parcours en profondeur, on ne fait en général pas seulement un parcours élémentaire (sur un seul sommet comme dans le parcours générique) mais un parcours en profondeur « complet », qui consiste à relancer des parcours élémentaires tant que tout le graphe n'a pas été découvert. Pour assurer une complexité $O(|V| + |E|)$, on mutualise le tableau des booléens, comme dans le pseudo-code ci-dessous.

Algorithme 12.19 : Parcours en profondeur complet du graphe

Entrée : Un graphe G donné par listes d'adjacence
`deja_vu`$[v] \leftarrow Faux$ pour tout sommet v;
Fonction *pp(u)* **:**
  `deja_vu`$[u] \leftarrow Vrai$;
  pour *tout voisin v de u* **faire**
    si `deja_vu`$[v] = Faux$ **alors**
      pp(v)
pour *tout sommet u du graphe* **faire**
  si `deja_vu`$[u] = Faux$ **alors**
    pp(u)

Il est parfois utile de stocker plus d'informations pendant le parcours, ce qu'on verra à la section suivante.

On va voir dans la section suivante des variantes du parcours en profondeur permettant de résoudre les problèmes suivants :

— Test de la connexité d'un graphe non orienté ;

— Détection de circuit (resp. cycle) dans un graphe orienté (resp. non orienté).

Complexité. Tout comme le parcours en largeur, le parcours en profondeur a une complexité $O(|V| + |E|)$, pour les mêmes raisons : l'initialisation de `deja_vu` prend un temps $O(|V|)$, et ensuite la complexité est linéaire en le nombre d'arêtes/arcs $|E|$.

12.6 Applications du parcours en profondeur

12.6.1 Connexité d'un graphe non orienté

Un parcours en profondeur élémentaire (l'équivalent de la fonction `pp`) lancé sur un sommet permet de découvrir tous les sommets accessibles depuis ce sommet. Pour tester la connexité, il suffit donc d'effectuer un parcours élémentaire et de vérifier que l'on a atteint les n sommets.

Plus généralement, dans l'algorithme 18.19, appliqué à un graphe non orienté, chaque parcours élémentaire lancé dans la boucle `Pour` permet de découvrir un ensemble de sommets accessibles les uns les autres (et aucun autre sommet n'est accessible) : un tel ensemble de sommets s'appelle une composante connexe. L'algorithme suivant calcule les composantes connexes d'un graphe non orienté. Il complète simplement le parcours « théorique » avec une liste de listes.

```
def compo_connexes(G):
    """ Parcours en profondeur pour calculer les composantes connexes """
    n=len(G)
    deja_vus = [False]*n
    liste_cc = []
    def pp(v):
        deja_vus[v]=True
        for w in G[v]:
            if not deja_vus[w]:
                pp(w)
        liste_cc[-1].append(v)
    for i in range(n):
        if not deja_vus[i]:
            liste_cc.append([])
            pp(i)
    return liste_cc
```

Voici le résultat, avec le graphe de la figure 12.2.

```
>>> G = [[1, 2, 4, 5], [0, 2], [0, 3], [2], [0, 5], [0, 4], [7], [6, 8], [7], []]
>>> compo_connexes(G)
[[3, 2, 1, 5, 4, 0], [8, 7, 6], [9]]
```

Remarque 12.20. *Le programme officiel stipule la présence possible de boucles : cela ne change rien à cet algorithme.*

12.6.2 Détection de circuit dans un graphe orienté

Contextualisons un peu : une application possible des graphes orientés et la construction de graphes de dépendance, essentielle dans la compilation de programmes. Dans un tel graphe, un sommet est une tâche à effectuer, et il y a un arc entre le sommet i et le sommet j si la tâche i doit être effectuée avant la tâche j. L'existence d'un circuit dans un tel graphe est problématique : il est impossible d'effectuer les tâches du circuit en respectant les contraintes ! Pour éviter les *bugs*, il faut donc savoir les détecter, c'est le problème de la section.

Pour détecter l'existence d'un circuit, on va simplement se baser sur le parcours en profondeur. Rajoutons une couleur aux sommets du graphe, dépendant de leur statut durant le parcours :

— un sommet pas encore découvert sera *blanc* ;

— un sommet v en cours de traitement, c'est-à-dire que l'appel $pp(v)$ est en cours mais ne s'est pas terminé, sera *gris* ;

— un sommet v dont le traitement est terminé, c'est-à-dire que l'appel $pp(v)$ a été effectué et s'est terminé, sera *noir*.

La proposition suivante permet de donner une caractérisation des graphes possédant un circuit.

Proposition 12.21. *Un graphe orienté possède un circuit si et seulement si, lors du parcours en profondeur, un des appels $pp(v)$ trouve un sommet* gris *dans la liste d'adjacence de v.*

Démonstration.

- Si c'est le cas, notons w le sommet *gris* découvert dans l'appel $pp(v)$. Cela signifie que l'appel $pp(w)$ a engendré une série d'appels non terminés $(pp(w_i))_{0\leq i\leq k}$, avec $w_0 = w$ et $w_k = v$, avec pour tout $i \in [\![0, k-1]\!]$, w_{i+1} dans la liste d'adjacence de w_i : il y a bien un circuit !
- Réciproquement, supposons que le graphe possède un circuit $w_0, \dots, w_k = w_0$ ($k \geq 1$, w_{i+1} dans la liste d'adjacence de w_i pour tout $i \in [\![0, k-1]\!]$). Puisque l'algorithme du parcours en profondeur examine tous les sommets, il en découvre un de ce circuit. Quite à renuméroter les sommets du circuit, supposons que c'est w_0. Alors puisque tous les sommets du circuit sont accessibles depuis w_0, ils seront découverts pendant l'appel $pp(w_0)$. En particulier, l'arc (w_{k-1}, w_0) sera découvert dans l'appel $pp(w_{k-1})$, alors que w_0 est gris.

□

En conséquence, il suffit d'attribuer des couleurs aux sommets comme précisé au dessus, pour être capable de détecter un circuit. Dans le code suivant on utilise une liste supplémentaire pour encoder la couleur, avec la convention $(0, 1, 2)$ pour (blanc, gris, noir). On convient que `pp(v)` renvoie `True` si et seulement si aucun circuit n'a été détecté dans l'appel `pp(v)`.

```
def sans_circuit(G):
    """ renvoie un booléen indiquant si le graphe est sans circuit """
    n=len(G)
    couleurs = [0] * n #blanc, gris, noir = 0, 1, 2
    def pp(v):
        couleurs[v]=1
        for w in G[v]:
            if couleurs[w]==0:
                if not pp(w): #on découvre un circuit dans pp(w)
                    return False
            elif couleurs[w]==1: #sommet gris !
                return False
        couleurs[v]=2
        return True
    for i in range(n):
        if couleurs[i] == 0:
            if not pp(i): #on découvre un circuit dans pp(i)
                return False
    return True
```

Remarque 12.22. *Si on autorise les boucles, la présence d'une boucle sera détectée et considérée comme un circuit.*

12.6.3 Détection de cycle dans un graphe non orienté

Du point de vue de la connexité dans un graphe non orienté, une arête d'un cycle peut être supprimée sans nuire à la connexité. Dans une application où les arêtes coûtent des ressources (une route, un cable électrique, etc...), et qu'on ne s'intéresse qu'à la connexité, il est utile de savoir repérer l'existence d'un circuit, cela montre que l'on peut économiser des ressources. Le même algorithme que celui de la section précédente peut être utilisé avec quelques adaptations pour détecter la présence de cycles dans des graphes non orientés.

La seule petite difficulté réside dans le fait que dans un graphe non orienté, une arête $\{u, v\}$ est encodée à la manière de deux arcs $u \to v$ et $v \to u$: le circuit $u \to v \to u$ ne doit pas être compté. Pour ce faire, on peut simplement rajouter un paramètre à la fonction `pp(v)`, indiquant le sommet u ayant permis la découverte de v : on peut alors ignorer l'arc $v \to u$. Voici le code :

```
def sans_cycle(G):
    n=len(G)
    couleurs = [0] * n #blanc, gris, noir = 0, 1, 2
    def pp(v, parent):
        couleurs[v]=1
        for w in G[v]:
            if couleurs[w]==0:
                if not pp(w,v):
                    return False
            elif couleurs[w]==1 and parent!=w: #on ignore l'arc v -> parent
                return False
        couleurs[v]=2
        return True
    for i in range(n):
        if couleurs[i] == 0:
            if not pp(i, None): #le paramètre parent initial n'est pas pertinent
                return False
    return True
```

Remarque 12.23. *Dans le cas d'un graphe où les boucles sont autorisées, une boucle ne sera pas détectée comme un cycle dans le code ci-dessus, mais on pourrait facilement traiter ce cas avec un test* `w == v`.

12.7 Graphes pondérés

On va maintenant considérer des graphes où les arcs / arêtes sont munis d'un *poids*. Les applications sont nombreuses, car un poids sur un arc peut symboliser une capacité (quantité maximale que l'on peut faire transiter sur l'arc dans une unité de temps), ou un coût (durée ou distance d'un trajet, quantité de carburant nécessaire, etc...)

On a vu précédemment comment calculer la distance $\delta(s, t)$ entre un sommet source s et un sommet quelconque t dans un graphe non pondéré, à l'aide d'un parcours en largeur. Le but va être de généraliser cet algorithme au cas des graphes pondérés, on discute d'abord dans cette section des définitions et de l'implémentation.

12.7.1 Pondération

Définition 12.24. *On considère un graphe $G = (V, E)$ orienté ou non. Une* fonction de pondération *de G est une fonction $\omega : E \to \mathbb{R}$. Le réel $\omega(e)$ est appelé le* poids *de l'arête ou de l'arc e. Le graphe $G = (V, E, \omega)$ est appelé un graphe* pondéré.

On étend naturellement la fonction de pondération à tout couple de sommets pour obtenir une fonction $V^2 \to \mathbb{R} \cup \{+\infty\}$, avec la définition suivante :

$$\omega'(u, v) = \begin{cases} \omega((u, v)) & \text{si } (u, v) \in E \\ 0 & \text{si } u = v \\ +\infty & \text{sinon.} \end{cases}$$

Au programme officiel ne figure que l'algorithme de Dijkstra et une de ses variantes, qui nécessitent l'hypothèse suivante.

On suppose dans la suite que les arcs/arêtes ont un poids positif.

12.7.2 Poids d'un chemin

Définition 12.25. *La notion de poids d'un arc s'étend au poids d'un chemin : pour $p = v_0, v_1, \dots, v_n$ un chemin dans un graphe, on définit son poids comme $\omega(p) = \sum_{i=0}^{n-1} \omega(v_i, v_{i+1}) \in \mathbb{R}$.*

Définition 12.26. *Pour s et t deux sommets dans le graphe, on notera $\delta(s,t)$ le plus petit poids d'un chemin de s à t :*

$$\delta(s,t) = \inf\{\omega(p) \mid p \text{ est un chemin de } s \text{ à } t\} \in \mathbb{R}_+ \cup \{+\infty\}$$

Remarque 12.27. *Pour s et t deux sommets du graphe :*

- *s'il existe au moins un chemin de s à t, l'ensemble ci-dessus est non vide et $\delta(s,t) \in \mathbb{R}_+$ (car les poids sont positifs).*
- *sinon, cet ensemble est vide et $\delta(s,t) = +\infty$.*

Définition 12.28. *Pour s et t deux sommets tels que $\delta(s,t) \in \mathbb{R}_+$, on appelle chemin de plus petit poids de s à t un chemin de poids $\delta(s,t)$ entre s et t.*

Remarque 12.29. *Dans un cadre où les poids sont par exemple des distances (routes...), on pourra parler de distance et de plus court chemin, mais il ne faut pas confondre $\delta(s,t)$ avec le nombre d'arcs minimal reliant deux sommets.*

12.7.3 Quelques propriétés sur les poids des chemins

Les propriétés suivantes, assez naturelles, seront utiles pour montrer la correction de l'algorithme de Dijkstra.

Proposition 12.30 (Inégalité triangulaire)**.** *Soient u, v, w trois sommets d'un graphe pondéré $G = (V, E, \omega)$. Alors $\delta(u,w) \leq \delta(u,v) + \delta(v,w)$.*

Démonstration. Si $\delta(u,v) = +\infty$ ou $\delta(v,w)$, alors l'inégalité est immédiate. Sinon, on obtient un chemin de poids $\delta(u,v) + \delta(v,w)$ de u à w en concaténant un chemin de poids $\delta(u,v)$ de u à v et un chemin de poids $\delta(v,w)$ de v à w. D'où $\delta(u,w) \leq \delta(u,v) + \delta(v,w)$. □

Corollaire 12.31. *Avec les mêmes notations, on a également $\delta(u,w) \leq \delta(u,v) + \omega(v,w)$.*

Démonstration. C'est immédiat, puisque $\delta(v,w) \leq \omega(v,w)$. □

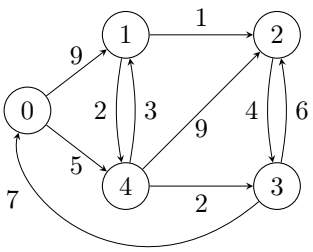
Optimalité des sous-chemins. Soient u et w deux sommets d'un graphe pondéré $G = (V, E, \omega)$. Supposons qu'un chemin c de plus petit poids de u à w passe par un sommet v. On note ces chemins $c_1 : u \rightsquigarrow v$ et $c_2 : v \rightsquigarrow w$ de sorte que c est la concaténation de c_1 et c_2. Alors le chemin c_1 (resp. c_2) est un chemin de plus petit poids de u à v (resp. de v à w). □

Démonstration. Supposons qu'il existe un chemin c_1' de poids strictement inférieur à $\omega(c_1)$ de u à v. Alors le chemin c' obtenu par concaténation de c_1' et c_2 est de poids strictement inférieur à $\omega(c_1) + \omega(c_2) = \omega(c)$, ce qui est absurde. De même pour l'autre morceau. □

12.7.4 Implémentation

- Dans le cas d'une implémentation creuse, il suffit dans la liste d'adjacence d'un sommet u, de stocker des couples (v,p) à la place du seul sommet v : ceci signifie que $(u,v) \in E$ et $\omega(u,v) = p$.
- Dans le cas d'une implémentation dense, on utilise la généralisation de la fonction ω à tout couple de sommets : la matrice d'adjacence est maintenant une matrices à valeurs dans $\mathbb{R} \cup \{+\infty\}$. On rappelle que le type flottant possède une valeur $+\infty$ que l'on peut obtenir comme `float('inf')`.

La figure suivante présente un exemple de graphe pondéré orienté.



En Python, sa représentation par liste d'adjacence serait par exemple :

```
G = [[(1, 9), (4, 5)], [(2, 1), (4, 2)], [(3, 4)], [(0, 7), (2, 6)], [(1, 3), (2, 9), (3, 2)]]
```

Alors que sa matrice d'adjacence et sa représentation en Python (sous forme de liste de listes) est la suivante, avec `inf = float('inf')` :

$$\begin{pmatrix} 0 & 9 & +\infty & +\infty & 5 \\ +\infty & 0 & 1 & +\infty & 2 \\ +\infty & +\infty & 0 & 4 & +\infty \\ 7 & +\infty & 6 & 0 & +\infty \\ +\infty & 3 & 9 & 2 & 0 \end{pmatrix}$$

```
G = [[0, 9, inf, inf, 5],
     [inf, 0, 1, inf, 2],
     [inf, inf, 0, 4, inf],
     [7, inf, 6, 0, inf],
     [inf, 3, 9, 2, 0]]
```

Remarque 12.32. *Attention à ne pas confondre la matrice d'adjacence dans les graphes pondérés et non pondérés : les zéros dans la matrice d'adjacence d'un graphe non pondéré deviennent des $+\infty$ dans la matrice d'un graphe pondéré, sauf sur la diagonale où ils restent des zéros. On excluera complètement la présence de boucles dans le contexte des graphes pondérés, la notion de coût pour aller d'un sommet à lui-même se devant d'être zéro !*

12.8 Algorithmes de Dijkstra et A*

12.8.1 Algorithme de Dijkstra

L'algorithme de Dijkstra [2] que l'on va voir est une généralisation du parcours en largeur : il consiste également à traiter les sommets un par un, par poids depuis l'origine s croissante. C'est donc un algorithme glouton. Lorsqu'on traite un nouveau sommet, on *relâche* tous les arcs issus de ce sommet.

Définition 12.33. *Les définitions suivantes sont utiles pour comprendre l'algorithme.*

- *Une* estimation des poids *depuis s est un tableau d vérifiant $d[t] \geq \delta(s,t)$ pour tout sommet t.*
- *Le* relâchement d'un arc $u \to v$ *consiste à réaliser l'opération* $d[v] \leftarrow \min(d[v], d[u] + \omega(u,v))$.

Algorithme 12.34 : Algorithme de Dijkstra

Entrée : Un graphe pondéré $G = (V, E, \omega)$, avec $\omega(E) \subset \mathbb{R}_+$, un sommet s
Sortie : Les poids minimaux $\delta(s,t)$ pour tout $t \in V$
$d[t] \leftarrow +\infty$ pour tout $t \in V$; $d[s] \leftarrow 0$;
$H \leftarrow \emptyset$; $F \leftarrow \{s\}$;
tant que $F \neq \emptyset$ **faire**
  $u \leftarrow$ Retirer de F un sommet v vérifiant $d[v]$ minimal parmi les sommets de F;
  pour *tout voisin v de u* **faire**
    si *v n'est ni dans F ni dans H* **alors**
      Ajouter v à F
    $d[v] \leftarrow \min(d[v], d[u] + \omega(u,v))$
  Ajouter u à H
Renvoyer d

Terminaison de l'algorithme. L'algorithme fait un parcours du graphe depuis le sommet s : on peut retrouver le parcours générique précédent en supprimant ce qui a trait au tableau d, et en remplaçant H par l'ensemble des sommets déja vus. Ainsi, l'algorithme termine.

Correction de l'algorithme. Faisons déja plusieurs observations :

— l'algorithme part d'une estimation (triviale) des poids $\delta(s,t)$. Puisque l'algorithme se contente de relâcher des arcs, on aura d'après le corollaire 12.31, $d[t] \geq \delta(s,t)$ pour tout sommet t à la fin de l'algorithme, cette propriété étant un invariant des boucles de l'algorithme.

— l'algorithme faisant notamment un parcours générique, à la fin de l'algorithme, tous les sommets t accessibles depuis s sont dans l'ensemble H, et vérifient tous $d[t] < +\infty$.

2. prononcer « Daïjkstra »

Montrons maintenant que l'algorithme est correct, via la proposition suivante :

Proposition 12.35. *Dans l'algorithme de Dijkstra, la propriété « tout sommet t de H vérifie $d[t] = \delta(s,t)$ » est un invariant de la boucle* **tant que**.

Démonstration. — la propriété est vraie avant la boucle, car l'ensemble H est vide.

— Pour montrer l'hérédité, il suffit de montrer qu'un sommet u de F vérifiant $d[u]$ minimal vérifie en fait $d[u] = \delta(s,u)$. Distinguons deux cas :

— si $u = s$ (c'est le premier tour de boucle), on a $d[s] = \delta(s,s) = 0$;

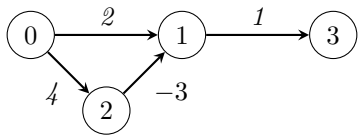
— sinon, considérons un chemin de plus petit poids de s à u, noté $s \rightsquigarrow u$. Considérons sur ce chemin le premier sommet y qui n'appartient pas à H (qui existe bien, car $s \in H$ et $u \notin H$) et x son prédeceseur. Le chemin se décompose en $s \rightsquigarrow x \to y \rightsquigarrow u$ (on peut avoir $s = x$ et/ou $y = u$). Puisque x est dans H, $\delta(s,x) = d[x]$ (hypothèse de récurrence) et lorsqu'on a considéré x dans la boucle, on a rajouté y à F s'il n'y était pas déja et relaché l'arc $x \to y$: ainsi $d[y] \leq \delta(s,x) + \omega(x,y) = \delta(s,y)$, donc il y a en fait égalité. Le morceau de s à y étant un chemin de plus petit poids de s à y et les poids positifs, on a $\delta(s,y) \leq \delta(s,u)$. Puisque $d[u]$ minimal parmi les sommets dans F, on a $d[u] \leq d[y]$. Ainsi :

$$d[u] \leq d[y] = \delta(s,y) \leq \delta(s,u) \leq d[u]$$

Il y a donc égalité partout, et en particulier $\delta(s,u) = d[u]$.

□

Remarque 12.36. *Le fait que les arcs aient un poids positif est un ingrédient essentiel de la preuve précédente, pour pouvoir affirmer que $\delta(s,y) \leq \delta(s,u)$. L'exemple minimaliste suivant montre un graphe pour lequel l'algorithme de Dijkstra lancé sur le sommet $s = 0$ ne fonctionne pas : l'algorithme relâche successivement les arcs $(0,1)$, $(1,3)$, $(0,2)$, $(2,1)$. Il faudrait relâcher à nouveau l'arc $(1,3)$ pour que le tableau d soit correct.*



Complexité de l'algorithme. Si on utilise un tableau de booléens pour marquer les éléments de F, retirer l'élément de F vérifiant $d[u]$ minimal a un coût $O(n)$. Cette action est effectuée au plus n fois pour un coût total $O(n^2)$. Ce coût majore le reste de l'algorithme, que le graphe soit implémenté via listes d'adjacence (le coût annexe est $O(a+n) = O(n^2)$ avec $a = |E|$) ou matrice d'adjacence (le coût est également $O(n^2)$). Ainsi l'algorithme de Dijkstra a un coût $O(n^2)$.

Remarque 12.37. *En utilisant une structure de donnée appelée* file de priorité *on peut accélérer l'algorithme, en particulier pour les graphes ayant peu d'arêtes et représentés par liste d'adjacence. Cette considération est hors programme !*

Calcul effectif de plus courts chemins. L'adaptation est la même que pour l'algorithme du parcours en largeur : il suffit d'utiliser une liste π de prédécesseurs. Lorsque l'on relâche un arc et que la valeur de $d[v]$ diminue (c'est-à-dire que l'on a $d[v] > d[u] + \omega(u,v)$), on réalise l'affectation $\pi[v] \leftarrow u$ conjointement à $d[v] \leftarrow d[u] + \omega(u,v)$.

Exemple. En figure 12.7 est représenté le déroulement de l'algorithme de Dijkstra sur un graphe à 5 sommets, dont la source s. Pour chaque sommet u on a fait figurer la valeur $d[u]$ à l'intétieur du cercle. Les arcs en gras représentent l'évolution de la liste des prédecesseurs.

Code Python. Voici une implémentation en Python de l'algorithme de Dijkstra. On suppose le graphe représenté par matrice d'adjacence. On utilise une fonction annexe permettant de trouver le sommet i vérifiant `d[i]` minimal parmi les sommets tels que `traites[i]` est faux (s'il existe un tel sommet vérifiant `d[i]` $< +\infty$, sinon -1 est renvoyé).

```
def cherche_min(d, traites):
    """ Renvoie le sommet i vérifiant d[i] minimal et traites[i] faux, s'il existe un tel sommet
          tel que d[i] != inf. Sinon, renvoie -1 """
    n=len(d)
    x=-1
    for i in range(n):
        if not traites[i] and d[i] != float('inf') and (x==-1 or d[x]>d[i]):
            x=i
    return x
```

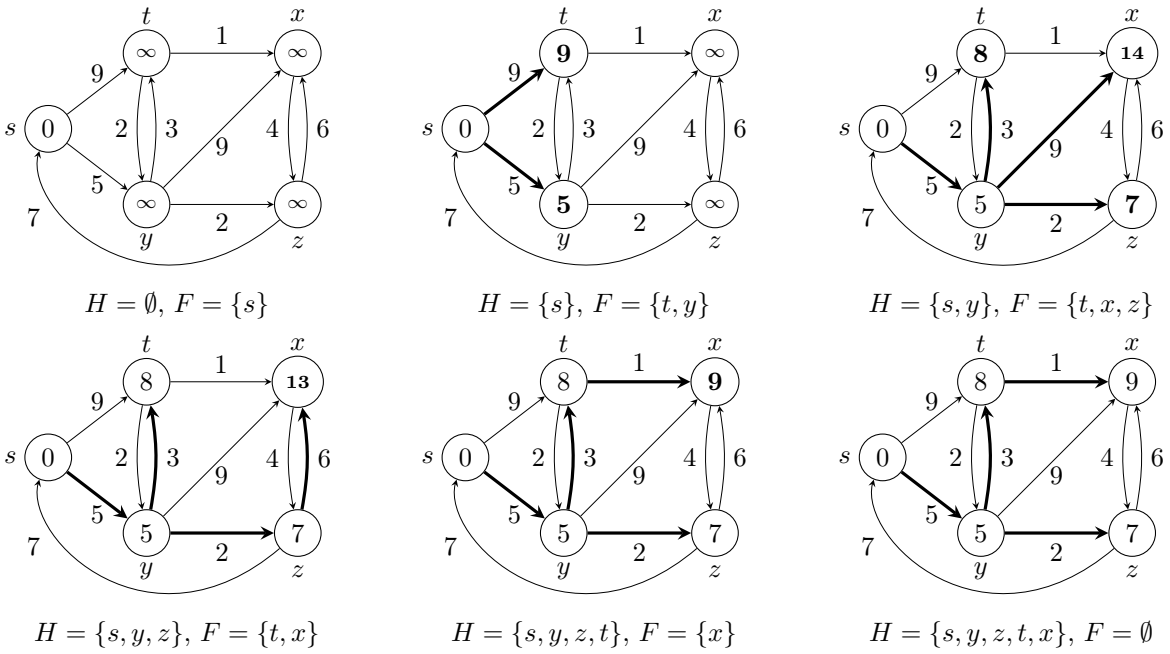


FIGURE 12.7 – Déroulement de l'algorithme de Dijkstra

Et voici l'algorithme de Dijkstra proprement dit.

```
def dijkstra_mat(G,s):
    """ G donné par matrice d'adjacence. Renvoie les poids chemins de plus petits poids depuis s. """
    n=len(G)
    d = [float('inf')]*n
    d[s]=0
    traites = [False]*n
    while True:
        x=cherche_min(d,traites)
        if x==-1:
            return d
        for i in range(n):
            d[i]=min(d[i], d[x]+G[x][i])
        traites[x]=True
```

12.8.2 Algorithme A*

Dans la pratique (par exemple, une recherche d'itinéraire via un outil comme Google Maps ou Waze), *tous* les poids depuis la source ne nous intéressent pas, on veut surtout le poids minimal entre deux sommets du graphe s et t (et un chemin de plus petit poids associé).

Pour optimiser l'algorithme de Dijkstra, on peut s'arrêter dès que le sommet t est traité dans la boucle principale du parcours, car on sait que $d[t]$ n'évoluera plus. Cette optimisation permet un gain de temps certain la plupart du temps, mais pas dans le pire cas (par exemple si t est le dernier sommet traité !)

Pour des graphes généraux, on ne peut pas tellement faire mieux dans le pire cas. Néanmoins dans la pratique, on peut tenter d'éviter des calculs inutiles. Par exemple, dans une recherche d'itinéraire le plus rapide entre Nice et Paris, on se doute qu'il est plutôt inutile d'effectuer des calculs vers Rome ou Barcelone (alors qu'ils sont plus proches en temps de trajet, donc seront examinés par l'algorithme de Dijkstra !)

L'algorithme A* essaie d'éviter des calculs inutiles via une *heuristique*, qui dans notre exemple privilégierait plutôt la direction nord-ouest et éviterait de considérer Rome ou Barcelone.

L'idée de l'algorithme A* est la suivante : on suppose connue une fonction f positive (l'heuristique), qui donne pour un nœud u une idée du poids du plus court chemin de u à t. Une telle heuristique est dite *admissible* si elle ne surestime jamais cette distance, c'est-à-dire que pour tout sommet u du graphe, on a $0 \leq f(u) \leq \delta(u, t)$. L'algorithme fonctionne alors comme celui de Dijkstra, sauf que dans la boucle principale, on retire de F un sommet v tel que $f(v) + d[v]$ est minimal (si f est la fonction identiquement nulle, on retrouve l'algorithme de Dijkstra). Voici deux exemples :

- Sur un graphe constitué de points du plan dont certains sont reliés par des segments, où l'on cherche à relier deux sommets s et t par un enchaînement le plus court possible, une fonction f possible serait la distance euclidienne entre le point considéré et la destination t : par inégalité triangulaire, f vérifie bien l'hypothèse $f(u) \leq \delta(u,t)$ pour tout sommet u du graphe et est donc admissible.
- Si on cherche à minimiser le temps de parcours en voiture, en supposant que l'on roule toujours à la vitesse maximale autorisée sur chacune des routes, une heuristique f possible serait la distance à vol d'oiseau divisée par la vitesse maximale sur autoroute (130 km $\cdot$ h^{-1}). Elle n'est admissible que si le conducteur respecte les limitations de vitesse !

