Cours d'Informatique pour Tous

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.

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 4 parties, elles-mêmes subdivisées en 13 chapitres. À ceux-ci s'ajoute un dernier chapitre explicitant brièvement l'usage des modules usuels en Python, notamment Numpy. Les trois premières parties sont relatives au programme de première année, la quatrième au programme de deuxième année. Le plan choisi est le suivant :

— La première partie est dévolue à l'« initiation ». Malgré ce nom, elle est fondamentale, notamment les chapitres 2 et 3. 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. Le cours présenté ici est habituellement 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. L'utilisation des modules n'est pas présentée en détail dans ce chapitre mais reléguée en fin de polycopié.
  — Le chapitre 2 présente la représentation des nombres (entier et flottants) dans un ordinateur. Il est plus détaillé que ce que préconise le programme officiel, mais les algorithmes de changements de base sur les entiers offrent une bonne introduction à l'algorithmique. L'addition des entiers relatifs sur des registres de taille fixée permet de plus de comprendre le fonctionnement d'un processeur. Enfin, la représentation des flottants offre un premier avertissement sur les erreurs d'arrondis.
  — Le chapitre 3 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 est dévolue à l'analyse numérique. Elle est très (peut-être trop) détaillée, mais c'est également un choix pédagogique motivé par l'importance qu'occupent les questions d'analyse numérique dans les concours. Elle se découpe en 5 chapitres.
  — Le chapitre 4 présente une sensibilisation aux erreurs numériques et fait suite au chapitre 2. Peut-être un peu rébarbative, il explique certains comportements « étranges » observés en TP, à cause de l'utilisation des nombres flottants.
  — Le chapitre 5 présente les deux méthodes au programme pour la résolution d'équations numériques : la méthode de la dichotomie et la méthode de Newton.
  — Le chapitre 6 présente la résolution d'équations linéaires via l'algorithme du pivot de Gauss. Là encore, on fait attention aux erreurs d'arrondis.
  — Le chapitre 7 présente des méthodes d'intégration de fonctions. Bien que la seule méthode au programme soit la méthode des rectangles (à gauche), on étudie également des méthodes d'ordre supérieur, notamment la méthode des trapèzes qui fait des apparitions régulières aux concours.
  — Le chapitre 8 présente des méthodes de résolution d'équations différentielles. On fait le lien avec le chapitre précédent : les méthodes de résolution sont vues comme des applications des méthodes d'intégration approchée. Là encore, seule la méthode d'Euler (explicite) est au programme, mais d'autres méthodes font également l'objet de questions aux concours.

— Le chapitre 9, unique chapitre de la partie 3 (Bases de données), présente les bases de SQL et de l'algèbre relationnelle. Le choix de présenter l'algèbre relationnelle (pas clairement au programme) est là aussi motivé par sa présence aux concours.

— La partie 4 (Algorithmique avancée) présente le programme de deuxième année.

  — Le chapitre 10 présente les algorithmes de tris « naïfs », c'est-à-dire quadratiques. C'est une bonne occasion de mettre en pratique les concepts introduits au chapitre 3.

  — Le chapitre 11 présente la structure de pile (seule structure abstraite au programme). Pour justifier le chapitre, plusieurs applications sont données. On présente également une autre structure abstraite, celle de file.

  — Le chapitre 12 introduit la récursivité, en s'appuyant sur le chapitre précédent. Pour ne pas se cantonner aux exemples « triviaux » d'algorithmes récursifs, on présente notamment un algorithme « Diviser pour régner ».

  — Enfin, le chapitre 13 présente, en lien avec le chapitre précédent, les deux algorithmes de tri efficaces au programme : le tri fusion et le tri rapide. On donne également un algorithme de calcul de la médiane en temps linéaire en moyenne, basé sur une variante du tri rapide.

— Le chapitre 14, relégué en annexe, donne une présentation des modules usuels. Bien que leur connaissance ne soit pas exigible à l'écrit des concours, les écrits proposent souvent des questions où ils sont utilisés (notamment Numpy), même si certaines fonctions sont données en formulaire. La connaissance des modules est également utile pour la deuxième épreuve de mathématiques à l'oral de Centrale, ou encore pour effectuer des modélisations dans un TIPE.

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.

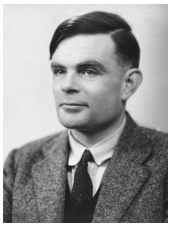


FIGURE 1 – Alan Turing

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).

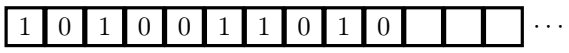


FIGURE 2 – Un ruban d'une machine de Turing

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.

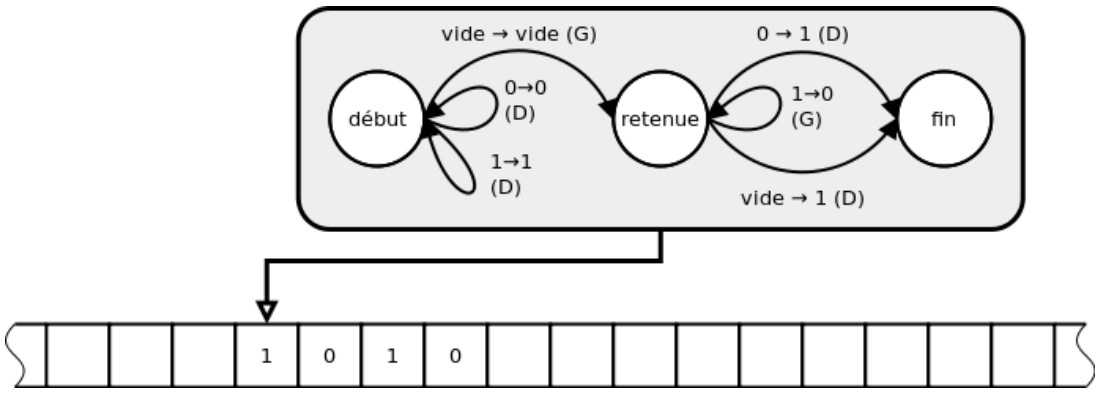


FIGURE 3 – Une machine de Turing permettant d'additionner 1 au nombre écrit sur le ruban

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

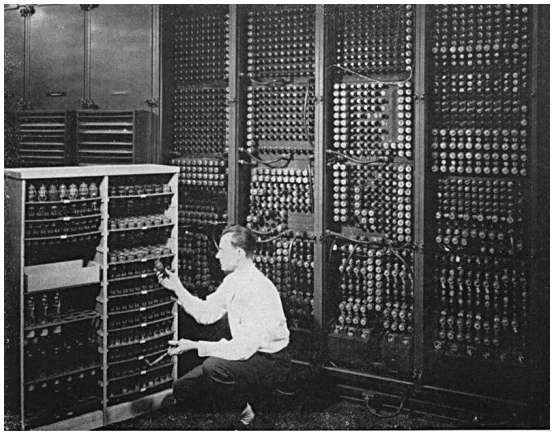


FIGURE 4 – L'ENIAC

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.

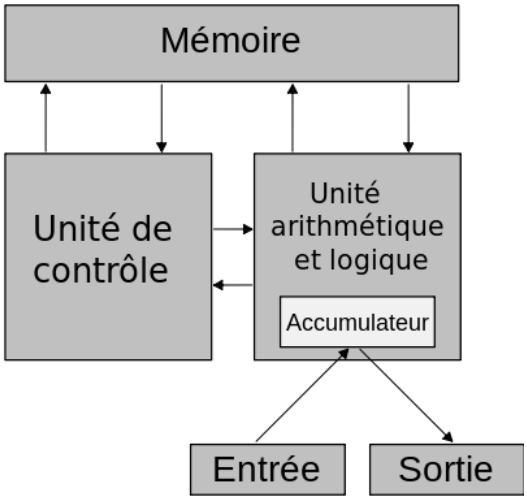


FIGURE 5 – Architecture de Von Neumann

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.

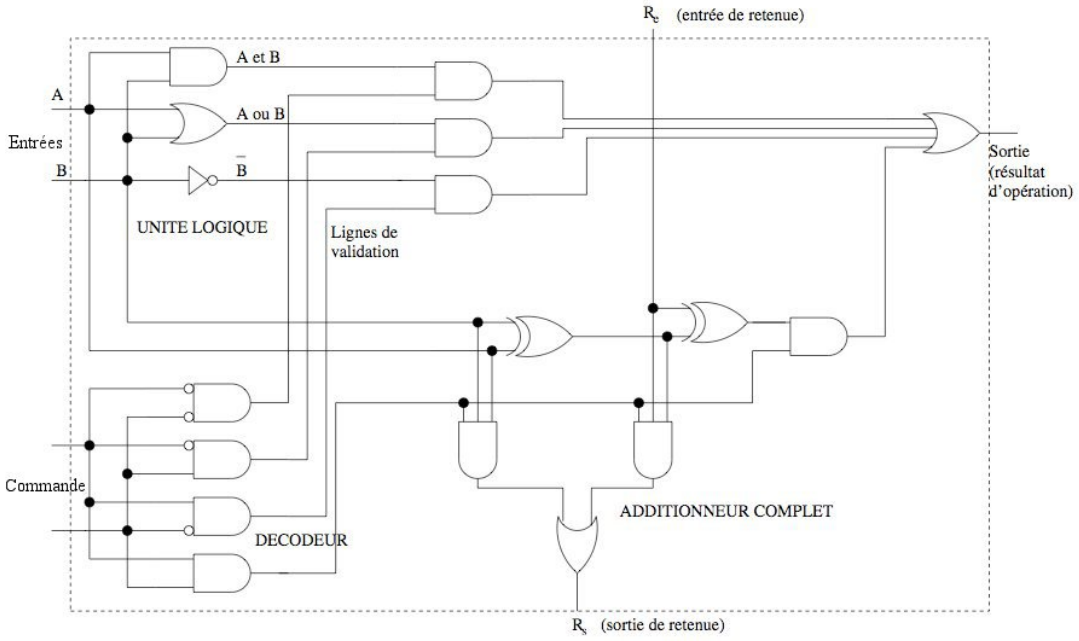


FIGURE 6 – Le circuit (sous forme de portes logiques) d'une unité arithmétique et logique sur des mots de 2 bits

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.

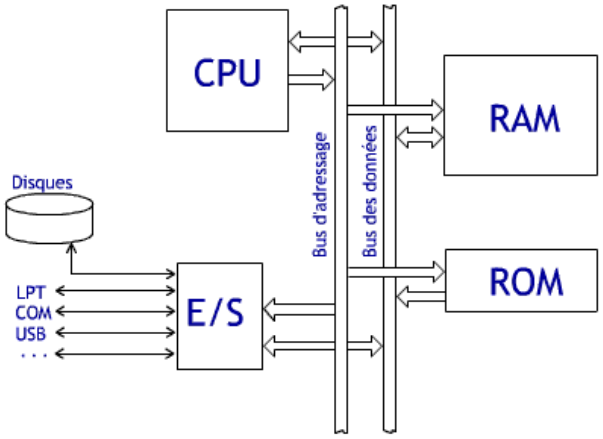


FIGURE 7 – Architecture réelle

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.

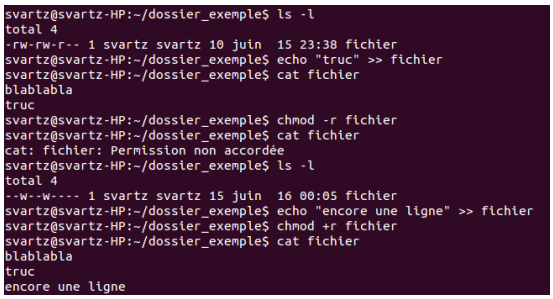


FIGURE 10 – Une suite de commandes modifiant les droits sur un fichier

— `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.

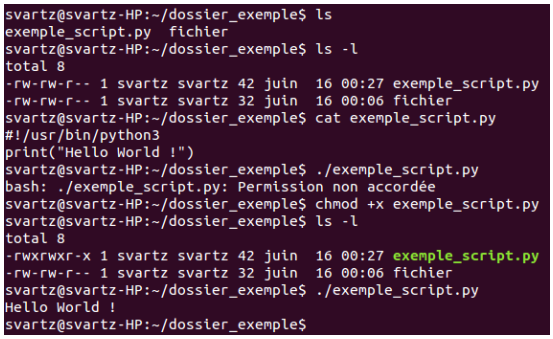


FIGURE 11 – Les droits d'exécution pour un script Python

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'administraeur 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.

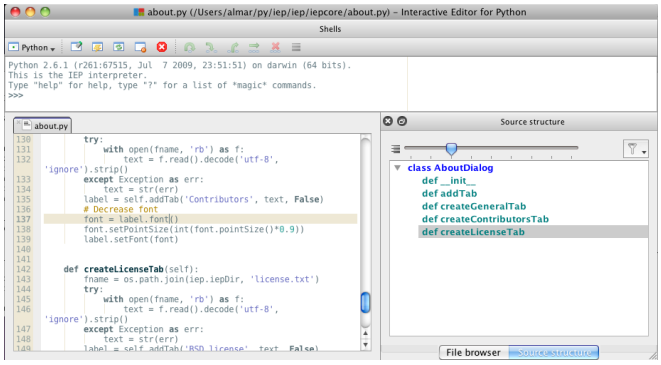


FIGURE 12 – L'environnement Pyzo

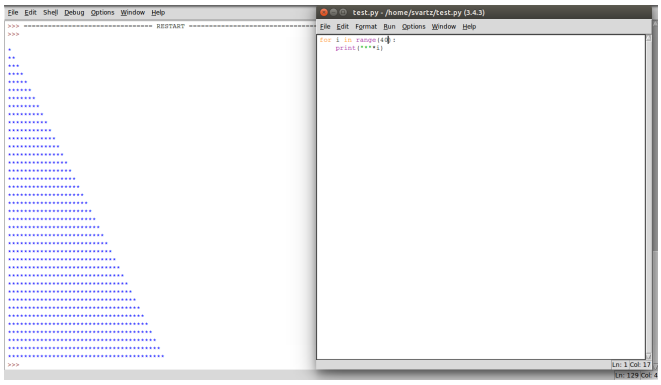


FIGURE 13 – L'environnement Idle

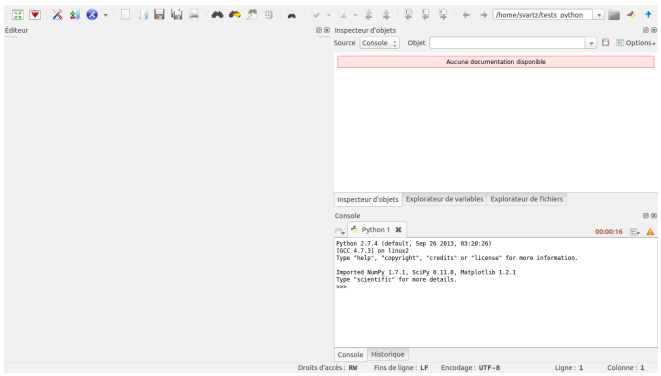


FIGURE 14 – L'environnement Spyder

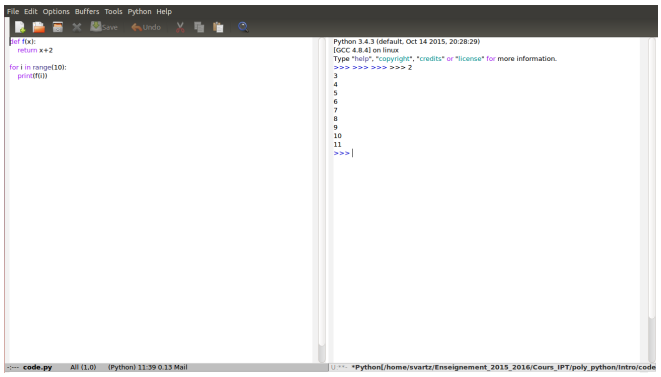


FIGURE 15 – L'environnement Emacs

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 des modules usuels, qui feront l'objet d'un chapitre à part. La version de Python utilisée est Python 3 (toutes les versions 3.x) : attention si dans votre système c'est une version 2.x qui est installée, il y a quelques différences.

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("blabla")
<class 'str'>
>>> type(4<7)
<class 'bool'>
>>> type([0,1,2])
<class 'list'>
```

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...) 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. Modulo.
3. Multiplication et division entières.
4. 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 (voir le cours sur la représentation des nombres). 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 polynômiale 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. 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 [7], 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 : `L=[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. Pour éviter tout test d'égalité sur des 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 [8]. 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).	$O(n\ln(n))$
`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.

8. 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 [9]. 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 l'UTF-8, 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 :

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

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

```
>>> 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[10]. `int`, `float`, `str`,... permettent de convertir un type en un autre.

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 :

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.
```

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

— 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.

— 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*[11], 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[12], 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[13], 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`.

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.
```

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

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[14]. 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
```

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[15].

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

14. Dans les exemples qui précèdent, je ne les ai pas respectées !
15. 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 !

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 Passage par références.

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. 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).

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
```

1.6 Entrées/Sorties

Cette section est consacrée 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.

1.6.1 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 (`' '`), et l'affichage se termine par un retour chariot (`\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
```

1.6.2 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.
`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 2

Entiers, Flottants

2.1 Représentation des entiers naturels

2.1.1 Écriture dans une base

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.

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 :

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

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 : $P(1), \ldots, P(b-1)$ sont vraies car pour N un des entiers parmi $1, \ldots, b-1$, 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 $(a_{n-1}, \ldots, a_0) = (c_{n-2}, \ldots, c_0, r)$ 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.

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 un 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 (base 8). On verra qu'il est très facile de passer du binaire à l'hexadécimal (ou à l'octal) et réciproquement.

2.1.2 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.

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 !

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*[3] (on commence par les moins significatifs, 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.

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 :

3. 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.

$n-2$ 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 :

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

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
```

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 $a_k \in \{0, \ldots, b-1\}$. 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 $b^{\ell-1}$.

2.2 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.

2.2.1 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 [4].

En interne. En pratique, un additionneur n bits correspond au chaînage de n additioneurs 1 bit de la forme de la figure 2.1.

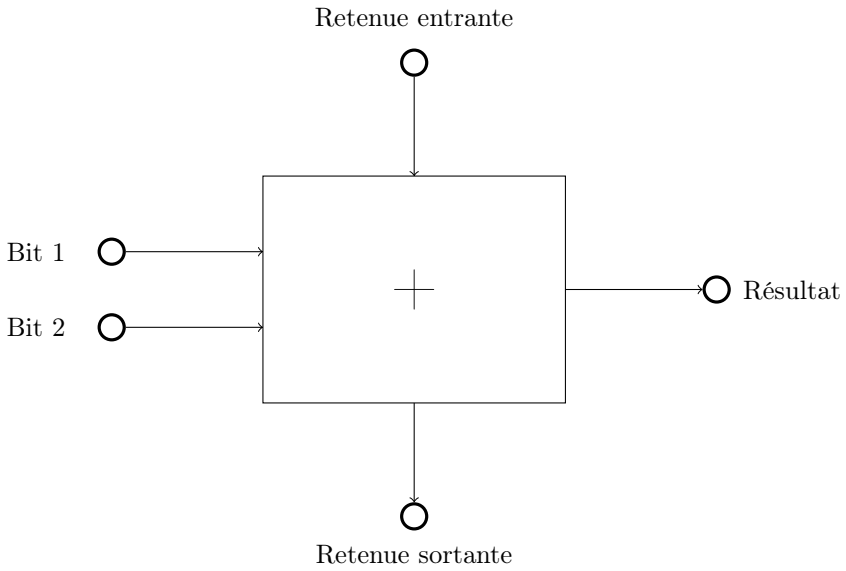


FIGURE 2.1 – Additionneur 1 bit

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

2.2.2 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 :

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

4. 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 2.6 (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*[5] *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 2.7. *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.

5. 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) :

	0^1	1	0	1
+	0	1	1	0
$\boxed{0}$	1	0	1	1

	1	0	0	0
+	1	1	1	1
$\boxed{1}$	0	1	1	1

Remarque 2.8. *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.*

2.2.3 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 [6]. 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 [7] correctement 4444^{4444}. Les longs entiers sont en fait codés par paquets de bits de longueur 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.

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

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.

2.3 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 [8] :

— 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.

2.3.1 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 [9] ». 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.

Définition 2.9. *Dans l'écriture signe$\times$ 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 :

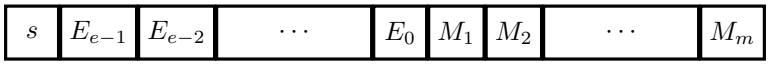


FIGURE 2.2 – La représentation des nombres flottants.

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 $E_0, \ldots, E_{e-1}$ 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).

8. Les chiffres sont tirés de Wikipédia. On choisit d'utiliser la syntaxe anglo-saxonne dans ce cours, la virgule étant notée par un point.
9. Tout cela sera précisé en cours de mathématiques.

2.3.2 Nombres flottants normalisés

On reprend la suite de bits de la figure 2.2, 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.

2.3.3 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 $S \times \underbrace{\overline{1.\cdots}^2}_{\text{mantisse}} \times 2^{-\mathrm{D}}$. 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. 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 par contre, on les obtient facilement !

2.3.4 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 [10]

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 $1 - 2^{-53}$).

10. Le lecteur voulant des précisions sur les règles d'approximation pourra se reporter à l'adresse : `http://fr.wikipedia.org/wiki/IEEE_754`.

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 $|P(x)|\leq 2^{-20}$.

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:
        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 $(x-0.1)^2=x^2-0.2+0.01$:

```
>>> 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 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. 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.

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[2] 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[3] ! 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. La notion essentielle pour montrer la correction des boucles est celle d'*invariant de boucle*[4].

Définition 3.3. *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.

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`.

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

2. 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 !
3. C'est la fameuse conjecture de Syracuse, toujours ouverte.
4. À ne pas confondre avec le *variant de boucle*...

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

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 not L==[],"la liste est vide !"
    return somme(L)/len(L)
```

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...