Proposition 12.38. *L'algorithme obtenu en modifiant l'algorithme de Dijkstra pour que l'élément u extrait de F soit celui pour lequel $f(u) + d[u]$ est minimal calcule correctement la valeur $\delta(s,t)$, pourvu que f soit admissible.*

Démonstration. Raisonnons par l'asburde. On se donne un graphe pondéré $G = (V, E, \omega)$, deux sommets s et t, et une heuristique f vérifiant $f(u) \leq \delta(u,t)$ pour tout $u \in V$. Supposons que t soit accessible depuis s, et que lorsque t est sorti de F, on n'ait pas $d[t] = \delta(s,t)$, on a alors nécessairement $d[t] > \delta(s,t)$. Considérons un chemin de plus petit poids de s à t, il existe le long du chemin au moins un sommet v pour lequel $d[v] > \delta(s,v)$ (car t appartient au chemin). Notons v le premier sommet de ce chemin vérifiant cette propriété, et u son prédecesseur. u est donc dans F puisque l'arc $u \to v$ n'a pas encore été relâché, et u vérifie $d[u] = \delta(s,u)$ au moment où t est sorti de F. Mais à ce moment là :

$$f(t) + d[t] = d[t] > \delta(s,t) = \delta(s,u) + \delta(u,t) \geq \delta(s,u) + f(u) = d[u] + f(u)$$

Puisque $f(t) = 0$ et $f(u) \leq \delta(u,t)$. Ainsi t ne peut être extrait de F à ce moment là, puisque u est dans F et vérifie $d[u] + f(u) < d[t] + f(t)$. □

12.8.3 Un exemple d'utilisation de A* dans un cadre concret

Cet exemple se trouve sur la page Wikipedia [3] dédiée à l'algorithme A*. Considérons la grille infinie $\mathbb{Z}^2$, dans laquelle on peut se déplacer en une étape d'un sommet vers un de ces huit voisins (un cran horizontalement ou verticalement, ou en diagonale). Le coût d'un déplacement est celui de la distance parcourue (1 ou $\sqrt{2}$). Comment trouver un plus court chemin du point $(0,0)$ au point $(16,16)$, sachant qu'une zone constituée de deux rectangles est interdite au déplacement ? (voir figure 12.8, à gauche).

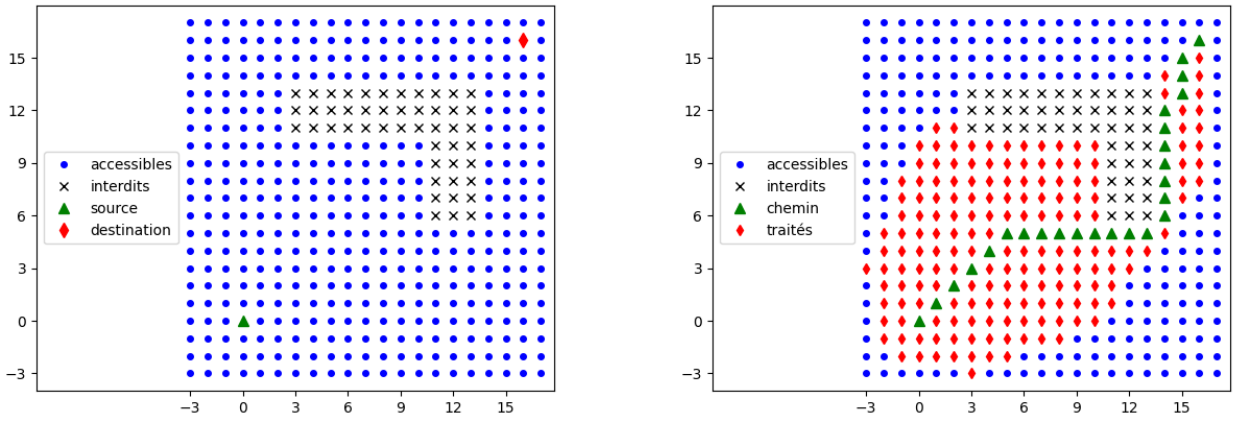


FIGURE 12.8 – À gauche, la grille du problème. À droite, le résultat fourni par A*.

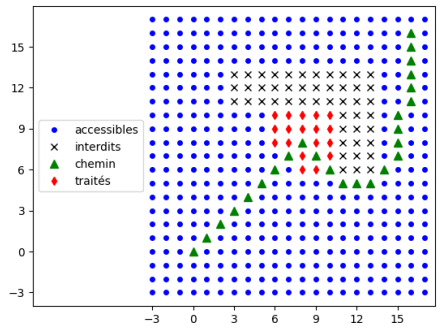
À droite de cette figure se trouve la même grille après application de l'algorithme A*, la fonction f associant simplement à un nœud sa distance euclidienne à la destination $(16,16)$. Les triangles donnent le chemin trouvé par l'algorithme, de longueur environ 27.31. On montre également les nœuds traités dans l'algorithme (qui sortent de l'ensemble F du pseudo-code), il y a 196 nœuds en comptant ceux du chemin, à ceux-ci s'ajoutent 95 sommets découverts mais qui auraient été traités après que la destination $(16,16)$ le soit.

L'algorithme de Dijkstra (ou l'algorithme A* avec une fonction f identiquement nulle), traite 2027 sommets (et en découvre 231 supplémentaires), soit environ 10 fois plus de sommets. Le chemin fourni par l'algorithme de Dijkstra a bien sûr la même longueur que celui trouvé par l'algorithme A*.

3. `https://fr.wikipedia.org/wiki/Algorithme_A*`

Remarque 12.39.

- *Si on se donne une fonction f non admissible, on trouvera en général un chemin encore plus rapidement, mais non optimal. Voici les sommets traités et le chemin trouvé par l'algorithme A^* dans l'exemple précédent lorsque la fonction f vaut 5 fois la distance euclidienne entre le nœud et la destination :*



Ce chemin est de longueur environ 29.8, un peu plus que le chemin optimal. Mais l'algorithme traite beaucoup moins de sommets !

- *Dans un graphe du même style mais avec beaucoup plus d'obstacles (comme un labyrinthe), l'algorithme ne sera pas significativement plus performant que l'algorithme de Dijkstra, puisqu'un chemin optimal peut avoir des directions opposées à celle favorisée par la fonction f.*

Cinquième partie

Bases de données

Chapitre 13

Requêtes sur une base de données à une table

13.1 Introduction : limite des structures de données plates pour la recherche d'informations

On souhaite représenter l'ensemble des élèves de deuxième année des classes préparatoires du lycée Masséna, sachant qu'ils sont regroupés par classes. Les élèves ont certains attributs :

— leur nom ;

— leur lycée d'origine en terminale ;

— la filière (MP, PC, PSI) suivie ;

— le numéro éventuel de la classe s'il y en a plusieurs (1 ou 2).

Prenons pour simplifier un ensemble d'élèves réduit :

Prénom	Nom	Filière	Numéro	Lycée d'origine
Mathilde	Dufour	PC	2	Calmette
Léa	Dupond	MP	2	Massena
Paul	Dugommier	PC	1	Massena
Mathilde	Dugommier	MP	1	Calmette
Clément	Durand	PC	1	Parc Imperial

On peut, en Python, choisir plusieurs représentations de ces données. On pourrait grouper les élèves par classe, puis par filière :

```
classes_de_spe=[
[
[['Mathilde', 'Dugommier', 'Calmette']],
[['Léa', 'Dupond', 'Massena']]
], [
[['Paul', 'Dugommier', 'Massena'], ['Clément', 'Durand', 'Parc Imperial']],
[['Mathilde', 'Dufour', 'Calmette']]
]
]
```

Ici, `classes_de_spe[1]` fournit par exemple les deux classes de PC, alors que `classes_de_spe[0][0]` donne la liste des élèves de MP 1. Dans les deux cas le lycée d'origine est présent. Si on veut extraire simplement la liste des élèves d'une classe (sans le lycée d'origine), on ferait quelque chose comme

```
>>> [x[:2] for x in classes_de_spe[1][0]]
[['Paul', 'Dugommier'], ['Clément', 'Durand']]
```

Par contre, si on veut la liste des élèves venant du lycée Calmette, il faut un peu plus travailler :

```
lycee_calmette=[]
for filiere in classes_de_spe:
    for classe in filiere:
        for eleve in classe:
            if eleve[2]=='Calmette':
                lycee_calmette.append(eleve[:2])
```

On obtient bien :

```
>>> lycee_calmette
[['Mathilde', 'Dugommier'], ['Mathilde', 'Dufour']]
```

On pourrait aussi regrouper les élèves par lycée d'origine, par contre la liste des élèves de MP 1 (par exemple) serait plus dure à établir. Il est aussi possible de stocker des 5-uplets comme dans le tableau ci-dessus, au prix par contre d'un stockage important de données (il faut imaginer qu'il y a beaucoup plus d'élèves...)

13.2 Présentation succincte des bases de données

13.2.1 Rôle des bases de données

Le rôle des bases de données est de simplifier le genre de considérations de la section précédente, en permettant :

— d'avoir une structure de données efficace ;
— une rapidité d'accès ;
— d'éviter à l'utilisateur d'avoir à s'interroger sur la manière dont sont stockées les données ;
— une sauvegarde des modifications ;
— une gestion des pannes ;
— une gestion des conflits si plusieurs utilisateurs modifient la base en même temps ;
— ...

13.2.2 Un exemple avec quelques requêtes

Par exemple, pour les élèves des différentes classes, on pourrait imaginer un système avec trois tables du type :

Table eleve

prenom	nom	id_classe	id_lycee
Mathilde	Dufour	934	1
Léa	Dupond	932	2
Paul	Dugommier	933	2
Mathilde	Dugommier	931	1
Clément	Durand	933	3

Table lycee

id_lycee	nom
1	Calmette
2	Massena
3	Parc Imperial

Table classe

id_classe	filiere	numero
931	MP	1
932	MP	2
933	PC	1
934	PC	2

La table des lycées ne possède que deux attributs (colonnes), mais on pourrait imaginer vouloir rajouter d'autres attributs (ville, nombre de classes de terminale...), on n'aurait pas à modifier toute la base. On remarque que notre découpage d'informations permet de limiter la redondance : on n'indique pour chaque élève que son numéro de classe, et pas sa filière et son numéro, informations qu'on peut retrouver facilement à partir du numéro de classe. Voici des exemples de requêtes qu'on peut réaliser sur ces tables :

sélectionner les élèves de MP 1, et les classer par ordre alphabétique

```
SELECT nom,prenom
FROM eleves
WHERE id_classe = 931
ORDER BY nom
```

Ici, on a supposé que l'on connaissait le code de la MP 1, si ce n'est pas le cas, il faut croiser les tables :

sélectionner les élèves de PC 1, et les classer par ordre alphabétique, version 2
```
SELECT nom, prenom
FROM eleves
JOIN classe ON eleves.id_classe=classe.id_classe
WHERE filiere="PC" AND numero=1
ORDER BY nom ;
```

De même, pour sélectionner les élèves venant de Calmette :

```
SELECT nom, prenom
FROM eleve
JOIN lycee ON eleve.id_lycee=lycee.id_lycee
WHERE lycee.nom="Calmette" ;
```

Pour connaître le nombre d'élèves en MP :

```
SELECT COUNT (*)
FROM eleve
JOIN classe ON eleve.id_classe=classes.id_classe
WHERE filiere="MP" ;
```

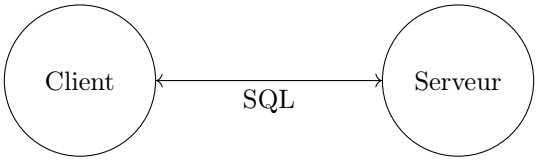
Le programme officiel impose uniquement l'enseignement des requêtes de recherche dans une base de données. Voici toutefois un exemple d'insertion dans une base :

```
INSERT INTO eleve (nom, prenom, id_classe)
VALUES (Ducourneau, Guillaume, 933) ;
```

(Il n'est pas obligatoire de renseigner tous les champs, mais ceci sort assez largement du programme).

13.2.3 Architecture client-serveur

Les bases de données sont articulées sous la forme client-serveur : l'utilisateur travaille sur un poste (client) qui peut être éloigné de l'ordinateur qui gère les données (serveur). Il est nécessaire de permettre un dialogue entre ces deux parties.



Une normalisation (rare en informatique) a permis d'unifier le langage utilisé dans les bases de données : SQL. Du point de vue des utilisateurs, la syntaxe est la même, en tout cas pour les fonctionnalités de base. Par contre, la programmation en interne de ces logiciels dépend de l'éditeur (Oracle, SAP, IBM, Microsoft, ...), mais cela nous préoccupera assez peu.

Dans ce cours, nous allons étudier les principales fonctions en commençant par les bases simples (ou plates) puis en étudiant ce qui fait l'intérêt des bases, le croisement de données. Essentiellement ici, on va voir comment interroger une base de données, sans en créer ni en modifier (ce n'est pas au programme).

En pratique, le langage SQL n'est en général pas visible pour les usagers. En effet, l'utilisation des bases de données se fait usuellement à travers une architecture « trois-tiers ». Entre l'usager et la base de données se trouve un serveur applicatif qui traduit les demandes de l'utilisateur (en général via une interface graphique) au gestionnaire de bases de données.

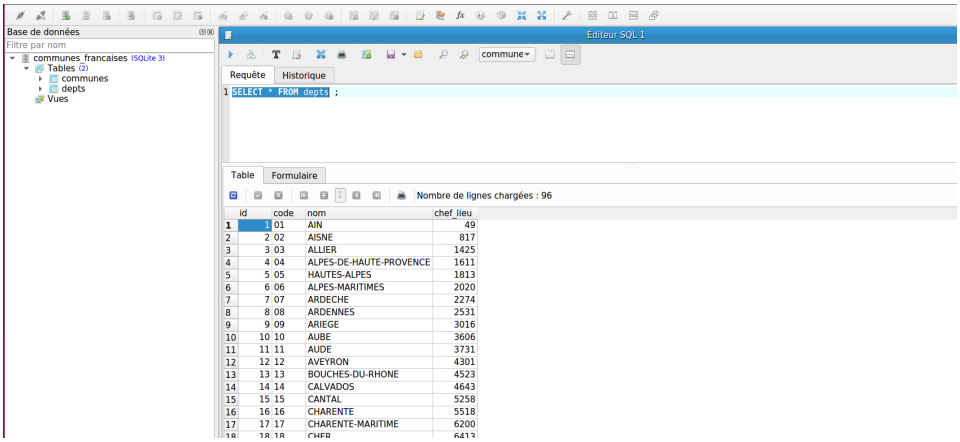
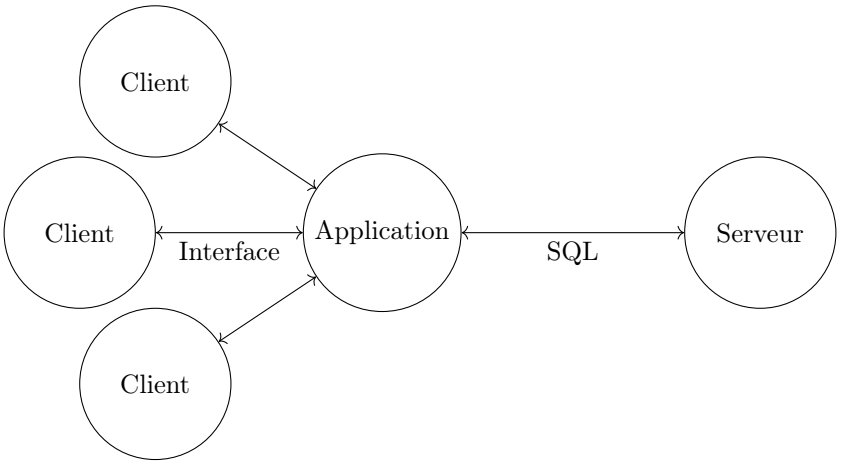


FIGURE 13.1 – La base de données utilisée en TP



Par exemple, lorsque l'on se connecte à Pronote, tout le côté « bases de données » est invisible pour l'utilisateur, seule l'application Pronote est disponible. C'est elle qui gère les droits d'accès : un élève ne peut consulter les notes d'un autre, tandis qu'un professeur de mathématiques ne peut rentrer des notes de physique-chimie...

13.2.4 Logiciels

En TP, nous utiliserons SqliteStudio [1] (figure 13.1) pour écrire nos requêtes. Ici, le client et la base seront sur le même poste !

13.2.5 Abstraction des bases de données

L'algèbre relationnelle, théorie inventée en 1970, est une théorie mathématique, proche de la théorie des ensembles, qui définit les opérations pouvant être effectuées sur des relations (ensemble de n-uplets). C'est cette théorie qui est le cœur des logiciels de base de données [2], bien qu'elle n'en soit qu'une abstraction. L'algèbre relationnelle est hors programme, mais on retrouve souvent le vocabulaire de l'algèbre relationnelle lorsqu'on traite de SQL, et les notations de l'algèbre relationnelle sont commodes pour la théorie. On indiquera le vocabulaire qui est plus propre au SQL et celui plus relatif à l'algèbre relationnelle.

13.3 Vocabulaire des bases de données

13.3.1 Modélisation en tableau

Le modèle utilisé est très simple : c'est celui d'un tableau à deux dimensions. Celui-ci se recontre souvent lorsque l'on traite de données, par exemple :

1. Téléchargeable ici : `https://sqlitestudio.pl/`
2. Un modèle équivalent est le calcul relationnel, que nous n'étudierons pas

— un répertoire (nom, téléphone, adresse, ...)
— une fiche de bibliothèque (auteur, titre, année,...)
— un carnet de commande (client, article, quantité, date, prix,...)

On représente ces données dans un tableau, par exemple dans un tableur. Par convention, dans ce cours, les données d'un élément (une fiche de bibliothèque, une commande, un individu du répertoire) seront sur une même ligne, les colonnes donnant les différents attributs (auteur, titre, année, par exemple). Voici un exemple dont on se servira tout au long de ce cours.

Table eleve					
prenom	nom	filiere	numero	lycee_origine	note_bac
Mathilde	Dufour	PC	2	Calmette	18
Léa	Dupond	MP	2	Massena	14
Paul	Dugommier	PC	1	Massena	12
Mathilde	Dugommier	MP	1	Calmette	15
Clément	Durand	PC	1	Parc Imperial	13

13.3.2 Vocabulaire

Colonnes et Attributs. On parle de *colonnes* en SQL et d'*attributs* en algèbre relationnelle. On notera formellement $A_1, A_2, \cdots, A_p$ les attributs. Les attributs forment un ensemble : *il n'y a pas d'attribut en double.* L'ordre des attributs n'est pas fixé : on ne parle pas du premier attribut mais de l'attribut nom (par exemple). La table précédente présente 6 attributs.

Domaine. L'ensemble des valeurs que peut prendre un attribut A est son domaine $\mathrm{Dom}(A)$. Par exemple, pour la table ci-dessus, quatre attributs peuvent avoir pour domaine des chaînes de caractères, le numéro est un entier, et la note du bac un flottant. Tout cela a été fixé à la création de la base, et ne va donc pas nous concerner dans nos requêtes de recherche. Que ce soit pour les chaînes de caractères, les entiers ou les flottants, il existe des sous-types, par exemple :

— `varchar(255)` pour des chaînes de caractères limitées à 255 caractères ;
— `int8` pour des entiers sur 8 octets ;
— `float8` pour des flottants sur 8 octets ;
— les dates sont représentées sous forme de chaîne de caractères, de la forme `"AAAA-MM-JJ"`, qu'on peut compléter avec une heure de la forme `"HH:MM:SS"`.

Tous ces sous-types sont hors programme, on se contentera de savoir que le type des attributs est fixé à la création, et qu'il est possible notamment d'avoir des attributs de type entier / flottant / chaîne de caractères. On peut imposer des conditions supplémentaires, comme le fait qu'une note soit entre 0 et 20, à la création de la table.

Champ NULL. Dans la pratique, tous les attributs ne sont pas nécessairement renseignés. Par exemple, lorsqu'on s'inscrit sur un site web, on ne renseigne pas systématiquement son numéro de téléphone ou son adresse. Dans une base de données, cela se traduit par un champ NULL (vide). Nous n'en parlerons pas plus, tout ce qui concerne la valeur NULL est hors-programme.

Schéma relationnel. On appelle schéma relationnel un p-uplet d'attributs, vérifiant toujours la contrainte que les attributs sont distincts deux à deux. On notera $\mathcal{S}$ dans ce cours un schéma relationnel.

Enregistrement. Chaque ligne s'appelle un *enregistrement.* C'est donc un élément de $\mathrm{Dom}(A_1) \times \mathrm{Dom}(A_2) \times \cdots \times \mathrm{Dom}(A_p)$. En fait, puisque l'ordre des attributs n'est pas fixé, une ligne est plutôt une fonction

$$\ell : \quad \{A_1, \ldots, A_p\} \longrightarrow \mathrm{Dom}(A_1) \cup \mathrm{Dom}(A_2) \cup \cdots \cup \mathrm{Dom}(A_p)$$

avec comme contrainte le fait que $\ell(A_i) \in \mathrm{Dom}(A_i)$. Comme il est plus facile de parler de la ligne (Mathilde, Dufour, PC, 2, Calmette, 18) que de l'application qui a chacun des attributs de la table associe sa valeur, on sacrifiera un peu à la rigueur mathématique ici.

Table ou Relation. On appelle *relation* ou *table* un ensemble fini d'enregistrements de $\mathrm{Dom}(A_1) \times \cdots \times \mathrm{Dom}(A_p)$, qu'on notera souvent $\mathcal{R}$ dans ce cours. Pour préciser que la relation est associée au schéma relationnel S, on notera $\mathcal{R}(S)$.

13.3.3 Contraintes

On a déja dit qu'il n'y avait pas d'attribut en double. Il n'en est en effet pas question, ils risqueraient d'être affectés de valeurs différentes dans des tuples. Pour t un tuple de $\mathcal{R}(\mathcal{S})$, on note $t[A_i]$ la composante du couple associée à l'attribut A_i. Par extension, si $X = (B_1, \dots, B_n) \subseteq \mathcal{S}$, on note $t[X]$ le n-uplet $(t[B_1], \dots, t[B_n])$

Définition 13.1. *Dans une relation $\mathcal{R}(\mathcal{S})$, les enregistrements forment un ensemble.*

Ceci a deux implications :

— Visuellement, on ne peut avoir deux lignes égales dans le tableau (ce qui serait d'ailleurs synonyme de redondance des données). Ceci correspond à l'aspect fonctionnel des tuples : avec $S = \{A_1, \dots, A_p\}$, si $t[A_1, \dots, A_p] = t'[A_1, \dots, A_p]$ alors $t = t'$. Une autre caractérisation est que deux enregistrements doivent différer d'au moins un attribut.

— L'ordre des lignes n'est pas fixé. Bien entendu une présentation des données dans un tableau aura un ordre des attributs et un ordre des tuple mais ces ordres de présentations sont du à l'ordre physique des données, ou le résultat d'un traitement particulier lors d'une requête. En particulier, il n'est pas garanti *a priori* par le SGBD.

13.3.4 Clés primaires

Pour garantir la non-répétition des enregistrement, les bases de données réelles contiennent un concept de *clé* qui doit être pensé dès la conception des bases de données, et indiqué à la création. Ce concept a également son imporance lors de requêtes concernant plusieurs relations, comme on le verra par la suite.

Définition 13.2. *La clé primaire d'une relation $\mathcal{R}$ est un ensemble X d'attributs tel que*

$$\forall t, t' \in R \qquad t[X] = t'[X] \Longrightarrow (t = t')$$

Autrement dit, si deux enregistrements sont égaux sur les attributs de l'ensemble X alors ils sont égaux partout. En raison de la contrainte d'ensemble pour les enregistrements, l'ensemble de tous les attributs pourrait toujours être une clé primaire. Néanmoins lorsqu'on crée une table, on a intérêt à prendre la clé primaire comme étant la plus petite possible, au sens de l'inclusion.

Exemple 13.3. *À Masséna, le numéro d'une classe (931, 933...) dans la table* `Classes` *est une clé primaire (lorsqu'on veut préciser la clé primaire, on souligne le ou les attributs qui en font partie).*

Table Classes		
classe	*filière*	*numéro*
931	*MP*	*1*
932	*MP*	*2*
933	*PC*	*1*
934	*PC*	*2*

Dans la pratique on évitera les clés primaires qui ne seraient primaires que « par accident » : on pourrait insérer des données supplémentaires qui feraient perdre le caractère primaire des clés (en pratique, le serveur interdira le rajout de ces données). Par exemple, dans la table Éleves, le couple {nom,filière} caractérise pour le moment chaque enregistrement, mais rien ne garantit que cette propriété soit conservée par des ajouts successifs dans la table. En particulier, déclarer ce couple comme clé primaire à la création interdit l'insertion de jumeaux.

On introduit souvent un attribut supplémentaire (commençant souvent par id, comme *index*), dans la définition de la relation qui sera un entier qui sera auto-incrémenté à chaque ajout d'un tuple (ceci est indiqué à la création de la base). Ce sera la clé primaire. Voici une modification de la relation eleve avec ce principe :

Table eleve'						
id_eleve	prenom	nom	filiere	numero	lycee_origine	note_bac
1	Mathilde	Dufour	PC	2	Calmette	18
2	Léa	Dupond	MP	2	Massena	14
3	Paul	Dugommier	PC	1	Massena	12
4	Mathilde	Dugommier	MP	1	Calmette	15
5	Clément	Durand	PC	1	Parc Imperial	13

Les clés primaires seront souvent formées d'un seul attribut, mais pas systématiquement. Par exemple, dans une table qui résume pour chque élève les cours qu'il suit, une clé primaire constituée du couple (numéro de l'élève, numéro du cours suivi) convient très bien.

13.4 Requêtes en SQL sur une seule table : sélection, projection et renommage

13.4.1 Introduction

Pour la communication entre le serveur applicatif et le serveur de bases de données, il s'est produit un événement rare en informatique : une norme universelle a été établie, qui permet d'écrire des requêtes de la même façon quel que soit le logiciel. Le langage utilisé est SQL, pour *Structured Query Langage*.

Bien entendu chaque éditeur optimise le traitement des questions posées en SQL pour donner des réponses le plus rapidement possible, et ajoute des fonctionnalités supplémentaires mais presque tous contiennent le langage SQL normalisé, que l'on va étudier dans la suite.

Au programme ne figurent que les requêtes de *recherche* dans une base de données déja créée, on ne verra donc pas les requêtes de création, suppression ou modification d'une base de données.

13.4.2 Syntaxe

La forme générale d'une requête de recherche en SQL sur une seule table est de la forme

```
SELECT ... FROM table ... ;
```

Les mots-clés de SQL sont usuellement écrits en majuscule mais ce n'est pas obligatoire. Le mot-clef principal d'une requête de recherche dans une base de données est `SELECT`, qu'on retrouvera en tête de toutes nos requêtes SQL. Le mot-clef `FROM` permet de spécifier le nom de la table à utiliser. Les requêtes se terminent par un point-virgule.

13.4.3 Projection

Définition 13.4. *Soit $\mathcal{R}$ une relation de schéma $\mathcal{S}$. Soit X un sous-ensemble non vide de $\mathcal{S}$. La projection de $\mathcal{R}$ sur X est la relation :*

$$\pi_X(\mathcal{R}) = \{t[X] \mid t \in \mathcal{R}\}$$

Autrement dit, pour projeter, on oublie les attributs qui ne sont pas dans l'ensemble X. Pour projeter en SQL, il suffit d'indiquer après le `SELECT` les attributs que l'on veut garder, séparés par des virgules :

```
SELECT A1,...,Ap FROM table ;
```

Si on ne veut pas effectuer de projection (c'est à dire garder tous les attributs), on peut utiliser le joker `*` au lieu d'énumérer tous les attributs un à un.

```
SELECT * FROM table ;
```

En algèbre relationnelle, le résultat d'un opérateur appliqué à une relation est une relation (en particulier, c'est un ensemble). En SQL, les doublons résultant de projections ne sont pas supprimés, il faut utiliser le mot clef `DISTINCT` pour supprimer les doublons.

```
>>> SELECT prenom FROM eleve ;
"Mathilde"
"Léa"
"Paul"
"Mathilde"
"Clément"

>>> SELECT DISTINCT prenom FROM eleve ;
"Mathilde"
"Léa"
"Paul"
"Clément"
```

13.4.4 Sélection

Définition 13.5. *Soit $\mathcal{R}$ une relation de schéma $\mathcal{S}$. Soit $\mathcal{C}$ une condition sur les enregistrements de la relation. La sélection de $\mathcal{R}$ suivant la condition $\mathcal{C}$ est la relation*

$$\sigma_{\mathcal{C}}(\mathcal{R}) = \{t \in \mathcal{R} \mid \mathcal{C}(t)\}$$

Autrement dit, on ne garde que les enregistrements vérifiant la condition. En SQL, la sélection se fait avec le mot-clef `WHERE`, placé après le nom de la table. Par exemple, la requête :

```
SELECT * FROM eleve WHERE lycee = 'Calmette' ;
```

extrait de la table `eleve` les éléves provenant du lycée Calmette. Comme en Python, on utilise les comparateurs `=` et `!=` pour l'égalité et la différence. Si le domaine de l'attribut le permet, on peut utiliser d'autres comparateurs (`>, <, >=, <=`) et même des fonctions arithmétiques. Une condition complexe peut être exprimée à l'aide de conditions plus simples et des des connecteurs logiques ET, OU, et NON (en anglais en SQL : `AND`, `OR`, `NOT`).

13.4.5 Sélection et projection

En pratique, sélection et projection sont souvent utilisées ensemble, sous la forme :

```
SELECT attributs FROM table WHERE condition ;
```

Attention : en pratique, la sélection est effectuée avant la projection : on ne garde que les enregistrements satisfaisant la condition, puis on projette. Par exemple, la requête suivante extrait les prénoms des élèves ayant eu au moins 17 au bac (sans éliminer les doublons éventuels).

```
SELECT prenom FROM eleve WHERE note_bac >= 17 ;
```

13.4.6 Renommage

Le renommage nous servira surtout lorsqu'on veut écourter les requêtes sur plusieurs tables, ce qu'on verra par la suite. En SQL, on peut renommer un attribut ou une table de la même manière.

Renommage d'attribut. On peut renommer un ou plusieurs attributs avec AS ou même en juxtaposant le nouveau nom à droite de l'ancien. La syntaxe générale est :

```
SELECT A1 AS B1,..,Ai AS Bi, C1,..,Cj FROM table ;
```

Le `AS` est facultatif, juxtaposer simplement le nouveau nom à droite de l'ancien est possible. Par exemple la requête

```
SELECT prenom p, nom n, notebac/2 note_sur_10 FROM eleve ;
```

produit la table suivante :

Table avec requête

p	n	note_sur_10
Mathilde	Dufour	9
Léa	Dupond	7
Paul	Dugommier	6
Mathilde	Dugommier	7
Clément	Durand	6

Renommage de tables. On travaillera parfois avec plusieurs tables où certains atttibuts ont les mêmes noms. Prenons l'exemple de deux tables `t1` et `t2` possédant toutes deux un attribut de nom `x`. Le logiciel de bases de données refusera toute requête impliquant `x`, puisqu'il y a ambiguïté et on ne peut savoir à quelle table on fait référence. Pour lever l'ambiguïté, il faudra utiliser `t1.x` ou `t2.x` (en général table.attribut) pour pouvoir effectuer une requête concernant l'attribut `x` d'une des tables. Pour écourter un peu cette syntaxe (ce qui est intéressant si les noms des tables sont longs), on peut renommer directement les tables de la même manière que les attributs, comme suit :

```
SELECT ... FROM nom_de_lancienne_table AS nom_de_la_nouvelle_table ...
```

On choisira un nom court pour la nouvelle table ! De même, le `AS` est facultatif.

13.5 Agrégats et fonctions d'agrégation

Lorsqu'on travaille avec des bases de données, on peut vouloir effectuer des opérations comme sur un tableur, et répondre à des questions comme les suivantes :

— combien y a-t-il de lignes dans la table ?

— quel est le prix maximal d'un objet ?

— quel acteur a joué dans le plus grand nombre de films ?

— combien y a-t-il d'élèves par classe en moyenne ?

On utilise pour cela des *fonctions d'agrégation*.

13.5.1 Fonction d'agrégation

Une fonction d'agrégation est essentiellement une fonction qui peut s'appliquer à un nombre d'arguments quelconque. On va voir une définition plus formelle.

Définition 13.6. *Soit $f : E^n \to F$. On dit que f est symétrique si pour tout n-uplet $(x_1, \dots, x_n)$ de E, tout $i \neq j$, on a $f(x_1, \dots, x_{i-1}, x_i, x_{i+1}, \cdots, x_{j-1}, x_j, x_{j+1}, \dots, x_n) = f(x_1, \dots, x_{i-1}, x_j, x_{i+1}, \cdots, x_{j-1}, x_i, x_{j+1}, \dots, x_n)$.*

Puisque les transpositions engendrent le groupe symétrique, il est équivalent de dire que le résultat d'une fonction symétrique ne change pas lorsqu'on permute les arguments.

Définition 13.7. *On appelle fonction d'agrégation $E \to F$ une suite $(f_n)_{n\geq 1}$ de fonctions symétriques de $E^n \to F$.*

Cette définition peut paraître étrange, mais c'est ce qui permet d'appliquer une fonction à un nombre quelconque d'arguments : on applique simplement la fonction f_n avec n le nombre d'arguments. Voici des exemples de fonctions d'agrégation :

— la somme, on a $f_n(x_1, \dots, x_n) = \sum_{i=1}^{n} x_i$. Elle est même définie avec aucun argument, en convenant que f_0 est la fonction nulle.

— de même, le produit, le max, le min et la moyenne sont des fonctions d'agrégation ;

— le cardinal, dans ce cas f_n est simplement la fonction constante égale à n (et est définie pour $n = 0$, avec $f_0 = 0$).

— le « et logique » où E est simplement l'ensemble $\{\text{Vrai}, \text{Faux}\}$, avec $f(x_1, \dots, x_n) = x_1$ ET $\cdots$ ET x_n.

13.5.2 Agrégats

On peut utiliser de telles fonctions en regroupant les données par paquets pour obtenir une nouvelle table. Formellement, si f est une fonction d'agrégation, X est un ensemble d'attributs d'une relation $\mathcal{R}$ de schéma S, et $A \in S$ un attribut n'appartenant pas à X on note, pour chaque tuple t_0 de valeurs de X, $f(t_0, A)$ la valeur de la fonction d'agrégation f appliquée à l'attribut A des tuples de la relation $\sigma_{t[X]=t_0}(\mathcal{R})$ c'est-à-dire appliquée à l'ensemble des tuples qui prennent les valeurs de t_0 pour les attributs de X.

On introduit une nouvelle opération, l'agrégation notée ${}_X\gamma_{f(A)}$. L'image de $\mathcal{R}$ par cette opération est une relation de schéma $X \cup \{f(A)\}$.

$$ {}_X\gamma_{f(A)}(\mathcal{R}) = \{t; \exists s \in \mathcal{R}, t[X] = s[X], t[f(A)] = f(s[X], A)\} $$

En pratique, on sépare les éléments de la relation $\mathcal{R}$ en paquets pour lesquels les valeurs sur les attributs de X sont les mêmes, et on calcule $f(A)$ sur chacun des paquets. Plus il y a d'attributs dans l'ensemble X, plus il y aura de paquets. Dans le cas extrême où X est vide, il n'y a qu'un seul paquet.
Par exemple :

— ${}_{filiere}\gamma_{MOYENNE(notebac)}(eleve)$ est une relation à deux attributs, donnant la moyenne des notes au bac suivant la filière.

— ${}_{\emptyset}\gamma_{MOYENNE(notebac)}(eleve)$ est une relation avec un unique tuple de taille 1, donnant la moyenne des notes au bac de tous les élèves de la table. On aura tendance à ne pas noter l'ensemble vide $\emptyset$. Remarquez qu'un tel tuple s'identifie avec un élément du domaine de son unique attribut.

13.5.3 SQL

Fonctions d'agrégation en SQL. Les fonctions disponibles (de base) sont, parmi d'autres :

- le comptage, c'est-à-dire le nombre de lignes : COUNT
- le maximum des éléments dans une colonne : MAX
- le minimum des éléments dans une colonne : MIN
- la somme des éléments d'une colonne : SUM
- la moyenne des éléments d'une colonne (sum/count) : AVG

Agrégation sans regroupement. Une requête SQL avec une simple fonction d'agrégation permet d'obtenir une table avec une unique ligne et une unique colonne (que l'on peut identifier à sa valeur). Si f est une de ces fonctions on l'emploie sous la forme

```
SELECT f(attribut) FROM table ;
```

Par exemple, avec la relation eleve, `SELECT AVG(notebac) FROM eleve` produit 14.4.

Regroupement. La syntaxe précédente correspond donc à un ensemble X vide : il n'y a qu'un seul paquet. Le mot-clef GROUP BY sert à indiquer sur quels attributs sont effectués les regroupements (les éléments de l'ensemble X précédent). La table ${}_X\gamma_{f(A)}(\mathcal{R})$ évoquée précédemment s'obtient via la requête :

```
SELECT A1, A2,..,Ap, f(A) FROM table GROUP BY A1, A2,...,Ap ;
```

Par exemple `SELECT filiere,AVG(notebac) FROM eleves GROUP BY filiere ;` produit :

filiere	AVG(notebac)
MP	14.5
PC	14.333333333333334

Pour la moyenne sur chaque classe, on écrirait `GROUP BY filiere, numero`.

13.5.4 Sélections et agrégation

Il est bien sûr possible de composer sélection et agrégation : par exemple ne garder que certaines lignes avant de réaliser l'agrégation, et/ou ne garder que certains résultats une fois l'agrégation effectuée.

L'agèbre relationnelle ne distingue pas une sélection faite avant une agrégation d'une sélection faite après une agrégation : l'opérateur est toujours σ. Toutefois en SQL la syntaxe est différente. Le mot-clé `WHERE` correspond à une sélection avant l'agrégation, et le mot-clé `HAVING` correspond à une sélection après agrégation. Par exemple, à la suite de la requête précédente, on pourrait vouloir garder uniquement les filières ayant une moyenne supérieure à 14.4. Cela correspond en algèbre relationnelle à la composition $\sigma_{MOYENNE(notebac)>14.4} \circ {}_{filiere}\gamma_{MOYENNE(notebac)}(eleve)$. La syntaxe correspondante est la suivante :

```
SELECT filiere,AVG(notebac) FROM eleves GROUP BY filiere HAVING AVG(notebac)>14.4 ;
```

qui produit :

filiere	AVG(notebac)
MP	14.5

Il est ici très commode de procéder à un renommage : garder un attribut qui s'appelle `AVG(notebac)` est un peu pénible.

```
SELECT filiere,AVG(notebac) AS moy FROM eleves GROUP BY filiere HAVING moy>14.4 ;
```

qui produit :

filiere	moy
MP	14.5

Rappellons qu'en SQL, le mot-clé AS est facultatif, on aurait pu juxtaposer `moy` à `AVG(notebac)`. Pour faire des sélections, on a donc deux outils à notre disposition : WHERE et HAVING. WHERE sélectionne avant une agrégation (on dit en amont), et HAVING en aval.

13.5.5 WHERE ou HAVING ?

Il ne faut pas confondre WHERE et HAVING.

— s'il n'y a pas de fonction d'agrégation dans la requête, une utilisation de HAVING sera rejetée par le logiciel de bases de données : seul WHERE est pertinent ;

— la condition dans le WHERE excluera des lignes avant de réaliser l'agrégation. Par exemple pour compter le nombre de lignes vérifiant une certaine propriété, WHERE est tout indiqué.

— inversement, HAVING est une sélection après l'agrégation. En algèbre relationnelle, la condition de sélection ne peut porter que sur les attributs de $X \cup \{f(A)\}$.

On a parfois le choix : par exemple, si au lycée Masséna, on veut calculer les moyennes au bac des classes de MP, on pourrait :

— calculer les moyennes au bac de toutes les classes, puis ne garder (avec HAVING) que les classes de MP, comme ceci :

```
SELECT numero, AVG(notebac) FROM eleves GROUP BY filiere, numero HAVING filiere = 'MP' ;
```

— calculer les moyennes au bac des élèves qui sont en MP seulement, regroupés par classe :

```
SELECT numero, AVG(notebac) FROM eleves WHERE filiere = 'MP' GROUP BY numero ;
```

Lorsqu'on a le choix, il vaut mieux sélectionner en amont, pour n'effectuer l'agrégation que sur un nombre minimal de lignes : la deuxième solution est préférable.

13.6 Affichage des résultats

En SQL, on peut ordonner les résultats suivant une certaine quantité, et ne garder que certaines lignes. Les mots-clés suivants viennent *après* les mots clés vus précédemment.

13.6.1 Ordonner les résultats avec ORDER BY

La syntaxe en SQL pour ordonner les résultats se fait à l'aide du mot-clé ORDER BY, couplé à une expression arithmétique des attributs. L'expression sera souvent un attribut lui-même :

```
SELECT * FROM eleve ORDER BY notebac ;
```

produit à l'affichage :

prenom	nom	filiere	numero	lycee_origine	note_bac
Paul	Dugommier	PC	1	Massena	12
Clément	Durand	PC	1	Parc Imperial	13
Léa	Dupond	MP	2	Massena	14
Mathilde	Dugommier	MP	1	Calmette	15
Mathilde	Dufour	PC	2	Calmette	18

Deux petites subtilités :

— on peut spécifier si l'on veut le résultat dans l'ordre croissant ou décroissant à l'aide des mots clés `ASC` et `DESC` (pour ascendant et descendant) juste après l'expression. Comme on le voit sur l'exemple précédent, le comportement par défaut est `ASC`.

— si on met plusieurs expressions après ORDER BY, séparés par des virgules (typiquement plusieurs arguments), la relation est triée pour l'ordre lexicographique (on compare la première expression, puis en cas d'égalité la deuxième, etc...)

Voici un exemple récapitulant un peu tout ça : des points du plan à coordonnées entières, avec un renommage.

Table point		
nom	abscisse	ordonnee
A	0	0
B	1	0
C	0	1
D	2	-3
E	-3	-2

La requête

```
SELECT nom,abscisse a, ordonnee o FROM point ORDER BY a*a+o*o DESC,ABS(a) ASC ;
```

produit la table suivante :

nom	a	o
D	2	-3
E	-3	-2
C	0	1
B	1	0
A	0	0

Les points sont triés par distance à l'origine décroissante. En cas d'égalité, ils sont triés par valeur absolue d'abscisse croissante.

13.6.2 Limiter l'affichage avec LIMIT et OFFSET

On peut limiter le nombre de résultats d'une requête SQL en utilisant LIMIT : en ajoutant LIMIT n avec n un entier, on limite le nombre de résultats à n. Il est aussi possible de préciser un OFFSET [3] : avec OFFSET m on ignore les m premiers résultats. Ne pas mettre OFFSET est donc équivalent à OFFSET 0. En reprenant l'exemple précédent, la requête :

```
SELECT nom,abscisse a,ordonnee o FROM point ORDER BY a*a+o*o DESC, ABS(a) ASC LIMIT 2 OFFSET 1 ;
```

produit :

nom	a	o
E	-3	-2
C	0	1

On limite le nombre de lignes à 2 au maximum, et on ignore la première (D ici). LIMIT et OFFSET sont utilisables sans préciser d'ordre, mais rappelez-vous que l'ordre de l'affichage n'est pas garanti (il dépend du stockage interne de la BDD) et c'est donc en général peu pertinent d'en faire usage sans ORDER BY.

13.7 Récapitulatif des requêtes sur une seule table

En SQL, une requête sur une seule table se présente invariablement sous la forme suivante. Tous les mots-clés n'ont pas à être présents, bien sûr, par contre l'ordre est toujours celui-ci, et il faut le connaître !

```
SELECT attributs
FROM table
WHERE condition
GROUP BY attributs
HAVING condition
ORDER BY quantité
LIMIT entier
OFFSET entier
```

Quelques rappels :

— on n'écrit pas de `GROUP BY` s'il n'y a pas de fonction d'agrégation ;

— on n'écrit pas de `HAVING` s'il n'y a pas de `GROUP BY` ;

— on ne confond pas `GROUP BY` et `ORDER BY`.

3. OFFSET ne s'utilise pas sans LIMIT !

13.8 Composition de requêtes

Ceci est un point important, et on commence à voir l'intérêt des bases de données. On a déja vu HAVING, qui effectue une sélection en aval d'une agrégation :

```
SELECT filiere,AVG(notebac) moy FROM eleve GROUP BY filiere HAVING moy>14.4 ;
```

Voyons un exemple équivalent à celui plus haut, mais sans HAVING :

```
SELECT filiere,moy FROM (SELECT filiere,AVG(notebac) moy FROM eleve GROUP BY filiere) WHERE moy>14.4 ;
```

Ici, on effectue une première requête (à l'intérieur des parenthèses) produisant des lignes de la forme filiere, moyenne, puis on réutilise immédiatement la table produite dans une nouvelle requête. Celle-ci est équivalente à la première et produit le même résultat, et peut-être vue comme une traduction SQL différente de la même opération en algèbre relationnelle.

Que ce soit en algèbre relationnelle ou en SQL, ceci se justifie bien : appliquer les opérateurs de l'algèbre relationnelle à une relation produit une relation, appliquer des requêtes à une table produit une table.

Un autre exemple est le suivant : quels sont les élèves ayant eu la plus haute note au bac ? Ici, il faut récupérer d'abord la plus haute note au bac, puis refaire une requête pour sélectionner les élèves ayant eu cette note. En SQL, on obtient :

```
SELECT * FROM eleve WHERE notebac=(SELECT MAX(notebac) FROM eleve) ;
```

Ici, on utilise l'identitification entre une table à 1 ligne et 1 colonne et la valeur de cette case. Attention, on pourrait être tenté d'écrire quelque chose comme

```
SELECT nom,prenom,MAX(notebac) FROM eleve  ;
```

qui n'a pas vraiment de sens en algèbre relationnelle, mais fonctionne en SQL (cela peut dépendre du logiciel utilisé). Le résultat produit est invariablement une table avec une seule ligne : on obtient les nom et prénom d'*un seul* élève ayant eu la meilleure note au bac (avec cette note).

Bref, on peut faire des sous-requêtes en les plaçant entre parenthèses. On a vu qu'on pouvait utiliser comme valeur une table à une ligne et une colonne pour faire une sélection. Par extension, on peut utiliser une requête produisant une table à une seule colonne avec le mot-clef IN. Par exemple, à la question « Quels sont les élèves de PC qui ont même prénom qu'un élève de MP ? », on peut répondre avec la requête :

```
SELECT * FROM eleve WHERE filiere="PC" AND prenom IN (SELECT prenom FROM eleve WHERE filiere="MP")
```

Le mot-clé IN est hors programme. Signalons enfin le mot-clé WITH (hors programme également), qui permet d'utiliser facilement le résultat d'une requête comme une table :

```
WITH r AS (SELECT ...) SELECT .. FROM r ...
```

Chapitre 14

Bases de données à plusieurs tables

L'intérêt des bases de données réside surtout dans la possibilité de découper l'information sur *plusieurs* tables, ce qui permet d'éviter la redondance d'information, et de garantir des contraintes d'intégrité. Par exemple, dans une base de données bien faite gérant les élèves d'un lycée, lorsqu'on rajoute un lycée à la base de données, il est impossible de le placer dans une classe qui n'existe pas.

14.1 Modèle entités-associations, clés étrangères

À partir de données brutes, il y a *a priori* plusieurs manières de les séparer en plusieurs tables de manière intelligente. Les principes généraux guidant la création d'une base de données à partir d'une situation concrète sont hors programme [1], néanmoins on va en donner une légère introduction. La conception d'une base de données (choix des tables, attributs...) consiste souvent en une modélisation préalable, la modélisation par entités-associations en est une très courante.

14.1.1 Entités et associations

Entité et type-entité. La notion d'entité a déja été vue implicitement au chapitre précédent :

— on parle d'entité pour désigner un objet, et on parle de propriétés pour désigner les composantes de l'objet. Par exemple, une personne peut être décrite par un numéro unique (disons id_personne), un nom, un prénom et une adresse, est une entité. Un livre d'une bibliothèque, décrit par un numéro unique, un titre, un auteur (on suppose l'auteur unique, c'est une simplification) est une autre entité. En SQL, l'objet correspond à un enregistrement d'une table, ses propriétés sont ses attributs.

— on parle de type entité pour désigner un ensemble d'entités qui possède les mêmes caractéristiques. On pourra parler du type entité Personne ou du type entité Livre. Il n'est pas possible de regrouper une personne et un livre dans un même type-entité, une personne n'ayant pas d'auteur ou un livre pas d'adresse. En SQL, un type-entité correspond à une une table d'une base de données : ici la table des personnes enregistrées et des livres de la bibliothèque.

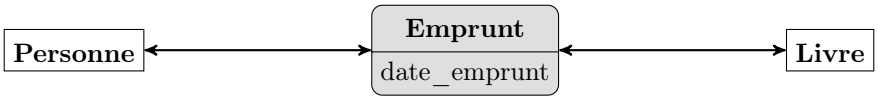
On représentera un type entité avec ses propriétés éventuelles comme suit :

Personne
id_personne
nom
prenom
adresse

Livre
id_livre
titre
auteur

Association. Une association consiste simplement à faire le lien entre deux entités : dans la gestion d'une bibliothèque par exemple, Robert Plant peut emprunter le livre « Le rock pour les nuls ». L'association sera ici l'emprunt, reliant une personne est un livre. L'ensemble de telles associations consiste en un type-association (dans la pratique, on confond souvent association et type-association, seul le type association est représenté graphiquement). Une telle association se représente comme suit :

1. Heureusement, car il y a matière à en faire un livre entier ! Le lecteur intéressé pourra consulter l'ouvrage de Laurent Audibert dont ce cours s'inspire : Bases de données, de la modélisation au SQL, ou sa version en ligne disponible ici : `https://laurent-audibert.developpez.com/Cours-BD/`



Comme on le voit, une association peut également posséder des propriétés, ici il n'y en a qu'une : la date d'emprunt. Les associations seront souvent binaires comme ci-dessus, bien qu'on puisse trouver des associations d'arité (nombre de types entité qui interviennent) supérieure à 2.

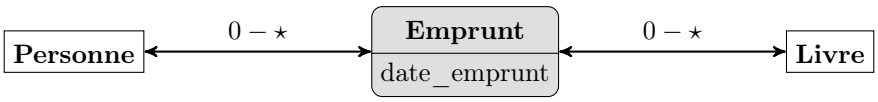
14.1.2 Cardinalités des associations

Une entité donnée peut figurer un certain nombre de fois dans l'association. On fait figurer sur le diagramme précédent le nombre de fois minimal (m) et maximal (M) que peut figurer une entité dans l'association, sous la forme $m - M$. En pratique :

— le nombre minimal m est toujours 0 ou 1 ;

— le nombre maximal M est toujours 1 ou strictement supérieur à 1, ce que l'on note par une étoile $\star$ (remarque : savoir que par exemple, une entité ne peut figurer qu'au plus 5 fois dans l'association n'est pas intéressant, ce qu'on veut savoir c'est si elle peut se trouver strictement plus d'une fois !).

Les cardinalités intéressantes sont donc $0-1$, $1-1$, $1-\star$ et $0-\star$ (le programme officiel ne mentionne que $1-1$ et $1-\star$). Dans notre exemple, une personne peut emprunter plusieurs livres (en même temps ou non), et un livre peut être emprunté par plusieurs personnes différentes. Il se peut aussi qu'une personne n'ait jamais fait d'emprunt ou qu'un livre ne soit jamais sorti de la bibliothèque. Les deux cardinalités sont donc $0-\star$, ce que l'on représente ainsi :

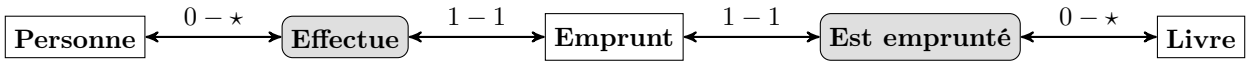


14.1.3 Transformation d'une cardinalité $\star \bullet \star$

Dans l'exemple précédent, les deux pattes de l'association binaire possède une cardinalité maximale $\star$. Une telle cardinalité traduit la nécessité, lorsque l'on va vouloir transformer la modélisation en base de données réelles, de scinder cette association en deux. Pour cela, on introduit un nouveau type-entité, qui ici correspond à l'emprunt, par exemple celui-ci :

Emprunt
id_emprunt
personne_emprunt
livre_emprunt
date_emprunt

On discutera des attributs plus précisément dans la suite, mais personne_emprunt (resp. livre_emprunt) est le numéro de la personne qui fait l'emprunt (resp. le livre emprunté).
L'association précédente se divise en deux nouvelles associations :



Les cardinalités $0-\star$ sont toujours présentes : une personne peut réaliser plusieurs emprunts, un livre peut être emprunté plusieurs fois. Maintenant qu'un emprunt est une entité, un emprunt donné (caractérisé notamment par un numéro unique) est réalisé une et une seule fois.

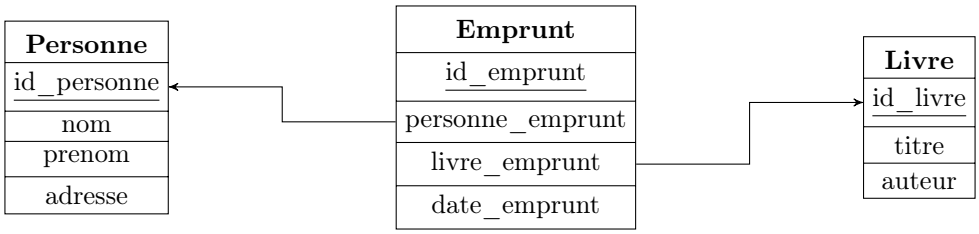
14.1.4 Clés primaires et clés étrangères

Définitions. On a déja parlé de clés primaires en SQL : un ensemble d'attributs qui caractérisent un enregistrement de manière unique (souvent réduit à un attribut, mais pas systématiquement). Dans les exemples ci-dessus, id_personne, id_livre et id_emprunt seraient des clés primaires des tables associées.

Comme on le voit dans l'entité Emprunt, deux propriétés (personne_emprunt et livre_emprunt), dépendent des deux entités Personne et Livre. Lorsqu'on crée une base de données, lors de la création de la table Emprunt, on peut

déclarer les attributs personne_emprunt et livre_emprunt comme des *clés étrangères*, qui *référencent* les attributs id_personne et id_livre des tables concernées.

Une clé étrangère est donc un (ou plusieurs) attribut(s) d'une table, référençant un ou plusieurs attributs d'une autre table, ce qui signifie notamment que les valeurs prises par la clé étrangère doivent être des valeurs prises par le(s) attribut(s) référencé(s). La plupart du temps, la clé étrangère référence une clé primaire de la table référencée. Voici un schéma d'une base de données complète, avec les clés primaires soulignées et les clés étrangères indiquées par des flêches.



La déclaration d'une clé étrangère se fait à la création de la table. Pour la culture, voici comment créer la table `Emprunt`, en supposant les tables `Personne` et `Livre` déja créées :

```
CREATE TABLE Emprunt (
    id_emprunt INTEGER PRIMARY KEY AUTOINCREMENT,
    personne_emprunt INT,
    livre_emprunt INT,
    date_emprunt DATE,
    FOREIGN KEY (
        personne_emprunt
    )
    REFERENCES Personne (id_personne),
    FOREIGN KEY (
        livre_emprunt
    )
    REFERENCES Livre (id_livre),
);
```

Une requête SQL rajoutant un enregistrement dans la table Emprunt avec une valeur de l'attribut personne_emprunt qui ne correspond à aucune valeur de l'attribut id_personne de la table Personne mènera à une erreur.

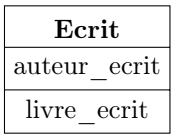
Cardinalité des associations et clés. Les cardinalités d'association où l'une au moins des pattes est $1-1$ traduit la possibilité d'avoir des clés étrangères ou primaires.

- Comme on le voit dans l'exemple ci-dessus, une association de la forme

  traduit la possibilité qu'un attribut de la table modélisant A soit une clé étrangère référençant une clé primaire de la table modélisant B.

- Une association de la forme

  traduit la possibilité d'avoir la même clé primaire (à renommage près) pour les tables modélisant A et B. Prenons un exemple concret : supposons que l'on ait également une table Auteur contenant les auteurs des livres (on suppose qu'un livre a un seul auteur pour simplifier, mais un auteur peut écrire plusieurs livres). On pourrait résumer le fait qu'un livre donné soit écrit par un auteur donné dans une table à deux attributs :



où auteur_ecrit serait une clé étrangère vers l'attribut id_auteur de la table Auteur, et l'attribut livre_ecrit serait (a priori) une clé étrangère vers l'attribut id_livre de Livre. Puisqu'un livre a exactement un auteur dans notre hypothèse, l'association entre Ecrit et Livre est bien de la forme $1-1$ des deux côtés : livre_ecrit est en fait une clé primaire de Ecrit [2] (et les deux tables ont exactement le même nombre d'enregistrements).

14.2 Produit cartésien et jointure

14.2.1 Introduction

On a déja évoqué le fait que pour éviter la redondance des informations il est en général préférable d'organiser nos données en différentes tables, plutôt qu'en une unique base de données (plate). Dans la suite, on prendra la base de données ci dessous en exemple.

Table eleve

id	prenom	nom	id_classe	id_lycee
1	Mathilde	Dufour	934	1
2	Léa	Dupond	932	2
3	Paul	Dugommier	933	2
4	Mathilde	Dugommier	931	1
5	Clément	Durand	933	3

Table lycee

id_lycee	nom_lycee
1	Calmette
2	Massena
3	Parc Imperial

Table classe

id_classe	filiere	numero
931	MP	1
932	MP	2
933	PC	1
934	PC	2

Dans cette base :

- L'attribut `id` de eleve est une clé primaire de cette table ;
- `id_classe` et `id_lycee` sont des clés primaires des tables `classe` et `lycee`, et les attributs de même nom dans la table `eleve` en sont des clés étrangères, référençant ces clés primaires. On aurait pu choisir des noms différents, on verra que cela n'est pas un problème.

14.2.2 Produit

Le premier moyen d'assembler deux relations est d'en faire le produit cartésien.

Définition 14.1 (Produit de deux relations)**.** *Si $\mathcal{R}$ est une relation de schéma S et $\mathcal{R}'$ une relation de schéma S' avec $S \cap S' = \emptyset$ alors le produit de $\mathcal{R}$ et $\mathcal{R}'$ est la relation $\mathcal{R} \times \mathcal{R}'$ de schéma $S \cup S'$ définie par*

$$\mathcal{R} \times \mathcal{R}' = \{u; u[S] \in \mathcal{R}, u[S'] \in \mathcal{R}'\}$$

La condition $\mathcal{S} \cap \mathcal{S}' = \emptyset$ n'est en fait pas contraignante : on peut procéder par renommage pour l'assurer. Pour éviter les homonymies, on notera en général $\mathcal{R}.A$ et $\mathcal{R}'.A$ les attributs de $\mathcal{R}$ et $\mathcal{R}'$.

Une telle table a donc pour cardinal le produit $|\mathcal{R}| \times |\mathcal{R}'|$, ce qui peut facilement devenir très lourd avec des tables de taille conséquente. Voici le début de la table *eleve* $\times$ *lycee* :

Table eleve×lycee

id	prenom	nom	id_classe	id_lycee	id_lycee	nom_lycee
1	Mathilde	Dufour	934	1	1	Calmette
1	Mathilde	Dufour	934	1	2	Massena
1	Mathilde	Dufour	934	1	3	Parc Imperial
2	Léa	Dupond	932	2	1	Calmette
2	Léa	Dupond	932	2	2	Massena

On remarque qu'il semble y avoir deux attributs de même nom (ce qui est impossible) : en fait les noms de ces attributs sont eleve.id_lycee et lycee.id_lycee (la présentation sous cette forme suit ce qu'il se passe en SQL).