Terminons cette section par la recherche d'un élément dans une liste, ce qui nous permet de préciser 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.

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.

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

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 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).

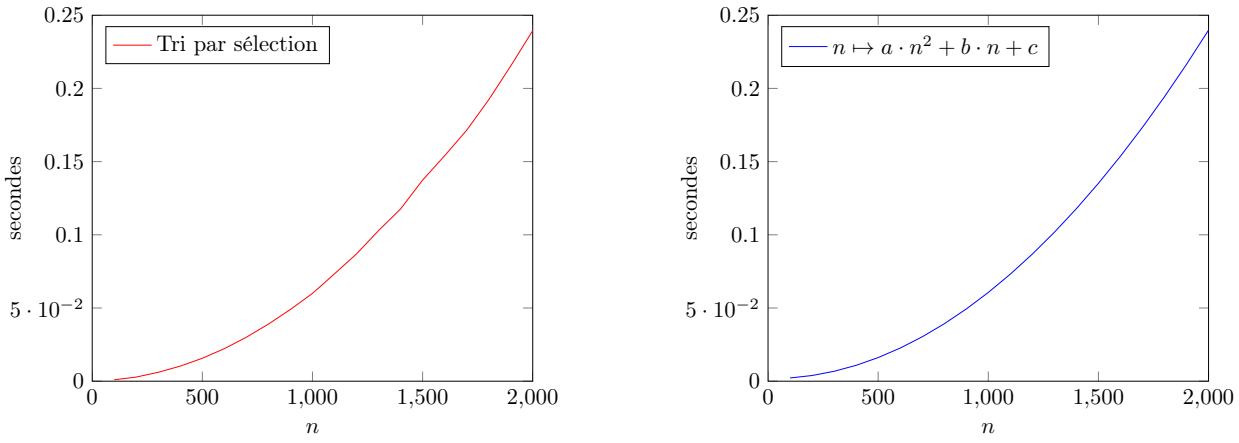


FIGURE 3.2 – Temps pour trier une liste de taille n et approximation par une fonction polynomiale.

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 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 :

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

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 3.4. *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 $n! \mod q$ où q est un nombre fixé, 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.

Maintenant que nous avons le bagage mathématique nécessaire, expliquons comment 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.

— Les opérations d'affectations, entrée/sortie de fonctions, opérations arithmétiques élémentaires... 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.

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 $a > 0$, 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 définit 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 »).

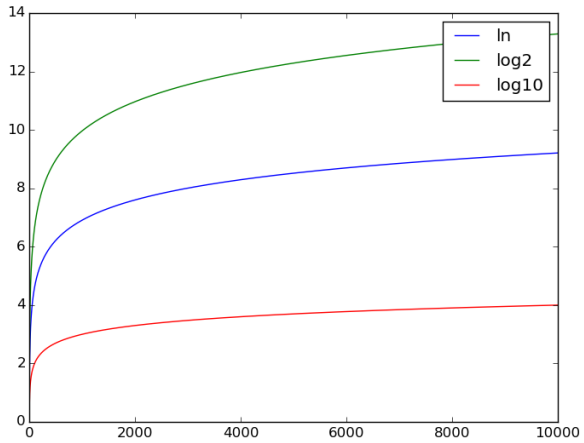


FIGURE 3.3 – Différents logarithmes

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 [5] !

`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).

5. 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 !

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
```

Deuxième partie

Analyse numérique

Chapitre 4

Estimation d'erreurs numériques

Puisque les réels ne sont représentés en machine que sous la forme de flottants, ils ne sont connus que de manière approchée. De plus, la somme ou le produit de deux flottants est également approchée. On va voir dans ce chapitre comment les erreurs se propagent, et on étudiera par l'exemple les phénomènes d'instabilité numérique, en essayant de voir comment limiter les effets néfastes de l'approximation.

4.1 Rappels sur la représentation des nombres réels

On rappelle qu'un flottant *normalisé* se représente avec m bits de mantisse sous la forme

$$x = (-1)^s \times \overline{1.M_1 \cdots M_m}^2 \times 2^E$$

où les $M_i \in \{0, 1\}$ sont les bits de la mantisse, stockés de manière contigue. L'exposant (décalé) et le signe sont également représentés en interne, contrairement au 1 précédant la mantisse. La finitude de m (nombre de bits de la mantisse) implique que les réels ne sont représentés que de manière approchée. Pour x un réel non nul qui n'est ni trop grand, ni trop petit en valeur absolue (pour que l'exposant associé à son développement en base 2 tienne sur le nombre de bits alloués), en considérant $\tilde{x}$ le flottant le plus proche, on s'aperçoit que

$$\left|\frac{x - \tilde{x}}{x}\right| \leq \frac{2^{E-m}}{2^E} = 2^{-m}$$

Ce nombre est la précision relative donnée par les m bits de mantisse, indépendante du réel représenté. Rappelons que sur 64 bits (ce qui est aujourd'hui le nombre de bits usuellement alloués à la représentation d'un flottant), m a pour valeur 52. Ainsi la précision relative est de $2^{-52} \simeq 2.22 \times 10^{-16}$.

Dans la suite, on notera ε cette précision relative.

4.2 Erreurs sur les sommes et produits

Lorsque l'on effectue une somme ou le produit de deux flottants, le résultat est arrondi au flottant le plus proche. Prenons un exemple : si l'on fait l'addition de 1 et de 2^{-100} (tous deux représentables exactement sur 64 bits) alors la somme est arrondie au flottant le plus proche, à savoir 1 lui-même. Peut-on majorer l'erreur effectuée ? C'est ce qu'on va voir maintenant.

4.2.1 Erreur sur la somme

Proposition 4.1. *Soient x et y deux réels représentés exactement. Alors en notant $\Delta(x+y) = \left|\widetilde{x+y} - x - y\right|$ la valeur absolue de la différence entre la « vraie » somme $x+y$ et le résultat calculé sous forme de flottant $\widetilde{x+y}$, on a $\Delta(x+y) \leq \varepsilon(|x| + |y|)$, avec ε la précision relative.*

Démonstration. Admise. □

Il n'est pas dur de se convaincre que la précision relative multipliée par la somme des valeurs absolues majore l'erreur : pour effectuer la somme, on décale le plus petit pour réaliser l'opération bit par bit : on perd juste les bits les moins significatifs, comme en base 10 :

Faisons l'addition de 123.4567 et 45.67834 avec 7 chiffres significatifs :

$$\begin{array}{cccccccccc} 1 & 2 & 3 & . & 4 & 5 & 6 & 7 & \\ & 4 & 5 & . & 6 & 7 & 8 & 3 & 4 \\ \hline 1 & 6 & 9 & . & 1 & 3 & 5 & 0 & 4 \end{array}$$

Le dernier chiffre sur 7 chiffres significatifs est perdu, on a donc une erreur de 4×10^{-5}. Or ici, avec 7 chiffres significatifs, on a une précision relative de 10^{-6} (on a seulement 6 chiffres de mantisse), et 169.13504×10^{-6} majore bien l'erreur commise.

Cumulation d'erreurs. En général, lorsqu'on effectue des calculs en série, les opérandes x et y eux-mêmes ne sont connus qu'à Δx et Δy près. On effectue donc la somme $x' + y'$ avec $|x' - x| = \Delta x$ et $|y' - y| = \Delta y$. L'erreur sur cette somme est majorée par $\varepsilon(|x'| + |y'|) \leq \varepsilon(|x| + |y| + \Delta x + \Delta y)$. En négligeant les termes de la forme $\varepsilon \Delta x$, on obtient que l'erreur entre le résultat obtenu en effectuant $x' + y'$ par rapport à la « vraie » somme $x + y$ se majore par $\varepsilon(|x| + |y|) + \Delta x + \Delta y$.

Somme sur plusieurs termes. Considérons ici une somme $\sum_{i=1}^{N} u_i$ de termes positifs, qu'on cherche à estimer. On peut (mathématiquement) sommer dans n'importe quel sens, mais qu'en est-il sur des flottants ?

Supposons que l'on somme dans l'ordre « naturel » ici. On effectue donc les opérations $s_k = u_k + s_{k-1}$ avec $s_k = \sum_{i=1}^{k} u_i$. D'après la proposition précédente, on a donc une erreur sur s_k qui est majorée par εs_k (les termes sont supposés positifs).

Ainsi, l'erreur cumulée est majorée par

$$\sum_{k=2}^{n} \varepsilon s_k = \varepsilon((n-2)u_1 + (n-2)u_2 + \cdots + u_n)$$

(la somme commence à 2 car $s_1 = u_1$ est sans erreur). On en déduit un principe simple :

> Lorsqu'on effectue une somme de plusieurs termes, il est préférable de sommer d'abord les termes les plus petits en valeur absolue.

Exemple. Illustrons ceci par un exemple. Vous savez peut-être que $\exp(x) = \sum_{k=0}^{+\infty} \frac{x^k}{k!}$. Cette série converge très vite vers sa limite, donc pour N assez grand, $\sum_{k=0}^{N} \frac{x^k}{k!}$ donne une bonne approximation. On définit ci-dessous deux fonctions exponentielles, faisant la somme jusqu'à 100. C'est largement suffisant pour dépasser la précision relative 2^{-52} si on ne s'éloigne pas trop de 0, on devrait donc calculer le flottant le plus proche de $\exp(x)$ de manière exacte.

Dans un sens, et dans l'autre

```
from math import factorial,exp

def exp_1(x):
    s=0
    for i in range(101):
        s+=x**i/factorial(i)
    return s

def exp_2(x):
    s=0
    for i in range(100,-1,-1):
        s+=x**i/factorial(i)
    return s
```

Comparons les deux fonctions avec la valeur donnée par Python pour e :

```
>>> exp(1)-exp_1(1)
-4.440892098500626e-16
>>> exp(1)-exp_2(1)
0.0
```

Dans la deuxième somme, on a sommé les termes les plus petits en premier : ceux de « la fin » de la série. Cet exemple n'est pas forcément le plus pertinent car on reste quand même très proche de la précision relative. Prenons en un autre.

Exemple. Vous savez probablement [1] que $\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$. Là encore, on peut sommer les termes indifféremment dans un sens ou dans l'autre.

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

def somme2(n):
    s=0.
    for i in range(n,0,-1):
        s+=i*i
    return s

def somme_exacte(n):
    return n*(n+1)*(2*n+1)//6
```

Remarquez le `s=0.` dans chacune des deux fonctions, pour forcer Python à travailler avec des flottants (si on avait mis `s=0`, il aurait travaillé avec des entiers, représentés de manière exacte...). Comparons la sortie des deux fonctions de sommation avec la formule exacte (qui donne le résultat exact, puisque c'est un entier) :

```
>>> N=10000000
>>> somme(N)-somme_exacte(N)
382074880.0
>>> somme2(N)-somme_exacte(N)
920649728.0
```

L'erreur reste du même ordre de grandeur, mais on est là aussi plus précis lorsqu'on somme les plus petits termes d'abord (ce qui correspond à `somme`).

4.2.2 Erreur sur le produit

On ne va pas trop s'étendre sur les erreurs apparaissant dans les produits, parce que contrairement aux sommes de plusieurs termes, l'erreur apparaissant dans un produit de plusieurs termes ne dépend pas vraiment de l'ordre dans lequel le produit est évalué.

Proposition 4.2. *Soient x et y deux réels représentés exactement. Alors en notant $\Delta(xy) = |\widetilde{xy} - xy|$ la valeur absolue de la différence entre le « vrai » produit xy et le résultat calculé sous forme de flottant $\widetilde{xy}$, on a $\Delta(xy) \leq \varepsilon\,|xy|$, avec ε la précision relative.*

Démonstration. Admise. □

Cumulation d'erreurs. De même qu'avec la somme, si x et y ne sont connus que de manière approchée à Δx et Δy près, alors en négligeant les termes de la forme $\Delta x \Delta y$ ou $\varepsilon \Delta x$ (qui sont « d'ordre 2 »), on obtient $\Delta(xy) \leq |y|\,\Delta x + |x|\,\Delta y + \varepsilon\,|xy|$.

Erreur sur le produit. Pour des réels $(x_i)_{1\leq i\leq n}$ représentés sans erreurs, on montre facilement comme pour la somme que l'erreur commise sur le produit est $\Delta(x_1 \cdots x_n) \leq (n-1)\varepsilon\,|x_1 \cdots x_n|$. Cette estimation ne dépend pas de l'ordre choisi pour les facteurs !

4.3 Phénomènes d'instabilité et remèdes

On étudie maintenant par l'exemple les phénomènes pouvant induire de l'instabilité numérique.

4.3.1 Phénomènes de compensation

Le phénomène de compensation se produit lorsque l'on soustrait deux nombres très proches : on perd beaucoup de précision sur le résultat. Ce problème se retrouve souvent en analyse numérique :

1. Ce qui se démontre aisément par récurrence, parmi d'autres méthodes.

— lorsque l'on veut estimer le nombre dérivé d'une fonction f en x, on peut par exemple calculer $\frac{f(x+h)-f(x)}{h}$ avec h petit. Si h est très petit, la précision est *a priori* meilleure mathématiquement, mais la différence $f(x+h)-f(x)$ peut subir une grand perte de précision ;

— dans un algorithme comme le pivot de Gauss, on est souvent amené à faire des soustractions. Il se peut que ce qu'on calcule soit assez imprécis.

Montrons quelques exemples, et essayons lorsque c'est possible de limiter les erreurs.

Exemple. Commençons par une évaluation simple :

```
>>> 1+10**-15-1
1.1102230246251565e-15
```

Rappelons que l'évaluation se fait de gauche à droite pour l'opérateur +. Lors du calcul de $1+10^{-15}$, on perd beaucoup d'information sur le 10^{-15}, qui est très proche de la précision relative ε (on travaille sur 64 bits). On soustrait ensuite 1, très proche de $1+10^{-15}$, pour obtenir une erreur de plus de 10% sur le résultat théorique 10^{-15}.

Exemple. Évaluation du nombre dérivé. On va prendre l'exemple de la fonction $f : x \mapsto \exp(x)$, dont on souhaite estimer le nombre dérivé en 1. On le connaît : c'est e lui même ! Mais chercher une approximation va nous permettre de vérifier l'erreur commise. On approche donc $\mathrm{e} = \exp'(1)$ par $\varphi(h) = \frac{\exp(1+h)-\mathrm{e}}{h}$, et il faut choisir h petit pour estimer précisément $\exp'(1)$. Mathématiquement, on a intérêt à prendre h le plus petit possible, mais ce n'est pas le cas en informatique :

```
>>> def phi(h): return (exp(1+h)-e)/h
...
>>> phi(2**-52)
4.0
>>> e
2.718281828459045
```

Pour h plus petit, c'est encore pire, car $\exp(1+h)$ et e sont indiscernables lorsqu'ils sont arrondis en flottants (sur 64 bits), on obtient donc 0. D'un point de vue informatique, on a intérêt à choisir h grand pour minimiser l'erreur commise par la soustraction de deux flottants qui seraient trop proches :

```
>>> phi(0.001)
2.7196414225332255
```

On a déja trois chiffres significatifs en prenant 10^{-3}, mais pour savoir quel h est le meilleur, faisons une petite analyse réalisant un compromis entre les deux erreurs.

- Comme $\exp(1+h) \simeq \mathrm{e} \times (1+h+\frac{h^2}{2})$, l'erreur mathématique commise est de l'ordre de $\frac{\mathrm{e}\times h}{2}$;
- D'un point de vue informatique, l'erreur commise dans la soustraction $\exp(1+h)-\mathrm{e}$ est majorée par $\varepsilon(\exp(1+h)+\mathrm{e}) \simeq 2\,\mathrm{e}\,\varepsilon$, donc on obtient une erreur d'approximation majorée par $\frac{2\,\mathrm{e}\,\varepsilon}{h}$.
- On a donc intérêt à avoir $\frac{\mathrm{e}\times h}{2} \leq \frac{2\,\mathrm{e}\,\varepsilon}{h}$ (car la deuxième estimation n'est qu'une majoration de l'erreur, alors que la première est un équivalent), les deux devant être assez proches. On obtient $h^2 \leq 4\varepsilon$, donc $h \leq 2^{-25}$, car $\varepsilon = 2^{-52}$. Essayons :

```
>>> e
2.718281828459045
>>> phi(2**-25)
2.7182818800210953
>>> phi(2**-26)
2.718281865119934
```

C'est mieux ! Le lecteur suspicieux pourra vérifier que notre raisonnement est très bon : $\varphi(2^{-26})$ donne la valeur de e la plus précise parmi les $f(2^{-i})$.

Remarques.

— Le h « optimal » dépend de la fonction à approcher, mais pas tant que ça : pour f une fonction deux fois dérivable en x, dont on cherche à approcher $f'(x)$ par la méthode ci-dessus, on trouve une erreur mathématique de l'ordre de $\frac{h}{2}|f''(x)|$ et une erreur informatique majorée par $\frac{2|f(x)|\varepsilon}{h}$. Si $|f''(x)|$ et $|f(x)|$ sont de l'ordre de 1, on trouve un h optimal sensiblement égal à celui trouvé pour exp.

— Pour approcher le nombre dérivé $f'(x)$ d'une fonction f en x, une formule bien meilleure est $\frac{f(x+h)-f(x-h)}{2h}$, car l'erreur mathématique est plus faible. On peut faire la même étude que précédemment avec la fonction exponentielle, et trouver un h optimal :

```
>>> def f(h): return (exp(1+h)-exp(1-h))/(2*h)
...
>>> e
2.718281828459045
>>> f(2**-18)
2.7182818284491077
```

Exemple : résolution d'une équation du second degré. Pour une équation de la forme $ax^2+bx+c=0$ dont le discriminant $\Delta = b^2-4ac$ est positif, les deux racines sont données par $\frac{-b\pm\sqrt{\Delta}}{2a}$. Si $|4ac|$ est petit devant b^2, alors on se retrouve face à un problème de compensation sur l'évaluation d'une des deux racines car $\sqrt{\Delta}$ et $|b|$ sont proches.

Prenons un exemple en base 10. Considérons l'équation $x^2-1634x+2=0$, et supposons que l'on dispose de 10 chiffres significatifs. Comme b est pair, on calcule plutôt $\Delta' = \frac{b^2}{4}-ac$. Ainsi :

$$\begin{array}{lll} \Delta' = 667487 & \text{donc} & \sqrt{\Delta'} \simeq 816.9987760 \\ x_1 = -\frac{b}{2}+\sqrt{\Delta'} & \text{et} & x_1 \simeq 1633,998776 \\ x_2 = -\frac{b}{2}-\sqrt{\Delta'} & \text{et} & x_2 \simeq 0.0012240 \end{array}$$

On n'a plus que 5 chiffres significatifs sur x_2 ! Pour y remédier, on peut remarquer que $x_1x_2 = 2$. Par suite, $x_2 = \frac{2}{x_1} \simeq 1.223991125\times10^{-3}$, et ce calcul permet de récupérer nos chiffres significatifs perdus.
On peut obtenir la même chose en Python.

Deux calculs de x_2

```
0.0012239911249025681  # obtenu avec la première version
0.001223991124941416 # obtenu avec la deuxième
```

Vérifions avec `scipy` :

```
>>> import scipy
>>> scipy.roots([1,-1634,2])[1]
0.0012239911249414159
```

On peut retenir un autre principe simple :

> On essaiera au maximum d'éviter les sommations dans lesquelles des termes proches en valeur absolue se compensent.

Exemple. Reprenons notre calcul de l'exponentielle par la formule $\exp(x) = \sum_{k=0}^{+\infty}\frac{x^k}{k!}$, qu'on tronquera encore à $k=100$. On veut ce coup-ci calculer $\exp(-10)$ (le reste de la série est beaucoup plus petit que $\varepsilon\exp(-10)$, avec ε la précision relative sur 64 bits).

Dans la somme $\sum_{k=0}^{100}\frac{(-10)^k}{k!}$, les deux termes les plus grands en valeur absolue sont donnés pour $k=9$ et $k=10$, avec $\frac{(-10)^{10}}{10!} = -\frac{(-10)^9}{9!} \simeq 2755.731922398589$. Comme $\exp(-10)$ est de l'ordre de 10^{-5} on a une perte de l'ordre de 9 chiffres significatifs si on évalue la somme telle quelle.

En effet, en reprenant la fonction `exp_2` donnée plus haut :

```
>>> exp_2(-10)
4.5399929291534136e-05
>>> from math import exp ; exp(-10)
4.5399929762484854e-05
```

Là encore, le remède est simple et utilise le principe évoqué plus haut : on utilise simplement la relation $\exp(-x) = \frac{1}{\exp(x)}$.

```
>>> 1/exp_2(10)
4.539992976248484e-05
```

4.3.2 Problèmes mal posés

On étudie ici des problèmes de suites définis par une récurrence, qui bien que mathématiquement corrects, donnent très vite des résultats abérrants à cause des erreurs d'approximation.