14.2.3 Jointure

Comme on le voit dans l'exemple ci-dessus, le produit cartésien entre deux tables donnent en général beaucoup d'enregistrements, et la plupart ne sont pas pertinents : ici, on voudrait ne garder que les enregistrements pour lesquels les valeurs des deux attributs eleve.id_lycee et lycee.id_lycee sont les mêmes, ce qu'on peut faire via une sélection.

La notion de jointure correspond à ceci : il ne s'agit d'un point de vue algébrique que d'une sélection après un produit cartésien.

2. Cette modélisation n'est pas très bonne : la table Ecrit ne sert pas à grand chose ! Par contre, on peut l'obtenir facilement comme résultat d'une requête SQL sur les tables Auteur et Livre, ce qu'on va apprendre à faire à la section suivante.

Définition 14.2 (Jointure de deux relations)**.** *Pour $\mathcal{R}$ et $\mathcal{R}'$ sont deux relations de schémas S et S' avec $S \cap S' = \emptyset$, et $\mathcal{C}$ une condition booléenne portant sur $S \cup S'$ alors la jointure de $\mathcal{R}$ et $\mathcal{R}'$ selon $\mathcal{C}$ est la relation $\mathcal{R} \bowtie_{\mathcal{C}} \mathcal{R}'$ de schéma $S \cup S'$ définie par*

$$\mathcal{R} \bowtie_{\mathcal{C}} \mathcal{R}' = \sigma_{\mathcal{C}}(\mathcal{R} \times \mathcal{R}')$$

On remarque que si $\mathcal{C}$ est la condition triviale (toujours vraie) on retrouve le produit cartésien standard. Reprenons l'exemple précédent :

Table eleve $\bowtie_{\text{eleve.id\_lycee=lycee.id\_lycee}}$ lycee						
id	prenom	nom	id_classe	id_lycee	id_lycee	nom_lycee
1	Mathilde	Dufour	934	1	1	Calmette
2	Léa	Dupond	932	2	2	Massena
3	Paul	Dugommier	933	2	2	Massena
4	Mathilde	Dugommier	931	1	1	Calmette
5	Clément	Durand	933	3	3	Parc Impérial

La table obtenue possède beaucoup moins de lignes que le produit cartésien !

Définition 14.3. *Si $\mathcal{C}$ est une conjonction d'égalités entre attributs, la jointure est appelée une* équi-jointure*.*

La plupart du temps, on considérera des équi-jointures, qui sont les seules au programme. Bien souvent, comme ci-dessus, ces jointures consisteront à imposer l'égalité entre une clé étrangère et la clé primaire référencée.

14.3 Tables multiples en SQL

14.3.1 Produit cartésien

La syntaxe en SQL pour le produit cartésien de tables est la suivante :

```
SELECT ... FROM table_1 JOIN table_2 JOIN ... JOIN table_n
```

Comme on le voit, il suffit de séparer les différentes tables par l'opérateur `JOIN`. Après un produit cartésien, rien n'empêche de procéder à des sélections, projections, agrégations... comme dans le chapitre précédent. Toutefois, lorsqu'on veut réaliser une jointure, on préfère séparer la condition de jointure d'une éventuelle sélection ultérieure.

14.3.2 Jointure

La jointure n'est qu'une sélection dans le produit cartésien, toutefois SQL propose le mot clé ON pour spécifier les conditions de jointures plutôt que dans une clause WHERE. La syntaxe d'une jointure est donc la suivante :

```
SELECT ... FROM table_1 JOIN table_2 JOIN ... JOIN table_n ON conditions de jointure
```

14.3.3 Quelques exemples

On rappelle qu'après une jointure, les attributs sont de la forme table.attribut. S'il n'y a pas ambiguïté (c'est-à-dire que l'attribut n'est présent que dans une seule des tables), on n'est pas obligé de préciser la table. Dans le cas où il y a ambiguïté, on préfère renommer les tables en des petits noms plus courts pour écourter les requêtes.

— Produit cartésien eleve × lycee :

```
SELECT * FROM eleve JOIN lycee
```

— Jointure entre eleve et lycee, identifiant les deux attributs id_lycee :

```
SELECT * FROM eleve JOIN lycee ON eleve.id_lycee = lycee.id_lycee
```

— Noms et prénoms des élèves provenant du lycée Calmette. On choisit de renommer les tables. Pour les attributs sans ambiguïté, on choisit de ne pas faire figurer la table de provenance.

```
SELECT nom,prenom FROM eleve e JOIN lycee L ON e.id_lycee=L.id_lycee
WHERE nom_lycee = 'Calmette'
```

— Noms et prénoms des élèves de MP 1 provenant du lycée Calmette. Il s'agit d'une jointure sur les 3 tables.

```
SELECT nom,prenom FROM eleve e JOIN lycee L JOIN classe c ON e.id_lycee=L.id_lycee AND
    e.id_classe=c.id_classe WHERE nom_lycee='Calmette' AND filiere='MP' AND numero=1 ;
```

— Pour chaque classe, le nombre d'élèves venant du lycée Calmette (on utilise juste le id_classe) :

```
SELECT id_classe, COUNT(*) FROM eleve e JOIN lycee L
ON e.id_lycee = L.id_lycee WHERE nom_lycee = 'Calmette'
GROUP BY id_classe
```

14.3.4 Auto-jointure

Il est possible d'utiliser deux fois (ou plus !) la même table. Cette possibilité est d'ailleurs très appréciée des sujets de concours... Pour ce faire, un renommage de table est obligatoire, puisque tous les attributs sont ambigus ! Voici un exemple d'utilisation : récupérer les couples d'identifiants d'élèves situés dans la même classe et provenant du même lycée.

```
SELECT e1.id, e2.id
FROM eleves e1 JOIN eleves e2
ON e1.id_classe = e2.id_classe AND e1.id_lycee = e2.id_lycee
WHERE e1.id < e2.id
```

On a choisit de faire une sélection supplémentaire : déja, il faut supprimer les couples qui proviennent d'un élève dupliqué (deux fois le même identifiant), et on choisit d'éliminer la moitié des couples en ne gardant que les couples où le premier identifiant est strictement inférieure au deuxième.

14.3.5 Pour conclure

L'ordre des mot-clés dans une requête de recherche dans une table SQL est toujours le même. Voici à quoi ressemble une requête complète, avec jointures.

```
SELECT attributs
FROM table1 JOIN table2 JOIN ... ON conditions de jointure
WHERE conditions de sélection
GROUP BY attributs de regroupement
HAVING conditions de sélection après agrégation
ORDER BY quantités pour l'ordonnancement
LIMIT n
OFFSET p
```

14.4 Opérateurs ensemblistes

Un autre manière de combiner deux relations consiste à utiliser les opérateurs ensemblistes, qui s'applique sur des relations *de même schéma* (c'est-à-dire ayant mêmes attributs).

Définition 14.4. *Soient $\mathcal{R}$ et $\mathcal{R}'$ deux relations de même schéma.*

— *$\mathcal{R} \cup \mathcal{R}'$ est la relation de même schéma dont les tuples sont ceux qui appartiennent à $\mathcal{R}$ ou à $\mathcal{R}'$*

— *$\mathcal{R} \cap \mathcal{R}'$ est la relation de même schéma dont les tuples sont ceux qui appartiennent à $\mathcal{R}$ et à $\mathcal{R}'$.*

— *$\mathcal{R} \setminus \mathcal{R}'$ est la relation de même schéma dont les tuples sont ceux qui appartiennent à $\mathcal{R}$ mais pas à $\mathcal{R}'$.*

En pratique, on obtient souvent des relations de même schéma en appliquant aux préalables des sélections. Par exemple, si l'on veut obtenir les prénoms d'élèves que l'on peut trouver à la fois en MP et en PC, on peut considérer

$$\pi_{\text{prenom}} \circ \sigma_{\text{id\_classe} \in \{931,932\}}(\text{eleve}) \cap \pi_{\text{prenom}} \circ \sigma_{\text{id\_classe} \in \{933,934\}}(\text{eleve})$$

En SQL, ces opérateurs sont `UNION`, `INTERSECT` et `EXCEPT`. La table précédente s'obtient comme ceci :

```
SELECT DISTINCT prenom FROM eleves WHERE id_classe = 931 OR id_classe = 932
INTERSECT
SELECT DISTINCT prenom FROM eleves WHERE id_classe = 933 OR id_classe = 934
```

En pratique, INTERSECT et EXCEPT peuvent se recoder en utilisant l'opérateur IN évoqué dans le chapitre précédent (hors programme). Par exemple, la requête précédente se réécrit :

```
SELECT DISTINCT prenom FROM eleves WHERE id_classe = 931 OR id_classe = 932
AND prenom IN SELECT prenom FROM eleves WHERE id_classe = 933 OR id_classe = 934
```

Sixième partie

Compléments d'algorithmique : dictionnaires et programmation dynamique

Chapitre 15

Implémentation des dictionnaires

15.1 Introduction

Dans ce chapitre, on voit comment implémenter la structure de dictionnaire vue en première année. Rappelons que cette structure permet de stocker des couples (k, e) où k est une *clé* et e un élément. Une contrainte est qu'une clé donnée ne peut être associée qu'à un seul couple. Les opérations qui doivent être permises sont les suivantes :

— création d'un dictionnaire vide ;
— recherche dans le dictionnaire d'un couple de clé k, accès à l'élément associé e si la clé est présente ;
— ajout d'un nouveau couple (k, e) ;
— modification de l'élément associé à une clé si elle est présente ;
— suppression du couple (k, e) à partir de la clé k.

On a vu en première année la syntaxe Python pour ces opérations, ce chapitre explique comment réaliser ces opérations de manière efficace, à l'aide de listes et de *fonctions de hachage*. En pratique, c'est de cette manière que sont implémentés les dictionnaires en Python.

Parenthèse. Une implémentation non efficace. On peut toujours implémenter un dictionnaire en utilisant simplement une liste de couples (k, e). Les opérations précédentes (à part la création) seront de complexité linéaire en le nombre de couples stockés dans le dictionnaire (en supposant que les clés sont de taille bornée pour que le test d'égalité s'effectue en temps constant). Dans la suite, on veut faire mieux !

15.2 Adressage direct et limites

15.2.1 Principe de l'adressage direct

Supposons que l'on sache que les clés à stocker soient des entiers de $K = [\![0, m[\![$, avec m un entier pas trop grand. Il est alors possible de réaliser la structure de dictionnaire dans une liste `L` de taille m. En supposant qu'un élément associé à une clé ne puisse jamais être `None`, on pourra réaliser le dictionnaire avec la convention :

$$\text{Pour } \texttt{k} \in [\![0, m-1]\!],\ \texttt{L[k]} = \begin{cases} \texttt{None} & \text{si la clé } \texttt{k} \text{ n'est pas présente;} \\ \text{l'élément } \texttt{e} \text{ associé} & \text{sinon.} \end{cases}$$

Voici par exemple un dictionnaire dont on sait que les clés ne peuvent être que des entiers de $[\![0, 9]\!]$. Le dictionnaire contient actuellement 3 couples : `(1, 'abc')`, `(5, 54)` et `(7, True)`.

```
[None, 'abc', None, None, None, 54, None, True, None, None]
```

Il est facile d'implémenter toutes les opérations de la liste ci-dessus avec cette structure, et elles s'effectuent toutes en temps constant, sauf la création. En effet, l'opération de création (qui devrait prendre en paramètre l'entier m bornant les clés possibles) s'effectue en temps $O(m)$.

15.2.2 Limites de l'adressage direct

L'adressage direct, outre le fait qu'il impose une contraite forte sur les clés (nécessairement des entiers positifs pas trop grands), demande également une complexité mémoire $O(m)$ quel que soit le nombre de clés effectivement contenues dans le dictionnaire. Il est préférable d'avoir une complexité mémoire linéaire en ce nombre, pour ne pas gaspiller de la mémoire. On va voir dans les sections qui suivent comment on peut réaliser ceci.

15.3 Tables de hachage de largeur fixe

Si l'ensemble des clés possibles est gros, voire même virtuellement infini, la stratégie précédente ne fonctionne pas. L'idée des *fonctions de hachage* est de ramener l'univers des clés possibles vers un ensemble fini $[\![0, \omega - 1]\!]$, avec ω pas trop gros. On utilisera alors une liste de taille ω pour stocker les couples (clé, élément). Plus précisément, avec `D` une telle liste, `D[i]` sera la liste des couples de clés dont l'image par la fonction de hachage est i. Comme une fonction de hachagetelle fonction n'est en général pas injective, il se peut que deux clés k et k' aient la même image via la fonction de hachage h. On verra comment gérer de telles *collisions*.

15.3.1 Fonctions de hachage

Une fonction de hachage est une fonction, qui, a partir d'une donnée fournie en entrée, renvoie un nombre de taille fixe. Dans une utilisation en cryptographie, une fonction de hachage doit vérifier les propriétés suivantes :

— le résultat *paraît* aléatoire : modifier un tout petit peu l'entrée modifie énormément la sortie. (Le processus est toutefois déterministe : le résultat d'un hachage est toujours le même sur la même entrée !)

— il est très difficile de construire des collisions : c'est-à-dire de trouver deux entrées différentes dont l'image par la fonction de hachage est la même [1].

Citons quelques fonctions de hachage usuelles : MD5 [2], la famille SHA-2 [3], et bien d'autres...

Voici, le résultat d'applications de MD5 à quelques chaînes de caractères (la syntaxe est celle du terminal Bash, sous Linux).

```
>>> echo "ceci est une chaine de caracteres" | md5sum
961742bb2d5f1b07e0e3cb2464075050  -
>>> echo "ceci est une chaine de caracteres!" | md5sum
a9c41cdd173a18a7d58d7e4dab4e5bee  -
```

Comme on le voit, le résultat est un nombre hexadécimal à 32 chiffres (soit 128 bits). L'ajout d'un caractère à la chaîne change drastiquement le résultat !

Citons deux applications usuelles des fonctions de hachage.

— lorsqu'on télécharge un fichier d'installation, on peut vouloir s'assurer que la transmission s'est faite sans erreur : procéder à une installation à partir d'un fichier corrompu peut s'avérer catastrophique. Le site officiel d'Ubuntu donne la valeur de hachage du fichier d'installation du système d'exploitation par la fonction de hachage SHA-256 : il suffit de vérifier que le hachage du fichier téléchargé coïncide bien.

— un site web permettant l'authentification des utilisateurs ne doit pas stocker « en clair » les mots de passe de ses utilisateurs pour des raisons de sécurité. (Une attaque pourrait récupérer tous les mots de passe !). Ainsi, celui-ci ne stocke que les images de ces mots de passe par une fonction de hachage donnée. Lorsqu'un utilisateur rentre son mot de passe, son image par la fonction de hachage est calculée et c'est cette valeur qui est comparée à ce qui est enregistré sur le site web.

15.3.2 Parenthèse : le hachage en Python

Python propose une fonction de hachage, il suffit d'appliquer la méthode `__hash__()` à l'objet que l'on veut hacher. L'entier renvoyé est un entier de 64 bits en complément à deux (donc un entier de $[\![-2^{63}, 2^{63} - 1]\!]$). Par exemple [4]

```
>>> "ceci est une chaine de caracteres".__hash__()
1792338306092055307
>>> "ceci est une chaine de caracteres!".__hash__()
2442840689297897247
```

1. Remarque : puisque la fonction de hachage est à valeurs dans un ensemble fini, les collisions existent, et on pourrait en trouver en calculant des images successives jusqu'à ce que l'on retombe sur une valeur déja obtenue. Ceci est impossible en pratique si l'espace d'arrivée est suffisamment grand, par exemple des nombres de 128 bits.

2. Obsolète aujourd'hui. `https://fr.wikipedia.org/wiki/MD5`

3. `https://fr.wikipedia.org/wiki/SHA-2`

4. Si vous essayez chez vous, vous aurez probablement un résultat différent. Il s'agit en fait d'une famille de fonctions, celle utilisée change à chaque lancement de Python.

15.3.3 Utilisation pour la structure de dictionnaire

Fixons un entier ω (la largeur de hachage), et proposons une impémentation de la structure de dictionnaire dans une liste `L` de taille ω fixée. On peut forcer la fonction de hachage à prendre des valeurs entre 0 et $\omega - 1$ en utilisant simplement le modulo. En supposant `omega` (variable globale) initialisée, on aura donc :

```
def hachage(x):
    return x.__hash__() % omega
```

Dans la suite, on note h cette fonction de hachage à valeurs dans $[\![0, \omega - 1]\!]$. La liste `L` contiendra ω listes, qu'on appelle des *alvéoles*. Lorsqu'on veut stocker un couple (k, e) dans le dictionnaire, on calcule la valeur $h(k)$. En notant $i = h(k)$, on rajoute à l'alvéole `L[i]` le couple (k, e). De même :

— lorsqu'on veut tester si un couple de clé k est présent, on calcule $i = h(k)$ et on cherche dans l'alvéole `L[i]` si un couple de clé k est présent, il faut parcourir toute l'alvéole dans le pire cas.

— on procède de manière similaire pour supprimer un couple à partir de sa clé, ou modifier l'élément associé à une clé.

La figure 15.1 présente un exemple de telle table de hachage de largeur $\omega = 5$, où pour l'exemple les clés sont simplement des entiers, et la fonction de hachage est simplement la congruence modulo 5. On n'a fait figurer que les clés et pas les éléments associés. Les alvéoles sont représentées comme des chaînes de maillons, même si on va les implémenter avec des listes classiques. On remarque que l'alvéole `L[2]` est vide.

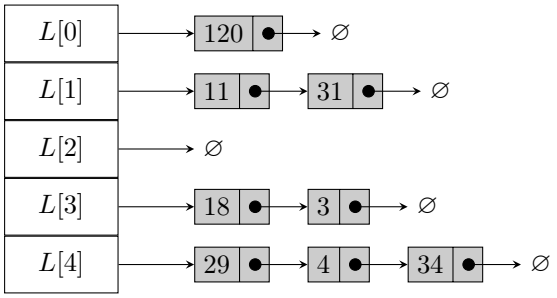


FIGURE 15.1 – Illustration d'une table de hachage de largeur $\omega = 5$

Voici, en Python, la table de hachage de la figure 15.1. Là aussi, pour simplifier, la table ne contient que des clés et pas des couples (clé, élément) :

```
L = [[120], [11, 31], [], [18, 3], [29, 4, 34]]
```

15.3.4 Implémentation en Python de la structure de dictionnaire

On peut maintenant écrire les fonctions de dictionnaire, en Python. On suppose la variable `omega` globale déclarée, et on utilisera la fonction `hachage` décrite plus haut.

- Création d'un dictionnaire vide : il s'agit de créer un ensemble de ω alvéoles, toutes vides.

```
def creer_dict():
    return [[] for i in range(omega)]
```

- Test de présence d'une clé k. Il s'agit de voir s'il existe un couple de clé k dans le dictionnaire. Il suffit de regarder dans l'alvéole d'indice $i = h(k)$, il est inutile de regarder toutes les alvéoles !

```
def est_presente(D,k):
    """ teste si la clé k est présente """
    i = hachage(k)
    for c in D[i]:
        if c[0]==k:
            return True
    return False
```

- Élément associé à une clé supposée présente :

```
def element(D,k):
    """ renvoie l'élément associé à une clé, supposée présente """
    assert est_presente(D,k)
    i = hachage(k)
    for c in D[i]:
        if c[0]==k:
            return c[1]
```

- Ajout du couple (k, e), k supposée non présente :

```
def ajout(D,k,e):
    """ ajoute le couple (k,e), k supposée non présente """
    assert not est_presente(D,k)
    i = hachage(k)
    D[i].append((k,e))
```

- Suppression du couple de clé k, k supposée présente.

```
def suppression(D,k):
    """ supprime le couple de clé k, supposée présente """
    assert est_presente(D,k)
    i = hachage(k)
    for j in range(len(D[i])):
        if D[i][j]==k:
            D[i].pop(j)
            return
```

- Modification de l'élément associé à une clé k, supposée présente.

```
def modification(D,k,e):
    """ modifie l'élément associé à une clé, supposée présente """
    assert est_presente(D,k)
    i = hachage(k)
    for j in range(len(D[i])):
        if D[i][j]==k:
            D[i].pop(j)
            D[i].append((k,e))
            return
```

15.3.5 Complexité : discussion

En supposant que l'on travaille sur des objets de taille bornée, (de sorte que la complexité de la fonction de hachage est constante, tout comme le test d'égalité de deux clés), la complexité des opérations précédentes (sauf la création) est linéaire en la taille de l'alvéole numéro $i = h(k)$, où k est la clé impliquée. Dans le pire cas, on peut imaginer que toutes les clés du dictionnaire sont présentes dans la même alvéole, et dans ce cas, la complexité n'est pas meilleure que si on avait utilisé naïvement une liste.

A contrario, si on suppose que la fonction de hachage a tendance à bien répartir les clés entre alvéoles, on peut imaginer que chacune d'elle a une taille environ n/ω, où n est le nombre de couples stockés et ω la largeur de hachage. Sans rentrer dans les détails probabilistes qui dépasseraient largement le cadre de ce cours, on admet que la complexité est de $O(1 + n/\omega)$ pour les opérations précédentes (sans la création en $O(\omega)$), sous des hypothèses naturelles.

15.4 Tables de hachage de largeur variable

On a dit en première année que les opérations de dictionnaire pouvait être considérée comme en $O(1)$, ce qui ne peut clairement pas être atteint avec la stratégie précédente si ω est fixe. Il suffit pour cela de faire varier la largeur de la table, de manière à avoir toujours n/ω borné, avec n le nombre d'entrées stockées dans le dictionnaire. On discute brièvement dans cette section d'une stratégie efficace, sans coder.

Famille de fonctions de hachage. On a dit que la fonction de hachage de Python était à valeur dans un ensemble de taille 2^{64} (les entiers relatifs de 64 bits en complément à deux). Cette taille étant très élevée, elle dominera toujours très nettement les largeurs de hachage dont on pourrait avoir besoin en pratique. En considérant des largeurs ω variables, on obtient modulo ω une famille de fonctions de hachage (h_ω), chacune à valeurs dans $[\![0, \omega - 1]\!]$.

Ajout d'un élément à une table de largeur variable. C'est essentiellement la seule opération à modifier, puisque test de présence, suppression, ou modification de l'élément associé à une clé n'augmentent pas le nombre n d'entrées stockées. Prenons une stratégie où l'on veut garantir que l'on a toujours $n/\omega \leq 2$ (la constante 2 importe peu, mais on veut une constante !), et rajoutons une clé non présente dans la table. Avec n le nombre d'entrée stockées :

— si $1+n \leq 2\omega$, on rajoute la clé dans l'alvéole idoine. Sous hypothèse de bonne répartition des clés, la complexité est $O(1)$.

— si $1+n \leq 2\omega$, on réarrange la table, par exemple en doublant sa largeur ω, c'est-à-dire que l'on travaille avec $\omega' = 2\omega$. Il faut alors, pour les n entrées (k, e) de la table, recalculer les valeurs $h_{\omega'}(k)$ pour placer les clés dans les bonnes alvéoles de la nouvelle table. On peut alors ajouter la nouvelle clé comme au point précédent. L'ajout ici se fait donc en temps $O(n)$, mais l'avantage est que le ratio n/ω' vaut environ 1 maintenant : on pourra réaliser de l'ordre de n ajouts supplémentaires sans avoir à changer la largeur de la table, donc chacun en $O(1)$. Les $1+O(n)$ ajouts évoqués dans ce point on donc au total une complexité $O(n)+O(n) = O(n)$: en moyenne, chacun a une complexité $O(1)$.

La stratégie évoquée précédemment est celle utilisée par Python pour implémenter efficacement les dictionnaires, bien que les constantes que l'on a évoquées ($n/\omega \leq 2$, multiplier la largeur d'un facteur 2) soient différentes. On dit que les opérations ont une complexité $O(1)$ *amortie*.

Une preuve du mécanisme en Python. La fonction `sys.getsizeof` permet de connaître la taille en bits d'un objet. Le code suivant crée un dictionnaire et ajoute successivement des entrées, on demande la taille du dictionnaire en bits à chaque étape [5]. Les sauts dans la taille montrent un réarrangement de la table. Les prochains sauts se produisent aux 22ème, 43ème, 86ème... éléments.

```
import sys
D={}
for i in range(1,13):
    D[i]=0
    print(i, " éléments, taille : ", sys.getsizeof(D))
```

```
1  éléments, taille :  232
2  éléments, taille :  232
3  éléments, taille :  232
4  éléments, taille :  232
5  éléments, taille :  232
6  éléments, taille :  360
7  éléments, taille :  360
8  éléments, taille :  360
9  éléments, taille :  360
10  éléments, taille :  360
11  éléments, taille :  640
12  éléments, taille :  640
```

5. Sans rentrer dans les détails, les tailles des clés et des éléments ne sont pas comptées ici, simplement la « structure » du dictionnaire.

Chapitre 16

Programmation dynamique

16.1 Introduction

Ce chapitre fait suite à celui sur les méthodes gloutonnes développées en première année. La programmation dynamique [1] est une technique pour résoudre des problèmes *d'optimisation* : sur un univers $\mathcal{U}$, on cherche à minimiser (ou maximiser, la situation est symétrique) une certaine fonction, souvent à valeurs dans les entiers. Concrètement, on se donne $f : \mathcal{U} \to \mathbb{Z}$, et on cherche à déterminer x tel que $f(x) = \min_{y\in\mathcal{U}}\{f(y)\}$ ou $f(x) = \max_{y\in\mathcal{U}}\{f(y)\}$, si cette quantité existe bien. On a vu en première année, en autres, les exemples suivants :

— l'algorithme de Dijkstra sur les graphes pondérés à poids positifs permet de trouver les poids des plus courts chemins d'un sommet source s vers les autres sommets du graphe. Le choix glouton consiste à traiter les sommets par poids croissant depuis s, et on a montré que ce choix glouton était optimal.

— dans le problème du rendu de monnaie, on a une somme d'argent s à décomposer en sommes de pièces de valeurs $v_0, \dots, v_{n-1}$, tout en minimisant le nombre total de pièces. En supposant ces valeurs croissantes et $v_0 = 1$ (ce qui assure qu'il y a toujours une solution), l'algorithme glouton consiste à prendre le plus de fois possible la pièce v_{n-1} (c'est à dire $\lfloor s/v_{n-1}\rfloor$), puis recommencer avec $s - v_{n-1}\lfloor s/v_{n-1}\rfloor$ et les autres valeurs. On a vu que cette méthode donnait une réponse optimale pour certaines valeurs de pièces (par exemple le système monétaire européen $1, 2, 5, 10, 20, 50, 100, 200, 500$), mais pas systématiquement : avec $s = 6$ et $(v_0, v_1, v_2) = (1, 3, 4)$, l'algorithme glouton propose de prendre une fois la pièce de valeur 4 et deux fois la pièce de valeur 1, alors qu'on peut prendre deux fois la pièce de valeur 3 pour obtenir une solution à deux pièces seulement.

La programmation dynamique, au prix d'une complexité en générale plus élevée qu'une approche gloutonne, permet de résoudre une plus grande variété de problème d'optimisation.

16.2 Un exemple complet : chemin de poids maximal dans une matrice

16.2.1 Le problème

On se donne une matrice $A = (a_{i,j})_{0\leq i<n, 0\leq j<m}$ constituée d'entiers positifs. On cherche un chemin, partant de la case en haut à gauche (indexée par $(0, 0)$ et aboutissant à la case en bas à droite (indexée par $(n-1, m-1)$ en n'utilisant que des déplacements d'un cran vers le bas ou d'un cran vers la droite, dont le *poids* (somme des entiers rencontrés) est maximal.

16.2.2 Recherche exhaustive ?

On peut éventuellement chercher à examiner tous les chemins possibles, car ils sont en nombre fini. Dénombrons les : pour construire un tel chemin, il suffit de savoir où placer les $n-1$ déplacements $\downarrow$ parmi les $n-1+m-1$ déplacements totaux. Ainsi il y a $N_{n,m} = \binom{n+m-2}{n-1}$ chemins possibles. Donnons un équivalent asymptotique de cette quantité lorsque $n = m$, grâce à la formule de Stirling :

$$N_{n,n} = \binom{2n-2}{n-1} = \frac{(2n-2)!}{(n-1)!^2} \underset{n\to+\infty}{\sim} \frac{(2n-2)^{2n-2}\,\mathrm{e}^{-2(n-1)}\sqrt{2\pi(2n-2)}}{2(n-1)^{2(n-1)}\,\mathrm{e}^{-2(n-1)}\,\pi(n-1)} = \frac{2^{2n-2}}{\sqrt{\pi(n-1)}} \underset{n\to+\infty}{\sim} \frac{2^{2n-2}}{\sqrt{\pi n}}$$

Le nombre de chemins possibles est donc exponentiel en n, déja pour $n = 30$ un algorithme qui explore tous les chemins possibles est impraticable.

1. qui n'est pas une méthode de programmation...

16.2.3 Solutions aux sous-problèmes

Considérons une solution à notre problème, c'est-à-dire un chemin c dans A de la case $(0,0)$ à la case $(n-1,m-1)$, dont le poids est maximal. Supposons que ce chemin passe par la case (i,j). Le chemin se décompose en deux morceaux $(0,0) \overset{c_1}{\rightsquigarrow} (i,j) \overset{c_2}{\rightsquigarrow} (n-1,m-1)$. Les deux chemins c_1 et c_2 sont des chemins de poids maximaux de la case $(0,0)$ à (i,j) et de la case (i,j) à $(n-1,m-1)$.