Un calcul d'intégrale menant à une suite récurrente. On souhaite calculer une approximation numérique de l'intégrale suivante :

$$I_n = \int_0^1 \frac{x^n \,\mathrm{d}x}{10+x}$$

Parmi plusieurs méthodes possibles, on peut chercher une relation de récurrence entre deux termes successifs de la suite. On obtient facilement :

$$I_0 = [\ln(10+x)]_0^1 = \ln\left(\frac{11}{10}\right) \qquad \text{et} \qquad I_n = \int_0^1 \frac{x^{n-1}(10+x-10)\,\mathrm{d}x}{10+x} = \int_0^1 x^{n-1}\,\mathrm{d}x - 10I_{n-1} = \frac{1}{n} - 10I_{n-1}$$

Observons la suite des valeurs données par Python avec ce calcul itératif, en supposant que l'on souhaite calculer I_{18} de manière approchée :

```
I_0=0.09531017980432493
I_1=0.04689820195675065
I_2=0.031017980432493486
I_3=0.023153529008398455
I_4=0.018464709916015454
I_5=0.015352900839845474
I_6=0.013137658268211921
I_7=0.011480560175023635
I_8=0.010194398249763648
I_9=0.009167128613474629
I_10=0.008328713865253717
I_11=0.007621952256553738
I_12=0.00711381076779595
I_13=0.00584969245117427
I_14=0.013578789773971 52
I_15=-0.06912212310730485
I_16=0.7537212310730486
I_17=-7.478388781318721
I_18=74.83944336874276
```

Si le calcul semble réaliste au début, cela devient très vite n'importe quoi. En effet, I_n est de manière évidente compris entre 0 et 1/10. Bien que la récurrence soit mathématiquement correcte, on peut expliquer ce comportement désastreux. Même en supposant que les $1/n$ soient calculés exactement, on s'aperçoit qu'une erreur (absolue) sur I_{n-1} est multipliée par 10 lors du calcul de I_n.

Un remède simple ici consiste à *renverser la récurrence*, en utilisant plutôt la relation $I_{n-1} = \frac{1}{10}\left(\frac{1}{n} - I_n\right)$. Il faut donc partir de plus loin, si l'on souhaite obtenir I_{18}. Pour ce faire, il nous faut une approximation de I_n qu'on peut calculer assez grossièrement :

$$I_n \simeq \int_0^1 \frac{x^n \,\mathrm{d}x}{10} = \frac{1}{10(n+1)}$$

En partant de $I_{35} \simeq 1/360$, on obtient :

```
I_35=0.002777777777777778
I_34=0.0025793650793650793
I_33=0.0026832399626517275
I_32=0.002761979034037858
```

```
I_31=0.0028488020965962146
I_30=0.002940926241953282
I_29=0.0030392407091380056
I_28=0.0031443517911551653
I_27=0.003256993392313055
I_26=0.00337800436447239 8
I_25=0.0035083534097066064
I_24=0.0036491646590293397
I_23=0.0038017502007637325
I_22=0.003967651066880149
I_21=0.004148689438766531
I_20=0.004347035818028109
I_19=0.0045652964181971895
I_18=0.004806628252917123
```

Calculons maintenant avec `scipy` :

```
>>> from scipy.integrate import quad ; quad(lambda x:x**18/(x+10),0,1)
(0.004806628252917125, 5.3364293572023825e-17)
```

Remarquez que `quad` fournit également un terme d'erreur. On est donc très précis avec notre renversement de récurrence. Comme ici, l'erreur sur I_{n-1} est à peu près celle sur I_n divisée par 10 (sans compter l'approximation sur $1/n$), on obtient théoriquement une valeur de I_{18} à 10^{-17} près. On aurait pu estimer beaucoup plus grossièrement le terme de départ, à condition de partir d'un peu plus loin : on obtient le même résultat sur I_{18} en partant de l'approximation [2] $I_{50} \simeq 10^{20}$.

Suites définies par une relation de récurrence de la forme $u_{n+1} = f(u_n)$. Dans la même veine que l'étude précédente, on peut se demander comment évoluent les erreurs dans un calcul d'une suite (u_n) définie par une récurrence de la forme $u_{n+1} = f(u_n)$. Donnons une estimation de l'erreur relative $\frac{\Delta u_{n+1}}{|u_{n+1}|}$ d'un terme en fonction du précédent.

Si $\tilde{x}$ et proche de x, alors $f(\tilde{x}) \simeq f(x) + f'(x)(\tilde{x} - x)$. L'erreur absolue est donc multipliée par $|f'(x)|$. Ce qui nous intéresse est plutôt l'erreur relative, qu'on calcule comme suit :

$$\frac{\Delta f(x)}{|f(x)|} \simeq \frac{|f(x) - f(\tilde{x})|}{|f(x)|} \simeq \left|\frac{xf'(x)}{f(x)}\right| \frac{\Delta x}{|x|}$$

La quantité $\kappa(x) = \left|\frac{xf'(x)}{f(x)}\right|$ se nomme le *conditionnement*. Si, lors du calcul de termes de la suite $(u_n)_{n\in\mathbb{N}}$, on se trouve dans un intervalle sur lequel $\kappa(x) > 1$, il faudra faire attention car les erreurs relatives seront dilatées.

Considérons par exemple la suite définie par :

$$\begin{cases} u_0 = 2 \\ u_{n+1} = |\ln(u_n)| \quad \text{pour } n \geq 0. \end{cases}$$

Le conditionnement associé à la fonction $x \mapsto |\ln(x)|$ en $x > 0$ est donné par $\kappa(x) = \frac{1}{\ln(x)}$. On donne ici quelques valeurs approchées de la suite et les conditionnements associés.

2. Oui, c'est extrèmement grossier !

$u_0 = 2$	
$u_1 = 0.6931471805599453$	$\kappa(u_0) = 1.4426950408889634$
$u_2 = 0.36651292058166435$	$\kappa(u_1) = 2.7284167729011495$
$u_3 = 1.00372150430231$	$\kappa(u_2) = 0.9962922939417376$
$u_4 = 0.003714596637805006$	$\kappa(u_3) = 269.2082337615291$
$u_5 = 5.595485183341199$	$\kappa(u_4) = 0.17871551210200434$
$u_6 = 1.7219600553126855$	$\kappa(u_5) = 0.5807335640073329$
$u_7 = 0.5434632090539631$	$\kappa(u_6) = 1.8400509608382805$
$u_8 = 0.6097932673451253$	$\kappa(u_7) = 1.6399000342423082$
$u_9 = 0.49463528524799216$	$\kappa(u_8) = 2.0216915974739575$
$u_{10} = 0.7039345854609151$	$\kappa(u_9) = 1.4205865440539907$
$u_{11} = 0.3510698455206566$	$\kappa(u_{10}) = 2.8484360384667697$
$u_{12} = 1.0467700852248318$	$\kappa(u_{11}) = 0.9553196199576278$
$u_{13} = 0.04570931391055714$	$\kappa(u_{12}) = 21.8773793445418$
$u_{16} = 0.11929171903512947$	$\kappa(u_{15}) = 8.382811548767407$
$u_{24} = 1.0009714389923408$	$\kappa(u_{23}) = 0.9990295037855239$
$u_{25} = 0.0009709674508406605$	$\kappa(u_{24}) = 1029.9006409887409$

De temps en temps, u_n est très proche de 1, et le conditionnement est grand. Si ici l'erreur est encore faible (le produit des conditionnements rencontrés reste bien inférieur à 10^{16}) ; très vite, les valeurs calculées ne seront plus pertinentes. Malheureusement, il n'y a pas vraiment de solution au problème ici !

Chapitre 5

Résolution d'équations numériques

On souhaite ici résoudre numériquement des équations sur les réels. On se limite à des équations à une seule variable, de la forme $f(x) = 0$, où f est une fonction qu'on supposera au minimum continue. On va principalement décrire deux algorithmes pour résoudre cette équation :

— la méthode dichotomique, qui fonctionne à tous les coups, et permet d'avoir une précision (nombre de bits significatifs) linéaire en le nombre d'itérations effectuées ;

— la méthode de Newton ; beaucoup plus efficace, car la précision double à chaque itération (on dit que la méthode est quadratique). Malheureusement, les conditions suffisantes d'application de la méthode sont assez draconiennes et pas forcément vérifiées en pratique.

Il existe en Python des fonctions du package numpy pour ces deux méthodes. Mais il est très intéressant de les programmer en pratique, d'autant que c'est exigible aux concours.

5.1 Motivation

Vous connaissez déja des méthodes *formelles* de résolution d'équations sur les réelles, par exemple la résolution des équations polynomiales de degré 2. Beaucoup d'exercices en classe préparatoire consistent à donner les solutions explicites à des équations numériques impliquant les fonctions usuelles (fonctions trigonométriques, exponentielles et logarithmes). Dans ce chapitre, on va travailler avec des méthodes de résolution *approchée* (numérique). Pourquoi résoudre des équations avec des méthodes numériques ? Deux raisons à cela :

— premièrement, parce que dans les sciences expérimentales et numériques, les données ne sont pas connues avec une précision infinie, il est donc inutile de chercher une solution exacte à nos équations : une solution approchée convient très bien et s'obtient en général plus rapidement ;

— deuxièmement, il n'est pas toujours possible de résoudre formellement les équations. Par exemple, il est impossible en général de résoudre formellement les équations polynomiales de degré supérieur ou égal à 5 [1]. Et même si il existe des méthodes formelles [2] pour résoudre les équations polynômiales de degré inférieur à 4, on peut s'interroger sur la pertinence pratique de savoir que l'unique solution réelle de l'équation

$$6x^3 - 6x^2 + 12x + 7 = 0$$

est $\frac{1}{3}\left(\sqrt[3]{\frac{5}{2}} - \sqrt[3]{50} + 1\right)$ plutôt qu'environ -0.442274230114311.

5.2 Méthode de la dichotomie

Le principe de cette méthode repose sur le théorème des valeurs intermédiaires. Si deux réels $a < b$ tels que f est continue sur $[a, b]$ vérifient $f(a)f(b) \leq 0$, alors f possède (au moins) un zéro dans l'intervalle $[a, b]$. Si on coupe l'intervalle en deux en posant $c = \frac{a+b}{2}$, alors f possède un zéro sur au moins l'un des deux intervalles $[a, c]$ et $[c, b]$. On choisit le bon intervalle qui maintient *l'invariant* stipulant que f change de signe entre les extrémités de l'intervalle, ce qui assure l'existence d'un zéro. Ainsi, pour déterminer un zéro de f avec une précision arbitraire, il suffit de raffiner l'intervalle d'encadrement d'un zéro par division successives par 2.

1. ce qui nous vient principalement des travaux d'Abel et de Galois...
2. appelées méthodes de Cardan et de Ferrari pour les degrés 3 et 4.

```
def dichotomie(f,a,b,eps):
    """Retourne une approximation d'un zéro de f par la méthode de la dichotomie.
    f supposée continue sur [a,b], à valeurs dans R, telle que f(a)f(b)<=0
    Calcul x tel que x soit un zéro de f à eps>0 près."""
    fa=f(a)
    fb=f(b)
    assert fa*fb<=0
    #Inv: f(a)f(b)<=0
    while b-a>2*eps:
        m=(a+b)/2
        fm=f(m)
        if fa*fm<=0:
            b,fb=m,fm
        else:
            a,fa=m,fm
    return (a+b)/2
```

Puisqu'on renvoie $\frac{a+b}{2}$ à la fin de l'algorithme, et que f possède un zéro sur l'intervalle $[a,b]$, avec $(b-a)\leq 2\varepsilon$, on est sûr d'avoir un zéro à ε près. Stocker les valeurs de $f(a)$ et de $f(b)$ permet de limiter le nombre d'appels à f qui peuvent être coûteux, mais on pourrait fort bien s'en passer dans l'écriture du code.

Correction. Admettons temporairement la terminaison de l'algorithme et montrons que la proposition $f(a)f(b)\leq 0$ évoquée est bien un invariant de la boucle `while` :

— La proposition est vraie avant la boucle (on le suppose !)

— Si elle est vrai en haut de la boucle, alors on considère $m=\frac{a+b}{2}$. Si $f(a)f(m)\leq 0$, alors on affecte la valeur m à b, donc la proposition est vérifiée en bas de la boucle. Sinon, c'est que $f(a)f(m)>0$. Mais alors $f(m)f(b)$ est du même signe que $f(m)f(b)\times f(a)f(m)=f(m)^2f(a)f(b)\leq 0$ (en effet, $f(m)^2$ est positif). Par suite, en affectant à a la valeur m, la proposition $f(a)f(b)\leq 0$ est bien vérifiée en bas de la boucle.

Ainsi, $f(a)f(b)\leq 0$ est un invariant de boucle, et est en particulier vrai *après* la boucle, donc f admet un zéro sur l'intervalle $[a,b]$. Or, après la boucle, on a $b-a\leq 2\varepsilon$, donc $\frac{a+b}{2}$ est à distance inférieure ou égale à ε d'un zéro de f. L'algorithme renvoie donc bien une approximation d'un zéro de f à ε près.

Exemple. Prenons un exemple classique, le calcul approché de $\sqrt{2}$. En affichant les valeurs successives prises par m dans le programme précédent, on obtient :

```
>>> print(sqrt(2)) ; dichotomie(lambda x:x*x-2,0,2,10**(-7))
1.4142135623730951
1.0
1.5
1.25
1.375
1.4375
1.40625
1.421875
1.4140625
1.41796875
1.416015625
1.4150390625
1.41455078125
1.414306640625
1.41418457031 25
1.41424560546875
1.414215087890625
1.4141998291015625
1.4142074584960938
1.4142112731933594
1.4142131805419922
1.4142141342163086
1.4142136573791504
1.4142134189605713
1.4142135381698608
1.4142135977745056
```

Remarquez l'utilisation de la fonction *anonyme* $x\mapsto x^2-2$ déclarée à l'aide de l'opérateur λ. Le nombre de chiffres décimaux exacts progresse d'un toutes les 3 ou 4 étapes, ce qui n'est pas étonnant car $2^3<10<2^4$.

Terminaison et complexité. Dans la même veine que la recherche dichotomique dans un tableau trié, l'intervalle sur lequel on travaille après k itérations de la boucle `while` est de longueur $\frac{b-a}{2^k}$ (ici, les valeurs de a et b sont celles passées en entrée de la fonction). On s'arrête lorsque $\frac{b-a}{2^k} \leq 2\varepsilon$, inégalité vérifiée pour $k \geq \log_2\left(\frac{b-a}{\varepsilon}\right) - 1$. Si $\varepsilon = 2^{-p}$ (pour avoir p bits significatifs), l'algorithme s'arrête dès que le nombre d'itération de la boucle atteint ou dépasse $p + \log_2(b-a) - 1$, ce qui montre du même coup la terminaison. Lorsque a et b sont fixés, il nous faut donc de l'ordre de p étapes pour avoir p bits significatifs : on dit que la méthode est d'ordre 1. En supposant que chaque appel à f ait un coût constant $O(1)$, la complexité est donc en $O(p)$.

La méthode de la dichotomie (*bisection* en anglais), est implémentée dans le sous-module `scipy.optimize` du module `scipy` (pour *scientific python*). Les modules ne sont pas à connaître, mais il est bon de savoir que ça existe déja !

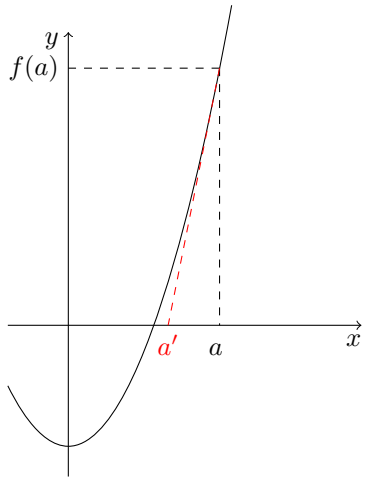
```
>>> import scipy.optimize
>>> scipy.optimize.bisect(lambda x:x*x-2,0,5)
1.4142135623734475
```

5.3 Méthode de Newton

5.3.1 Principe et code Python

La méthode de Newton est également une méthode de résolution numérique, beaucoup plus rapide que la méthode dichotomique, mais elle demande des conditions d'application assez restrictives pour assurer la convergence vers une solution de $f(x) = 0$.

La méthode est résumée dans le schéma suivant : On suppose la fonction f dérivable, on part d'une abscisse x_0 (a priori quelconque), et on itère à partir de x_0 le procédé $a \mapsto a'$ (voir figure), où a' est l'intersection de la tangente de f en a avec l'axe des abscisses.



Une itération de la méthode de Newton

Il faut savoir écrire l'itération effectuée : la tangente à f en a a pour équation $y - f(a) = f'(a)(x-a)$. Par suite, son intersection avec l'axe des abscisses est donnée par $x = a - \frac{f(a)}{f'(a)}$. La méthode de Newton avec un nombre fixé d'itérations est alors très facile à écrire :

```
def Newton(f,g,x0,n):
    """Retourne une approximation d'un zéro de f par la méthode de Newton.

    f: fonction dérivable.
    x0: un point pas trop éloigné d'un zéro de f
    g: dérivée de f
    n: nombre d'itérations
    La méthode marche en particulier si f C1, convexe sur un voisinage de x0,
    de dérivée strictement positive."""

    x=x0
    for i in range(n):
```

```
        x=x-f(x)/g(x)
    return x
```

Dans le code précédent, g est la dérivée de f, donnée en paramètre, et N est le nombre d'itérations. Donnons un petit exemple : $f : x \mapsto x^2 - 2$. En partant du réel 5, on obtient en 6 itérations une très bonne approximation de $\sqrt{2}$. Au départ, la convergence est un peu lente, mais très vite on double le nombre de chiffres significatifs à chaque étape. Voici les itérations effectuées (on a ajouté un `print` dans la fonction) :

```
>>> newton(lambda x: x*x-2,lambda x:2*x,5,6)
5.
2.7
1.7203703703703703
1.44145536817765
1.414470981367771
1.4142135857968836
1.4142135623730951
>>> from math import sqrt ; sqrt(2)
1.4142135623730951
```

On atteint très vite la valeur calculée par Python, ce qui est remarquable ! Il est possible de démontrer qu'effectivement la méthode donne rapidement une approximation d'un zéro de f sous certaines hypothèses.

5.3.2 Convergence de la méthode de Newton

Formule de Taylor-Lagrange. On va chercher à estimer à quelle vitesse les itérations de Newton permettent de se rapprocher de la racine de f, dans les cas favorables. L'ingrédient dans la suite est la formule de Taylor-Lagrange, que l'on rappelle ici.

Théorème 5.1. *Soit f une fonction $[a,b] \to \mathbb{R}$, de classe $\mathcal{C}^n$ sur $[a,b]$, et $n+1$ fois dérivable sur $]a,b[$. Alors il existe $\xi \in [a,b]$ tel que :*

$$f(b) = \sum_{k=0}^{n} \frac{f^k(a)}{k!}(b-a)^k + \frac{(b-a)^{n+1}}{(n+1)!} f^{n+1}(\xi)$$

Démonstration. Remarquons que pour $n = 0$, on obtient le théorème des accroissements finis. La preuve consiste également à appliquer le théorème de Rolle à une fonction bien choisie. Puisque $b \neq a$, on peut choisir K tel que

$$f(b) = \sum_{k=0}^{n} \frac{f^k(a)}{k!}(b-a)^k + \frac{(b-a)^{n+1}}{(n+1)!} K$$

Considérons maintenant la fonction Φ définie sur $[a,b]$ par $\Phi(x) = f(x) - \sum_{k=0}^{n} \frac{f^k(a)}{k!}(x-a)^k - \frac{(x-a)^{n+1}}{(n+1)!}K$. Par le choix effectué sur K, on a $\Phi(b) = 0$. Mais on a aussi $\Phi(a) = \Phi'(a) = \cdots = \Phi^{(n)}(a) = 0$. On peut donc appliquer $n+1$ fois le théorème de Rolle à $\Phi, \Phi', \ldots, \Phi^{(n)}$:

- il existe $c_1 \in]a,b[$ tel que $\Phi'(c_1) = 0$;
- il existe $c_2 \in]a,c_1[$ tel que $\Phi''(c_2) = 0$;
- ...
- il existe $c_{n+1} \in]a,c_n[$ tel que $\Phi^{(n+1)}(c_{n+1}) = 0$;

En notant $\xi = c_{n+1} \in]a,b[$, on a $0 = \Phi^{(n+1)}(\xi) = f^{(n+1)}(\xi) - K$. Donc $K = f^{(n+1)}(\xi)$ et on en déduit le théorème. □

Lien avec l'itération de Newton. Notons z une solution de $f(z) = 0$, et appliquons la formule de Taylor-Lagrange pour $n = 1$, en un point a quelconque, en supposant que f est de classe C^2 sur $[a,b]$. On obtient pour un certain ξ entre z et a :

$$0 = f(z) = f(a) + (z-a)f'(a) + \frac{(z-a)^2}{2} f''(\xi)$$

Maintenant, en supposant $f'(a)$ non nul, et en notant $N(a)$ le point obtenu par application de la méthode de Newton depuis a, on peut quantifier la distance entre $N(a)$ et z en fonction de celle entre a et z :

$$|N(a) - z| = \left| a - z - \frac{f(a)}{f'(a)} \right| = \frac{|(a-z)f'(a) - f(a)|}{|f'(a)|} = \frac{|f(a) + (z-a)f'(a)|}{|f'(a)|} = \frac{(z-a)^2\,|f''(\xi)|}{2\,|f'(a)|}$$

Si la quantité $\left|\frac{f''(\xi)}{2f'(a)}\right|$ reste bornée supérieurement par une constante strictement inférieure à 1 au cours des itérations, la discussion précédente montre que la méthode de Newton converge vers un zéro de f, et ce rapidement, pour peu que le point de départ soit suffisamment proche de z. Plus précisément, on a le théorème suivant :

Théorème 5.2. *Soit $\mathcal{I}$ un intervalle fermé, et f une fonction définie sur $\mathcal{I}$, de classe $\mathcal{C}^2$ sur $\mathcal{I}$, et où f' ne s'annule pas. Supposons que z soit un zéro de f sur l'intervalle $\mathcal{I}$. Soit M_2 une majoration de $|f''|$ et $m_1 > 0$ une minoration de $|f'|$ sur $\mathcal{I}$. Alors, en posant $K = \frac{M_2}{2m_1}$, si x_0 est un point de $\mathcal{I}$ vérifiant $K|x_0 - z| < 1$, la méthode de Newton avec point de départ x_0 converge vers z. De plus, en notant $(x_n)_{n\in\mathbb{N}}$ la suite des itérés obtenus, $K|x_n - z| \leq (K|x_0 - z|)^{2^n}$.*

Démonstration. En reprenant la discussion précédente, on a, en notant $N(a)$ le point obtenu à partir de a avec la méthode de Newton :

$$|N(a) - z| \leq \frac{(z-a)^2 M_2}{2m_1}$$

Posons $K = \frac{M_2}{2m_1}$, alors $|N(a) - z| \leq K(z-a)^2$, ce qu'on peut aussi écrire $K\,|N(a) - z| \leq (K(z-a))^2$. Par une récurrence immédiate, il vient $K\,|x_n - z| \leq (K\,|x_0 - z|)^{2^n}$. □

Commentaire. Ce théorème et sa démonstration ne sont pas à connaître. Il faut simplement retenir que sous de bonnes hypothèses, la méthode de Newton fonctionne, et est très efficace. De l'inégalité $K|x_n - z| \leq (K|x_0 - z|)^{2^n}$, on voit que $\log_2(|x_n - z|) \leq 2^n \log_2(K|x_0 - z|) - \log_2(K)$, ce qui signifie que la précision de l'approximation $x_n \simeq z$ est de l'ordre de 2^n bits. Contrairement à la méthode dichotomique (où pour p étapes, on a environ p bits significatifs), ici pour p étapes, on en a de l'ordre de 2^p : la convergence est très rapide ! On dit que la méthode est d'ordre 2.

5.3.3 Variations et extensions

La méthode de Newton admet plusieurs variantes, qu'on détaille brièvement ici.

Boucle while et condition d'arrêt. Il est possible de changer la boucle `for` en boucle `while` pour tester si on est assez près d'un zéro de f. Pour cela, on peut prendre comme condition d'arrêt $|f(a)| < \varepsilon$ ou encore que la distance entre deux termes successifs est plus petite que ε. Mais attention, la méthode de Newton ne converge pas forcément !

Estimation de la dérivée. Si on n'a pas d'expression de la dérivée (ce qui arrive si f est le résultat d'un calcul compliqué...), alors on peut remplacer le nombre dérivé de f en a par $\frac{f(a+\varepsilon)-f(a)}{\varepsilon}$, avec ε petit, ou mieux encore par $\frac{f(a+\varepsilon)-f(a-\varepsilon)}{2\varepsilon}$. La difficulté consiste à faire le bon choix sur ε : un petit ε apporte plus de précision, mais un trop petit ε engendre de l'instabilité numérique car le numérateur est la différence de deux termes très proches. Le lecteur se reportera au chapitre précédent pour avoir un « bon choix » de ε.

Méthode de la sécante. Une autre méthode consiste à se donner un deuxième point au départ x_1, et remplacer à chaque étape $f'(x_n)$ par $\frac{f(x_n)-f(x_{n-1})}{x_n-x_{n-1}}$. Là encore, il ne faut pas pousser trop loin les itérations sous peine d'instabilité. Cette dernière méthode a pour nom la *méthode de la sécante*, qui converge aussi vers un zéro de f si tout se passe bien. La convergence est un peu moins rapide qu'avec la méthode de Newton, mais reste bien plus rapide qu'avec la méthode dichotomique.

Fonctions à valeurs vectorielles. Contrairement à la méthode dichotomique, la méthode de Newton s'étend à des fonctions de $\mathbb{R}^k \to \mathbb{R}^k$ (avec $k \geq 2$), la dérivée étant remplacée par la différentielle [3], qui se doit d'être inversible pour que la méthode fonctionne.

En Python. La méthode de Newton s'effectue en Python avec la fonction `fsolve` du package `scipy.optimize`.

```
>>> import scipy.optimize ; scipy.optimize.fsolve(lambda x: x*x-2,5,fprime=lambda x:2*x)
array([ 1.41421356])
```

La réponse est sous forme de tableau numpy, car cela marche aussi avec des fonctions vectorielles. Si `fprime` n'est pas précisée, alors elle sera estimée [4].

3. hors programme !

4. C'est dans la documentation ! « By default, the Jacobian will be estimated. ». La matrice jacobienne est la matrice de la différentielle, c'est-à-dire peu ou prou un généralisation de la dérivée en dimension supérieure à 1. La méthode de Numpy s'applique donc aussi avec des fonctions vectorielles.

Chapitre 6

Pivot de Gauss et résolution de systèmes linéaires

On décrit ici un algorithme pour résoudre numériquement des systèmes linéaires. Après une phase d'introduction, on s'en remet au formalisme matriciel. On décrit les algorithmes permettant les opérations de base (transvections et échanges de lignes), puis l'algorithme du pivot en lui-même, dont on donne la complexité. Enfin, on montre quelques exemples montrant que du aux approximations avec des flottants, la réponse de l'algorithme n'est pas forcément à croire aveuglément.

6.1 Introduction

Prenons un système 3×3 que l'on veut résoudre :

$$\left\{\begin{array}{rcll} x+y+z & = & 2 & (L_1) \\ x-y+2z & = & 9 & (L_2) \\ 2x-y+z & = & 7 & (L_3) \end{array}\right.$$

Pour résoudre le système et éliminer la variable x dans les deux dernières équations, on commence par réaliser les opérations $L_2 \leftarrow L_2 - L_1$ et $L_3 \leftarrow L_3 - 2 \times L_1$. On obtient alors le système :

$$\left\{\begin{array}{rcll} x+y+z & = & 2 & (L_1) \\ -2y+z & = & 7 & (L_2) \\ -3y-z & = & 3 & (L_3) \end{array}\right.$$

On peut ensuite éliminer y dans la dernière équation en réalisant l'opération $L_3 \leftarrow L_3 - \frac{3}{2}L_2$. On obtient

$$\left\{\begin{array}{rcll} x+y+z & = & 2 & (L_1) \\ -2y+z & = & 7 & (L_2) \\ -\frac{5}{2}z & = & -\frac{15}{2} & (L_3) \end{array}\right.$$

Maintenant que le système est sous forme triangulaire, on va pouvoir effectuer une phase de *remontée* pour déterminer la solution : on calcule z dans la dernière équation, puis y en utilisant la valeur de z que l'on vient de déterminer, puis enfin la valeur de x avec la première équation :

$$\left\{\begin{array}{rcl} x+y+z & = & 2 \\ -2y+z & = & 7 \\ z & = & 3 \end{array}\right. \qquad \left\{\begin{array}{rcl} x+y+z & = & 2 \\ y & = & -2 \\ z & = & 3 \end{array}\right. \qquad \left\{\begin{array}{rcl} x & = & 1 \\ y & = & -2 \\ z & = & 3 \end{array}\right.$$

6.2 Formalisme matriciel et opérations élémentaires

6.2.1 Formalisme matriciel

Le système précédent peut s'écrire sous forme matricielle

$$AX = Y \qquad \text{avec} \qquad A = \begin{pmatrix} 1 & 1 & 1 \\ 1 & -1 & 2 \\ 2 & -1 & 1 \end{pmatrix}, \quad X = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \quad \text{et} \quad Y = \begin{pmatrix} 2 \\ 9 \\ 7 \end{pmatrix}$$

On a vu que la résolution se fait en deux phases : mise du système sous forme triangulaire, et phase de remontée. Pour la première phase, l'opération principale consiste à ajouter à une ligne une autre ligne, multipliée par un certain facteur : cette opération s'appelle une transvection. Remarquez qu'il faut faire la même opération sur le *vecteur colonne* Y. Ajoutons à cela une autre opération : l'échange de deux lignes. Nous n'en avons pas eu besoin dans l'exemple introductif, mais si nous étions par exemple partis du système

$$\left\{\begin{array}{rcll} y+z & = & 2 & (L_1)\\ x-y+2z & = & 9 & (L_2)\\ 2x-y+z & = & 7 & (L_3)\end{array}\right.$$

il aurait été impossible d'éliminer x dans les deuxième et troisième équation puisque le coefficient devant x est nul dans la première. Pour ce faire, il suffit d'échanger la première ligne avec une des deux autres, ce qui se traduit matriciellement par l'échange de deux lignes de la matrice A et des lignes correspondantes du vecteur colonne Y.

6.2.2 Opérations élémentaires

Dans la suite, on suppose que l'on représente une matrice en Python par la liste de ses lignes, chacune étant elle-même donnée par la liste de ses éléments. De même, un vecteur de $\mathbb{R}^n$ peut-être vu comme une matrice de taille $n \times 1$, donc sera représenté par une liste de listes à un seul élément. Par exemple, la matrice `A` et le vecteur `Y` de l'introduction se définissent ainsi :

```
A = [[1, 1, 1], [1, -1, 2], [2, -1, 1]]
Y = [[2], [9], [7]]
```

Écrivons maintenant les deux fonctions `transvection` et `echange` ainsi décrites. Elles fonctionnent pour des matrices rectangulaires de tailles quelconques.

Fonction de transvection

```
def transvection(A,i,j,mu):
    """On ajoute à la ligne i de A mu fois la ligne j. """
    m=len(A[0]) #m est le nombre de colonnes de A.
    for k in range(m):
        A[i][k]+=mu*A[j][k]
```

Fonction d'échange

```
def echange_lignes(A,i,j):
    """On échange les deux lignes i et j de A."""
    A[i],A[j]=A[j],A[i]
```

Remarquez que dans la fonction d'échange, on ne manipule pas les coefficients : `A[i]` et `A[j]` sont toutes deux des références vers des listes en mémoire, on construit le tuple de références `(A[j],A[i])` qu'on déconstruit pour affecter la première composante à `A[i]` et la deuxième à `A[j]`. Cette opération se fait donc en temps constant. La fonction de transvection est par contre linéaire en m, le nombre de colonnes de A.

6.3 L'algorithme du pivot de Gauss

Dans ce chapitre, on va s'intéresser ici uniquement aux cas où le système $AX = Y$ possède *une unique solution*. On parle de système de *Cramer*. Tout d'abord, ce cas ne peut se produire que si la matrice A possède autant de lignes que de colonnes (la matrice est dite carrée). Ceci sera vu en cours de mathématiques. Ce n'est cependant pas une condition suffisante, comme le montrent les deux exemples qui suivent :

$$(S_1)\quad \left\{\begin{array}{rcl} x+y & = & 2\\ 2x+2y & = & 3\end{array}\right. \qquad \text{et} \qquad (S_2)\quad \left\{\begin{array}{rcl} x+y & = & 2\\ 2x+2y & = & 4\end{array}\right.$$

Le système (S_1) ne possède *aucune* solution : en effet, si on soustrait deux fois la première ligne à la deuxième, on obtient l'équation $0 = -1$. *A contrario*, le système (S_2) en possède *une infinité* : tous les couples (x, y) de la forme $(t, 2-t)$.

L'algorithme du pivot de Gauss consiste à mettre peu à peu le système sous forme triangulaire. Supposons que ceci soit déja partiellement éffectué. On a alors une matrice A de la forme (les $\star$ sont des coefficients qui peuvent être arbitraires) :

$$\begin{pmatrix} \lambda_0 & \star & \star & \star & \star & \cdots & \star & \star \\ 0 & \lambda_1 & \star & \star & \star & \cdots & \star & \star \\ 0 & 0 & \lambda_2 & \star & \star & \cdots & \star & \star \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & \lambda_i & \star & \cdots & \star \\ 0 & 0 & \cdots & 0 & \star & \star & \cdots & \star \\ \vdots & \vdots & \cdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & \star & \star & \cdots & \star \end{pmatrix}$$

où $\lambda_0, \ldots, \lambda_{i-1}$ sont tous non nuls. On cherche maintenant à « mettre des 0 en dessous de λ_i », ce qui revient à éliminer une variable des équations en dessous de celle associée à i. Si λ_i est non nul, c'est possible : il suffit de réaliser des transvections de la forme $L_j \leftarrow L_j - \frac{a_{j,i}}{\lambda_i} L_i$, pour tout j entre $i+1$ et $n-1$. Sinon, il faut aller chercher une ligne en dessous où le coefficient en colonne i est non nul, pour procéder d'abord à un échange de lignes avec la ligne i. Ce coefficient *existe toujours* si le système possède une solution et une seule. En fait, la réciproque est vraie dans le sens où si le système ne possède pas une unique solution, on trouvera lors de la mise sous forme triangulaire un certain i pour lequel la colonne est remplie de zéros en dessous de l'indice i inclus.

Bref, il existe un coefficient en dessous de λ_i inclus, qui est non nul. On a dit que si λ_i était nul, il fallait procéder à un échange de lignes. Rappelons qu'**il ne faut pas tester l'égalité entre deux flottants**, sauf très bonne raison. Il se peut en effet que λ_i soit très petit, mais non nul. Procéder à des transvections pour mettre des 0 en dessous est catastrophique du point de vue de la proximité de la solution qu'on va obtenir avec la solution exacte, puisqu'on est amené à faire des multiplications par $1/\lambda_i$. Une méthode simple pour y remédier (bien qu'elle puisse aussi être mise en défaut dans certains cas) consiste à effectuer un échange de lignes systématique entre la ligne d'indice i et celle située en dessous, qui possède le plus grand coefficient en valeur absolue dans la colonne i (sauf si c'est la ligne i elle-même).

Cette méthode porte le nom du *pivot partiel*. Écrivons donc une fonction recherchant quel est l'indice de la ligne située en dessous de i inclus, possédant le plus grand coefficient en valeur absolue dans la colonne i.

```
def indice_pivot(A,i):
    """Recherche de la ligne d'indice >= i possédant le plus grand élément en colonne i"""
    n=len(A)
    i_max=i
    for j in range(i+1,n):
        if abs(A[j][i])>abs(A[i_max][i]):
            i_max=j
    return i_max
```

On a maintenant tout ce qu'il nous faut pour décrire la première partie du pivot : la mise sous forme triangulaire. On donne ici un algorithme à part, intéressant en soit.

Algorithme 6.1 : Mise d'une matrice sous forme triangulaire par opérations sur les lignes

Entrées : Une matrice A de taille $n \times n$
pour chaque i *allant de* 0 *à* $n-1$ **faire**
  $j \leftarrow$ indice_pivot(A,i);
  si $j \neq i$ **alors**
    echange_ligne(A, i, j)
  pour chaque j *allant de* $i+1$ *à* $n-1$ **faire**
    transvection$(A, j, i, -A[j][i]/A[i][i])$

Remarquez que la complexité d'une transvection étant en $O(n)$, la complexité d'une mise sous forme triangulaire d'une matrice carrée de taille $n \times n$ est donc en $O(n^3)$.

Dans l'algorithme du pivot pour résoudre $AX = Y$, on effectue ces opérations, en effectuant les mêmes transvections sur Y que sur A. La seconde phase est celle de la remontée. Comme son nom l'indique, il suffit de calculer les composantes de X en commençant par la dernière. Posons $A = (a_{i,j})_{0 \leq i,j \leq n-1}$ avec $a_{i,j} = 0$ pour $i > j$. En posant X le vecteur colonne de composantes $x_0, \ldots, x_{n-1}$, le produit AX s'écrit :

$$\begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & \cdots & a_{0,n-1} \\ 0 & a_{1,1} & a_{1,2} & \cdots & a_{1,n-1} \\ 0 & 0 & a_{2,2} & \cdots & a_{2,n-1} \\ \vdots & \cdots & \ddots & \cdots & \cdots \\ 0 & \cdots & \cdots & 0 & a_{n-1,n-1} \end{pmatrix} \times \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} a_{0,0}x_0 + \sum_{j=1}^{n-1} a_{0,j}x_j \\ a_{1,1}x_1 + \sum_{j=2}^{n-1} a_{1,j}x_j \\ a_{2,2}x_2 + \sum_{j=3}^{n-1} a_{2,j}x_j \\ \vdots \\ a_{n-1,n-1}x_{n-1} \end{pmatrix}$$

Pour tout i entre 0 et $n-1$, on tire de l'égalité $a_{i,i}x_i + \sum_{j=i+1}^{n-1} a_{i,j}x_j = y_i$ l'expression $x_i = \frac{1}{a_{i,i}}\left(y_i - \sum_{j=i+1}^{n-1} a_{i,j}x_j\right)$, qui ne fait intervenir que les x_j avec $j > i$. On a donc tout ce qu'il faut pour résoudre le système.

Avant de l'écrire, concluons par une fonction permettant de réaliser la copie d'une matrice. En effet, *a priori* on ne veut pas toucher à la matrice de départ A, ni au vecteur colonne Y.

Copie d'une matrice

```
def copie_matrice(A):
    return([A[i][:] for i in range(len(A))])
```

6.4 Écriture du pivot

Il suffit de mettre en forme tout ce qu'on a vu pour le moment :

— copier les matrices et le vecteur colonne de départ ;

— mettre A sous forme triangulaire en faisant les mêmes opérations (échanges de lignes et transvections) sur Y ;

— obtenir une à une les composantes de X en partant de la fin.

Le pivot de Gauss, version pivot partiel

```
def gauss(A0,Y0):
    """Renvoie le vecteur colonne X tel que A0*X=Y0, on suppose qu'il existe et est unique."""
    A=copie_matrice(A0)
    Y=copie_matrice(Y0)
    n=len(A)
    for i in range(n):
        j=indice_pivot(A,i)
        if j!=i:
            echange_lignes(A,i,j)
            echange_lignes(Y,i,j)
        for j in range(i+1,n):
            mu=-A[j][i]/A[i][i]
            transvection(A,j,i,mu)
            transvection(Y,j,i,mu)
    X=[[0] for i in range(n)]
    for i in range(n-1,-1,-1):
        X[i][0]=1/A[i][i]*(Y[i][0]-sum([A[i][j]*X[j][0] for j in range(i+1,n)]))
    return X
```

En terme de complexité, la phase la plus coûteuse est de mettre le système sous forme triangulaire : on a déja vu que cette étape était en $O(n^3)$. La copie des matrices est en $O(n^2)$ (nombre de coefficients), de même que la phase de remontée : en effet on effectue une boucle de n étapes, et il y a $O(n)$ opérations à effectuer à l'intérieur.

6.5 Quelques exemples

Dans l'ensemble, notre algorithme fonctionne plutôt bien. Reprenons l'exemple donné en introduction :

```
>>> A=[[1,1,1],[1,-1,2],[2,-1,1]]
>>> Y=[[2],[9],[7]]
>>> print(gauss(A,Y)
[[1.0], [-2.0], [3.0]]
```

Notez qu'ici, la solution est *exactement* la bonne, puisqu'on a travaillé tout au long du calcul uniquement avec des flottants exactement représentables (dyadiques). Ce n'est pas le cas en général. Le code suivant produit une matrice A générée aléatoirement avec des entiers entre 0 et 50, et un vecteur Y dont la coordonnée d'indice i est la somme des coefficients de la ligne i de A.

```
from random import randint
n=5
A=[[0]*n for _ in range(n)]
for i in range(n):
    for j in range(n):
        A[i][j]=randint(0,50)
Y=[[sum(A[i])] for i in range(n)]
X=gauss(A,Y)
```

Si le système $AX = Y$ possède une unique solution (et c'est le cas avec forte probabilité), c'est nécessairement le vecteur colonne dont toutes les composantes sont égales à 1 (donc représentables exactement en flottants !) Comme on peut le voir, la solution calculée est proche, mais pas exactement celle-ci :

```
>>> print(X)
[[0.9999999999999989], [0.9999999999999992], [1.0000000000000007], [1.0000000000000009], [0.9999999999999988]]
```

Notez que si l'on avait travaillé avec des *fractions* donc sur $\mathbb{Q}$, on aurait pu résoudre le système exactement. En fait, sur $\mathbb{Q}$, le test d'égalité à zéro est légitime, et on pourrait parfaitement écrire un algorithme de résolution de système fonctionnant aussi avec des systèmes qui ne sont pas de Cramer, pouvant dire si le système n'a aucune solution ou donner une paramétrisation de l'ensemble des solutions dans le cas où il en a une infinité.

Normalement, si le système n'a pas de solution, une division par zéro se produira. Remarquez qu'un système peut ne pas avoir de solutions, bien que le programme en donne une :

```
>>> A=[[0.1,0.1,0.1],[0.3,0.2,0.1],[0.4,0.3,0.2]]
>>> Y=[[2],[9],[7]]
>>> print(gauss(A,Y))
[[-3.602879701896395e+16], [7.205759403792797e+16], [-3.6028797018964e+16]]
```

Cela vient du fait que le coefficient en bas à droite après mise sous forme triangulaire est sensé être nul, mais ne l'est pas tout à fait du aux erreurs d'approximations (1/10 n'est pas représentable exactement en flottants). Voici à quoi ressemble la matrice après mise sous forme triangulaire :

```
[[  4.00000000e-01   3.00000000e-01   2.00000000e-01]
 [  0.00000000e+00   2.50000000e-02   5.00000000e-02]
 [  0.00000000e+00   0.00000000e+00  -1.24900090e-16]]
```

Enfin, il se peut que le système ait une solution, mais que les erreurs d'approximations font qu'on tombe sur 0. Par exemple, le système suivant

$$\begin{cases} x + (1+10^{-15})y + z &= 1 \quad (L_1) \\ (1-10^{-15})x + y + 2z &= 0 \quad (L_2) \\ z &= 0 \quad (L_3) \end{cases}$$

possède une unique solution, mais lorsque l'on utilise notre pivot, on obtient :

```
>>> A=[[1,1+10.**(-15),1],[1-10**(-15),1,2],[0,0,1]]
>>> Y=[[1],[0],[0]]
>>> print(gauss(A,Y))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/svartz/Enseignement/Cours_IPT/premiere_annee/cours/analyse_numerique/gauss.py", line 72, in gauss
    mu=-A[j][i]/A[i][i]
ZeroDivisionError: float division by zero
```

Les coefficients en 10^{-15} n'ont pas été choisis au hasard. Rappelez vous (voir le chapitre numéro 2), que la représentation d'un flottant normalisé sur 64 bits est de la forme $(-1)^s \times \overline{1.m_1 m_2 \cdots m_{52}}^2 \times 2^E$, c'est à dire que l'on a que 52 bits de mantisse. Or $10^{-15} = 1000^{-5} \simeq (2^{-10})^5 = 2^{-50}$. Autrement dit, $1 + 10^{-15}$ et 1 sont très proche en machine, et on est quasiment à la limite entre le flottant 1 et celui immédiatement supérieur.