La technique de démonstration est classique, et à retenir : supposons que c_1 ne soit pas optimal, il existe donc un chemin c_1' de poids strictement supérieur de la case $(0,0)$ à la case (i,j). Alors le chemin $(0,0) \overset{c_1'}{\rightsquigarrow} (i,j) \overset{c_2}{\rightsquigarrow} (n-1,m-1)$ est un chemin de poids strictement supérieur à c, ce qui est absurde. De même pour c_2.

Le problème d'optimisation d'un chemin de la case $(0,0)$ à une case (i,j) peut-être qualifié de *sous-problème* au problème initial, car la matrice à considérer est plus petite (en effet comme on se restreint aux déplacements $\downarrow$ et $\rightarrow$, tout se passe sur une matrice de taille $(i+1)\times(j+1)$).

Une solution au problème initial (sur la matrice $n \times m$) donne une solution à de multiples sous-problèmes : c'est une caractéristique que la programmation dynamique peut être utilisée pour résoudre le problème.

16.2.4 Une relation récursive pour le poids maximal d'un chemin

Notons $(p_{i,j})_{0\leq i<n,0\leq j<m}$ le poids maximal d'un chemin de $(0,0)$ à (i,j). Résoudre le problème consiste à trouver un chemin de poids $p_{n-1,m-1}$ de $(0,0)$ à $(n-1,m-1)$. Concentrons nous d'abord sur le calcul de $p_{n-1,m-1}$, et d'une manière générale de tous les $(p_{i,j})_{0\leq i<n,0\leq j<m}$. Remarquons que les $(p_{i,j})_{0\leq i<n,0\leq j<m}$ satisfont la relation suivante :

$$p_{i,j} = a_{i,j} + \begin{cases} 0 & \text{si } i=j=0 \\ p_{i,j-1} & \text{si } i=0, \text{ et } j>0 \\ p_{i-1,j} & \text{si } j=0, \text{ et } i>0 \\ \max\{p_{i-1,j}, p_{i,j-1}\} & \text{sinon.} \end{cases}$$

En effet :

- la relation pour les trois premiers points est évidente, car dans ce cas il n'y a qu'un seul chemin licite de la case $(0,0)$ à la case (i,j). On suppose dorénavant $i>0$ et $j>0$;
- À partir d'un chemin $\mathcal{C}$ menant à la case $(i-1,j)$, on en construit un menant à la case (i,j) via un déplacement $\downarrow$. En choisissant $\mathcal{C}$ de poids maximal $p_{i-1,j}$, on obtient $p_{i,j} \geq a_{i,j} + p_{i-1,j}$. Comme on obtient de manière symétrique $p_{i,j} \geq a_{i,j} + p_{i,j-1}$, on conclut que $p_{i,j} \geq a_{i,j} + \max\{p_{i-1,j}, p_{i,j-1}\}$;
- Réciproquement, tout chemin licite menant à la case (i,j) passe par $(i-1,j)$ ou $(i,j-1)$. Prenons-en un de poids maximal $p_{i,j}$, et supprimons le dernier mouvement, on obtient un chemin de poids $p_{i,j}-a_{i,j}$ menant à la case $(i-1,j)$ ou à la case $(i,j-1)$. Ainsi $\max\{p_{i-1,j}, p_{i,j-1}\} \geq p_{i,j} - a_{i,j}$.

Et la relation est démontrée. Cette relation fournit un algorithme récursif pour le calcul de $p_{n-1,m-1}$. Néanmoins cet algorithme revient à explorer tous les chemins possibles et a la même complexité que la recherche exhaustive.

16.2.5 Un calcul itératif des $p_{i,j}$

Il est toutefois possible de calculer tous les coefficients $p_{i,j}$ très simplement en utilisant une matrice P, de même taille que A, que l'on remplit en temps $O(nm)$ à l'aide des coefficients de la matrice A en suivant la relation précédente. Voici un code Python, où la matrice A est supposée stockée comme une liste de n listes de taille m :

```
def calcul_p(A):
    n,m=len(A), len(A[0])
    P = [[0]*m for _ in range(n)]
    P[0][0]=A[0][0]
    for i in range(1,n):
        P[i][0] = P[i-1][0]+A[i][0]
    for j in range(1,m):
        P[0][j]=P[0][j-1]+A[0][j]
    for i in range(1,n):
        for j in range(1,m):
            P[i][j] = A[i][j]+max(P[i-1][j],P[i][j-1])
    return P
```

L'appliquer à la matrice A de l'introduction donne la matrice P suivante :

$$A = \begin{pmatrix} 2 & 39 & 12 & 49 & 47 & 18 & 22 & 19 \\ 37 & 21 & 34 & 26 & 10 & 2 & 35 & 39 \\ 31 & 21 & 12 & 26 & 34 & 27 & 7 & 22 \\ 20 & 46 & 16 & 2 & 11 & 40 & 36 & 13 \\ 18 & 30 & 32 & 37 & 28 & 24 & 9 & 6 \end{pmatrix} \quad \text{et} \quad P = \begin{pmatrix} 2 & 41 & 53 & 102 & 149 & 167 & 189 & 208 \\ 39 & 62 & 96 & 128 & 159 & 169 & 224 & 263 \\ 70 & 91 & 108 & 154 & 193 & 220 & 231 & 285 \\ 90 & 137 & 153 & 156 & 204 & 260 & 296 & 309 \\ 108 & 167 & 199 & 236 & 264 & 288 & 305 & 315 \end{pmatrix}$$

16.2.6 Détermination d'une solution au problème initial

On sait maintenant calculer les $p_{i,j}$ dans un temps acceptable, il reste à déterminer un chemin de poids $p_{n-1,m-1}$ dans la matrice A, de la case $(0,0)$ à la case $(n-1,m-1)$.

Parmi plusieurs solutions, on propose la suivante : il suffit de remonter de la case $(n-1,m-1)$ à la case $(0,0)$ dans la matrice P. Depuis une case (i,j), on a le choix entre remonter à la case $(i-1,j)$ et remonter à la case $(i,j-1)$: il suffit de choisir la case $(i',j') \in \{(i-1,j),(i,j-1)\}$ telle que $p_{i',j'}$ est maximal : on obtient donc un chemin convenable avec une complexité supplémentaire $O(n+m)$.

Terminons cet exemple par une implémentation. On encode un chemin comme une suite de caractères « > » ou « v » indiquant à chaque étape s'il faut se diriger sur la case de droite ou celle d'en dessous. Comme on remonte depuis la case $(n-1,m-1)$, le chemin sera construit à l'envers, à la fin on retourne la liste (et on concatène les éléments en une chaîne de caractères, par pur souci esthétique). Voici le code :

```
def max_chemin(A):
    n,m=len(A), len(A[0])
    P=calcul_p(A)
    L = []
    i,j=n-1, m-1
    while i>0 and j>0:
        if P[i-1][j]>P[i][j-1]:
            L.append('v')
            i-=1
        else:
            L.append('>')
            j-=1
    while i>0:
        L.append('v')
        i-=1
    while j>0:
        L.append('>')
        j-=1
    return "".join(L[::-1])
```

Remarquez que les deux dernières boucles `while` servent simplement à remonter d'un des bords (gauche ou supérieur) à la case initiale, et seule l'une des deux est utile. Appliquons l'algorithme à la matrice A de l'exemple :

```
>>> max_chemin(A)
'>>>>vv>v>>v'
```

Autrement dit, un chemin de poids maximal est le suivant :

$$A = \begin{pmatrix} \mathbf{2} & \mathbf{39} & \mathbf{12} & \mathbf{49} & \mathbf{47} & 18 & 22 & 19 \\ 37 & 21 & 34 & 26 & \mathbf{10} & 2 & 35 & 39 \\ 31 & 21 & 12 & 26 & \mathbf{34} & \mathbf{27} & 7 & 22 \\ 20 & 46 & 16 & 2 & 11 & \mathbf{40} & \mathbf{36} & \mathbf{13} \\ 18 & 30 & 32 & 37 & 28 & 24 & 9 & \mathbf{6} \end{pmatrix}$$

Le lecteur non convaincu vérifiera que ce chemin est de poids 315, ce qui correspond à $p_{4,7} = p_{n-1,m-1}$.

16.2.7 Une parenthèse sur une approche gloutonne

Pour construire un chemin de poids élevé, on pourrait utiliser l'approche gloutonne suivante. On part de la case en haut à gauche, et à chaque étape, on choisit d'aller vers la case en dessous ou à droite possédant le plus grand nombre : c'est le choix localement optimal. Sur l'exemple précédent, le chemin produit est le suivant :

$$\begin{pmatrix} \mathbf{2} & \mathbf{39} & 12 & 49 & 47 & 18 & 22 & 19 \\ 37 & \mathbf{21} & \mathbf{34} & \mathbf{26} & 10 & 2 & 35 & 39 \\ 31 & 21 & 12 & \mathbf{26} & \mathbf{34} & \mathbf{27} & 7 & 22 \\ 20 & 46 & 16 & 2 & 11 & \mathbf{40} & \mathbf{36} & \mathbf{13} \\ 18 & 30 & 32 & 37 & 28 & 24 & 9 & \mathbf{6} \end{pmatrix}$$

On peut vérifier que ce chemin a un poids 304, ce qui n'est pas optimal. Néanmoins, la complexité du calcul d'un tel chemin est $O(n+m)$, contrairement au calcul en $O(nm)$ d'un chemin optimal par programmation dynamique. Cette observation est générale : les algorithmes gloutons sont plus rapides que leurs équivalents dynamiques.

Visuellement, on peut comparer les approches gloutonnes et dynamiques, voir la figure suivante. Un algorithme dynamique procède « de bas en haut », un algorithme glouton ramène un problème à un problème plus petit via le choix glouton.

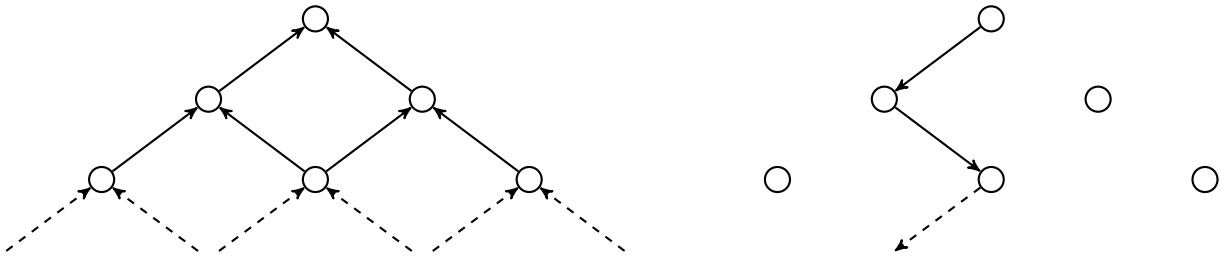


FIGURE 16.1 – Comparaison des stratégies dynamique et gloutonne

16.3 Principes de la programmation dynamique

L'exemple précédent est typique d'une résolution de problème par programmation dynamique. Donnons un résumé de la démarche, dans un cadre un peu plus abstrait.

16.3.1 La démarche d'une résolution de problème par programmation dynamique

On se donne donc $f : \mathcal{U} \to \mathbb{Z}$, et on cherche à déterminer x tel que $f(x) = \max_{y\in\mathcal{U}}\{f(y)\}$ (la démarche est la même si on cherche à minimiser f sur $\mathcal{U}$). Dans la suite, on notera $M_{f,\mathcal{U}}$ la quantité $\max_{y\in\mathcal{U}}\{f(y)\}$.

Il y a quatre étapes dans la résolution d'un tel problème par programmation dynamique.

1. Identifier une *sous-structure optimale* : c'est un indice de l'application de la programmation dynamique. Il s'agit de voir que si l'on connaît $x \in \mathcal{U}$ tel que $f(x) = M_{f,\mathcal{U}}$, alors de x on déduit des solutions $(x_i)_{i\in I}$ à des sous-problèmes de la forme « trouver $x_i \in \mathcal{U}_i$, tel que $f_i(x_i) = \max_{y\in\mathcal{U}_i}\{f(y)\} = M_{f_i,\mathcal{U}_i}$ ». Les problèmes associés aux $(f_i,\mathcal{U}_i)$ doivent être plus simples à résoudre que le problème associé à $(f,\mathcal{U})$. Dans l'exemple précédent, les problèmes $(f_i,\mathcal{U}_i)$ étaient des déterminations de chemins de poids maximal dans des matrices plus petites.
2. Déduire de la sous-structure optimale une relation récursive permettant le calcul de $M_{f,\mathcal{U}}$ à partir de certains $M_{f_i,\mathcal{U}_i}$. Dans l'exemple précédent, on a exhibé une relation entre $p_{n-1,m-1}$, $p_{n-2,m-1}$ et $p_{n-1,m-2}$.

   Ce qui distingue une résolution par programmation dynamique d'un algorithme « diviser pour régner » est le fait que les calculs des différents $M_{f_i,\mathcal{U}_i}$ ne sont pas *du tout* indépendants : écrire un algorithme récursif calculant tel quel $M_{f,\mathcal{U}}$ mène en général à une solution très coûteuse, ce qui peut être résumé au travers du schéma de la figure 16.1 à gauche, calqué sur le problème étudié précédemment.
3. Pour pallier le problème évoqué au point (2), on calcule alors les $M_{f_i,\mathcal{U}_i}$ utiles (y compris $M_{f,\mathcal{U}}$) itérativement, en faisant usage d'un tableau pour stocker tous ces éléments. On peut parfois se contenter de n'en stocker que certains, par exemple dans le problème précédent un espace $O(n)$ en plus de la matrice A (au lieu de $O(nm)$) suffirait [2] pour calculer $p_{n-1,m-1}$. En toute généralité, il peut être commode d'utiliser un dictionnaire pour stocker ces $M_{f_i,\mathcal{U}_i}$.
4. Enfin, on modifie légèrement le calcul des $M_{f_i,\mathcal{U}_i}$ pour obtenir en même temps [3] pour tout i un x_i satisfaisant $f_i(x_i) = M_{f_i,\mathcal{U}_i}$. En général cette étape n'est pas difficile.

2. Comment procéder ?

3. Dans le problème précédent, on a choisi de ne calculer un chemin convenable qu'après le calcul des $p_{i,j}$, mais on aurait pu par exemple stocker en parallèle des $p_{i,j}$ (dans une troisième matrice) le dernier déplacement à effectuer pour arriver en (i,j) en suivant un chemin optimal.

16.3.2 Une parenthèse sur les problèmes de combinatoire

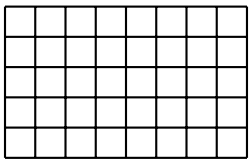
La technique évoquée ci-dessus pour résoudre un problème d'optimisation s'applique aussi pour la résolution de certains problèmes de combinatoire. Dans ce cas, on cherche plutôt à calculer la taille d'un certain ensemble $\mathcal{U}$. La démarche ressemble à celle ci-dessus :

1. On partitionne l'ensemble $\mathcal{U}$ en sous-ensembles disjoints $(\widetilde{\mathcal{U}_i})_i$, les $(\mathcal{U}_i)$ étant des ensembles associés à des « sous-problèmes combinatoires » et les $(\widetilde{\mathcal{U}_i})$ des ensembles en bijection avec les $(\mathcal{U}_i)$: une légère modification d'un élément de $\mathcal{U}_i$ donne un élément de $\widetilde{\mathcal{U}_i}$.
2. Écrire que $\mathcal{U} = \cup_i \widetilde{\mathcal{U}_i}$ (l'union étant disjointe) fournit une relation de récurrence permettant de calculer $|\mathcal{U}|$, à savoir $|\mathcal{U}| = \sum_i |\mathcal{U}_i|$.
3. On calcule plutôt $|\mathcal{U}|$ itérativement, en faisant usage d'un tableau.

Pour résumer, la résolution d'un problème de combinatoire suit essentiellement les points (1) à (3) évoqués pour la résolution d'un problème d'optimisation par programmation dynamique, le point (4) n'ayant pas de sens ici. Détaillons rapidement deux exemples.

Nombre de chemins sur un quadrillage

On considère un quadrillage de taille $n \times m$, comme celui-ci dessous :



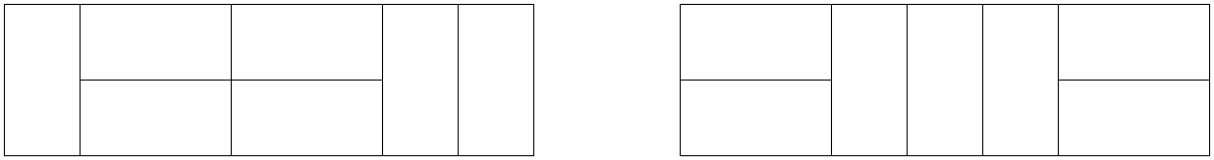
On cherche le nombre de chemins sur partant du coin en haut à gauche jusqu'au coin en bas à droite, en suivant seulement les directions [4] $\rightarrow$ et $\downarrow$. Résolvons le rapidement :

1. On peut indexer les points de la grille de $(0,0)$ (en haut à gauche) à (n,m) (en bas à droite). Notons $\mathcal{C}_{i,j}$ l'ensemble des chemins de $(0,0)$ à (i,j), utilisant seulement les déplacements autorisés. Alors pour tout $i,j \geq 0$, on a
$$\mathcal{C}_{i,j} = \widetilde{\mathcal{C}_{i-1,j}} \cup \widetilde{\mathcal{C}_{i,j-1}}$$
où $\widetilde{\mathcal{C}_{i-1,j}}$ est l'ensemble des chemins de $\mathcal{C}_{i-1,j}$, complétés par le segment $(i-1,j) \rightarrow (i,j)$, et de même pour $\widetilde{\mathcal{C}_{i,j-1}}$. On convient que $\mathcal{C}_{-1,j} = \mathcal{C}_{i,-1} = \emptyset$ et que $C_{0,0}$ contient comme unique élément le chemin réduit au point $(0,0)$.
2. En notant $N_{i,j} = |\mathcal{C}_{i,j}|$, on a donc la relation de récurrence $N_{i,j} = N_{i-1,j} + N_{i,j-1}$, valable pour $i,j > 0$, sinon $N_{i,0} = N_{0,j} = 1$.
3. On peut donc tabuler les $N_{i,j}$ dans un tableau de taille $(n+1) \times (m+1)$, car c'est $N_{n,m}$ qui nous intéresse.

Le lecteur pourra vérifier qu'il y a 1287 chemins convenables, pour l'exemple [5] de la grille de taille 5×8.

Un problème de pavage

On considère un rectangle de taille $2 \times n$, et on s'intéresse aux *pavages* de ce rectangle par des dominos 1×2. La figure suivante montre deux exemples de pavage d'un rectangle 2×7.



Notons F_n le nombre de pavages possibles d'un rectangle de taille $2 \times n$. Cherchons une relation de récurrence nous permettant de calculer F_n efficacement. Supposons $n \geq 2$ et considèrons le domino occupant le coin en haut à droite du rectangle.

— si ce domino est placé verticalement (comme dans le pavage de gauche dans la figure ci-dessus), alors il reste à paver un rectangle $2 \times (n-1)$: le nombre de tels pavages est donc F_{n-1} ;

4. Oui, ce problème de combinatoire ressemble fortement au problème d'optimisation vu précédemment : c'est fait exprès !
5. Comme déja évoqué, il y a en fait $\binom{n+m}{n}$ chemins possibles, et $1287 = \binom{13}{5}$...

— sinon, le domino est placé horizontalaement (comme dans le pavage de droite). Nécessairement, un autre domino horizonal est placé en dessous, il reste donc à paver un rectangle de taille $2 \times (n-2)$, et il y a F_{n-2} tels pavages.

Ainsi, $(F_n)_n$ satisfait la relation de récurrence $F_n = F_{n-1} + F_{n-2}$ pour $n \geq 2$ (on reconnaît la suite de Fibonacci...), avec conditions initiales $F_0 = F_1 = 1$. Comme on l'a vu au chapitre 3, il vaut mieux faire usage d'un tableau que de récursivité pour calculer efficacement F_n.

Remarque 16.1. *Ce problème était facile. Le lecteur pourra chercher le nombre de pavages possibles d'un rectangle* 3×30 *avec des dominos* 1×2*, c'est moins évident !*

Utilisation d'un dictionnaire

On peut garder la formulation récursive, en faisant en sorte de *mémoïzer* les résultats obtenus. En faisant usage d'un dictionnaire, on garde une trace de toutes les valeurs calculées, ce qui évite l'explosion de la complexité due au chevauchement des sous-problèmes. Voici à titre d'exemple, une fonction `fibo`, calculant de manière efficace le terme F_n de la suite de Fibonacci, définie par $F_0 = F_1 = 1$, et $F_n = F_{n-1} + F_{n-2}$ pour $n \geq 2$:

```
def fibo(n):
    D={0:1, 1:1}
    def f(n):
        if not n in D.keys():
            D[n]=f(n-1)+f(n-2)
        return D[n]
    return f(n)
```

La fonction `f` interne est récursive. Elle fait usage d'un dictionnaire, défini en dehors [6]. Elle n'est pas de complexité exponentielle, contrairement à la version naïve !

```
>>> fibo(100)
573147844013817084101
```

16.4 Autres exemples de résolution par programmation dynamique

16.4.1 Le problème de la plus longue sous séquence commune

Si s est une chaîne de caractère, on appelle sous-séquence de s une chaîne de caractères x dont les caractères se retrouvent dans s, à des indices strictement croissants. Formellement, si $s = s_0 s_1 \cdots s_{n-1}$, une sous-séquence de s de taille ℓ est une chaîne $x = x_0 \cdots x_{\ell-1}$ telle qu'il existe [7] $\varphi : [\![0, \ell-1]\!] \to [\![0, n-1]\!]$ strictement croissante, avec $x_i = s_{\varphi(i)}$ pour tout $i \in [\![0, \ell-1]\!]$.

Le problème de la plus longue sous-séquence commune (PLSSC) consiste à trouver un mot x qui est une sous-séquence commune maximale à deux mots s et t. Par exemple, une sous-séquence commune maximale à « arythmie » et « rhomboédrique » est « rhmie », de longueur 5. Ce problème a des applications pratiques, notamment en génétique : la proximité de deux individus peut être évaluée en calculant une sous-séquence commune [8] à deux séquences d'ADN prises sur les individus.

Sous-structure optimale. Notons n et m les longueurs de s et t. Pour $0 \leq i \leq n$ et $0 \leq j \leq m$, on note $\ell_{i,j}$ la longueur d'une plus longue sous-séquence commune aux préfixes de tailles i et j de s et t. Ce qui nous intéresse est $\ell_{n,m}$, et une sous-séquence associée. Supposons que l'on connaisse une sous-séquence commune x de longueur $\ell_{n,m}$, supposé strictement positif.

— si les derniers caractères de s et t sont les mêmes, alors x termine par ce caractère (sinon on pourrait le rajouter !). Mais alors x privé de son dernier caractère est une sous séquence commune à s et t tous deux privés de leur dernier caractère (notés s' et t' dans la suite), et c'est même une plus longue sous-séquence commune à ces deux mots (l'argument est classique : si ce n'était pas le cas, on pourrait trouver une sous-séquence commune à s et t plus longue que x en rajoutant le dernier caractère commun à s et t à une sous-séquence commune maximale de s' et t').

— si les derniers caractères de s et t diffèrent, alors x est une sous-séquence commune à s et t' ou à s' et t (voire au deux), et c'est même une plus longue sous-séquence commune de manière évidente.

Nous avons exhibé une sous-structure optimale !

6. Une erreur courante serait de le définir dans `f`, ce qui ne fonctionnerait évidemment pas car il serait réinitialisé à chaque appel.
7. En prenant $\ell = 0$, on obtient l'application vide, la chaîne de caractères vide est bien une sous-séquence de toute chaîne de caractères.
8. La *distance d'édition* entre deux séquences est également intéressante, voir la suite.

Une relation de récurrence. La discussion précédente nous fournit une relation de récurrence sur les $\ell_{i,j}$, à savoir :

$$\ell_{i,j} = \begin{cases} 0 & \text{si } i=0 \text{ ou } j=0\,; \\ 1+\ell_{i-1,j-1} & \text{si les préfixes de tailles } i \text{ et } j \text{ de } s \text{ et } t \text{ terminent par la même lettre;} \\ \max\{\ell_{i-1,j}, \ell_{i,j-1}\} & \text{sinon.} \end{cases}$$

Calcul itératif des $\ell_{i,j}$ et construction d'une sous-séquence commune. Pour calculer les $(\ell_{i,j})_{0\leq i\leq n, 0\leq j\leq m}$, on procède itérativement en remplissant une matrice `L` de taille $(n+1)\times(m+1)$. De manière similaire au problème du chemin de poids maximal, il suffit de remonter depuis la case (n,m) jusqu'à la case $(0,0)$ (ou plus simplement à un bord supérieur ou inférieur du tableau) pour construire une sous-séquence commune, à l'envers. Voici une implémentation :

```
def plssc(s,t):
    n,m=len(s), len(t)
    L = [[0]*(m+1) for _ in range(n+1)]
    for i in range(1,n+1):
        for j in range(1,m+1):
            if s[i-1]==t[j-1]:
                L[i][j]=1+L[i-1][j-1]
            else:
                L[i][j]=max(L[i-1][j], L[i][j-1])
    x=[]
    i, j, k=n,m,L[n][m]
    while k>0:
        if L[i][j]==L[i-1][j]:
            i-=1
        elif L[i][j]==L[i][j-1]:
            j-=1
        else:
            x.append(s[i-1])
            i-=1
            j-=1
            k-=1
    return "".join(x[::-1])
```

Une fois les $\ell_{i,j}$ calculés, on connaît la longueur d'une plus longue sous-séquence commune. On reconstruit alors la sous-séquence commune, à l'envers. Pour cela, on remonte depuis la case (n,m). Si $\ell_{i,j} = \ell_{i-1,j}$ ou $\ell_{i,j-1}$, on peut remonter d'un cran vers le haut ou vers la gauche. Sinon, on a trouvé un nouveau caractère, et on remonte en diagonale à la case $(i-1, j-1)$. Testons :

```
>>> plssc('arythmie', 'rhomboedrique')
'rhmie'
```

Complexité. La détermination des $\ell_{i,j}$ se fait en temps $O(nm)$, alors que la construction d'une plus longue sous-séquence commune ne prend qu'un temps $O(n+m)$.

16.4.2 Distance d'édition

Définition du problème. Un problème important en génétique, évoqué plus haut, est le calcul de la *distance d'édition* entre deux chaînes de caractères. On convient que sur une chaîne de caractères, on peut effectuer les opérations *d'édition* suivantes :

— insérer un caractère à un endroit donné ;

— supprimer un caractère donné ;

— modifier un caractère en un autre.

La distance d'édition (ou distance de Levenshtein) entre deux chaînes s et t est définie par le nombre minimal $d(s,t)$ d'opérations [9] à effectuer pour passer de s à t. Il est facile de vérifier que d est bien une distance au sens mathématique, c'est-à-dire que pour toutes chaînes s et t :

— $d(s,s) = 0$;

— $d(s,t) = d(t,s)$;

— $d(s,t) \leq d(s,u) + d(t,u)$ pour toute chaîne u (inégalité triangulaire).

9. On peut généraliser en introduisant un coût différent, par exemple en considérant qu'une insertion / suppression coûte plus cher qu'une modification. L'approche proposée s'étend facilement.

On peut montrer que la distance entre les chaînes $s =$ « ACTGTAA » et $t =$ « AACTGC » est 4. Une suite possible d'opérations d'édition permettant de passer de s à t est la suivante :

$$\text{ACTGTAA} \underset{\text{suppression}}{\longrightarrow} \text{ACTGTA} \underset{\text{suppression}}{\longrightarrow} \text{ACTGT} \underset{\text{modification}}{\longrightarrow} \text{ACTGC} \underset{\text{insertion}}{\longrightarrow} \text{AACTGC}$$

Sous-structure optimale. Comme souvent lorsqu'il est question de distances, une sous-structure optimale est facile à observer. Si on connaît une suite d'opérations possibles pour passer de s à t de longueur minimale, et que cette suite passe par la chaîne u (disons $s \rightsquigarrow u \rightsquigarrow t$, avec $\rightsquigarrow$ signifiant une certaine succession d'opérations d'édition), alors les deux suites d'opérations $s \rightsquigarrow u$ et $u \rightsquigarrow t$ sont optimales : en effet, s'il existait par exemple une suite plus courte d'opérations de s à u, disons $s \Rightarrow u$, alors la suite $s \Rightarrow u \rightsquigarrow t$ serait plus courte que la suite initiale : c'est absurde. De même pour $t \rightsquigarrow u$.

Relation de récurrence. De même que pour le problème de la PLSSC, on peut introduire $d_{i,j}$, distance d'édition entre les préfixes de taille i de s et j de t, pour $0 \leq i \leq n$ et $0 \leq j \leq m$, avec n et m les tailles de s et t. On a alors la relation suivante :

$$d_{i,j} = \begin{cases} i & \text{si } j = 0 \\ j & \text{si } i = 0 \\ d_{i-1,j-1} & \text{si } s_{i-1} = t_{j-1} \\ 1 + \min(d_{i-1,j}, d_{i,j-1}, d_{i-1,j-1}) & \text{sinon.} \end{cases}$$

Preuve : les trois premiers points sont assez évidents, prouvons le dernier.