Lorsqu'on résout le système à la main, on obtient en faisant l'opération $L_2 \leftarrow L_2 - (1-10^{-15})L_1$:

$$\begin{cases} x + (1+10^{15})y + z &= 1 \quad (L_1) \\ 10^{-30}y + (1+10^{-15})z &= 0 \quad (L_2) \\ z &= 0 \quad (L_3) \end{cases}$$

Or, il se trouve que 10^{-30} est calculé comme $1-(1+10^{-15})(1-10^{-15})$ et donc est arrondi à zéro. En effet, du point de vue des flottants, $(1+10^{-15})(1-10^{-15})=1$, car on n'a pas assez de bits de mantisse pour distinguer 1 et $1-10^{-30}$. En effet, le plus grand flottant strictement inférieur à 1 est $\frac{\sum_{k=0}^{52}2^{-k}}{2}=1-2^{-53}$, et $1-10^{-30}\simeq 1-2^{100}$ est beaucoup plus proche de 1 que de $1-2^{-53}$, donc il est arrondi à 1.

Ce genre de cas se produit lorsqu'on est très proche d'une matrice *non inversible* (ceci sera revu en cours de mathématiques), à savoir ici la matrice

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix}.$$

Pour conclure, il faut essentiellement retenir que :

— dans l'ensemble, l'algorithme du pivot (partiel) fonctionne bien ;

— ce n'est pas parce que l'algorithme renvoie une solution que le système en a une ;

— il se peut que le système ait bien une unique solution mais que l'algorithme échoue à la calculer.

6.6 En Python

Pour résoudre un système linéaire en Python, on utilise Numpy, et plus précisément son sous-module `linalg` (pour *linear algebra*, c'est-à-dire algèbre linéaire). Une matrice est simplement représenté par un tableau Numpy. La fonction `fsolve` prend en entrée une matrice carrée inversible A, et un vecteur Y, et renvoie la solution de $AX=Y$:

```
>>> import numpy as np
>>> a = np.array([[3,1], [1,2]])
>>> b = np.array([9,8])
>>> x = np.linalg.solve(a, b)
>>> x
array([ 2.,  3.])
```

En effet, on a bien $\begin{pmatrix} 3 & 1 \\ 1 & 2 \end{pmatrix} \times \begin{pmatrix} 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 9 \\ 8 \end{pmatrix}$.

Deux remarques :

— Python est souple : si on lui donne un tableau Numpy « simple » comme second membre, il peut calculer quand même, comme on le voit ci-dessous. Si on lui donne le second membre comme une matrice Numpy de taille $n\times 1$ (dans la lignée du cours), il renvoie la solution sous la même forme.

— Sans surprise, Python a les mêmes problèmes que nous avec les nombres flottants :

```
>>> np.linalg.solve([[0.1,0.1,0.1],[0.3,0.2,0.1],[0.4,0.3,0.2]], [2, 9, 7])
array([ -4.11757680e+16,   8.23515360e+16,  -4.11757680e+16])
```

Chapitre 7

Calcul approché d'intégrales

7.1 Introduction

Le but de ce chapitre est de présenter quelques méthodes de calcul approché d'intégrales. De manière similaire à la problématique de la résolution d'équations numériques, il y a notamment deux motivations pour chercher de telles méthodes :

— les fonctions usuelles (fonctions trigonométriques, exponentielles, logarithmes...) qu'on manipule possèdent souvent des primitives que l'on peut exprimer à l'aide des fonctions usuelles. Cependant, ce n'est pas toujours le cas : il est par exemple impossible d'exprimer une primitive de $x \mapsto \exp(x^2)$ à l'aide des fonctions usuelles. Pour estimer une intégrale comme $\int_0^1 \exp(x^2)\,\mathrm{d}x$, il est donc nécessaire de recourir à des méthodes numériques ;

— si une fonction n'est connue que par ses valeurs prises en certains points (par exemple issus d'une série de mesures physiques), le calcul exact d'intégrales n'a pas de sens et on ne peut que chercher une estimation de l'intégrale d'une fonction « raisonnable » passant par ces points (ou s'en approchant suffisamment).

De plus, les idées développées ici nous seront utiles pour aborder la résolution numérique d'équations différentielles. En effet, une équation différentielle (d'inconnue la fonction x) de la forme

$$\begin{cases} x(t_0) = x_0 \\ x'(t) = f(x(t), t) \end{cases}$$

peut se réécrire $x(t) = x(t_0) + \int_{t_0}^{t} f(x(u), u)\,\mathrm{d}u$. Les méthodes développées dans ce chapitre s'appliqueront avec quelques adapatations pour le calcul approché de valeurs de la fonction x.

7.2 Idée générale de l'approximation d'intégrales.

Méthodes élémentaires et composées. L'idée générale de l'approximation d'intégrales est la suivante. On se donne une fonction f continue sur un intervalle $[a, b]$ dont on cherche à calculer une approximation de $\int_a^b f(t)\,\mathrm{d}t$. On peut partitionner l'intervalle $[a, b]$ en petits intervalles à l'aide d'une subdivision $a = x_0 < x_1 < \cdots < x_n = b$, on a alors $\int_a^b f(t)\,\mathrm{d}t = \sum_{k=0}^{n-1} \int_{x_i}^{x_{i+1}} f(t)\,\mathrm{d}t$. Il ne reste plus qu'à estimer les $\int_{x_i}^{x_{i+1}} f(t)\,\mathrm{d}t$, par une méthode *de quadrature élémentaire* : en général on prend la même méthode pour estimer toutes ces intégrales. La décomposition de $\int_a^b f(t)\,\mathrm{d}t$ en ces petites intégrales s'appelle une méthode *de quadrature composée*. La notion de « petit intervalle » et de « gros intervalle » dépend de la fonction f et de la précision souhaitée.

Avertissement. Dans la suite, on notera $[a, b]$ un « petit » intervalle, sur lequel on applique une méthode élémentaire, ou un « gros » intervalle, qu'on découpe en morceaux pour appliquer une méthode élémentaire. En pratique, le découpage en morceaux se fera avec une subdivision régulière.

Méthode élémentaire. Une méthode élémentaire de quadrature de $\int_a^b f(t)\,\mathrm{d}t$ se présente sous la forme

$$\int_a^b f(t)\,\mathrm{d}t \simeq (b-a) \sum_{i=0}^{N-1} \omega_i f(\xi_i)$$

où les ξ_i sont des points de l'intervalle $[a, b]$ et ω_i des réels, qu'on appelle les poids.

Définition 7.1. *Une méthode d'intégration est dite* d'ordre n *si elle est exacte pour tout polynôme de degré inférieur ou égal à n et inexacte pour un polynôme de degré $n+1$.*

Autrement dit, par linéarité par rapport à f des deux quantités $\int_a^b f(t)\,\mathrm{d}t$ et $(b-a)\sum_{i=0}^{N-1}\omega_i f(\xi_i)$, une méthode élémentaire est d'ordre n si et seulement si :

$$\text{pour tout } 0\leq j\leq n,\quad \int_a^b x^j\,\mathrm{d}t=(b-a)\sum_{i=0}^{N-1}\omega_i\xi_i^j \qquad \text{et} \qquad \int_a^b x^{n+1}\,\mathrm{d}t\neq(b-a)\sum_{i=0}^{N-1}\omega_i\xi_i^{n+1}$$

Proposition 7.2 (Méthode d'ordre 0)**.** *Puisque $\int_a^b 1\,\mathrm{d}t=(b-a)$, la méthode est d'ordre au moins zéro si et seulement si $\sum_{i=0}^{N-1}\omega_i=1$, ce qu'on supposera par la suite.*

7.3 Méthodes des rectangles

7.3.1 Principe général des méthodes élémentaire et composée

Méthode élémentaire. La méthode de quadrature élémentaire repose sur un principe simple : pour estimer une intégrale de la forme $\int_a^b f(t)\,\mathrm{d}t$, on choisit un seul point ξ de l'intervalle $[a,b]$, et on approche l'intégrale par

$$\int_a^b f(t)\,\mathrm{d}t\simeq(b-a)f(\xi)$$

Il y a *a priori* trois choix sensés pour le point ξ : le point a, le point b, ou le point milieu $\frac{a+b}{2}$, comme montré en figure 7.1.

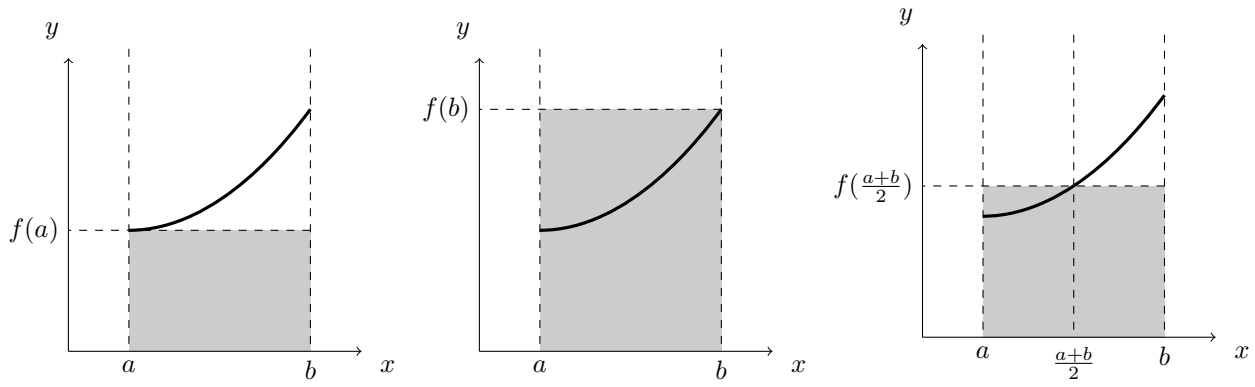


FIGURE 7.1 – Méthode des rectangles élémentaire : à gauche, à droite et au point milieu

Méthode composée. Ces trois méthodes de quadrature élémentaire donnent des méthodes de quadrature composée : on découpe l'intervalle $[a,b]$ en morceaux égaux, et on applique la méthode des rectangles à gauche/à droite/du point milieu sur chacun d'eux. Soit n un entier strictement positif, et soit $h=\frac{b-a}{n}$, qu'on appelle le pas. On écrit donc $\int_a^b f(t)\,\mathrm{d}t=\sum_{k=0}^{n-1}\int_{a+kh}^{a+(k+1)h} f(t)\,\mathrm{d}t$ et on applique la méthode élémentaire à chacune des $\int_{a+kh}^{a+(k+1)h} f(t)\,\mathrm{d}t$. Ce principe d'approximation est connu dans le cours de mathématiques comme « sommes de Riemann ».

7.3.2 Codes Python pour les méthodes composées

Rectangles à gauche, quadrature composée. Elle consiste à faire l'approximation $\int_{a+kh}^{a+(k+1)h} f(t)\,\mathrm{d}t\simeq h\times f(a+kh)$. On a donc $\int_a^b f(t)\,\mathrm{d}t\simeq h\times\sum_{k=0}^{n-1} f(a+kh)$. Écrivons un code Python calculant cette approximation : la fonction prend en entrée une fonction, deux réels a et b, et un entier n, et renvoie cette somme.

```
def int_rec_gauche(f,a,b,n):
    h=(b-a)/n
    somme=0
    x=a
    for k in range(n):
        somme+=f(x)
        x+=h
    return h*somme
```

Rectangles à droite, quadrature composée. Il suffit juste de prendre $\int_{a+kh}^{a+(k+1)h} f(t)\,\mathrm{d}t \simeq h \times f(a+(k+1)h)$ à la place de $h \times f(a+kh)$ dans la formule précédente. Pour le code Python, on peut réutiliser le même, en commençant avec $x = a + h$ au lieu de $x = a$.

```
def int_rec_droite(f,a,b,n):
    h=(b-a)/n
    somme=0
    x=a+h
    for k in range(n):
        somme+=f(x)
        x+=h
    return h*somme
```

Point milieu, quadrature composée. De même, on prend ici $\int_{a+kh}^{a+(k+1)h} f(t)\,\mathrm{d}t \simeq h \times f(a+kh+h/2)$

```
def int_rec_milieu(f,a,b,n):
    h=(b-a)/n
    somme=0
    x=a+h/2
    for k in range(n):
        somme+=f(x)
        x+=h
    return h*somme
```

7.3.3 Étude théorique

Ordre de ces méthodes. On se convainc rapidement avec un dessin que la méthode des rectangles à gauche ou à droite n'est pas exacte sur les fonctions affines : ces deux méthodes sont donc d'ordre zéro. En revanche, la méthode du point milieu est d'ordre 1 : on vérifie aisément avec un dessin qu'elle est exacte sur les fonctions affines, et le lecteur suspicieux vérifiera que $\int_a^b t^2\,\mathrm{d}t \neq (b-a)\left(\frac{a+b}{2}\right)^2$ si $a \neq b$: la méthode n'est donc pas d'ordre 2.

On peut se demander si, lorsque n tend vers l'infini, la méthode des rectangles permet de se rapprocher de la valeur exacte de l'intégrale. La réponse est oui, vous connaissez [1] le théorème suivant sous le nom des « sommes de Riemann ».

Théorème 7.3 (Sommes de Riemann)**.** *Soit f une fonction continue de $[a,b] \to \mathbb{R}$. Lorsque n tend vers l'infini, la méthode des rectangles composée obtenue en découpant l'intégrale en n morceaux converge vers l'intégrale de f entre a et b. c'est-à-dire*

$$\frac{b-a}{n}\sum_{k=0}^{n-1} f\left(\xi_{k,n}\right) \underset{n\to+\infty}{\longrightarrow} \int_a^b f(t)\,\mathrm{d}t$$

avec $\xi_{k,n}$ un point quelconque de l'intervalle $[a+\frac{k(b-a)}{n}, a+\frac{(k+1)(b-a)}{n}]$.

Démonstration. [2] Évaluons la différence entre la somme et l'intégrale :

$$\frac{b-a}{n}\sum_{k=0}^{n-1} f\left(\xi_{k,n}\right) - \int_a^b f(t)\,\mathrm{d}t = \sum_{k=0}^{n-1}\int_{a+k(b-a)/n}^{a+(k+1)(b-a)/n} \left[f\left(\xi_{k,n}\right) - f(t)\right]\mathrm{d}t$$

D'après le théorème de Heine, une fonction continue sur un segment y est uniformément continue. Fixons $\varepsilon > 0$. Alors, il existe $\eta > 0$, tel que pour tout couple $(x,y) \in [a,b]$, si $|x-y| < \eta$, alors $|f(x) - f(y)| \leq \varepsilon$. Soit n_0 assez grand tel que $\frac{b-a}{n_0} < \eta$. Alors pour tout $n \geq n_0$:

$$\begin{aligned}\left|\frac{b-a}{n}\sum_{k=0}^{n-1} f\left(\xi_{k,n}\right) - \int_a^b f(t)\,\mathrm{d}t\right| &\leq \sum_{k=0}^{n-1}\int_{a+k(b-a)/n}^{a+(k+1)(b-a)/n} \left|f\left(\xi_{k,n}\right) - f(t)\right|\mathrm{d}t \\ &\leq \sum_{k=0}^{n-1}\int_{a+k(b-a)/n}^{a+(k+1)(b-a)/n} \varepsilon\,\mathrm{d}t = \sum_{k=0}^{n-1}\frac{b-a}{n}\varepsilon \\ \left|\frac{b-a}{n}\sum_{k=0}^{n-1} f\left(\xi_{k,n}\right) - \int_a^b f(t)\,\mathrm{d}t\right| &\leq (b-a)\varepsilon\end{aligned}$$

Comme ε peut-être pris arbitrairement proche de 0, la conclusion suit.

□

1. ou le verrez bientôt, avec des hypothèses plus fortes pour les PCSI.
2. Les PCSI peuvent ignorer cette démonstration, hors programme.

7.4 Introduction aux méthodes de Newton-Cotes

7.4.1 Principe général

Les méthodes de Newton-Cotes ont pour principe de chercher une formule de quadrature élémentaire sur l'intervalle $[a,b]$ de la forme

$$\int_a^b f(t)\,\mathrm{d}t \simeq (b-a)\sum_{i=0}^{N-1}\omega_i f(\xi_i)$$

avec les ξ_i formant un ensemble de $N \geq 2$ points régulièrement répartis sur l'intervalle $[a,b]$, c'est-à-dire $a = \xi_0 < \xi_1 < \cdots < \xi_{N-1} = b$, et $\xi_{i+1} - \xi_i = \frac{b-a}{N-1}$ pour tout $i < N-1$. Le problème est de chercher quels poids (ω_i) prendre pour que l'ordre de la méthode soit le plus élevé possible. Comme on a N points ξ_i, il est naturel d'espérer qu'avec les bons poids (ω_i) la méthode soit d'ordre $N-1$ au moins : l'exactitude sur les fonctions polynomiales de degré inférieur ou égal à $N-1$ impose N contraintes linéaires en les (ω_i), dont on peut raisonnablement espérer qu'elles donnent un système ayant une unique solution.

Avant d'écrire ce système linéaire pour trouver ces ω_i, simplifions un peu le problème. Par changement de variable $t = \frac{a+b}{2} + u\frac{b-a}{2}$, on se ramène de l'intervalle $[a,b]$ à l'intervalle $[-1,1]$. En effet,

$$\int_a^b f(t)\,\mathrm{d}t = \int_{-1}^1 f\left(\frac{a+b}{2} + u\frac{b-a}{2}\right)\frac{b-a}{2}\,\mathrm{d}u$$

Une formule de quadrature élémentaire sur l'intervalle $[-1,1]$ avec les points ξ régulièrement espacés est de la forme :

$$\int_{-1}^1 g(v)\,\mathrm{d}v \simeq 2\sum_{i=0}^{N-1}\omega_i g(\xi_i)$$

Utiliser cette méthode donne immédiatement une formule avec points régulièrement espacés sur l'intervalle $[a,b]$, et elle est exacte sur $[-1,1]$ pour les fonctions polynomiales de degré au plus d si et seulement si elle l'est sur $[a,b]$; les fonctions polynomiales de degré borné par d étant stables par changement de variables affine. On cherche donc une formule de quadrature élémentaire sur l'intervalle $[-1,1]$.

7.4.2 Méthode des trapèzes

Méthode élémentaire. La méthode des trapèzes est la méthode de Newton-Cotes avec deux points d'interpolation, à savoir les bornes de l'intervalle. Sur $[-1,1]$, on a donc :

$$\int_{-1}^1 f(t)\,\mathrm{d}t \simeq 2\left(\omega_0 f(-1) + \omega_1 f(1)\right)$$

Pour que la formule soit exacte sur les fonctions polynomiales affines (de degré au plus 1), on doit donc avoir :

$$\int_{-1}^1 1\,\mathrm{d}t = 2\left(\omega_0+\omega_1\right) \qquad \text{et} \qquad \int_{-1}^1 t\,\mathrm{d}t = 2\left(-\omega_0+\omega_1\right)$$

Puisque $\int_{-1}^1 1\,\mathrm{d}t = 2$ et $\int_{-1}^1 t\,\mathrm{d}t = 0$, ce système est très simple à résoudre et donne une unique solution : $\omega_0 = \omega_1 = \frac{1}{2}$. On comprend *a posteriori* le nom de méthode des trapèzes avec le dessin de la figure 7.2.

Méthode composée. À partir de la méthode élémentaire, on tire immédiatement une méthode composée : soit n un entier strictement positif, on découpe l'intégrale $\int_a^b f(t)\,\mathrm{d}t$ en morceaux sur lesquels on applique la méthode des trapèzes, on a donc avec $h = \frac{b-a}{n}$:

$$\begin{aligned}
\int_a^b f(t)\,\mathrm{d}t &= \sum_{k=0}^{n-1}\int_{a+kh}^{a+(k+1)h} f(t)\,\mathrm{d}t \\
&\simeq \sum_{k=0}^{n-1}\frac{h}{2}(f(a+(k+1)h)+f(a+kh)) \\
\int_a^b f(t)\,\mathrm{d}t &\simeq h\left(\frac{f(a)+f(b)}{2} + \sum_{k=1}^{n-1} f(a+kh)\right)
\end{aligned}$$

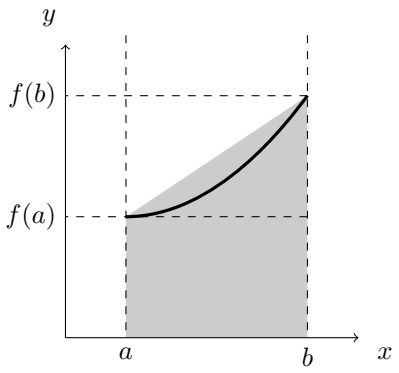


FIGURE 7.2 – Méthode des trapèzes

Code Python. Voici le code Python correspondant :

```
def int_trapezes(f,a,b,n):
    """On approche l'intégrale de f sur [a,b] par des trapèzes"""
    h=(b-a)/n
    somme=(f(a)+f(b))/2
    x=a+h
    for k in range(1,n):
        somme+=f(x)
        x+=h
    return h*somme
```

Ordre de la méthode. Étudions maintenant l'ordre de la méthode : il est d'au moins 1 puisque la méthode est exacte sur les fonctions affines. Un dessin permet de se convaincre qu'elle est inexacte sur les fonctions polynomiales de degré 2, ou alors un simple calcul :

$$\int_{-1}^{1} t^2 \,\mathrm{d}t = \left[\frac{t^3}{3}\right]_{-1}^{1} = \frac{2}{3} \qquad \text{alors que} \qquad 2\left(\frac{1}{2}(-1)^2 + \frac{1}{2}1^2\right) = 2$$

La méthode est d'ordre 1.

7.4.3 Méthode de Simpson

Méthode élémentaire. On utilise ici trois points d'interpolation. c'est-à-dire que sur l'intervalle $[-1, 1]$:

$$\int_{-1}^{1} f(t) \,\mathrm{d}t \simeq 2\left(\omega_0 f(-1) + \omega_1 f(0) + \omega_2 f(1)\right)$$

Pour que la méthode soit d'ordre au moins deux, il faut que l'égalité soit vérifiée pour $f = t \mapsto t^k$ avec $0 \leq k \leq 2$. On en déduit le système suivant :

$$\begin{cases} \omega_0 + \omega_1 + \omega_2 &= 1 \\ -\omega_0 + \omega_2 &= 0 \\ \omega_0 + \omega_2 &= \frac{1}{3} \end{cases}$$

dont l'unique solution est $\omega_0 = \omega_2 = \frac{1}{6}$ et $\omega_1 = \frac{2}{3}$.

Ordre de la méthode. La méthode est donc au moins d'ordre 2. En fait, elle est exacte également pour la fonction $t \mapsto t^3$, car cette fonction impaire vérifie

$$\int_{-1}^{1} t^3 \,\mathrm{d}t = 0 = 2\left(\omega_0 \times (-1)^3 + \omega_1 \times 0^3 + \omega_2 \times 1^3\right)$$

On vérifie facilement qu'elle est par contre inexacte pour la fonction $t \mapsto t^4$ car

$$\int_{-1}^{1} t^4 \,\mathrm{d}t = \frac{2}{5} \neq 2\left(\omega_0 \times (-1)^4 + \omega_1 \times 0^4 + \omega_2 \times 1^4\right) = \frac{2}{3}$$

La méthode est donc d'ordre exactement 3.

Méthode composée. Comme d'habitude, on fixe maintenant un entier $n > 0$, et on découpe l'intervalle $[a, b]$ pour appliquer la méthode de Simpson sur les petits intervalles $[a + kh, a + (k + 1)h]$, avec $h = \frac{b-a}{n}$. On obtient la formule suivante :

$$\begin{aligned}\int_a^b f(t)\,\mathrm{d}t &= \sum_{k=0}^{n-1}\int_{a+kh}^{a+(k+1)h} f(t)\,\mathrm{d}t \\ &\simeq \sum_{k=0}^{n-1} h\left(\frac{1}{6}f(a+kh)+\frac{2}{3}f(a+kh+h/2)+\frac{1}{6}f(a+(k+1)h)\right) \\ \int_a^b f(t)\,\mathrm{d}t &\simeq h\left(\frac{f(a)+f(b)}{6}+\frac{1}{3}\sum_{k=1}^{n-1} f(a+kh)+\frac{2}{3}\sum_{k=0}^{n-1} f(a+kh+h/2)\right)\end{aligned}$$

Code Python. Une fois la formule établie, il est facile de l'implémenter. Le code suivant reprend la formule précédente.

```
def int_simpson(f,a,b,n):
    h=(b-a)/n
    x=a
    y=a+h/2
    s1=0
    s2=f(y)
    for i in range(n-1):
        x+=h
        y+=h
        s1+=f(x)
        s2+=f(y)
    return h*((f(a)+f(b))/6+s1/3+2*s2/3)
```

Il est un peu complexe, car on a cherché à minimiser le nombre d'appels à f en suivant la formule précédemment écrite. Bien sûr, un code Python qui se contenterait d'appliquer la méthode élémentaire sur chacun des petits intervalles serait tout aussi valable, mais ferait deux fois plus d'appels à la fonction.

7.4.4 Méthodes d'ordre supérieur

On va généraliser les méthodes précédentes rapidement, sans donner de code Python, parce que c'est un peu répétitif et exploite toujours le même principe. On cherche donc une formule pour l'approximation

$$\int_{-1}^{1} f(t)\,\mathrm{d}t \simeq 2\sum_{i=0}^{N-1}\omega_i f(\xi_i) \qquad \text{avec} \quad \xi_i = -1 + \frac{2i}{N-1}.$$

avec les ω_i choisis de sorte que la méthode soit exacte pour les fonctions polynomiales de degré au plus $N - 1$. On se ramène donc au système linéaire à N équations et N inconnues suivant :

$$\forall\ 0 \leq k \leq N-1, \qquad \frac{1}{2}\int_{-1}^{1} t^k\,\mathrm{d}t = \sum_{i=0}^{N-1}\omega_i\xi_i^k \qquad \text{avec} \quad \xi_i = -1 + \frac{2i}{N-1}.$$

Matriciellement, on a donc à résoudre $MX = Y$, avec :

$$M = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ \xi_0 & \xi_1 & \xi_2 & \cdots & \xi_{N-1} \\ \xi_0^2 & \xi_1^2 & \xi_2^2 & \cdots & \xi_{N-1}^2 \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ \xi_0^{N-1} & \xi_1^{N-1} & \xi_2^{N-1} & \cdots & \xi_{N-1}^{N-1} \end{pmatrix} \qquad X = \begin{pmatrix} \omega_0 \\ \omega_1 \\ \omega_2 \\ \vdots \\ \omega_{N-1} \end{pmatrix} \qquad \text{et} \qquad Y = \frac{1}{2}\begin{pmatrix} \int_{-1}^{1}\mathrm{d}t \\ \int_{-1}^{1} t\,\mathrm{d}t \\ \int_{-1}^{1} t^2\,\mathrm{d}t \\ \vdots \\ \int_{-1}^{1} t^{N-1}\,\mathrm{d}t \end{pmatrix}$$

La matrice M est la matrice de *Van der Monde* associée aux ξ_i : ce genre de matrice se retrouve souvent en mathématiques, et si vous ne l'avez pas encore croisée, ça viendra. Un résultat classique est que c'est une matrice inversible (car les ξ_i sont distincts) : le système possède donc une unique solution. On appelle « méthode de Newton Cotes à N points » la méthode obtenue en utilisant les poids solution du système.

Proposition 7.4. *La méthode de Newton-Cotes à N points est d'ordre au moins $N - 1$, et au moins N si N est impair.*

Démonstration. Le premier point est clair : les poids sont solution du système donc la méthode est d'ordre au moins $N-1$. Ensuite, montrons que l'unique solution $X = (\omega_i)$ du système précédent est symétrique, c'est-à-dire que $\omega_{N-1-i} = \omega_i$ pour tout $i \in [\![0, N-1]\!]$. Tout d'abord, les ξ_i vérifient $\xi_{N-1-i} = -\xi_i$ pour tout $i \in [\![0, N-1]\!]$. Par suite, si on considère le vecteur $\widetilde{X} = (\omega_{N-1-i})$ obtenu par renversement des coordonnées de X, on remarque que la multiplication d'une ligne de M d'indice pair $2k$ par $\widetilde{X}$ donne le même résultat que la multiplication par X, car $\xi_i^{2k} = \xi_{N-i-1}^{2k}$. De plus, la multiplication d'une ligne de M d'indice impair $2k+1$ par $\widetilde{X}$ donne :

$$\sum_{i=0}^{N-1} \xi_i^{2k+1}\omega_{N-1-i} = \sum_{i=0}^{N-1} (-\xi_{N-1-i})^{2k+1}\omega_{N-i-1} = -\sum_{i=0}^{N-1} \xi_i^{2k+1}\omega_i$$

On obtient donc $-\frac{1}{2}\int_{-1}^{1} t^{2k+1}\,\mathrm{d}t$, or cette intégrale est nulle. Ainsi, $\widetilde{X}$ est également solution du système. Or cette solution est unique, donc $X = \widetilde{X}$ et les coefficients ω_i sont bien symétriques.

Revenons à l'ordre de la méthode pour N impair : puisque les coefficients de X sont symétriques, on a $\sum_{i=0}^{N-1} \xi_i^N\omega_i = \sum_{i=0}^{\lfloor \frac{N}{2} \rfloor - 1} (\xi_i^N + \xi_{N-1-i}^N)\omega_i + \xi_{\lfloor \frac{N}{2} \rfloor}^N \omega_{\lfloor \frac{N}{2} \rfloor}$. Or cette somme est nulle car $\xi_{N-1-i} = -\xi_i$ et N impair, et de plus $\xi_{\lfloor \frac{N}{2} \rfloor} = 0$. Cette somme nulle correspond à $\int_{-1}^{1} t^N\,\mathrm{d}t$, donc la méthode est bien d'ordre au moins N. □

En fait, on peut montrer qu'il y a égalité, ce que l'on admettra :

Théorème 7.5. *La méthode de Newton-Cotes à N points est d'ordre $N-1$ pour N pair et N pour N impair.*

Démonstration. admis. □

En pratique, la méthode est donc plutôt utilisée avec un nombre impair de points (à part la méthode des trapèzes). Passons aux résultats de la résolution du système linéaire présenté plus haut [3] :

- $N = 2$. On retrouve
$$\omega_0 = \omega_1 = \frac{1}{2}$$
C'est la méthode des trapèzes.
- $N = 3$. On retrouve
$$\omega_0 = \omega_2 = \frac{1}{6} \qquad \text{et} \qquad \omega_1 = \frac{2}{3}$$
C'est la méthode de Simpson.
- $N = 5$. On trouve
$$\omega_0 = \omega_4 = \frac{7}{90}, \qquad \omega_1 = \omega_3 = \frac{16}{45} \qquad \text{et} \qquad \omega_2 = \frac{2}{15}$$
Cette méthode est appelée méthode de *Boole-Villarceau*.
- $N = 7$. On trouve
$$\omega_0 = \omega_6 = \frac{41}{840}, \qquad \omega_1 = \omega_5 = \frac{9}{35}, \qquad \omega_2 = \omega_4 = \frac{9}{280} \qquad \text{et} \qquad \omega_3 = \frac{34}{105}$$
Cette méthode est appelée méthode de *Weddle-Hardy*.
- $N = 9$. On trouve
$$\omega_0 = \omega_8 = \frac{989}{28350}, \qquad \omega_1 = \omega_7 = \frac{2944}{14175}, \qquad \omega_2 = \omega_4 = \frac{-464}{14175}, \qquad \omega_3 = \omega_5 = \frac{5248}{14175}, \qquad \text{et} \qquad \omega_4 = \frac{-454}{2835}$$

Remarquez la présence de poids *négatifs* qui commencent à apparaître pour $N = 9$. On va voir que d'un point de vue de stabilité, ces poids négatifs sont mauvais. En pratique, les méthodes de Newton-Cotes sont donc utilisées pour $N \in \{2, 3, 5, 7\}$.

7.4.5 Peut-on faire mieux ?

On peut se poser la question de la pertinence des (ξ_i) régulièrement répartis sur l'intervalle $[a, b]$. En fait, autant pour des questions d'ordre que de stabilité numérique, ce choix n'est pas le meilleur : il vaut mieux utiliser d'autres points ξ_i. On n'en dira pas plus [4] !

3. À vérifier en TP !
4. En fait si : les meilleurs points ξ sur $[-1, 1]$ sont donnés par les racines des polynômes de Chebyshev.

7.4.6 Influence des erreurs d'arrondi

Supposons qu'à la place de $f(x)$, on calcule $\tilde{f}(x)$, avec $\left|f(x)-\tilde{f}(x)\right| \leq \varepsilon$. Quel écart a-t-on lors de l'application d'une méthode de quadrature élémentaire ? Notons $S_f = 2\sum_{i=0}^{N-1} \omega_i f(\xi_i)$. On a alors

$$\begin{aligned}\left|S_f - S_{\tilde{f}}\right| &= \left|2\sum_{i=0}^{N-1}\omega_i f(\xi_i) - 2\sum_{i=0}^{N-1}\omega_i \tilde{f}(\xi_i)\right| \\ &= \left|2\sum_{i=0}^{N-1}\omega_i (f(\xi_i)-\tilde{f}(\xi_i))\right| \\ \left|S_f - S_{\tilde{f}}\right| &\leq 2\varepsilon\sum_{i=0}^{N-1}|\omega_i|\end{aligned}$$

Si les ω_i sont tous positifs, on a $\sum_{i=0}^{N-1}|\omega_i| = \sum_{i=0}^{N-1}\omega_i = 1$. Mais si les (ω_i) ne sont pas tous positifs, on a $\sum_{i=0}^{N-1}|\omega_i| > 1$. Voila pourquoi il faut privilégier les méthodes à poids positifs : si les ω_i ne sont pas tous de même signe, l'erreur est potentiellement dilatée.

7.5 Quelques estimations d'erreurs

On présente ici quelques estimations d'erreurs pour les méthodes des rectangles, lorsque f est de classe $\mathcal{C}^1$ ou $\mathcal{C}^2$. De telles estimations peuvent être explicitées pour toutes les méthodes ci-dessus, mais sortent très largement du programme et de ce que vous devez savoir ! On va voir que la méthode du point milieu avec un découpage de $[a,b]$ en n petits intervalles permet d'avoir une erreur en $O\left(\frac{1}{n^2}\right)$, contrairement à la méthode des rectangles à gauche qui n'est qu'en $O\left(\frac{1}{n}\right)$.

Notation. Dans ce qui suit, pour g une fonction continue sur l'intervalle $[a,b]$, on notera $\|g\|_\infty$ la *norme infinie* de g, c'est à dire $\|g\|_\infty = \sup_{t\in[a,b]}|g(t)|$. Cette borne supérieure est bien définie et est en fait un maximum, puisqu'une fonction continue sur un segment est bornée et y atteint ses bornes.

Les démonstrations que l'on va effectuer font usage de l'égalité de Taylor-reste intégrale, qu'on énonce ici.

Lemme 7.6 (Formule de Taylor-reste intégral)**.** *Soit f une fonction de classe $\mathcal{C}^n$ sur un intervalle $[a,b]$, avec $n \geq 1$. Alors pour $x \in [a,b]$, on a :*

$$f(x) = \sum_{k=0}^{n-1}\frac{(x-a)^k}{k!}f^{(k)}(a) + \int_a^x \frac{(x-t)^{n-1}}{(n-1)!}f^{(n)}(t)\,\mathrm{d}t$$

Démonstration. La démonstration se fait par récurrence sur n.

- Si $n=1$, la formule se réécrit simplement $f(x) = f(a) + \int_a^x f'(t)\,\mathrm{d}t$, ce qui est correct.
- Supposons $n \geq 2$ et la propriété démontrée jusqu'au rang $n-1$. Puisque f est de classe $\mathcal{C}^n$, elle est en particulier de classe $\mathcal{C}^{n-1}$ et
$$f(x) = \sum_{k=0}^{n-2}\frac{(x-a)^k}{k!}f^{(k)}(a) + \int_a^x \frac{(x-t)^{n-2}}{(n-2)!}f^{(n-1)}(t)\,\mathrm{d}t$$
On intègre par partie le terme de droite :
$$\int_a^x \frac{(x-t)^{n-2}}{(n-2)!}f^{(n-1)}(t)\,\mathrm{d}t = \left[-\frac{(x-t)^{n-1}}{(n-1)!}f^{(n-1)}(t)\right]_a^x - \int_a^x \frac{-(x-t)^{n-1}}{(n-1)!}f^{(n)}(t)\,\mathrm{d}t$$
Ce qui donne immédiatement la propriété au rang n.
- Par principe de récurrence, la propriété est démontrée pour tout $n \geq 1$.

□

Lemme 7.7. *Soit f de classe $\mathcal{C}^1$ sur $[a,b]$. On a alors*

$$\left|\int_a^b f(t)\,\mathrm{d}t - (b-a)f(a)\right| \leq \frac{(b-a)^2}{2}\|f'\|_\infty$$

Démonstration. On applique la formule de Taylor reste intégral [5] à la fonction $F : x \mapsto \int_a^x f(t)\,\mathrm{d}t$ qui est de classe $\mathcal{C}^2$. On en déduit :

$$F(b) = F(a) + (b-a)F'(a) + \int_a^b (b-t)F''(t)\,\mathrm{d}t$$

Par suite, $\left|\int_a^b f(t)\,\mathrm{d}t - (b-a)f(a)\right| \leq \int_a^b (b-t)\,|f'(t)|\,\mathrm{d}t \leq \|f'\|_\infty \int_a^b (b-t)\,\mathrm{d}t = \frac{(b-a)^2}{2}\,\|f'\|_\infty$. □

Le lemme 7.7 donne une estimation de l'erreur commise dans la méthode élémentaire des rectangles à gauche pour f de classe $\mathcal{C}^1$. On en déduit immédiatement le résultat suivant pour la méthode composée :

Proposition 7.8. *Soit f de classe $\mathcal{C}^1$ sur $[a,b]$. En appliquant la méthode des rectangles à gauche sur l'intervalle $[a,b]$ découpé en n morceaux, on commet une erreur bornée par :*

$$\left|\int_a^b f(t)\,\mathrm{d}t - h\sum_{k=0}^{n-1} f(a+kh)\right| \leq \frac{(b-a)^2}{2n}\|f'\|_\infty$$

avec $h = \frac{b-a}{n}$.

Démonstration. Il suffit d'appliquer le lemme précédent à $\left|\int_{a+kh}^{a+(k+1)h} f(t)\,\mathrm{d}t - hf(a+kh)\right| \leq \frac{h^2}{2}\,\|f'\|_\infty = \frac{(b-a)^2}{2n^2}\,\|f'\|_\infty$ et de sommer les n termes après une inégalité triangulaire. □

Remarque 7.9. *Ceci est une nouvelle démonstration de la convergence des sommes de Riemann, dans le cas $\mathcal{C}^1$.*

Lemme 7.10. *Soit f de classe $\mathcal{C}^2$ sur $[a,b]$. On a alors*

$$\left|\int_a^b f(t)\,\mathrm{d}t - (b-a)f\left(\frac{a+b}{2}\right)\right| \leq \frac{(b-a)^3}{24}\|f''\|_\infty$$

Démonstration. On va aussi appliquer l'égalité de Taylor reste intégral, mais à une primitive F de la fonction $x \mapsto f(x) - \varphi(x)$, où $\varphi : x \mapsto f\left(\frac{a+b}{2}\right) + \left[x - \left(\frac{a+b}{2}\right)\right] f'\left(\frac{a+b}{2}\right)$ est l'équation de la tangente à f au point milieu $\left(\frac{a+b}{2}\right)$. Remarquons que $F'\left(\frac{a+b}{2}\right) = F''\left(\frac{a+b}{2}\right) = 0$. On a alors, en notant $c = \frac{a+b}{2}$:

$$\begin{aligned} F(x) &= F(c) + (x-c)F'(c) + \frac{(x-c)^2}{2}F''(c) + \int_c^x \frac{(x-t)^2F'''(t)}{2}\,\mathrm{d}t \\ &= F(c) + \int_c^x \frac{(x-t)^2F'''(t)}{2}\,\mathrm{d}t \end{aligned}$$

Ainsi, $F(b) - F(c) = \int_c^b \frac{(b-t)^2F'''(t)}{2}\,\mathrm{d}t$ et $F(a) - F(c) = \int_c^a \frac{(a-t)^2F'''(t)}{2}\,\mathrm{d}t$, donc $F(b) - F(a) = \int_c^b \frac{(b-t)^2F'''(t)}{2}\,\mathrm{d}t - \int_c^a \frac{(a-t)^2F'''(t)}{2}\,\mathrm{d}t$. Or $F(b) - F(a) = \int_a^b f(t)\,\mathrm{d}t - (b-a)f\left(\frac{a+b}{2}\right)$, et comme $F''' = f''$ car φ est affine :

$$\begin{aligned} \left|\int_c^b \frac{(b-t)^2F'''(t)}{2}\,\mathrm{d}t\right| &\leq \left[-\frac{(b-t)^3}{6}\right]_c^b \|f''\|_\infty \\ &\leq \frac{(b-a)^3}{6\times 8}\,\|f''\|_\infty = \frac{(b-a)^3}{2\times 24}\,\|f''\|_\infty \end{aligned}$$

et de même pour $\left|\int_c^a \frac{(a-t)^2F'''(t)}{2}\,\mathrm{d}t\right|$. Par somme, le résultat suit. □

Proposition 7.11. *Soit f de classe $\mathcal{C}^2$ sur $[a,b]$. En appliquant la méthode du point milieu sur l'intervalle $[a,b]$ découpé en n morceaux, on commet une erreur bornée par :*

$$\left|\int_a^b f(t)\,\mathrm{d}t - \frac{h}{2}\sum_{k=0}^{n-1}\Big(f(a+kh) + f(a+(k+1)h)\Big)\right| \leq \frac{(b-a)^3}{24n^2}\|f''\|_\infty$$

avec $h = \frac{b-a}{n}$.

Démonstration. Il suffit d'appliquer le lemme précédent, en découpant en n petits intervalles de longueur $h = \frac{b-a}{n}$. □

5. Les MPSI pourront utiliser Taylor-Lagrange.

En appliquant des méthodes similaires, on peut montrer les majorations suivantes pour la méthode des trapèzes et la méthode de Simpson (qui font parfois l'objet d'un long exercice de mathématiques). Les résultats qui suivent ne sont pas à retenir et sont donnés à titre culturel.

Proposition 7.12. *Soit f de classe $\mathcal{C}^2$ sur $[a,b]$. En appliquant la méthode des trapèzes sur l'intervalle $[a,b]$ découpé en n morceaux, on commet une erreur bornée par :*

$$\left|\int_a^b f(t)\,\mathrm{d}t - h\left(\frac{f(a)+f(b)}{2} + \sum_{k=1}^{n-1} f(a+kh)\right)\right| \leq \frac{(b-a)^3}{12n^2}\|f''\|_\infty$$

avec $h = \frac{b-a}{n}$.