- Déja, $d_{i,j} \leq 1 + \min(d_{i-1,j}, d_{i,j-1}, d_{i-1,j-1})$. En effet, on a par exemple $d_{i,j} \leq 1 + d_{i-1,j}$, car à partir d'une suite d'opérations de longueur $d_{i-1,j}$ permettant de passer de $s[:i-1]$ à $t[:j]$, on en obtient une de longueur $1 + d_{i-1,j}$ de $s[:i]$ à $t[:j]$ en lui rajoutant au début la suppression du dernier caractère de $s[:i]$. On montre de même que $d_{i,j} \leq 1 + d_{i,j-1}$ et $d_{i,j} \leq 1 + d_{i-1,j-1}$, d'où la conclusion.
- Pour montrer l'inégalité inverse, considérons une suite d'opérations de longueur $d_{i,j}$ menant de $s[:i]$ à $t[:j]$. Le dernier caractère de $t[:j]$ apparaît dans cette suite après une insertion, une modification, ou une suppression. Quite à modifier un peu la suite d'opérations, supposons que cette opération est faite en dernier. Si par exemple c'est une insertion, la dernière étape passe de $t[:j-1]$ à $t[:j]$, en enlevant cette étape on obtient une suite de longueur $d_{i,j} - 1$ de $s[:i]$ à $t[:j-1]$. Et donc, $\min(d_{i-1,j}, d_{i,j-1}, d_{i-1,j-1}) \leq d_{i,j-1} \leq d_{i,j} - 1$. De même s'il s'agit d'une insertion ou d'une modification.

Exercice 4. Écrire une fonction calculant la distance d'édition entre deux chaînes s et t. Avec n et m les longueurs de ces chaînes, votre fonction calculera tous les $(d_{i,j})_{0 \leq i \leq n, 0 \leq j \leq m}$. Voici la matrice obtenue avec les chaînes « ACTGTAA » et « AACTGC » :

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 0 & 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 1 & 1 & 2 & 3 & 4 \\ 3 & 2 & 2 & 2 & 1 & 2 & 3 \\ 4 & 3 & 3 & 3 & 2 & 1 & 2 \\ 5 & 4 & 4 & 4 & 3 & 2 & 2 \\ 6 & 5 & 4 & 5 & 4 & 3 & 3 \\ 7 & 6 & 5 & 5 & 5 & 4 & 4 \end{pmatrix}$$

Quelle est la complexité de votre fonction ?

Exercice 5. *Plus difficile.* Écrire une fonction `affiche_transformations(s,t)` affichant une suite optimale de tranformations permettant de passer de s à t. Par exemple :

```
>>> affiche_transformations('ACTGTAA', 'AACTGC')
départ, s =  ACTGTAA. But : t =  AACTGC.
suppression :  ACTGTA
suppression :  ACTGT
modification :  ACTGC
insertion :  AACTGC
```

Indication : après calcul de la matrice des distances, utiliser une boucle `while` comme pour le calcul de la PLSSC. Outre les indices i et j, maintenir une chaîne de caractères `reste` initialement vide, telle que les opérations restantes soient le passage de `s[:i]+reste` à `t[:j]+reste` (on rappelle que `+` est la concaténation de chaînes).

16.4.3 L'algorithme de Floyd-Warshall

Rappel sur les graphes. On considère dans cette section des graphes pondérés orientés. Un tel graphe $G = (V, E, \omega)$ sera ici considéré sans boucles (c'est-à-dire que les éléments de E sont des couples d'éléments distincts de V). La fonction de pondération $\omega : E \to \mathbb{R}$ n'est pas supposée positive comme dans l'algorithme de Dijkstra. Elle s'étend en une fonction définie sur V^2 en posant $\omega_{i,j} = \begin{cases} 0 & \text{si } i = j \\ +\infty & \text{si } (i,j) \notin E \end{cases}$.

Un tel graphe sera représenté ici via sa matrice d'adjacence $A = (\omega_{i,j})_{0 \leq i,j \leq n-1}$ avec $n = |V|$ (en Python, $+\infty$ s'encode via `float('inf')`, ce flottant infini est plus grand que tout flottant / entier). Le poids d'un chemin c dans le graphe, noté encore $\omega(c)$, est la somme des poids des arcs du chemin.

Chemins de plus petits poids entre tout couple de sommets. Le but de cette section est de calculer, pour tout $(i,j) \in V^2$, la quantité $p_{i,j} = \inf\{\omega(c) \mid c \text{ chemin de } i \text{ à } j\}$. Notons que si j n'est pas accessible depuis i, on a $p_{i,j} = +\infty$, et si, sur un chemin de i à j, il existe un circuit (chemin ayant même sommet aux deux extrémités) de poids total strictement négatif, alors $p_{i,j} = -\infty$ (en effet, il suffit de boucler sur le chemin pour obtenir un poids arbitrairement petit). On suppose dans la suite qu'il n'y a pas de tels circuits, mais l'algorithme que l'on va proposer est capable de les détecter.

Sous structure optimale. On est ici en présence d'un problème où il s'agit de trouver des plus courts chemins (au sens de la somme des poids). Comme on l'a dit plus haut, dans ce genre de problème réside une sous-structure optimale évidente : les techniques de programmation dynamique peuvent s'appliquer.

Idée de l'algorithme de Floyd-Warshall. On a dit que l'on supposait l'absence de circuit de poids total strictement négatif dans le graphe. Par suite, un chemin de plus petit poids entre i et j peut être supposé ne passer que par des sommets distincts. En effet, on pourrait supprimer les circuits éventuels (nécessairement de poids nuls) d'un plus court chemin de i à j pour obtenir un chemin constitué de sommets distincts. L'idée de l'algorithme consiste à calculer les poids des chemins entre tout couple i et j, mais dont les sommets intermédiaires sont dans un ensemble qui croît au fur et à mesure.

Suite de matrices. On pose pour tout $k \in [\![0, n]\!]$:

$P_k = (p^k_{i,j})_{(i,j) \in [\![0,n-1]\!]^2}$ avec $p^k_{i,j} = \inf\{\omega(c) \mid c \text{ chemin de } i \text{ à } j \text{ dont les sommets intermédiaires sont dans } [\![0, k-1]\!]\}$

Par sommet intermédiaire, on entend tous les sommets sauf les extrémités i et j. On a donc $P_0 = A$ (matrice d'adjacence) et on veut calculer P_n. Ces matrices sont à valeurs dans $\mathbb{R} \cup \{+\infty\}$, (la valeur $-\infty$ ne se produit pas s'il n'y a pas de circuit de poids total strictement négatif). La proposition suivante indique comment calculer M^{k+1} à partir de M^k :

Proposition 16.2. *En l'absence de circuit de poids strictement négatif dans le graphe, on a pour tout* $(i,j,k) \in [\![0, n-1]\!]^3$ *:* $p^{k+1}_{i,j} = \min(p^k_{i,j}, p^k_{i,k} + p^k_{k,j})$.

Démonstration.
- S'il n'y a pas de chemin entre i et j dont les sommets intermédiaires sont dans $[\![0,k]\!]$, $p^k_{i,j} = p^{k+1}_{i,j} = +\infty$, et l'un des deux $p^k_{i,k}$ ou $p^k_{k,j}$ vaut aussi $+\infty$ (car sinon on aurait un chemin entre i et k et un autre entre k et j, dont la concaténation fournirait un chemin entre i et j).
- Sinon, un chemin de poids minimal entre i et j ne passant que par des sommets de $[\![0,k]\!]$ peut être supposé ne passer qu'au plus une fois par k, comme on l'a dit plus haut. Si ce chemin ne passe pas par k, on a $p^{k+1}_{i,j} = p^k_{i,j}$, et sinon le chemin se décompose en $i \overset{c_1}{\rightsquigarrow} k \overset{c_2}{\rightsquigarrow} j$ avec c_1 et c_2 des chemins dont les sommets intermédiaires sont dans $[\![0, k-1]\!]$, ainsi $p^{k+1}_{i,j} = p^k_{i,k} + p^k_{j,k}$.

□

On obtient ainsi un algorithme de complexité $O(n^3)$ pour le calcul de tous les $p_{i,j}$. La remarque suivante indique que l'on peut se contenter de mettre à jour une unique matrice :

Remarque 16.3. *En l'abscence de circuit de poids total négatif, on a également, pour tout* $(i,j,k) \in [\![0, n-1]\!]^3$ *:* $p^{k+1}_{i,j} = \min(p^k_{i,j}, p^k_{i,k} + p^k_{k,j}) = \min(p^k_{i,j}, p^{k+1}_{i,k} + p^k_{k,j}) = \min(p^k_{i,j}, p^k_{i,k} + p^{k+1}_{k,j}) = \min(p^k_{i,j}, p^{k+1}_{i,k} + p^{k+1}_{k,j})$.

En effet, remplacer $p^k_{i,k}$ par $p^{k+1}_{i,k}$ ne change rien, car il n'y a pas de circuit de poids strictement négatif bouclant sur k. L'algorithme de Floyd-Warshall est donc facile à écrire, voir algorithme 16.4.

Complexité. Elle est clairement de $O(n^3)$, en temps comme en mémoire.

Algorithme 16.4 : Algorithme de Floyd-Warshall

Entrée : Un graphe pondéré $G = (V, E, \omega)$ donné par sa matrice d'adjacence A
Sortie : La matrice $(p_{i,j})_{0 \leq i,j < n}$ des plus courtes distances entre deux sommets quelconques du graphe
$P \leftarrow \text{copie}(A)$;
pour *tout k entre 0 et $n-1$ faire* **faire**
  pour *tout i entre 0 et $n-1$ faire* **faire**
    pour *tout j entre 0 et $n-1$ faire* **faire**
      $p_{i,j} \leftarrow \min(p_{i,j}, p_{i,k} + p_{k,j})$
Renvoyer A

Détection des circuits de poids total négatif. S'il existe un circuit de poids total négatif, prenons en un sans sommet en double excepté les extrémités. Soit k le sommet aux extrémités, que l'on peut supposer supérieur aux autres sommets du circuit (c'est possible quite à « faire tourner » le chemin). Notons $i < k$ un autre sommet quelconque du circuit. Juste avant l'itération k de l'algorithme, on a donc $p_{k,i} + p_{i,k} < 0$, et donc à l'itération k, $p_{k,k}$ sera strictement négatif, et ce sera encore le cas à la fin de l'algorithme. Conclusion : il suffit de tester l'existence d'un élément diagonal strictement négatif à la fin de l'algorithme pour tester l'existence d'un circuit de poids total strictement négatif.

Calcul effectif des chemins de plus petits poids. Parmi d'autres méthodes, on peut calculer une *matrice de liaison* $\Pi = (\pi_{i,j})_{0 \leq i,j \leq n-1}$, avec $\pi_{i,j}$ défini comme le prédécesseur de j dans un chemin de plus petit poids de i à j (et n'est pas défini s'il n'existe pas de tel chemin). Pour calculer Π, il suffit d'initialiser $\pi_{i,j}$ à i s'il existe un arc entre i et j (et une valeur arbitraire sinon). Ensuite, dans l'algorithme, lorsqu'on a $p_{i,k} + p_{k,j} < p_{i,j}$, on réalise l'affectation $\pi_{i,j} \leftarrow \pi_{k,j}$ parallèlement à $p_{i,j} \leftarrow p_{i,k} + p_{k,j}$.

Exemple. La figure 16.2 présente un tel exemple. On remarque que le graphe possède un circuit de poids nul, mais pas de circuit de poids total strictement négatif. On montre le calcul de Π en parallèle de la matrice P.

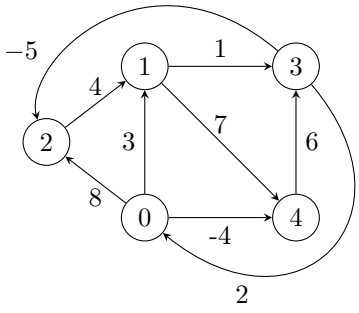
Code Python. Le code suivant renvoie les deux matrices P et Π.

```
def floyd_warshall(G):
    n=len(G)
    P = [L[:] for L in G] #copie de la matrice d'adjacence
    Pi = [[i for _ in range(n)] for i in range(n)] #matrice de liaison : on pose Pi[i,j]=i même si
                                                   #l'arc i -> j n'existe pas
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if P[i][k]+P[k][j]<P[i][j]:
                    P[i][j] = P[i][k]+P[k][j]
                    Pi[i][j] = Pi[k][j]
    return P, Pi
```

Vérifions :

```
>>> inf=float('inf')
>>> G = [[0,  3,  8,  inf,  -4], [inf,  0,  inf,  1,  7],[inf,  4,  0,  inf,  inf],
         [2,  inf,  -5,  0,  inf], [inf,  inf,  inf,  6,  0]] #graphe exemple
>>> P,Pi=floyd_warshall(G)
>>> np.array(P) #en Numpy simplement pour l'affichage.
array([[ 0,  1, -3,  2, -4],
       [ 3,  0, -4,  1, -1],
       [ 7,  4,  0,  5,  3],
       [ 2, -1, -5,  0, -2],
       [ 8,  5,  1,  6,  0]])
>>> np.array(Pi) #seules les valeurs hors diagonale sont pertinentes.
array([[0, 2, 3, 4, 0],
       [3, 1, 3, 1, 0],
       [3, 2, 2, 1, 0],
       [3, 2, 3, 3, 0],
       [3, 2, 3, 4, 4]])
```

Exercice 6. Écrire une fonction `chemin(Pi, i, j)` prenant en entrée une matrice de liaison, deux sommets `i` et `j`, et renvoyant un chemin de plus petit poids de `i` à `j`, sous la forme d'une liste débutant par `i` et terminant par `j`. On impose une complexité linéaire en la taille de la sortie. Par exemple avec la matrice `Pi` précédente :



$$P_0 = A = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi = \begin{pmatrix} . & 0 & 0 & . & 0 \\ . & . & . & 1 & 1 \\ . & 2 & . & . & . \\ 3 & . & 3 & . & . \\ . & . & . & 4 & . \end{pmatrix}$$

$$P_1 = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi = \begin{pmatrix} . & 0 & 0 & . & 0 \\ . & . & . & 1 & 1 \\ . & 2 & . & . & . \\ 3 & 0 & 3 & . & 0 \\ . & . & . & 4 & . \end{pmatrix}$$

$$P_2 = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi = \begin{pmatrix} . & 0 & 0 & 1 & 0 \\ . & . & . & 1 & 1 \\ . & 2 & . & 1 & 1 \\ 3 & 0 & 3 & . & 0 \\ . & . & . & 4 & . \end{pmatrix}$$

$$P_3 = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi = \begin{pmatrix} . & 0 & 0 & 1 & 0 \\ . & . & . & 1 & 1 \\ . & 2 & . & 1 & 1 \\ 3 & 2 & 3 & . & 0 \\ . & . & . & 4 & . \end{pmatrix}$$

$$P_4 = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi = \begin{pmatrix} . & 0 & 3 & 1 & 0 \\ 3 & . & 3 & 1 & 0 \\ 3 & 2 & . & 1 & 0 \\ 3 & 2 & 3 & . & 0 \\ 3 & 2 & 3 & 4 & . \end{pmatrix}$$

$$P_5 = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi = \begin{pmatrix} . & 2 & 3 & 4 & 0 \\ 3 & . & 3 & 1 & 0 \\ 3 & 2 & . & 1 & 0 \\ 3 & 2 & 3 & . & 0 \\ 3 & 2 & 3 & 4 & . \end{pmatrix}$$

FIGURE 16.2 – Calcul de chemins de plus petits poids dans un graphe, à l'aide de l'algorithme de Floyd-Warshall

```
>>> chemin(Pi, 0, 2)
[0, 4, 3, 2]
```

Septième partie

Enjeux contemporains : introduction à l'intelligence artificielle et à la théorie des jeux

Chapitre 17

Apprentissage supervisé ou non

Dans les journaux ou dans les romans de science fiction, le terme « intelligence artificielle » relève du mystère et a une vague connotation ésotérique, qui nourrit beaucoup de fantasmes ! Il ne s'agit pourtant pas « d'apprendre à l'ordinateur à penser par lui-même », mais simplement de résoudre des problèmes *a priori* non évidents, que l'humain effectue sans peine, comme reconnaître des lettres, des chiffres ou un chien sur une image, distinguer des sons, faire de la traduction automatique, etc... Ce chapitre donne des idées simples mais efficaces pour aborder les plus faciles de ces problèmes.

17.1 Algorithme des k plus proches voisins

17.1.1 Un exemple concret

Pour reconnaître une plaque minéralogique, il est nécessaire de savoir reconnaître un chiffre entre 0 et 9 dans une image x (comme celle de la figure 17.1, à droite). Une idée possible consiste à partir d'une banque [1] d'images représentant des chiffres (on sait quels chiffres sont représentés), comme celle de la figure 17.1, à gauche. On cherche alors les k images les plus proches (au sens d'une certaine distance) de l'image x dans la banque. Par exemple, avec $k = 5$, dans la banque précédente, on trouve 4 sept et un 9 en plus proches voisins : on peut détecter automatiquement que c'est un 7 ! On parle ici *d'apprentissage surpervisé*, car on part d'une banque de données où la classe d'appartenance est connue.

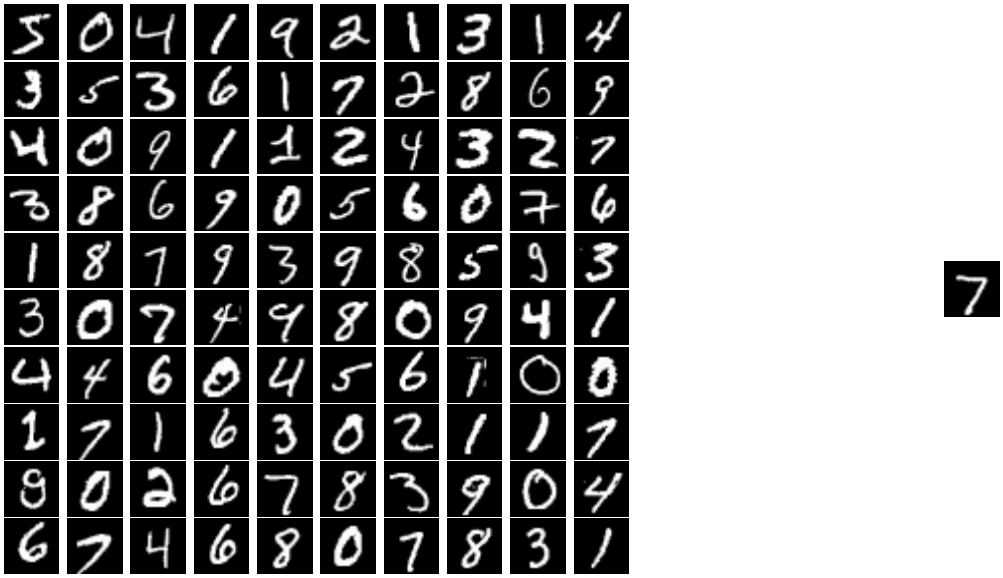


FIGURE 17.1 – Une banque de 100 images, dont on connaît déja les chiffres associés. À droite, un chiffre inconnu

Ici, la distance entre deux images, que l'on voit comme des tableau de pixels (il n'y a qu'une valeur par pixel car les images sont en noir et blanc) à valeurs dans $[\![0, 255]\!]$, est la somme des valeurs absolues des différences des pixels

1. Les images présentées ici sont extraites d'une très grande base proposée par Yann le Cun (un des pionniers de la reconnaissance d'images), située ici : `http://yann.lecun.com/exdb/mnist/`

de même emplacement, à savoir :

$$d(I,J) = \sum_{i=0}^{n-1}\sum_{j=0}^{m-1} |I_{i,j} - J_{i,j}|$$

pour deux images de taille $n \times m$. Il est possible de choisir d'autres distances également.

17.1.2 Un exemple dans le plan

Pour simplifier, nous allons implémenter l'algorithme dans le plan, avec la distance euclidienne. On suppose de plus qu'il n'y a que deux classes d'objets, les points rouges et les points bleus. L'ensemble d'apprentissage (la base) sera constituée d'une liste de couples `(p,e)` où :

— `p` est un point du plan euclidien, représenté comme un couple de valeurs numériques ;

— `e` est une étiquette, pouvant valoir `'r'` ou `'b'`, pour rouge et bleu.

L'algorithme prendra également en entrée un point du plan dont on veut décider s'il est rouge ou bleu, et un entier k supposé impair : on attribuera la couleur associée à la couleur majoritaire parmi les k plus proches voisins, comme k est impair on n'a pas à se préoccuper des cas d'égalité.

On implémente d'abord la fonction de distance (avec `sqrt` importée du module `math`) :

```
def d(p1,p2):
    x1,y1=p1
    x2,y2=p2
    return sqrt((x1-x2)**2+(y1-y2)**2)
```

La fonction suivante prend en entrée la base (`X`), un point du plan `p`, et un entier k impair, et renvoie la couleur majoritaire des k plus proches voisins de `p`.

```
def couleur(X,p,k):
    """ calcule la couleur de p en prenant la couleur majoritaire parmi ses k plus proches voisins """
    L = [(d(p,c[0]),c[1]) for c in X]
    L.sort()
    nb = 0
    for i in range(k):
        if L[i][1]=='r':
            nb+=1
        else:
            nb-=1
    if nb>0:
        return 'r'
    else:
        return 'b'
```

Remarque : on crée la liste `L` contenant des couples (distance au point p, couleur) pour chaque élément de l'ensemble `X`. On utilise ensuite la méthode `sort` de Python. On rappelle que pour des couples, la méthode trie par défaut suivant l'ordre lexicographique, donc en comparant la distance d'abord, ce qui nous convient ici.

Appliquons cette méthode à un exemple [2]. La figure 17.2 montre une distribution de 80 points rouges et 80 points bleus.
La figure 17.3 montre, pour les valeurs $k = 1, 7, 21, 159$ (159 étant la valeur maximale possible puisque le nuage est constitué de 160 points), pour chaque point du carré $[-4,4] \times [-4,4]$, la couleur attribuée par l'algorithme. Les distributions ayant permis d'obtenir le nuage de points X sont deux lois gaussiennes centrées en les points $(1,-1)$ et $(-1,1)$, de même rayon. Ainsi, la meilleure manière de réduire les probabilités d'erreurs et de décider qu'au dessus de la bissectrice $y = x$, un point doit être bleu, sinon il est rouge. Dans la pratique, on ne connaît pas les distributions de probabilités ayant mené à l'ensemble d'apprentissage X. On remarque néanmoins sur cet exemple les faits suivants.

— Pour $k = 1$, l'algorithme attribue correctement sa couleur à chaque point de l'ensemble X (qui est son unique plus proche voisin !). Néanmoins, la répartition obtenue n'est pas très régulière, et est complètement dépendante de X.

— Plus k grandit ($k = 7$, $k = 21$), plus la frontière entre les zones bleues et rouges tend à devenir régulière, et proche de la distribution optimale ($y = x$).

2. Cet exemple se trouve dans le livre Apprentissage statistique de G. Dreyfus, J.-M. Martinez, M. Samuelides, M. B. Gordon, F. Badran, S. Thiria, aux éditions Eyrolles.

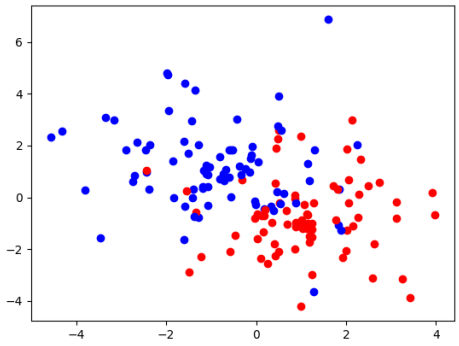


FIGURE 17.2 – Un nuage de points issus de deux distributions gaussiennes

— Néanmoins, pour k trop grand, la frontière est très régulière mais peut s'éloigner de la frontière théorique (pour $k = 159$, on associe à un point p la couleur opposée au point de X le plus éloigné de p, ce qui est assez dépendant de X également. Ici la couleur est majoritairement rouge à cause du point bleu extrême en haut du graphique...).

Il s'ensuit que trouver une valeur « optimale » de k est crucial (ici, on sent bien que $k = 7$ ou $k = 21$ sont préférables à $k = 1$ ou $k = 159$), c'est un problème difficile.

17.1.3 Matrice de confusion

Pour aider à choisir un bon paramètre k, il est courant de scinder en deux l'ensemble d'apprentissage. Imaginons que l'on ait encore un ensemble de points Y issus des mêmes distributions (on connaît leur couleur rouge ou bleu), que l'on avait écartés dans l'analyse précédente. À k fixé, on peut voir quelle couleur est attribuée à chacun via l'algorithme des k plus proches voisins, et construire une matrice 2×2 appelée *matrice de confusion*, de la forme $\begin{pmatrix} \text{RR} & \text{RB} \\ \text{BR} & \text{BB} \end{pmatrix}$ où :

— RR (resp. BB) est le nombre de points de Y qui sont rouges (resp. bleus) et étiquetés comme tel par l'algorithme.

— RB (resp. BR) est le nombre de points de Y qui sont rouges (resp. bleus) et étiquetés en bleu (resp. rouge) par l'algorithme.

Remarque 17.1. *Dans le cas où il y a p classes, la matrice de confusion est une matrice $p \times p$.*

Plus la matrice de confusion ressemble à une matrice diagonale, plus *a priori* le choix de k est pertinent ! Voici les matrices de confusion M_k obtenues pour $k \in \{1, 7, 21, 159\}$ sur un ensemble Y de 100 points, du même style que le nuage de la figure 17.2.

$$M_1 = \begin{pmatrix} 38 & 12 \\ 10 & 40 \end{pmatrix} \quad M_7 = \begin{pmatrix} 42 & 8 \\ 11 & 39 \end{pmatrix} \quad M_{21} = \begin{pmatrix} 43 & 7 \\ 8 & 42 \end{pmatrix} \quad M_{159} = \begin{pmatrix} 47 & 3 \\ 20 & 30 \end{pmatrix}$$

Sur cet exemple, prendre $k = 21$ semble plus intéressant.

17.2 Algorithme des k-moyennes

17.2.1 L'algorithme en pseudo-code

À l'oeil, il est facile de voir que le nuage de points de la figure 17.4 est composé de 3 morceaux, auquel on a donné des couleurs distinctes. Si dans cet exemple, l'appartenance d'un point à l'amas orange ou à l'amas bleu peut se discuter pour les points situés à la frontière, l'oeil humain est très bon pour répartir instinctivement les points en trois morceaux. Le but de cette section est de décrire un algorithme prenant en entrée un tel nuage de points dans un espace euclidien, un entier k et renvoyant une partition du nuage en k sous-parties « proches ». On parle ici d'*apprentissage non supervisé* car contrairement à la section précédente, on n'a pas d'ensemble d'apprentissage déja étiqueté. Concrètement, le problème est le suivant :

Soit X un ensemble fini de points d'un espace euclidien, et k un entier tel que $|X| \geq k$. Trouver une partition de X en sous-ensembles $X_0, \ldots, X_{k-1}$ non vides qui minimise la quantité $\sum_{i=0}^{k-1} \sum_{x \in X_i} \|x - m_i\|^2$, avec m_i le barycentre des points de X_i.

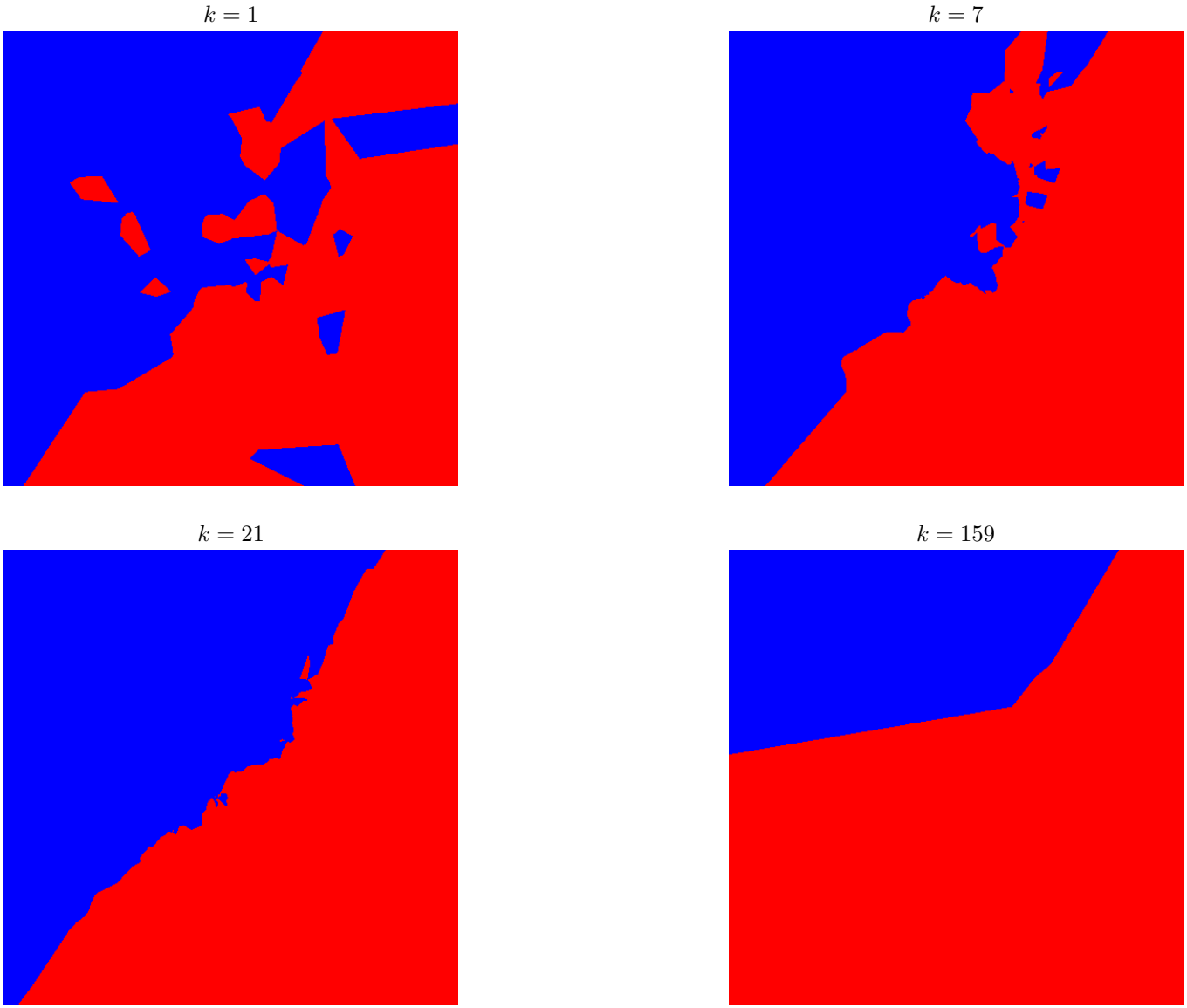


FIGURE 17.3 – Les point du carré $[-4, 4] \times [-4, 4]$ attribués aux classes rouge et bleu en fonction de k

Remarque 17.2. *Dans un tel espace euclidien E, le barycentre de p points $x_0, \dots, x_{p-1}$ est $\frac{1}{p}\sum_{i=0}^{p-1} x_i$. (La notion de barycentre est plus générale, ici on parle en fait d'*isobarycentre*).*

Résoudre ce problème exactement est difficile du fait du nombre de partitions possibles : en effet, pour chaque point de X, on a k choix pour le placement dans l'un des X_i, ce qui fait k^n partitions avec $n = |X|$. Bien sûr, on pourrait réduire un peu le nombre de partitions à examiner en prenant en compte les symétries (si on a examiné une partition $(X_0, \dots, X_{k-1})$, pas besoin d'examiner une partition où les X_i sont les mêmes, mais permutés). Toutefois ce nombre resterait exponentiel en n, même pour $k = 2$.