Proposition 7.13. *Soit f de classe $\mathcal{C}^4$ sur $[a,b]$. En appliquant la méthode de Simpson sur l'intervalle $[a,b]$ découpé en n morceaux, on commet une erreur bornée par :*

$$\left|\int_a^b f(t)\,\mathrm{d}t - h\left(\frac{f(a)+f(b)}{6} + \frac{1}{3}\sum_{k=1}^{n-1} f(a+kh) + \frac{2}{3}\sum_{k=0}^{n-1} f(a+kh+h/2)\right)\right| \leq \frac{(b-a)^5}{2880n^4}\left\|f^{(4)}\right\|_\infty$$

avec $h = \frac{b-a}{n}$ et $f^{(4)}$ la dérivée quatrième de f.

On remarque que la méthode des trapèzes est légèrement moins précise que celle du point milieu. Elle est cependant plus facile à mettre en oeuvre si les valeurs de f ne sont pas calculées mais mesurées. La méthode de Simpson est très précise : diviser le pas h d'un facteur 2 (c'est à dire doubler n) fait gagner un facteur 16 (2^4). Si la fonction est suffisamment régulière, on peut obtenir des majorations similaires pour toutes les méthodes de Newton-Cotes : la puissance de n au dénominateur est égale à l'ordre de la méthode, plus 1.

La figure 7.3 présente l'évolution de l'erreur dans l'intégration de la fonction de Runge $\mathcal{R} : x \mapsto \frac{1}{1+25x^2}$ sur l'intervalle $[0,1]$, avec 4 méthodes différentes. Comme on a tracé un diagramme log-log, les (valeurs absolues des) pentes donnent l'ordre de la méthode : on retrouve l'ordre 1 pour la méthode des rectangles à gauche, 2 pour les méthodes des trapèzes et du points milieu et 4 pour la méthode de Simpson.

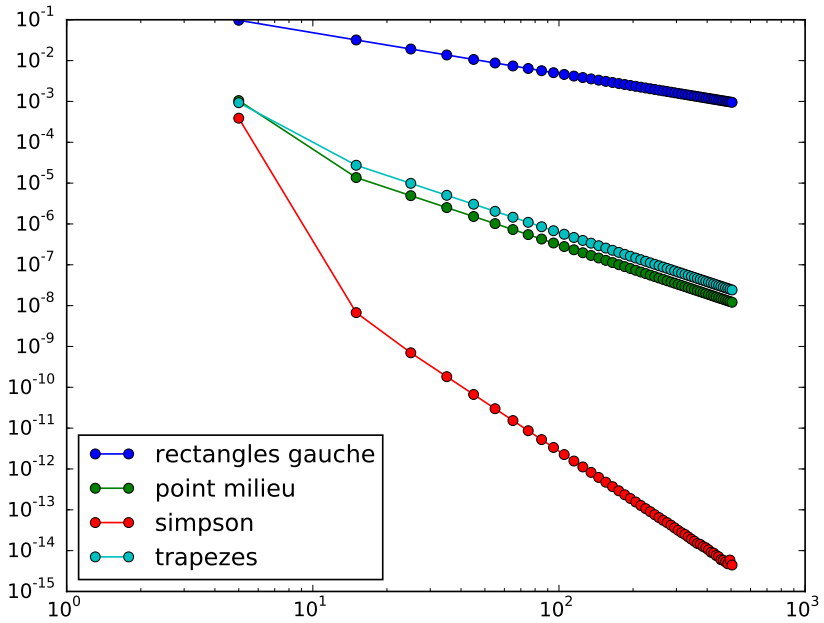


FIGURE 7.3 – Évolution de l'erreur en fonction de n (diagramme log-log) pour les 4 méthodes vues en cours, sur la fonction de Runge intégrée sur $[0,1]$. Les méthodes des trapèzes et du point milieu ont une convergence semblable car elles ont le même ordre.

7.6 Et les méthodes intégrées en Python ?

La documentation se trouve ici[6] Pour intégrer en Python, on utilise généralement la méthode `quad` du module `scipy.integrate`. On a en prime une estimation de l'erreur commise.

```
>>> int_rec_gauche(lambda x: x*x,0,1,20)
0.30875000000000014
>>> int_rec_milieu(lambda x: x*x,0,1,20)
0.3331250000000001
>>> int_trapezes(lambda x: x*x,0,1,20)
0.3337500000000001
>>> int_simpson(lambda x: x*x,0,1,20)
0.3333333333333335
>>> import scipy.integrate as sci ; sci.quad(lambda x: x*x,0,1)
(0.33333333333333337, 3.700743415417189e-15)
```

La méthode utilisée est assez ésotérique (c'est l'aide Python qui le dit !) et utilise pas mal de méthodes différentes suivant ce qu'on lui donne en paramètre, bien plus complexes que celles présentées ici. Remarquez que notre méthode de Simpson est bien exacte sur la fonction $x \mapsto x^2$.

Étant donné deux tableaux de points `X` et `Y`, avec `Y` le tableau des valeurs prises par une fonction f en les points de `X`, on peut explicitement utiliser la méthode des trapèzes ou la méthode de Simpson :

```
>>> X=[1,3,4] ; Y=[x*x for x in X]
>>> sci.trapz(x=X,y=Y)
22.5
>>> sci.simps(x=X,y=Y)
21.0
>>> int_simpson(lambda x:x*x,1,4,1)
21.0
```

On retrouve d'ailleurs bien l'exactitude de la méthode de Simpson sur les fonctions polynomiales de degré 2. Remarquez que Python se débrouille même si les points ne sont pas régulièrement espacés : en fait il prend les points par groupe de trois et calcule (l'unique) parabole passant par ces points et en déduit une méthode d'intégration.

Plus généralement, il peut faire pareil avec les méthodes de Newton-Cotes (il y a une méthode `newton_cotes` dans le module `scipy.optimize`).

6. `https://docs.scipy.org/doc/scipy-0.15.1/reference/tutorial/integrate.html`

Chapitre 8

Résolution approchée d'équations différentielles

8.1 Introduction

8.1.1 Motivation

De même qu'il n'est pas possible d'exprimer les primitives de certaines fonctions à l'aide des fonctions usuelles, il n'est en général pas possible de déterminer la solution formelle d'un système différentiel. C'est le cas par exemple de l'équation du pendule amorti :

$$\ddot{\theta}(t) + k_1 \sin(\theta(t)) + k_2 \dot{\theta}(t) = 0$$

De tels systèmes non linéaires interviennent par exemple en chimie ou en mécanique des fluides. La résolution formelle étant impossible, il est nécessaire de résoudre numériquement le système pour visualiser la solution.

8.1.2 Reformulation

Une équation différentielle ordinaire s'écrit sous la forme [1] :

$$(\star) \qquad \left\{ \begin{array}{ll} x(t_0) = x_0 & \text{(condition initiale)} \\ x'(t) = f(x(t), t) & \text{pour } t \text{ au voisinage de } t_0. \end{array} \right.$$

où la fonction f est une fonction continue, *a priori* quelconque. Notez que l'inconnue x peut aussi bien être une fonction à valeurs dans $\mathbb{R}$ qu'une fonction à valeurs dans $\mathbb{R}^k$, avec $k > 1$: cette formulation [2] permet de traiter aussi bien des équations scalaires d'ordre supérieur à 1 que des systèmes d'équations différentielles.
Par exemple, l'équation du système du pendule amorti se reformule en l'équation vectorielle d'ordre 1 :

$$\left\{ \begin{array}{l} x(t_0) = x_0 \\ x'(t) = f(x(t), t) \end{array} \right. \quad \text{où} \quad x_0 = \begin{pmatrix} \theta(t_0) \\ \dot{\theta}(t_0) \end{pmatrix} \quad \text{et} \quad f : \begin{pmatrix} a \\ b \end{pmatrix}, t \mapsto \begin{pmatrix} b \\ -k_1 \sin(a) - k_2 b \end{pmatrix}$$

Notez que dans ce cas, l'équation est *autonome* : la fonction f ne dépend pas du temps. Mais on sera obligé de la déclarer sous cette forme en Python lorsqu'on utilisera `odeint`. Un autre exemple serait l'équation régissant la tension u_C aux bornes d'un condensateur d'un circuit RLC :

$$u_{\mathrm{C}}(t) + \mathrm{RC}\frac{\mathrm{d}u_{\mathrm{C}}}{\mathrm{d}t}(t) + \mathrm{LC}\frac{\mathrm{d}^2 u_{\mathrm{C}}}{\mathrm{d}t^2}(t) = \mathrm{E}(t)$$

Dans ce cas, l'équation différentielle se reformule en

$$\left\{ \begin{array}{l} x(0) = x_0 \\ x'(t) = f(x(t), t) \end{array} \right. \quad \text{où} \quad x_0 = \begin{pmatrix} u_C(0) \\ \dot{u_C}(0) \end{pmatrix} \quad \text{et} \quad f : \begin{pmatrix} a \\ b \end{pmatrix}, t \mapsto \begin{pmatrix} b \\ \frac{1}{\mathrm{LC}} \left[E(t) - a - \mathrm{RC}b \right] \end{pmatrix}$$

L'équation différentielle n'est pas autonome si la force électromotrice du générateur n'est pas constante.

1. On suit ici la même formulation que la syntaxe de la fonction `odeint` en Python, bien que la convention mathématique soit plutôt d'écrire l'équation différentielle sous la forme $x'(t) = f(t, x(t))$. Cela ne change pas grand chose en définitive, mais attention à ne pas faire de confusion en Python !

2. Au programme de deuxième année.

Les théorèmes mathématiques donnent en général l'existence et l'unicité d'une solution x à l'équation $(\star)$, définie au voisinage de t_0 (avec quelques hypothèses sur f, qu'on ne détaillera pas ici). il est nécessaire de procéder à une résolution numérique (approchée) si l'on souhaite visualiser l'allure de cette solution.

8.1.3 Lien avec l'intégration

On souhaite donc résoudre l'équation $(\star)$ de manière approchée. On connaît la valeur $x(t_0) = x_0$ de la solution au temps initial t_0. L'idée de la résolution approchée que l'on va développer dans ce chapitre est de calculer de proche en proche des approximations de x aux temps $t_0 + h, t_0 + 2h, t_0 + 3h, \dots$, avec h un « petit » pas de temps.

Faisons le lien avec le chapitre précédent : connaissant une approximation de $x(t)$, on peut écrire que :

$$x(t+h) = x(t) + \int_t^{t+h} x'(u)\,\mathrm{d}u = x(t) + \int_t^{t+h} f(x(u),u)\,\mathrm{d}u$$

Pour estimer $x(t+h)$ à partir d'une estimation de $x(t)$, on est donc ramené à approcher l'intégrale $\int_t^{t+h} f(x(u),u)\,\mathrm{d}u$, ce qu'on sait faire depuis le chapitre précédent ! La difficulté ici est que l'on ne connaît (une approximation de) la valeur de la solution x qu'au temps t, et pas sur le reste de l'intervalle $[t, t+h]$. On peut donc appliquer facilement la méthode des rectangles à gauche, et on verra comment contourner cette difficulté pour appliquer les autres méthodes d'intégration.

8.1.4 Formulation « à la `odeint` »

La fonction `odeint` du module `scipy.integrate` permet de résoudre des équations différentielles sous la forme $(\star)$ déja évoquée :

$$(\star) \qquad \begin{cases} x(t_0) = x_0 & \text{(condition initiale)} \\ x'(t) = f(x(t),t) & \text{pour } t \text{ au voisinage de } t_0. \end{cases}$$

La méthode utilisée est plus complexe (et plus efficace) que celles développées dans ce chapitre. `odeint` prend en paramètres :

- la fonction `f` ;
- une valeur `x0` (qui peut-être un vecteur représenté par un tableau Numpy) ;
- un tableau de temps `[`$t_0, t_1, \dots, t_{n-1}$`]` (le premier élément de ce tableau est le temps t_0 du système $(\star)$).

et renvoie via l'appel `odeint(f, x0, T)` une approximation de la solution aux temps t_i, sous la forme d'un tableau Numpy de taille n. Bien sûr, le premier élément de ce tableau est `x0`. L'idée est de calculer une approximation de $x(t_{i+1})$ à partir d'une approximation de $x(t_i)$.

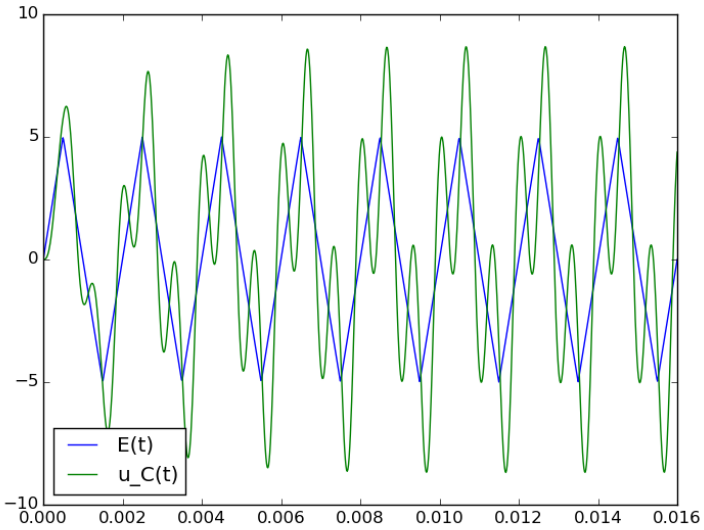
Donnons par exemple la tension aux bornes d'un condensateur soumis à un échelon de tension « en triangles ». On suppose les constantes déja définies, ainsi que la fonction `E(t)` donnant la tension en fonctions du temps. Voici une définition de la fonction `f` :

```
def fRLC(X,t):
    u,up=X[0], X[1] #tension, dérivée de la tension.
    return np.array([up, (E(t)-R*C*up-u)/(L*C)])   #dérivée et dérivée seconde.
```

On note `p` la période du générateur, et on résout en prenant 1000 points sur l'intervalle $[0, 8p]$. Le code suivant trace conjointement la tension aux bornes du générateur et la tension aux bornes du condensateur, avec condition initiale $u_C(0) = 0$ et $\dot{u_C}(0) = 0$.

```
T=np.linspace(0, 8*p, 1000)
X0=np.array([0.,0.]) #initialement, tension et dérivée de la tension sont nulles.
Sode=odeint(fRLC,X0, T) #solution odeint
plt.plot(T,[E(t) for t in T], label="E(t)")
plt.plot(T,[u[0] for u in Sode], label="u_C(t)") #on ne veut que la tension, pas sa dérivée.
plt.legend(loc="lower left")
plt.show()
```

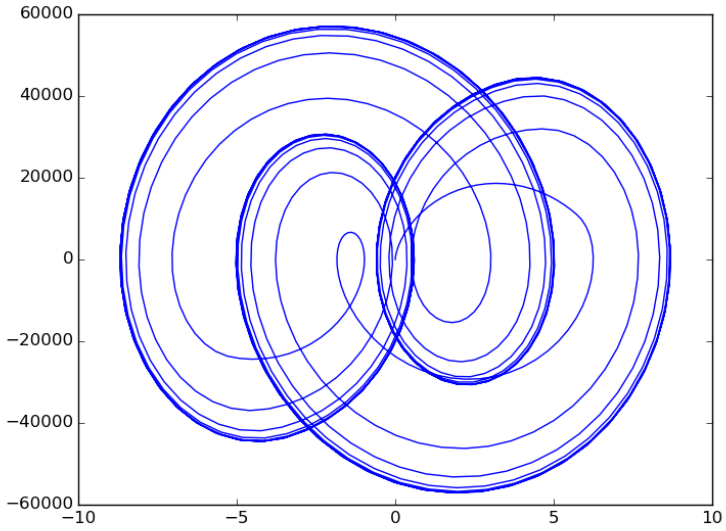
Ici, le résultat de `odeint`, stocké dans `Sode`, est un tableau Numpy à deux dimensions : 1000 « lignes » (correspondant à la taille de `T`), et deux « colonnes », puisqu'ici on travaille en dimension 2. Dans cet exemple, on n'est intéressé que par la tension aux bornes du condensateur, et pas sa dérivée. On veut donc garder la première colonne, et la tracer en fonction du temps. Voici le graphe obtenu :



Pour avoir le portrait de phase ($u_C(t)$ en abscisse, $\frac{\mathrm{d}u_C}{\mathrm{d}t}(t)$ en ordonnée), il faut écrire :

```
plt.plot([u[0] for u in Sode],[u[1] for u in Sode])
```

Voici le résultat :



La syntaxe utilisée pour le tracé pourra être utilisée avec les fonctions que l'on va écrire, qui suivront la même spécification de `odeint`.

8.2 Méthode d'Euler explicite

8.2.1 Principe de la méthode : les rectangles à gauche

Cette méthode est la seule au programme, et est donc à savoir refaire sans hésiter. Elle est basée sur la méthode des rectangles à gauche pour le calcul d'intégrale, qui repose sur l'identité :

$$\int_{t_i}^{t_{i+1}} f(x(u),u)\,\mathrm{d}u \simeq (t_{i+1}-t_i)f(x(t_i),t_i)$$

On voit ici qu'on peut calculer une approximation de $x(t_{i+1})$ à partir d'un approximation de $x(t_i)$, puisque

$$x(t_{i+1}) = x(t_i) + \int_{t_i}^{t_{i+1}} f(x(u), u)\,\mathrm{d}u \simeq x(t_i) + (t_{i+1} - t_i) f(x(t_i), t_i)$$

Le *schéma numérique* de résolution approchée est donc le suivant, pour un calcul de x_i, valeur approchée de $x(t_i)$:

$$\begin{cases} x_0 \text{ donné} \\ x_{i+1} = x_i + (t_{i+1} - t_i) f(x_i, t_i) \end{cases}$$

On voit pourquoi cette méthode porte le nom d'*explicite* : contrairement à la formulation générale de l'introduction, le second membre ici ne comporte pas de « valeurs inconnues » de la fonction x puisqu'il n'y a que $x(t_i)$, supposé déja calculé (de manière approchée).

8.2.2 Code(s) Python

Version générale (type `odeint`). La fonction Python suivante prend en entrée une telle fonction f, une liste de temps T contenant $t_0 < t_1 < \cdots < t_{n-1}$, et un point x_0, et calcule itérativement les valeurs $x_1, \ldots, x_{n-1}$, approximations de la solution de l'équation différentielle aux temps t_i.

```
def euler_explicite(f,x0,T):
    x=x0
    X=[x]
    for i in range(len(T)-1):
        x+=(T[i+1]-T[i])*f(x,T[i])
        X.append(x)
    return X
```

Version temps équirépartis. La fonction suivante, très similaire, se place dans le cas où l'on ne se donne pas une liste de temps, mais un seul temps initial t_0, ainsi qu'un *pas* h et le nombre de valeurs désirées N. On renvoie un couple de deux listes de tailles N :

— une liste contenant les N temps $t_0 < t_0 + h < \cdots < t_0 + (N-1)h$;

— les valeurs approchées de x en ces temps.

```
def euler_explicite_equirepartis(f,t0,x0,N,h):
    t=t0
    x=x0
    X=[x]
    for i in range(N-1):
        x+=h*f(x,t)
        t+=h
        X.append(x)
    return T,X
```

L'une ou l'autre version est à préférer suivant les ciconstances, il faut savoir s'adapter [3].

Exemple : fonction exponentielle. Prenons l'exemple de la classique fonction exponentielle, solution du système suivant :

$$\begin{cases} \exp(0) = 1 \\ \exp'(t) = f(\exp(t), t) \end{cases}$$

où f est simplement la fonction $f : (x, t) \mapsto x$ (le système différentiel est autonome). Le script Python suivant permet d'obtenir (et d'afficher) une représentation graphique de l'exponentionnelle, approchée par la méthode d'Euler sur $[0, 1]$ avec 10 points.

```
import numpy as np
from math import *
import matplotlib.pyplot as plt
T=np.linspace(0,1,10)
f=lambda x,t: x
X=euler_explicite(f,1,T)
plt.plot(T,X,label="exp euler")
plt.legend(loc="upper left") #localisation de la légende en haut à gauche
plt.show()
```

3. Et s'adapter en particulier aux sujets des concours...

Remarquez qu'on a utilisé une définition anonyme pour définir f, on aurait également pu utiliser classiquement `def`. Des graphes similaires à ceux qu'on peut obtenir en Python sont donnés en figure 8.1, pour 5, 10 ou 30 points d'interpolation. Même si, peu à peu, la solution s'écarte de la « vraie » fonction exponentielle (en noir), l'approximation n'est pas trop mauvaise. De plus, lorsqu'on diminue le pas, on s'aperçoit qu'il y a convergence de l'approximation vers la fonction exponentielle.

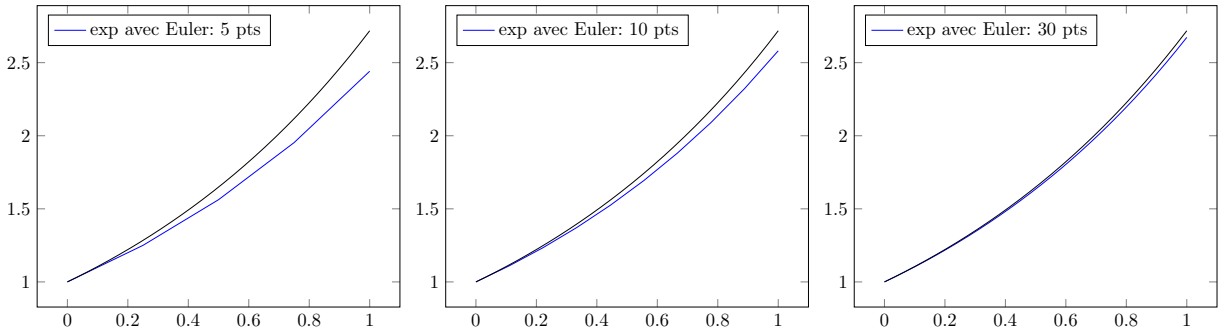


FIGURE 8.1 – Comparaison : Euler explicite sur $[0,1]$: 5, 10 et 30 points. La courbe en noir est la « vraie » fonction exponentielle.