L'algorithme des k-moyennes est un algorithme approché, qui s'exécute rapidement en général. Voici son pseudo-code.

17.2.2 Analyse de l'algorithme

Excluons les « cas pathologiques » problématiques (plusieurs m_i égaux, égalité entre plusieurs $\|x - m_i\|$...), qui mathématiquement peuvent se produire, mais

— d'une part, peuvent être évités en rajoutant quelques lourdeurs à l'algorithme, ce qu'on ne fera pas ici ;

— d'autre part, ne se produisent pas dans la pratique lorsqu'on travaille avec des nombres flottants, ce qui est le cadre naturel,

et montrons la terminaison de l'algorithme.

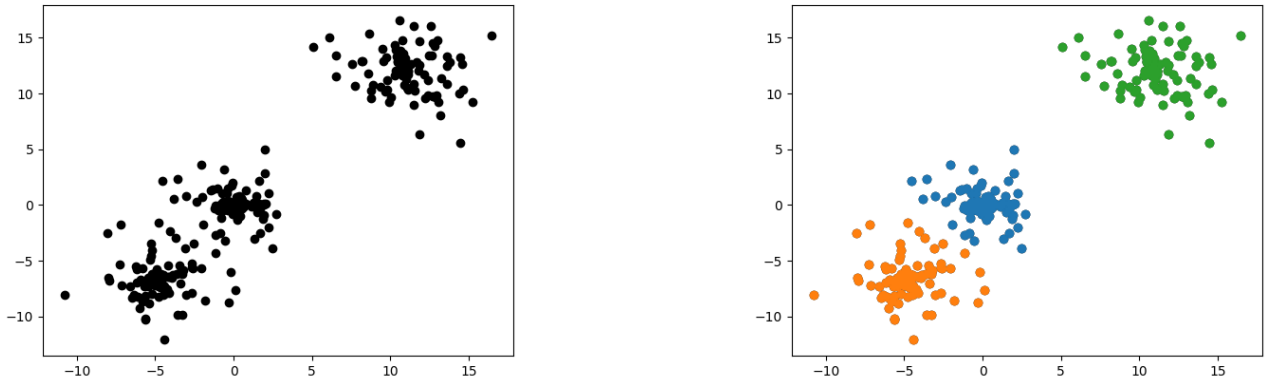


FIGURE 17.4 – Un nuage de points du plan. À droite, la partition en trois morceaux par l'algorithme des k-moyennes.

Algorithme 17.3 : Algorithme des k-moyennes

Entrée : Un ensemble fini de points X d'un espace euclidien, un entier $k \leq |X|$
Sortie : Une partition de X en parties non vides $X_0, \dots, X_{k-1}$
Répartir les points de X en sous-ensembles disjoints non vides $X_0, \dots, X_{k-1}$;
tant que *la situation évolue faire* **faire**
  Calculer $m_0, \dots, m_{k-1}$, barycentres de $X_0, \dots, X_{k-1}$;
  pour *chaque point x de X* **faire**
    Trouver i tel que $\|x - m_i\|$ est minimal parmi les $\|x - m_j\|$, et placer x dans X_i
Renvoyer $X_0, \dots, X_{k-1}$

Terminaison de l'algorithme (HP). Ce paragraphe peut être ignoré car admis dans le programme officiel. Il s'adresse plutôt aux bons élèves en mathématiques. Notons que l'ensemble des partitions possibles de X en sous-ensembles disjoints non vides est fini. Ainsi la fonction qui a une telle partition associe le coût $\sum_{i=0}^{k-1} \sum_{x \in X_i} \|x - m_i\|^2$ où les m_i sont barycentres des X_i, ne peut prendre qu'un nombre fini de valeurs. Il suffit alors de montrer que la valeur prise par la fonction de coût décroît strictement à chaque tour de la boucle Tant que. Les évolutions de cette quantité peuvent être décomposées en deux étapes différentes.

— Déplacer un élément x de X_j à X_i s'il vérifie $\|x - m_i\| < \|x - m_j\|$ fait baisser strictement $\sum_{i=0}^{k-1} \sum_{x \in X_i} \|x - m_i\|^2$;

— Recalculer les barycentres des X_i fait également baisser chaque quantité $\sum_{x \in X_i} \|x - m_i\|^2$, donc la somme, ce qui est moins évident. Cela repose sur le lemme suivant.

Lemme 17.4. *Soient $x_0, \dots, x_{p-1}$ des points d'un espace euclidien $(E, \langle ., . \rangle)$. Alors la fonction $f : \gamma \mapsto \sum_{i=0}^{p-1} \|x_i - \gamma\|^2$ admet un unique minimum global sur E, en le barycentre des x_i.*

Démonstration. Il est clair que $f(\gamma) \underset{\|\gamma\| \to +\infty}{\to} +\infty$. Par suite, il existe $r \in \mathbb{R}_+$ tel que $\|\gamma\| \geq r \Rightarrow f(\gamma) \geq f(0) + 1$. La boule fermée centrée en 0 et de rayon r étant compacte, f y admet un minimum. Ce minimum est inférieur à $f(0)$, et c'est donc un minimum global de f sur E car f est supérieure à $f(0) + 1$ en dehors de la boule. De plus, un tel minimum est nécessairement atteint dans la boule ouverte centrée en 0 et de rayon r (puisque f est supérieure à $f(0) + 1$ sur la sphère de centre 0 et de rayon r). La fonction f étant $\mathcal{C}^1$ sur E, la différentielle de f doit s'annuler en cet extremum. Or pour $\gamma, h \in E$,

$$\begin{aligned} f(\gamma + h) - f(\gamma) &= \sum_{i=0}^{p-1} \left(\|x_i - \gamma - h\|^2 - \|x_i - \gamma\|^2 \right) \\ &= \sum_{i=0}^{p-1} \left(-2\langle x_i - \gamma, h \rangle + \|h\|^2 \right) \\ &= \langle -2 \sum_{i=0}^{p-1} (x_i - \gamma), h \rangle + o(\|h\|) \end{aligned}$$

Il s'ensuit que $\vec{\text{grad}}_\gamma(f) = -2 \sum_{i=0}^{p-1} (x_i - \gamma)$. Or, ce gradient s'annule si et seulement si $\gamma = \frac{1}{p} \sum_{i=0}^{p-1} x_i$, autrement dit si γ est le barycentre des x_i. En conclusion, le barycentre des x_i est le seul point où f admet un extremum local, et c'est un minimum global. □

Résultat de l'algorithme. On a donc montré (ou admis) que l'algorithme terminait. Le résultat n'est toutefois pas nécessairement une partition optimale de l'ensemble X, mais possiblement un « minimum local » au sens où les étapes de l'algorithme ne modifient pas la partition. La figure 17.5 montre un exemple à 6 points partitionné en 3 classes [3]. Les points sont les ronds, les croix sont les barycentres des partitions (à gauche, le point bleu est seul dans sa partition, donc égal au barycentre de sa classe). À gauche comme à droite, pour chaque point, le barycentre le plus proche est celui de sa couleur : l'algorithme s'arrête dans les deux cas. La fonction de coût vaut $13/6 \simeq 2.17$ à gauche et $3/2$ à droite. On peut vérifier que la partition de droite est optimale.

FIGURE 17.5 – Deux partitions de taille 3 de l'ensemble $\{(\cos(a\pi/3), \sin(a\pi/3)) \mid 0 \leq a \leq 5\}$ sur lequel l'algorithme des k plus proches voisins n'évolue plus. À gauche une partition non optimale, à droite une partition optimale.

Complexité de l'algorithme. Dans le pire cas, l'algorithme est de complexité exponentielle en le nombre de points à répartir. Néanmoins, on peut signaler pour la culture que cet algorithme a à k fixé une *complexité lisse* [4] polynomiale. Cela signifie que sur tout ensemble X fixé, si on applique l'algorithme à X auquel on rajoute une perturbation aléatoire sur chaque point – un *bruit gaussien* –, on obtient une complexité polynomiale en moyenne. Ceci explique les très bonnes performances pratiques de l'algorithme.

17.2.3 Implémentation

Représentation et fonctions utilitaires. On travaille ici avec des tableaux Numpy représentant des vecteurs dans $\mathbb{R}^k$ pour un certain k. On importe les modules classiques comme suit :

```
import random as rd
import numpy as np
```

Le produit scalaire sur cet espace s'écrit simplement :

```
def ps(x,y):
    return (x*y).sum()
```

On en dérive la distance entre deux points :

```
def dist(x,y):
    return ps(x-y,x-y)**0.5
```

Avec X l'ensemble de points de taille n, une partition de X en sous-ensembles X_i sera représentée comme une liste de listes d'éléments de $[\![0, n-1]\!]$. Par exemple, les deux partitions de la figure 17.5 sont `[[0, 1, 2], [3, 4], [5]]` et `[[0, 1], [2, 3], [4, 5]]`. On peut facilement écrire une fonction calculant le barycentre des X_i avec cette représentation.

```
def barycentre(X,xi):
    """ X ensemble de points, xi liste d'indices de X.
    Renvoie le barycentre des X[k] pour k dans xi """
    s=0
    for k in xi:
        s=s+X[k]
    return s/len(xi)
```

3. On laisse au lecteur le soin de trouver un exemple minimal à trois points et deux classes !
4. Ce concept est récent, et a valu a ses inventeurs le prestigieux prix Gödel. Voir `https://fr.wikipedia.org/wiki/Analyse_lisse_d%27algorithme`

Pour l'initialisation de l'algorithme, on utilisera par exemple la fonction suivante, permettant de placer les entiers de $[\![0, n-1]\!]$ dans k sous-ensembles disjoints, non vides (représentés comme des listes). Pour faire ceci, on mélange aléatoirement la liste constituée des entiers de $[\![0, n-1]\!]$. Les k premiers éléments obtenus sont les indices des points qu'on ajoute à $X_1, \dots, X_k$. Pour les éléments suivants, on choisit aléatoirement dans quel X_j on les place.

```
def initialisation(n,k):
    """ initialisation de l'algorithme : renvoie une partition de [0,n-1] en k morceaux non vides.
    On choisit le processus suivant : mélanger aléatoirement la liste [0,...,n-1], les k premiers
    vont dans k ensembles différents. Les autres sont placés ensuite aléatoirement dans l'un des Xi.
    """
    assert n>=k
    L=list(range(n))
    rd.shuffle(L) #mélange aléatoire de L
    X=[[] for _ in range(k)]
    for i in range(k): #L[i] dans X[i] pour les k premiers
        X[i].append(L[i])
    for i in range(k,n): #ensuite, L[i] dans un X[j] aléatoire
        X[rd.randrange(0,k)].append(L[i])
    return X
```

Par exemple :

```
>>> initialisation(10,3)
[[9, 0], [3, 8, 6, 5, 2], [7, 1, 4]]
```

Voici enfin une fonction prenant en entrée un point `p` et une liste d'autres points `M` (dans l'idée, des barycentres), et renvoyant l'indice du point le plus proche de `p` dans `M`.

```
def plus_proche(p,M):
    i, d = 0, dist(p,M[0])
    for j in range(1,len(M)):
        if dist(p,M[j])<d:
            i, d = j, dist(p,M[j])
    return i
```

Algorithme en lui-même. On a maintenant ce qu'il faut pour écrire l'algorithme des k-moyennes. La condition « tant que la situation évolue » peut se traduire par une boucle infinie, qui s'arrête dès que les partitions X_i sont identiques entre deux itérations [5].

```
def k_moyennes(X,k):
    n=len(X)
    assert n>=k, "le cardinal de X doit être plus grand que k !"
    part = initialisation(n,k)
    while True:
        M = [barycentre(X,xi) for xi in part]
        part2 = [[] for _ in range(k)]
        for i in range(n):
            part2[plus_proche(X[i],M)].append(i)
        if part == part2:
            return part
        else:
            part = part2
```

17.2.4 Une application : compression d'images

Cadre euclidien pour les pixels d'une image. Les pixels d'une image couleur sont en général encodés comme des triplets (r, g, b), chacune des valeur r, g, b étant un entier de $[\![0, 255]\!]$. Il y a donc 256^3 couleurs possibles, et chaque pixel est encodé sur 24 bits (il faut 8 bits pour une valeur entre 0 et 255). Via l'inclusion $[\![0, 255]\!]^3 \subset \mathbb{R}^3$, on peut considérer un pixel comme un point de l'espace euclidien $\mathbb{R}^3$ muni de son produit scalaire canonique.

Application à la compression. Une méthode possible pour compresser une image est de réduire le nombre de couleurs utilisées à k, avec k un (petit) entier. En supposant que $k = 2^p$, il suffit pour stocker l'image de :

— stocker les valeurs $c_0, \dots, c_{k-1}$ des couleurs utilisées (24 bits par couleur utilisée) ;

5. Mise à part à l'initialisation, les X_i sont triés dans l'ordre croissant, ce qui permet d'utiliser le test d'égalité sur des listes.

— stocker pour chaque pixel de l'image le numéro de la couleur qu'on lui donne (seulement p bits au lieu de 24).

En négligeant le stockage des k couleurs (qui compte peu par rapport à l'image à moins qu'elle ne soit toute petite), on obtient un ratio de compression $\frac{\text{poids de l'image compressée}}{\text{poids de l'image initiale}} = \frac{p}{24}$.

Utilisation de l'algorithme des k-moyennes. Pour savoir quelles couleurs seront utilisées, et quelle couleur associer à chaque pixel, on peut utiliser l'algorithme des k-moyennes. L'ensemble X de l'algorithme est constitué de toutes les couleurs présentes dans l'image initiale. Après application de l'algorithme, on remplace chacune de ces couleurs par le barycentre de sa classe (arrondi à la couleur la plus proche)

La figure 17.6 montre un tel exemple :

— à gauche figure l'image originelle (un lycée quelconque de province [6]), de largeur 512 pixels et hauteur 288 pixels. Au total il y a donc 147456 pixels, mais seulement 46283 couleurs distinctes.

— à droite figure l'image après application de l'algorithme des k-moyennes, avec $k = 8 = 2^3$ (il n'y a vraiment que 8 couleurs différentes dans toute l'image, elles sont présentées figure 17.7), qui nécessite donc seulement un huitième d'espace de stockage par rapport à l'image de gauche.

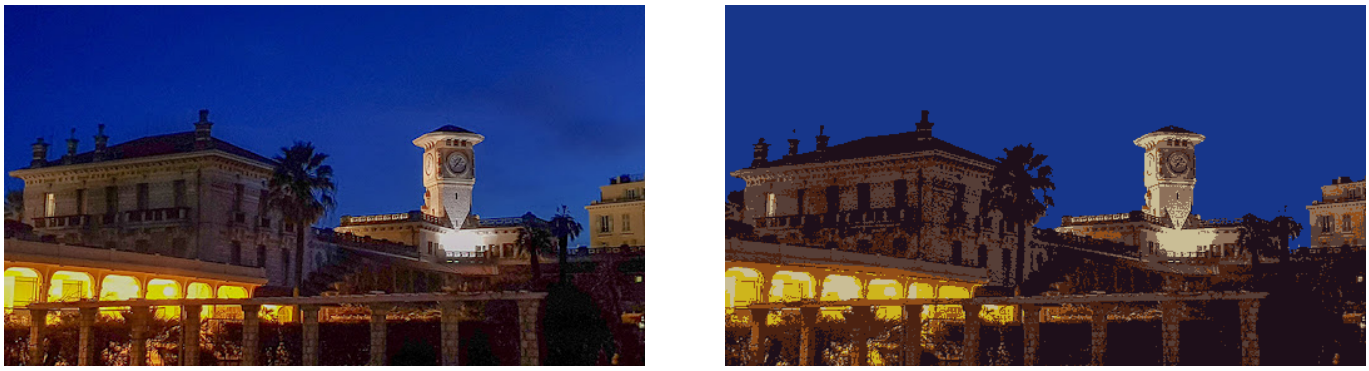


FIGURE 17.6 – Application de l'algorithme des k-moyennes à la compression d'images. Il n'y a que 8 couleurs différentes dans l'image de droite !

FIGURE 17.7 – Les 8 couleurs utilisées pour la compression de l'image de la figure 17.6

6. Franchement, c'est le plus beau lycée de France, non ?

Chapitre 18

Introduction à la théorie des jeux

Dans ce chapitre, on donne une introduction à l'étude de la théorie des jeux. Les jeux que l'on va étudier sont des jeux à *information complète*, ce qui exclut la plupart des jeux de cartes (chaque joueur n'a pas connaissance du jeu adverse et éventuellement du reste du paquet), et sans hasard. On ne s'intéresse qu'à des *jeux d'accessibilité* à deux joueurs, qui peuvent être modélisés par des graphes orientés, jouer un coup consiste à suivre un arc. Dans un premier temps, on discute de jeux qui sont suffisamment simples pour pouvoir être résolus complètement : l'espace des états possibles n'est pas trop grand. Dans un second temps, on aborde la notion de stratégie avec heuristique, qui peut s'appliquer aux jeux qu'il est impossible (à l'heure actuelle, au vu des contraintes matérielles) de résoudre complètement, comme le jeu d'échecs ou le jeux de go.

18.1 Jeu d'accessibilité sur un graphe

18.1.1 Exemple : un jeu de Nim

Introduction. Dans l'émission *Fort Boyard*, le *duel de bâtonnets*[1] avec le *maître des ténèbres* est un jeu à deux joueurs. Au départ, un tas comporte 20 bâtonnets, et à son tour un joueur peut en enlever 1, 2 ou 3. Le perdant est celui qui prend le dernier bâtonnet. Voici un exemple de *partie* :

$$20 \longrightarrow 17 \longrightarrow 16 \longrightarrow 15 \longrightarrow 12 \longrightarrow 10 \longrightarrow 8 \longrightarrow 7 \longrightarrow 4 \longrightarrow 3 \longrightarrow 1 \longrightarrow 0$$

Ce jeu est particulièrement simple, et il est facile de voir que c'est le joueur qui joue en premier qui gagne, s'il suit la stratégie de laisser à l'autre joueur un nombre de bâtonnets qui est de la forme $4n+1$.

Modélisation par un graphe. À un tel jeu, on peut associer un graphe orienté $G = (V, E)$:

— les sommets V de G sont les positions atteignables dans le jeu. Pour le jeu précédent, on pourrait utiliser 21 sommets numérotés de 0 à 20, indiquant le nombre de bâtonnets restants.

— les arcs E de G indiquent quel sommet est atteignable depuis un autre en jouant un unique coup. Dans le jeu précédent, les arcs issus du sommet 7 pointent vers 6, 5 et 4, comme on le voit figure 18.1.

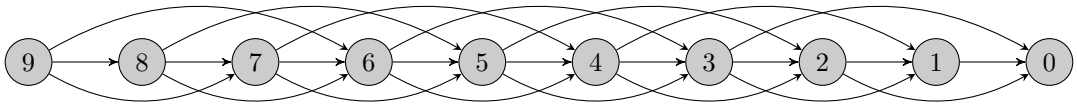


FIGURE 18.1 – Un jeu de Nim avec seulement 9 bâtonnets au départ

Une *partie* sur un tel graphe est un chemin où le premier joueur, depuis le sommet de départ, suit un arc, et ensuite chaque joueur suit à tour de rôle un arc, si c'est possible. Une partie est *finie* si ce chemin termine sur un sommet sans successeur, *infinie* sinon.

Proposition 18.1. *Si le graphe G est sans circuit, toute partie est finie.*

1. Un exemple de partie où ni le maître du jeu, ni la la joueuse n'ont étudié la théorie des jeux : `https://www.youtube.com/watch?v=dfFSpesQyNg`

Généralisation : graphe biparti. Il est utile dans la pratique de « dédoubler » un graphe comme le précédent en un graphe *biparti*, de sorte qu'une partie dans le jeu se résume simplement à un chemin dans le graphe biparti, sans avoir à distinguer quel joueur joue.

Définition 18.2. *Un graphe $G = (V, E)$ est dit* biparti *s'il existe une partition de V en deux sous-ensembles V_a et V_b, telle qu'aucun arc du graphe ne relie deux sommets de V_a, ou deux sommets de V_b.*

Exemple 18.3. *Le graphe biparti associé au graphe de la figure 18.1 est le suivant.*

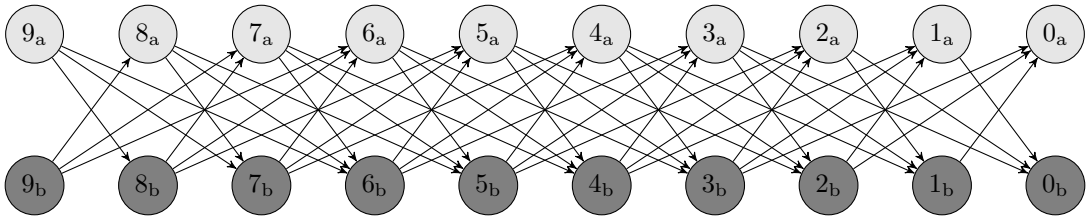


FIGURE 18.2 – Le graphe biparti associé au jeu de Nim avec 9 bâtonnets au départ. Le joueur a commence (en 9_a), et donc le sommet 9_b ne sera jamais atteint.

18.1.2 Vocabulaire : arène, condition de victoire, positions gagnantes et stratégies

On commence par définir une arène dans un jeu à deux joueurs.

Définition 18.4 (Arène)**.** *Un graphe de jeu (aussi appelé une* arène*) est un triplet (G, V_1, V_2), où $G = (V, E)$ est un graphe orienté, avec V partitionné en $V_1 \cup V_2$.*

Les sommets d'un tel graphe se partitionnent donc en deux ensembles V_1 et V_2, V_1 étant l'ensemble des sommets contrôlés par le premier joueur et V_2 ceux contrôlés par le second.

Définition 18.5 (Partie)**.** *Une* partie *sur un tel graphe depuis un sommet v_0 se déroule comme suit :*

- *le joueur contrôlant v_0 choisit, s'il en existe un, un sommet v_1 tel que $(v_0, v_1) \in E$;*
- *le joueur contrôlant v_1 choisit, s'il en existe un, un sommet v_2 tel que $(v_1, v_2) \in E$;*
- *etc...*

Autrement dit, une partie $\mathcal{P}$ est un chemin $v_0, v_1, \dots$, maximal dans le graphe, possiblement infini, où chaque joueur contrôlant un sommet choisit un arc vers un autre sommet du graphe si c'est possible, sinon la partie s'arête.

Remarque 18.6. *Souvent, comme dans la section précédente, le graphe considéré sera biparti, de sorte que les joueurs jouent alternativement : aucun arc du graphe ne relie deux sommets de V_1 ou de V_2.*

Définition 18.7 (Condition de victoire)**.** *Une condition de victoire (pour le premier joueur) est un sous-ensemble des parties possibles. Le complémentaire est la condition de victoire du second joueur.*

En théorie des jeux, il existe de multiples conditions de victoire. La plus simple (et la seule au programme) est la condition d'*atteignabilité*.

Définition 18.8 (Condition d'atteignabilité)**.** *Un condition d'atteignabilité pour le premier joueur est un sous-ensemble W de V. La partie est remportée par le premier joueur si celle-ci visite un sommet de W, remportée par le second joueur dans le cas contraire.*

Remarque 18.9 (Sommet de degré sortant nul)**.** *Dans la modélisation précédente, on a implicitement supposé qu'un sommet v de degré sortant nul contrôlé par le second joueur est dans l'ensemble W : si le deuxième joueur ne peut pas jouer, le premier gagne. Ceci n'est pas restrictif : si on voulait déclarer le second joueur vainqueur lorsque la partie aboutit à v, on pourrait rajouter un sommet v' contrôlé par le second joueur, et des arcs (v, v') et (v', v).*

Exemple 18.10 (Retour sur le jeu des bâtonnets)**.** *Puisque le jeu des bâtonnets est considéré comme perdu par le joueur qui prend le dernier bâtonnet, on peut modéliser le jeu comme un jeu d'accessibilité en enlevant les sommets 0_x, et en prenant $W = \{1_b\}$.*

Définition 18.11 (Stratégie)**.** *Pour $G = (V, E)$ un graphe, notons $V_i^{>0}$ l'ensemble des sommets de degré sortant non nul contrôlés par le joueur $i \in \{1, 2\}$. Une stratégie est une application $\varphi : V_i^{>0} \to V$, telle que $\forall\, s \in V_i^{>0}$, $(s, \varphi(s)) \in E$.*

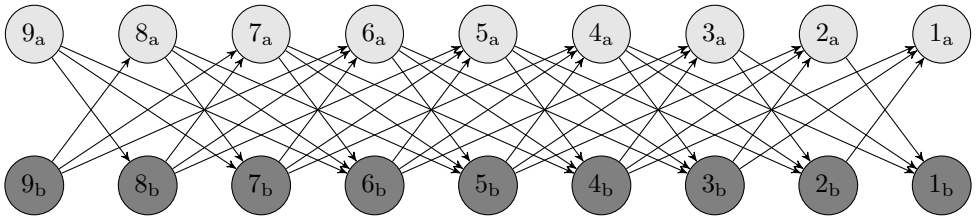


FIGURE 18.3 – Jeu d'accessibilité des bâtonnets : la condition de victoire pour le premier joueur se résume à atteindre 1_b. Le premier joueur gagne si ce sommet est atteint, le second gagne dans le cas contraire.

On dit que le joueur i ($i \in \{1,2\}$) *respecte* la stratégie φ lors d'une partie $\mathcal{P} = (v_0, v_1, \dots)$ si

$$\forall\, j \quad v_j \in V_i^{>0} \Longrightarrow v_{j+1} = \varphi(v_j)$$

Autrement dit, lorsque c'est au joueur i de jouer (et qu'il peut jouer !), il choisit le sommet donné par la stratégie.

Remarque 18.12. *Les stratégies considérées ici sont en fait* sans mémoire*, c'est-à-dire que le prochain sommet joué ne dépend que du sommet courant et pas des précédents. C'est tout à fait adapté pour des conditions d'accessibilité. Avec d'autres conditions de victoire, il faudrait envisager des stratégies où le choix du prochain sommet dépend de tous les précédents.*

Définition 18.13 (Stratégie gagnante)**.** *Une stratégie φ est dite* gagnante *pour le joueur i ($i \in \{1,2\}$) depuis le sommet $v_0 \in V$, si toute partie jouée depuis v_0 où le joueur i respecte la stratégie φ est gagnante pour i.*

Exemple 18.14 (Jeu des bâtonnets)**.** *On considère le graphe des bâtonnets, avec n bâtonnets au départ, de sommets $V = \{1_a, \dots, n_a, 1_b, \dots, n_b\} = V_1 \cup V_2$ avec $V_1 = \{1_a, \dots, n_a\}$ et $V_2 = \{1_b, \dots, n_b\}$ comme défini précédemment. La stratégie consistant (si possible) à laisser à l'autre joueur un nombre de bâtonnets qui est de la forme $4n+1$ est une stratégie gagnante pour chaque joueur, depuis un sommet qui n'est pas de la forme $(4n+1)_a$ ou $(4n+1)_b$.*

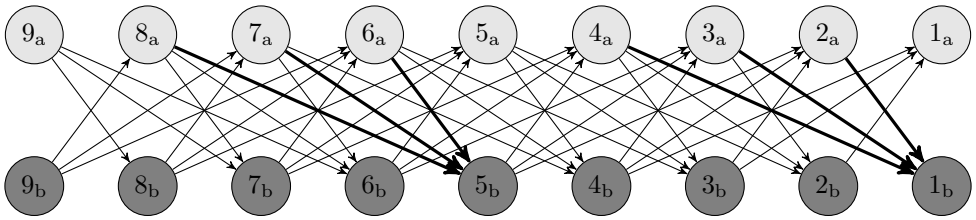


FIGURE 18.4 – Jeu d'accessibilité des bâtonnets : en gras, les arcs d'une stratégie gagnante pour le premier joueur. Les positions 9_a et 5_a étant perdantes, la définition de la stratégie sur ces sommets n'a pas d'importance.

Définition 18.15 (Position gagnante)**.** *Un sommet $v \in V_i$ est appelé* position gagnante *pour le joueur i si celui-ci possède une stratégie gagnante depuis v.*

Dans l'exemple précédent, on voit que 9_a n'est pas une position gagnante pour le joueur a, contrairement à 8_a, 7_a et 6_a.

18.1.3 Attracteur

Dans un jeu d'accessibilité, *résoudre* le jeu consiste essentiellement à calculer les positions gagnantes de chacun des joueurs, et une stratégie pour chacun (définie sur chaque position gagnante que le joueur contrôle). On se concentre d'abord sur les positions gagnantes du premier joueur, qui doit pour gagner atteindre un ensemble W. L'outil adéquat est *l'attracteur*.

Ensembles définissant l'attracteur. On se donne donc un graphe $G = (V, E)$, et deux ensembles V_1, V_2 formant une partition de V, et W l'ensemble de sommets définissant la condition de victoire du premier joueur. On définit incrémentalement une suite de sous-ensembles de sommets comme suit :

$$\mathcal{A}_0 = W \quad \text{et} \quad \forall\, j \geq 0 \quad \mathcal{A}_{j+1} = \mathcal{A}_j \cup \{v \in V_1 \mid \exists v' \in \mathcal{A}_i \mid (v, v') \in E\} \cup \{v \in V_2 \mid \forall\, v' \in V \mid (v, v') \in E \Rightarrow v' \in \mathcal{A}_i\}$$

En français : l'ensemble $\mathcal{A}_{j+1}$ est constitué

— des sommets de $\mathcal{A}_j$;
— des sommets contrôlés par le premier joueur pour lesquels il existe au moins un arc permettant de rejoindre un sommet de $\mathcal{A}_j$;
— des sommets contrôlés par le second joueur dont *tous* les arcs aboutissent à des sommets de $\mathcal{A}_j$.

La suite $(\mathcal{A}_j)_{j\geq 0}$ est clairement croissante au sens de l'inclusion, *l'attracteur* de W est défini par l'union croissante des $(\mathcal{A}_j)$.

Définition 18.16. L'attracteur *de W est l'ensemble* $\mathcal{A} = \cup_{j=0}^{+\infty}\mathcal{A}_j$.

On va voir que l'attracteur contient précisément les positions gagnantes du premier joueur.

Théorème 18.17. *Avec les notations précédentes, le premier joueur possède une stratégie gagnante pour tout sommet de $\mathcal{A}$, son adversaire possède une stratégie gagante pour tout sommet de ${}^c\mathcal{A}$.*

Démonstration. Pour tout sommet v de V, on pose

$$\mathrm{Rang}(v) = \inf\{j \geq 0 \mid v \in \mathcal{A}_j\} \in \mathbb{N} \cup \{+\infty\}$$

(le rang est fini pour chaque sommet de l'attracteur, sinon il vaut $+\infty$). Montrons par récurrence sur le rang que les sommets de $\mathcal{A}$ sont des positions gagnantes du premier joueur :

- C'est évident pour les sommets de rang 0, qui sont dans W ;
- Supposons la propriété démontrée jusqu'au rang $j \geq 0$, et considérons un sommet v de rang $j+1$ (qui est donc dans $\mathcal{A}_{j+1}\backslash\mathcal{A}_j$).
  — s'il est contrôlé par le premier joueur, alors par définition il existe un sommet v' de $\mathcal{A}_j$ et l'arc (v, v') appartient à E. Le joueur peut donc jouer cet arc. Puisqu'il possède par hypothèse de récurrence une stratégie gagnante depuis v', il en possède une depuis v ;
  — s'il est contrôlé par le second joueur, par définition, tout arc depuis v aboutit à un sommet de $\mathcal{A}_j$: quel que soit le choix du deuxième joueur, le premier possède une stratégie gagnante depuis le sommet choisi, donc v est également une position gagnante pour le premier joueur.

Inversement, si $v \notin \mathcal{A}$, on vérifie que v est une position gagnante du second joueur en construisant de la même manière une stratégie gagnante pour ce joueur : si v est contrôlé par le second joueur, celui-ci peut joueur un sommet v' qui n'est pas non plus dans $\mathcal{A}$, et si v est contrôlé par le premier joueur, tout choix de celui-ci (s'il peut jouer) aboutit à un sommet qui n'est pas non plus dans $\mathcal{A}$. Le second joueur peut donc faire en sorte que la partie évite W, il remporte donc la partie. □

Remarque 18.18. *Dans la preuve précédente, on a supposé que les sommets de degré sortant nul contrôlés par le deuxième joueur sont dans W (voir remarque 18.9).*

18.1.4 Algorithme de calcul de l'attracteur

Pour calculer l'attracteur, il suffit essentiellement de faire un parcours de graphe, mais sur le graphe transposé de G : en effet on part de l'ensemble W et on remonte les arcs à l'envers. Il est utile de calculer les degrés sortants des sommets dans G (qui sont les degrés entrants dans ${}^tG$), pour gérer la condition « tout arc sortant d'un sommet contrôlé par le deuxième joueur aboutit à un sommet de l'attracteur ».

Terminaison de l'algorithme. Il s'agit essentiellement d'un parcours de graphe : en dehors des appels à la fonction récursive *parcours*, l'algorithme termine. De plus, un appel à $parcours(u)$ n'est réllement utile que si $u \notin \mathcal{A}$, il y aura donc au plus un appel utile pour chaque sommet. Ainsi, l'algorithme termine.

Correction de l'algorithme. On veut montrer qu'à la fin de l'algorithme, l'ensemble $\mathcal{A}$ contient précisément l'attracteur associé à W. Montrons déja que la propriété « tout élément de $\mathcal{A}$ est un élément de l'attracteur » est un invariant de l'algorithme.

— au début, $\mathcal{A}$ est vide, donc la propriété est vérifiée.
— si $parcours(u)$ est lancé dans la boucle principale, c'est que $u \in W$, donc u est bien dans l'attracteur.
— sinon, c'est que $parcours(v)$ est lancé pendant $parcours(u)$, avec u dans l'attracteur. Si $v \in V_1$, cela signifie que l'arc $v \to u$ est présent dans le graphe G, donc par définition v est dans l'attracteur. Sinon, cela signifie que $n_v = 0$. Or n_v est décrémenté (au plus) une fois par sommet u de l'attracteur tel que l'arc $v \to u$ est présent dans le graphe : si n_v atteint 0, cela signifie que *tous* les sommets que l'on peut atteindre depuis v sont dans l'attracteur, donc par définition v l'est aussi.

Algorithme 18.19 : Calcul de l'attracteur

Entrée : Un graphe $G = (V, E)$ donné par listes d'adjacence, une partition de V en deux ensembles $V_1 \cup V_2$, un ensemble W
Sortie : L'attracteur associé à W
$\mathcal{A} \leftarrow \emptyset$;
Pour chaque sommet v, calculer n_v, le degré sortant de v dans G;
Calculer ${}^tG$, graphe transposé de G;
Fonction *parcours(u)* **:**
  si $u \notin \mathcal{A}$ **alors**
    $\mathcal{A} \leftarrow \mathcal{A} \cup \{u\}$;
    pour *tout voisin* v *de* u *dans* ${}^tG$ **faire**
      $n_v \leftarrow n_v - 1$;
      si $v \in V_1$ *ou* $n_v = 0$ **alors**
        parcours(v)
pour *tout sommet* u *de* W **faire**
  parcours(u)

Montrons maintenant que tout sommet de l'attracteur se retrouve dans l'ensemble $\mathcal{A}$ à la fin de l'algorithme. Supposons que ce ne soit pas le cas et considérons un sommet v de l'attracteur, de rang minimal parmi ceux qui ne se retrouvent pas dans $\mathcal{A}$.

— v n'est pas de rang 0 ;
— si $v \in V_1$, alors par hypothèse il existe un sommet u dans l'attracteur, de rang strictement inférieur, tel que l'arc (v, u) soit présent dans le graphe. Dans l'appel $parcours(u)$ où u est ajouté à $\mathcal{A}$, $parcours(v)$ aurait du être lancé, c'est absurde.
— sinon, $v \in V_2$. Par hypothèse, tous les sommets accessibles depuis v dans G sont dans l'attracteur, et ont un rang strictement inférieur par définition. Notons u le dernier à être ajouté à $\mathcal{A}$: lorsque v est examiné dans la liste d'adjacence de u dans ${}^tG$, n_v passe à zéro et $parcours(v)$ aurait du être lancé, c'est donc absurde.

Complexité de l'algorithme. L'algorithme peut être implémenté avec une complexité $O(|V| + |E|)$: il suffit pour cela d'encoder V_1, W et $\mathcal{A}$ comme des listes de booléens.

Implémentation Python. Voici un code Python qui suit le pseudo-code ci-dessus (utilisant une fonction de calcul du graphe transposé laissée en exercice).

```
def calcul_attracteur(G, V1, W):
    """ G : graphe de jeu (V,E) donné par liste d'adjacence.
    V1 : liste de booléens indiquant si un sommet est dans V1 (complémentaire dans V2)
    W : liste de booléens indiquant si un sommet est dans W (gagnée par le joueur 1).
    Renvoie l'attracteur comme une liste de booléens """
    n=len(G)
    A=[False]*n #attracteur
    nG = [len(x) for x in G] #degrés sortant dans G
    tG=transpose(G) #graphe transposé
    def parcours(u):
        if not A[u]:
            A[u]=True
            for v in tG[u]:
                nG[v]=nG[v]-1
                if V1[v] or nG[v]==0:
                    parcours(v)
    for u in range(n):
        if W[u]:
            parcours(u)
    return A
```

Modification de l'algorithme pour le calcul d'une stratégie Pour calculer en parallèle une stratégie pour le joueur 1, il suffit de reprendre l'algorithme : lorsqu'on lance $parcours(v)$ dans $parcours(u)$, on peut poser $\varphi(v) = u$. Pour calculer une stratégie gagnante pour le joueur 2 sur un sommet $u \in V_2$ n'appartenant pas à l'attracteur, il suffit

de parcourir la liste d'adjacence de u à la recherche d'un sommet v qui n'est pas non plus dans l'attracteur (le coût total du calcul de la stratégie est également en $O(|V| + |E|)$). Ceci est laissé en exercice !

Remarque 18.20. *Dans cette partie, on s'est placé dans le cadre le plus simple des jeux d'accessibilité : le joueur 1 gagne s'il atteint un certain ensemble d'états, sinon c'est le joueur 2. Une situation un peu plus complexe, mentionnée dans le programme officiel, est la suivante : le joueur 1 gagne s'il atteint un certain ensemble W_1, le joueur 2 gagne s'il atteint un ensemble W_2, autrement la partie est nulle. Dans ce cas, en supposant le graphe sans circuit*[2]*, il suffit essentiellement d'appliquer deux fois l'algorithme : les sommets du graphes se répartissent en trois morceaux :*

- *les sommets gagnants pour le premier joueur (l'attracteur de W_1) ;*
- *les sommets gagnants pour le second joueur (l'attracteur de W_2) ;*
- *les autres sommets, où si chaque joueur joue de manière optimale, la partie est nulle.*

Un exemple est le jeu du morpion, sur un carré 3×3 : la position initiale n'est une position gagnante ni pour le premier joueur, ni pour le second.

18.2 Algorithme Minimax

Les jeux précédents, que l'on peut résoudre intégralement à l'aide d'un parcours de graphe, ne sont pas très intéressants : il n'existe pas de championnat du monde de jeu des batonnets ou de Morpion, puisqu'une stratégie optimale est connue ! Des jeux plus compliqués sont par exemple les échecs ou le jeu de go : ces jeux ne sont pas résolus entièrement dans le sens où on ne connaît pas de stratégie optimale. En effet, le nombre de parties possibles est de l'ordre de 10^{120} pour le jeu d'échecs et 10^{600} pour le jeu de go, ce qui rend une exploration exhaustive impossible.

Malgré cette impossibilité, les humains sont aujourd'hui dépassés par la machine[3]. Cette partie vise à donner un aperçu d'un programme informatique capable de jouer à de tels jeux où l'espace des positions est trop grand pour être exploré exhaustivement.

Dans cette partie, on étudie les jeux à information complète, à deux joueurs, à somme nulle, et à coups asynchrones en nombre fini. Précisons un peu ces termes :

- un *jeu à information complète* est un jeu où chaque joueur connaît les actions qu'il peut entreprendre, ainsi que ses adversaires, et les gains résultants de telles actions.
- un *jeu à coups asynchrones* est un jeu où chaque joueur joue alternativement. *En nombre fini* signifie qu'une partie ne peut être infinie (aux échecs, une règle impose que si aucun pion n'a avancé ou aucune pièce n'a été capturée en 50 coups, la partie est nulle).
- un *jeu à somme nulle* est un jeu où le gain du premier joueur correspond à la perte de l'autre joueur. En pratique, le gain du premier joueur est souvent réduit aux trois valeurs $+\infty$ (il gagne), 0 (partie nulle), $-\infty$ (le deuxième joueur gagne), mais dans la suite ce gain pourra être toute valeur de $\mathbb{R}\cup\{\pm\infty\}$.

Dans la suite, on appelera un tel jeu un jeu Min-Max, et les deux joueurs seront appelés Max et Min : le but de Max est de maximiser son gain, le but de Min est de minimiser le gain du joueur Max.

18.2.1 Représentation par un arbre

Notion d'arbre. Un tel jeu se représente sous forme arborescente. Formellement, un arbre est un graphe connexe acyclique, dont on choisit un nœud particulier (la racine), qui oriente l'arbre. En informatique, les arbres « poussent » de haut en bas : la racine est donc située en haut.

La profondeur d'un nœud est sa distance à la racine (la racine est donc l'unique nœud à profondeur zéro). Pour la représentation graphique, tous les nœuds à une même profondeur sont représentés sur une même ligne horizontale. Le fait que le graphe soit connexe et acyclique impose qu'il existe un unique chemin simple (sans sommet en double) de la racine r à un nœud n quelconque de l'arbre. Le long de ce chemin il y a une relation de parenté entre nœuds successifs : si p précède q, p est le *père* de q et q est un *fils* de p. Un nœud sans fils est appelé une *feuille*, sinon c'est un *nœud interne*. Sur la figure 18.5, r est la racine, n est un nœud interne, f est une feuille.

2. Si le graphe possède des circuits, il faut être plus précis. C'est possible mais il me semble que l'on s'éloigne un peu de l'introduction à la théorie des jeux !

3. Aux jeux d'échecs, Deeper Blue bat le champion du monde en titre Garry Kasparov en 1997. Il a fallu attendre 2017 pour que le champion du monde du jeu de Go Ke Jie soit battu par le programme informatique AlphaGo. Depuis, la supériorité des algorithmes sur les humains est indéniable.

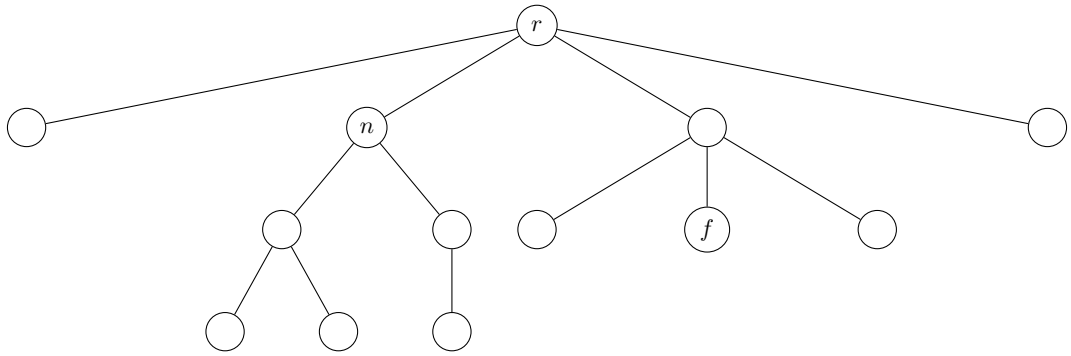


FIGURE 18.5 – Un arbre

Arbre pour un jeu Min-Max. Un jeu Min-Max se représente comme un arbre, où les nœuds représentent des positions du jeu. Puisque les joueurs jouent alternativement, les nœuds d'un même niveau sont alternativement contrôlés par un même joueur. Par symétrie, on peut supposer que :

— la racine est contrôlée par le joueur Max ;

— les nœuds à profondeur 1 sont contrôlés par le joueur Min ;

— les nœuds à profondeur 2 sont contrôlés par le joueur Max ;

— etc...

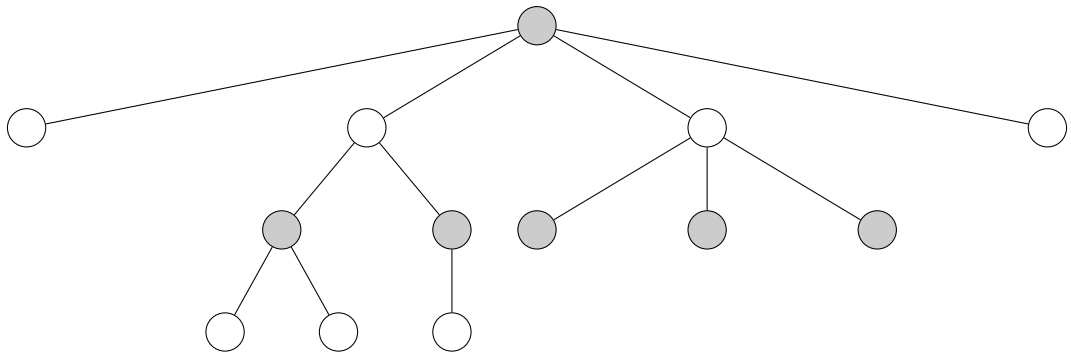


FIGURE 18.6 – Un arbre représentant un jeu Min-Max : les nœuds contrôlés par Max sont en gris, les autres sont contrôlés par Min.

La figure 18.6 représente un tel arbre Min-Max.

18.2.2 Stratégie optimale pour un jeu Min-Max

Évaluation des feuilles. Le jeu étant fini, l'arbre associé l'est aussi. Le jeu se termine lorsqu'on aboutit à une feuille f. Pour toute feuille f, on note $\mathcal{S}(f) \in \mathbb{R} \cup \{\pm\infty\}$ le score du joueur Max si cette feuille est atteinte (rappel : le but du joueur Max est de maximiser cette quantité, Min de la minimiser).

Évaluation des nœuds. On peut définir récursivement (en langage savant, on dit par induction) une extension de la fonction $\mathcal{S}$, notée $\mathcal{E}$, à tous les nœuds du graphe de la façon suivante :

$$\mathcal{E}(n) = \begin{cases} \mathcal{S}(n) & \text{si } n \text{ est une feuille ;} \\ \max\{\mathcal{E}(x) \mid x \text{ fils de } n\} & \text{si } n \text{ est contrôlé par le joueur Max ;} \\ \min\{\mathcal{E}(x) \mid x \text{ fils de } n\} & \text{si } n \text{ est contrôlé par le joueur Min.} \end{cases}$$

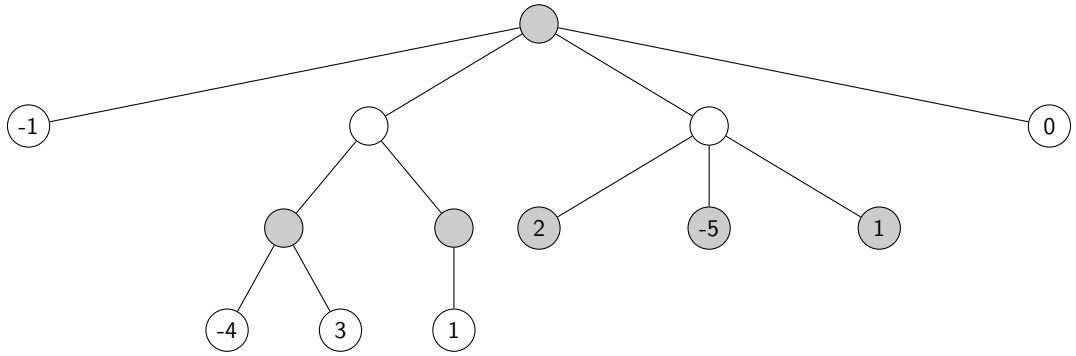
Il est facile de montrer par récurrence que :

— si, lorsque Max doit jouer en un nœud interne n, il joue un fils y de n tel que $\mathcal{E}(y) = \max\{\mathcal{E}(x) \mid x \text{ fils de } n\}$, alors son score à la fin de la partie vaut au moins $\mathcal{E}(n)$;

— si, lorsque Min doit jouer en un nœud interne n, il joue un fils y de n tel que $\mathcal{E}(y) = \min\{\mathcal{E}(x) \mid x \text{ fils de } n\}$, alors le score de Max à la fin de la partie vaut au plus $\mathcal{E}(n)$;

Stratégie optimale pour chaque joueur. Si chaque joueur joue en essayant de maximiser (resp. minimiser) la quantité $\mathcal{E}$ précédente, le score final du joueur Max à la fin de la partie sera $\mathcal{E}(r)$, avec r la racine.

Exercice 7. Dans l'arbre suivant, on a fait figurer dans chaque feuille son score $\mathcal{S}(f)$. Si chaque joueur joue optimalement, quel sera le score du joueur Max à la fin de la partie ? (Calculer la valeur de $\mathcal{E}(n)$ pour chaque nœud n).



18.2.3 Algorithme avec heuristique

Bien sûr, dans un jeu intéressant, l'arbre est beaucoup trop gros pour être exploré entièrement. Si on veut créer une intelligence artificielle capable de jouer assez intelligemment au jeu, il faut :

— être capable d'approximer intelligemment $\mathcal{E}(n)$ pour un nœud n qui n'est pas une feuille : on suppose connue une fonction $\mathcal{H}$ (appelée heuristique) définie sur ces nœuds. Au jeux d'échecs par exemple, il est courant de compter une valeur 9 pour la dame, 5 pour une tour, 3.25 pour un fou ou un cavalier, et 1 pour un pion. En faisant la différence entre les qualités des pièces blanches et noires, on obtient une valeur raisonnable $\mathcal{H}(n)$ d'une position n.

— réduire la profondeur de recherche dans l'arbre à une certaine valeur p : dans l'évaluation $\mathcal{E}(n)$ du score d'un nœud n, on remplace l'évaluation $\mathcal{E}(n')$ des nœuds n' situés à une distance p de n par la valeur $\mathcal{H}(n')$. Ainsi, la complexité n'est pas trop élevée : Aux échecs, une profondeur de 20 est déja conséquente.

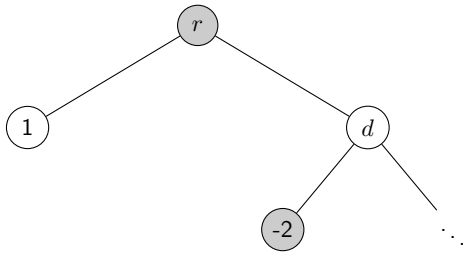
L'algorithme 18.21, récursif, résume cette démarche. On renvoie ici simplement une estimation de $\mathcal{E}(n)$, pour écrire une intelligence artificielle capable de jouer, il suffirait de renvoyer également le prochain coup à jouer, dans le cas d'un nœud interne.

Algorithme 18.21 : $MiniMax(n, p, \mathcal{S}, \mathcal{H})$

Entrée : Un nœud n d'un arbre associé à un jeu Min-Max, un entier $p \geq 0$ (profondeur de recherche), une fonction $\mathcal{S}$ de score sur les feuilles, une heuristique $\mathcal{H}$ sur les autres nœuds.
Sortie : Une estimation de $\mathcal{E}(n)$, score sur le nœud n de l'arbre.
si *n est une feuille* **alors**
  Renvoyer $\mathcal{S}(n)$
si $p = 0$ **alors**
  Renvoyer $\mathcal{H}(n)$
si *n est un nœud max* **alors**
  Renvoyer $\max\{Minimax(f, p-1, \mathcal{S}, \mathcal{H}) \mid f \text{ fils de } n\}$
sinon
  Renvoyer $\min\{Minimax(f, p-1, \mathcal{S}, \mathcal{H}) \mid f \text{ fils de } n\}$
;

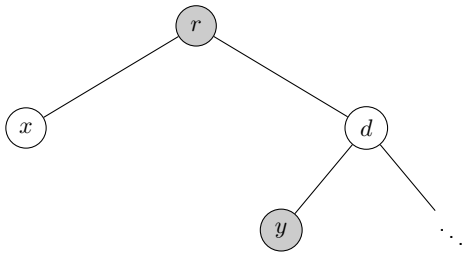
18.2.4 Élagage $\alpha - \beta$ (HP)

Principe. La complexité de l'algorithme 18.21 est en général exponentielle en p : en effet, si par exemple tout nœud interne de l'arbre possède a fils, la complexité est en $O(a^p)$. L'élagage $\alpha - \beta$ est une technique permettant en pratique d'éviter l'évaluation d'un bon nombre de nœuds. Prenons un exemple minimal :

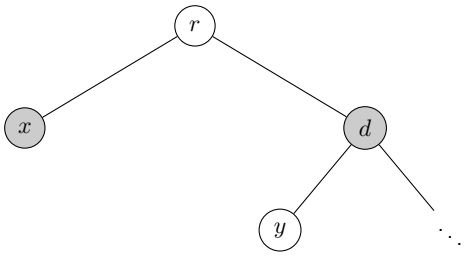


Ici, l'algorithme MiniMax a déja donné la valeur 1 au fils gauche de la racine, et est en train d'évaluer son fils droit d, qui est un nœud Min. Le fils gauche de d a déja été évalué (en -2). Puisque d est un nœud Min, on sait que la valeur qui lui sera attribuée est inférieure ou égale à -2. Il n'est donc pas nécessaire d'évaluer la valeur de d précisément : elle n'interviendra pas dans la valeur attribuée à la racine r, puisque c'est un nœud Max qui aura donc une valeur supérieure ou égale à 1. En particulier, l'exploration du sous-arbre droit de d peut être évitée.

Coupure α et coupure β. La figure 18.7 présente les deux coupures possibles. La coupure α est la coupure présentée précédemment, qui permet de ne pas évaluer certains petits-enfants d'un nœud Max. Symétriquement, une coupure β permet de ne pas évaluer certains petits-enfants d'un nœud Min.



Coupure α : si $\mathcal{E}(x) \geq \mathcal{E}(y)$, il est inutile d'évaluer le reste de la descendance de d.



Coupure β : si $\mathcal{E}(x) \leq \mathcal{E}(y)$, il est inutile d'évaluer le reste de la descendance de d.

FIGURE 18.7 – Coupure α et coupure β

Modification de l'algorithme. On introduit deux nouveaux paramètres α et β dans l'algorithme, qui encadrent les valeurs intéressantes que peut prendre $\mathcal{E}(n)$ pour chaque nœud n, pour obtenir l'algorithme 18.22.

- Pour l'appel initial, il suffit d'utiliser $\alpha = -\infty$ et $\beta = +\infty$;
- Pour un nœud Max n, on introduit un minorant m de la valeur $\mathcal{E}(n)$ (initialisée à $-\infty$). Cette valeur est actualisée à chaque calcul $\mathcal{E}(f)$ sur les fils de n. Si m dépasse β, on a une coupure β : la valeur $\mathcal{E}(n)$ dépassera la valeur intéressante maximale. Si m dépasse simplement α, on peut donner à α la valeur m, ce qui sera utile dans les appels récursifs sur les autres fils de n pour restreindre la plage de valeurs intéressantes.
- Symétriquement, pour un nœud Min n, on introduit un majorant M de la valeur $\mathcal{E}(n)$ (initialisée à $+\infty$). Cette valeur est actualisée à chaque calcul $\mathcal{E}(f)$ sur les fils de n. Si M descend en dessous de α, on a une coupure α. Si M descend simplement en dessous de β, on peut donner à β la valeur M.

Algorithme 18.22 : $MiniMax(n, p, \alpha, \beta, \mathcal{S}, \mathcal{H})$

Entrée : Un nœud n d'un arbre associé à un jeu Min-Max, un entier $p \geq 0$ (profondeur de recherche), une fonction $\mathcal{S}$ de score sur les feuilles, une heuristique $\mathcal{H}$ sur les autres nœuds. Deux valeurs $\alpha < \beta$ qui encadrent les valeurs intéressantes de $\mathcal{E}(n)$.

Sortie : Une estimation de $\mathcal{E}(n)$, score sur le nœud n de l'arbre.

```
si n est une feuille alors
    Renvoyer S(n)
si p = 0 alors
    Renvoyer H(n)
si n est un nœud max alors
    m ← −∞;
    pour chaque fils f de n faire
        v ← Minimax(f, p − 1, α, β, S, H);
        si v > m alors
            m ← v
        si m ≥ β alors
            Renvoyer m   # coupure beta !
        α ← max(α, m);
    Renvoyer m
sinon
    M ← +∞;
    pour chaque fils f de n faire
        v ← Minimax(f, p − 1, α, β, S, H);
        si v < M alors
            M ← v
        si M ≤ α alors
            Renvoyer M   # coupure alpha !
        β ← min(β, M);
    Renvoyer M
;
```