8.2.3 Variante pour les fonctions à valeurs vectorielles

On a en fait supposé que l'on cherchait une fonction à valeurs réelles : `x` est un flottant dans le code précédent. Il n'y a en fait pas grand chose à changer pour travailler avec une fonction à valeurs vectorielles, représentés sous forme de tableau Numpy : il faut juste faire attention dans le code Python. Les lignes :

```
x+=(T[i]-T[i-1])*f(x,T[i-1])
X.append(x)
```

fonctionnent aussi, mais ici les éléments du tableau `X` sont tous des références vers un unique tableau Numpy en mémoire : il faut donc les modifier. On indique deux possibilités. La première consiste à faire une copie de `x` avant de l'ajouter à `X`.

```
x+=(T[i+1]-T[i])*f(x,T[i])
X.append(np.copy(x))
```

La seconde consiste à changer `x+=(T[i+1]-T[i])*f(x,T[i])` en `x=x+(T[i+1]-T[i])*f(x,T[i])`. En effet dans ce cas, on crée un nouveau tableau numpy dans l'évaluation, puis on affecte le résultat à la variable `x`.

```
x=x+(T[i+1]-T[i])*f(x,T[i])
X.append(x)
```

La remarque fonctionne également pour la version « équirépartie ».

8.2.4 Exemple complet : le pendule amorti

On rappelle l'équation du pendule amorti (la solution X est une fonction à valeurs dans $\mathbb{R}^2$, la première composante est l'angle du pendule, la seconde est la vitesse angulaire) :

$$\begin{cases} x(t_0) = x_0 \\ x'(t) = f(x(t),t) \end{cases} \quad \text{où} \quad x_0 = \begin{pmatrix} \theta(t_0) \\ \dot{\theta}(t_0) \end{pmatrix} \quad \text{et} \quad f : \begin{pmatrix} a \\ b \end{pmatrix}, t \mapsto \begin{pmatrix} b \\ -k_1 \sin(a) - k_2 b \end{pmatrix}$$

Donnons le code Python pour résoudre l'équation du pendule amorti, avec [4] $k_1 = 1, k_2 = 0.5$, $t_0 = 0$, $\theta(0) = 0$ et $\dot{\theta}(0) = 1$. On résout sur l'intervalle $[0,20]$, en prenant 100 points.

```
k1=1
k2=0.5

def fpendule(x,t):
    a,b=x[0],x[1]
```

4. Un physicien ralerait parce que les unités ne sont pas présentes, et il aurait raison.

```
    return np.array([b,-k1*sin(a)-k2*b])

T=np.linspace(0,20,100)
X=euler_explicite(fpendule,np.array([0,1]),T)
plt.plot(T,[x[0] for x in X])
plt.show()
```

Pour le tracé (voir figure 8.2 à gauche), on extrait ici les premières composantes des éléments du tableau `X` : en effet ; celles-ci contiennent les valeurs de $(\theta(t), \dot{\theta}(t))$, *a priori* seule la première composante nous intéresse. Sans surprise, l'angle du pendule tend vers zéro.

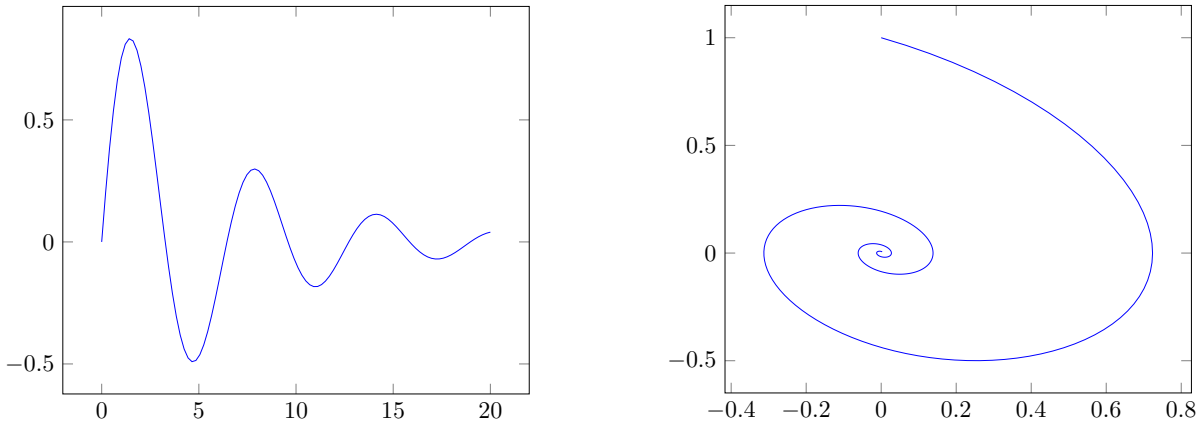


FIGURE 8.2 – Résolution de l'équation du pendule amorti, sur l'intervalle $[0, 20]$. (a) le tracé de $\theta(t)$ en fonction de t. (b) le tracé du portrait de phase, $\dot{\theta}$ en fonction de θ.

On peut aussi tracer le portrait de phase ($\dot{\theta}$ en fonction de θ), voir figure 8.2 à droite.

```
T=np.linspace(0,20,1000)
X=euler_explicite(fpendule,np.array([0.,1.]),T)
plt.plot([u[0] for u in X],[u[1] for u in X])
plt.show()
```

8.3 Quelques notions d'analyse numérique

8.3.1 Erreur sur l'exponentielle

Revenons sur l'exponentielle, approchée par la méthode d'Euler sur $[0, 1]$. Prenons un pas régulier égal à $1/n$, et calculons les itérations effectuées. On a $x_{i+1} = x_i + \frac{1}{n} \times x_i = (1 + \frac{1}{n})x_i$. Un récurrence immédiate donne $x_i = (1 + \frac{1}{n})^i$. Il s'ensuit que l'approximation de $\mathrm{e} = \exp(1)$ que l'on obtient avec la méthode d'Euler et un pas de $1/n$ est $x_n = (1 + \frac{1}{n})^n$. Un développement limité classique donne l'erreur commise sur e :

$$\begin{aligned} x_n &= \exp\left(n \ln\left(1 + \tfrac{1}{n}\right)\right) \\ &= \exp\left(n\left(\tfrac{1}{n} - \tfrac{1}{2n^2} + o\left(\tfrac{1}{n^2}\right)\right)\right) \\ &= \exp\left(1 - \tfrac{1}{2n} + o\left(\tfrac{1}{n}\right)\right) \\ x_n &= \mathrm{e}\left(1 - \tfrac{1}{2n}\right) + o\left(\tfrac{1}{n}\right) \end{aligned}$$

Ainsi l'erreur commise est de l'ordre de $\frac{\mathrm{e}}{2n} = \frac{\mathrm{e} \times h}{2}$, avec h le pas. Ainsi, diminuer le pas réduit l'erreur, mais évidemment c'est plus coûteux car la complexité est linéaire en n, donc en $O(1/h)$. On remarque que l'erreur ici est linéaire en h, ce qui est en fait le cas dès qu'on utilise la méthode d'Euler explicite.

8.3.2 Notion d'erreur de consistance

Remarque 8.1. *On ne propose dans ce chapitre que des schémas numériques à un pas, c'est à dire que l'on obtient une approximation de $x(t_{i+1})$ à partir d'une approximation de $x(t_i)$ et pas des approximations des $x(t_k)$ précédents.*

Définition 8.2. *Dans un schéma numérique à un pas, on définit l'erreur de consistance du schéma comme la somme des erreurs commises entre la valeur réelle $x(t_i)$ et $\widetilde{x_i}$, celle calculée à partir de $x(t_{i-1})$ en utilisant le schéma. C'est à dire*

$$e(h) = \sum_{i=1}^{N} |x(t_i) - \widetilde{x_i}|$$

L'erreur de consistance consiste ainsi à sommer les erreurs commises à chaque étape lors du calcul de $x(t_{i+1})$, en supposant à chaque fois qu'on applique le schéma à partir de la valeur exacte $x(t_i)$ en t_i.

Exemple 8.3. *Calculons l'erreur de consistance dans le calcul de l'exponentielle en 1, à partir de 0 avec un pas $h = 1/n$ en utilisant la méthode d'Euler explicite. On a $\widetilde{x_{i+1}} = \exp(t_i) + (t_{i+1} - t_i)\int_{t_i}^{t_{i+1}} \exp(t_i)\,\mathrm{d}u = \exp(ih)(1+h)$. L'erreur de consistance est ici :*

$$\begin{aligned} e(h) &= \sum_{i=0}^{n-1} \left|\exp((i+1)h) - (1+h)e^{ih}\right| \\ &= (\exp(h) - (1+h)) \sum_{i=0}^{n-1} e^{ih} \\ &= (\exp(h) - (1+h)) \frac{\mathrm{e}^{nh} - 1}{\mathrm{e}^{h} - 1} \\ e(h) &= (\exp(h) - (1+h)) \frac{\mathrm{e} - 1}{\exp(h) - 1} \end{aligned}$$

car $\exp(h) - (1+h) \geq 0$ et $nh = 1$. Un développement limité au voisinage de 0 montre que cette erreur de consistance vérifie $e(h) \sim \frac{h(\mathrm{e}-1)}{2}$.

L'erreur de consistance est un peu plus faible que l'erreur sur e de la section précédente, ce qui est bien normal : dans l'erreur de consistance on ne somme que les erreurs obtenues pour le calcul de $x(t_{i+1})$ en appliquant la méthode d'Euler explicite à partir de la vraie valeur en t_i, alors que dans la méthode d'Euler on ne calcule une approximation de $x(t_{i+1})$ qu'à partir d'une approximation de $x(t_i)$.

Définition 8.4. *On dit qu'un schéma est consistant si l'erreur $e(h)$ tend vers 0 lorsque $h \to 0$.*

On dit qu'un schéma numérique est stable s'il n'est pas trop sensible aux erreurs d'arrondis [5] ; on en dira pas plus. Étudier la consistance est important : on peut montrer que si le schéma est stable et consistant, alors le schéma est convergent : les x_i calculés comme $x(t_i)$ convergent vers $x(t_i)$ lorsque le pas tend vers 0.

8.3.3 Ordre d'un schéma

Définition 8.5. *On dit qu'un schéma numérique est d'ordre au moins p si l'erreur de consistance $e(h)$ vérifie $e(h) = O(h^p)$, et d'ordre exactement p si de plus $e(h) \neq O(h^{p+1})$.*

Proposition 8.6. *Supposons que $\varphi : u \mapsto f(x(u), u)$ soit de classe $\mathcal{C}^1$ sur l'intervalle $[t_0, t]$, alors le schéma de la méthode d'Euler est d'ordre 1 sur cet intervalle.*

Démonstration. Soit $N > 0$. Posons $h = (t - t_0)/N$ et $t_i = t_0 + ih$ pour $1 \leq i \leq N$. On a alors

$$\begin{aligned} e(h) &= \sum_{i=0}^{N-1} |x(t_{i+1}) - x(t_i) - hf(x(t_i), t_i)| \\ &= \sum_{i=0}^{N-1} \left| \int_{t_i}^{t_{i+1}} f(x(u), u) - f(x(t_i), t_i)\,\mathrm{d}u \right| \end{aligned}$$

On reconnaît l'erreur dans la méthode des rectangles à gauche. On a vu dans le chapitre précédent que l'erreur sur chacun des intervalles $[t_i, t_{i+1}]$ était inférieure ou égale à $\frac{(t_{i+1}-t_i)^2}{2} \|\varphi'\|_\infty$. Comme $t_{i+1} - t_i = h$, on a par somme $e(h) \leq \frac{Nh^2}{2} \|\varphi'\|_\infty = \frac{h(t-t_0)}{2} \|\varphi'\|_\infty$. La méthode est bien d'ordre au moins 1. On a vu avec l'exemple de l'exponentielle que la méthode était d'ordre au plus 1, il y a donc égalité. □

5. Cette notion peut se définir de manière rigoureuse, néanmoins cela nous emmenerait assez loin, et très en dehors du programme.

8.4 Autres méthodes

On explique ici brièvement d'autres méthodes possible. Essentiellement, elles dérivent des autres méthodes d'intégration du chapitre précédent, mais il faut contourner la difficulté de l'impossibilité de les appliquer directement : sur l'intervalle $[t_i, t_{i+1}]$, on ne connaît *a priori* la valeur (approchée) de $f(x(u), u)$ qu'en t_i.

8.4.1 Méthode d'Euler implicite

Principe (en dimension 1)

La méthode d'Euler implicite repose sur la méthode des rectangles à droite. On a donc $\int_{t_i}^{t_{i+1}} f(x(u), u)\,\mathrm{d}u \simeq (t_{i+1} - t_i) f(x(t_{i+1}), t_{i+1})$. Pour calculer l'approximation de x_{i+1} avec cette méthode, il faut donc résoudre l'équation numérique

$$x_{i+1} = x_i + (t_{i+1} - t_i) f(x_{i+1}, t_{i+1})$$

ce qu'on peut faire avec une des méthodes du chapitre 5 pour la résolution d'une équation différentielle ou x est à valeurs dans $\mathbb{R}$: méthode dichotomique ou méthode de Newton. Cette méthode est également d'ordre 1, et un peu plus compliquée à mettre en oeuvre, mais elle est un peu plus stable que la méthode d'Euler explicite.

Voici un code Python utilisant la méthode dichotomique du chapitre 5. On cherche arbitrairement sur l'intervalle $[x_i - 1, x_i + 1]$. A priori cela devrait être largement suffisant si le pas n'est pas trop petit. Si ça ne marche pas, il faut changer d'intervalle.

```
def euler_implicite(f,x0,T):
    x=x0
    X=[x]
    for i in range(len(T)-1):
        g=lambda y: y-x-(T[i+1]-T[i])*f(y,T[i+1])
        x=dichotomie(g,x-1,x+1,0.001)
        X.append(x)
    return X
```

Pour appliquer la méthode, on procède essentiellement comme avec la méthode d'Euler explicite. Les résultats obtenus avec la méthode d'Euler implicite se trouvent en figure 8.3.

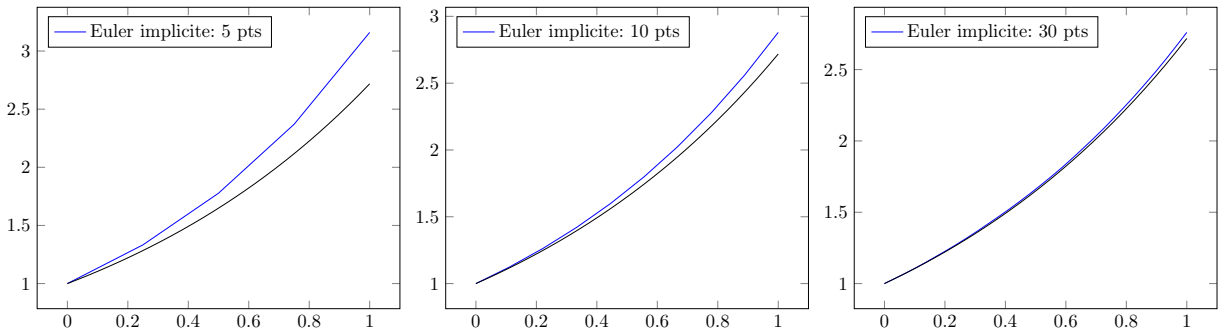


FIGURE 8.3 – Comparaison : Euler implicite pour exp sur $[0, 1]$: 5, 10 et 30 points.

Extension en dimensions supérieures

La méthode dichotomique n'admet pas d'extensions en dimension supérieure à 1, par contre la méthode de Newton en admet une. On peut donc étendre la méthode d'Euler implicite en dimension supérieure ou égale à 2, via la fonction `fsolve` du module `scipy.optimize`. Voici le code d'une méthode d'Euler implicite valable en toute dimension, à la `odeint` :

```
import scipy.optimize as sco

def euler_implicite(f, x0, T):
    x=x0
    X=[x]
    for i in range(len(T)-1):
        g=lambda y: y-x-(T[i+1]-T[i])*f(y, T[i+1])
        x=sco.fsolve(g, x)
        X.append(x)
    return X
```

Remarquez qu'on n'a pas besoin de préciser la dérivée de g, `scipy` arrive à l'estimer pour appliquer la méthode. Comme pour la méthode vue au chapitre 5, il faut préciser un « point de départ » : si le pas n'est pas trop grand, $x(t_{i+1})$ devrait être proche de $x(t_i)$, ainsi l'approximation de $x(t_i)$ est un bon point de départ.

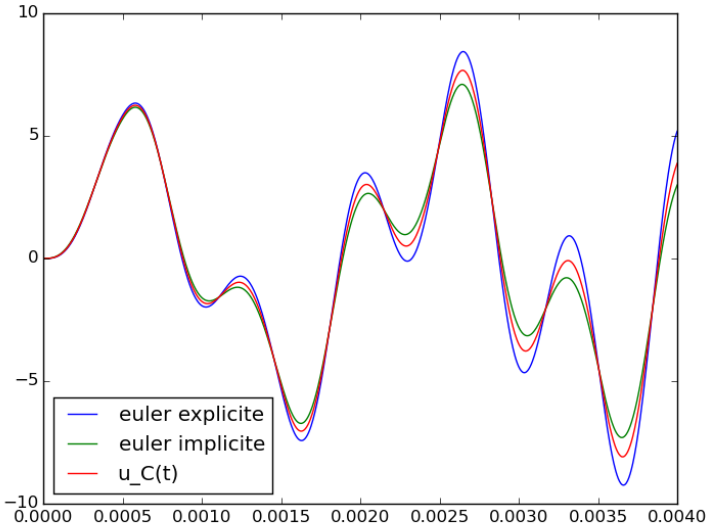
Stabilité de la méthode implicite

Même si la méthode d'Euler implicite a le même ordre (1) que la méthode explicite, elle est en générale préférable pour des raisons de *stabilité*. Si on regarde ce qui se passe vis à vis de la résolution de l'équation dont l'exponentielle est solution :

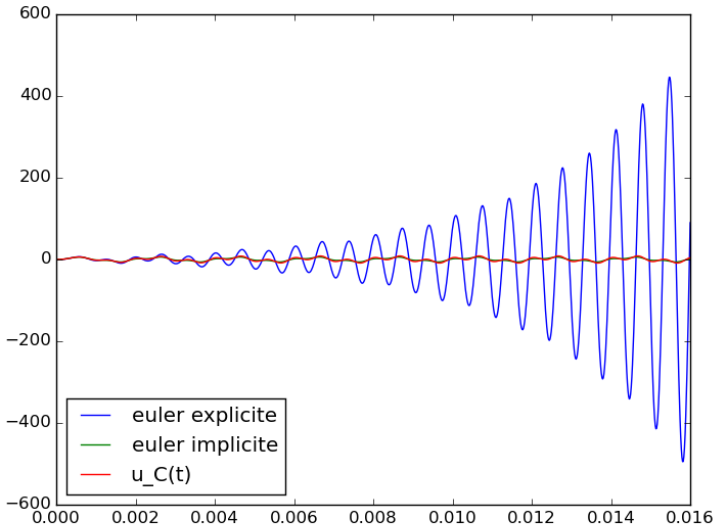
— la méthode explicite a tendance à suivre la tangente : on se retrouve donc « à l'extérieur » de la courbe décrite par l'exponentielle ;

— à l'inverse, la méthode implicite à tendance à positionner les approximations « à l'intérieur » de la courbe.

Si on travaille avec une équation dont la solution est une fonction oscillante bornée, la méthode explicite va avoir tendance à écarter la solution approchée de la solution exacte au niveau des pics, et donc s'éloigner de plus en plus de la vraie solution. À l'inverse la méthode implicite donnera une approximation qui restera bornée. Reprenons l'exemple du circuit RLC présenté en section 8.1, dont nous avions résolu l'équation avec `odeint`. On prend ici 1000 points d'interpolation sur l'intervalle $[0, 2p]$, avec p la période. On considère que la solution fournie par `odeint` est exacte. Voici le résultat :



Si on augmente la durée de simulation (ou qu'on diminue le pas), on s'aperçoit que la solution fournie par la méthode d'Euler explicite a tendance à s'éloigner drastiquement de la solution exacte, au contraire de la méthode implicite. Voici la même simultation sur l'intervalle $[0, 8p]$, avec toujours 1000 points d'interpolation.



Comme on le voit, la solution obtenue avec la méthode d'Euler explicite diverge [6], celle obtenue avec la méthode d'Euler implicite reste bornée. C'est pour cette raison que la méthode d'Euler implicite est en général préférable à la méthode explicite, car plus stable.

8.4.2 Schéma prédicteur correcteur explicite (Runge-Kutta 2)

Ce schéma repose sur la méthode du point milieu. Il faut donc estimer la dérivée au point milieu. La valeur de x au milieu de l'intervalle $[t_i, t_{i+1}]$ s'estime par $x_{i+1/2} = x_i + \frac{h}{2} f(x_i, t_i)$. On en déduit une approximation de la dérivée au point milieu $x'_{i+1/2} = f(x_{i+1/2}, t_i + h/2)$. La méthode consiste à utiliser cette approximation à partir de x_i pour en déduire x_{i+1} : $x_{i+1} = x_i + h x'_{i+1/2}$. C'est pour cela que le schéma s'appelle *prédicteur correcteur* : on fait une première estimation pour obtenir la dérivée au point milieu, puis on corrige l'estimation pour obtenir une approximation à l'extrémité droite de l'intervalle. Récapitulons :

$$x_{i+1} = x_i + hf\left(x_i + \frac{h}{2} f(x_i, t_i), t_i + \frac{h}{2}\right)$$

Le code Python s'écrit facilement :

```
def RK2(f,x0,T):
    x=x0
    X=[x]
    for i in range(len(T)-1):
        h=T[i+1]-T[i]
        x=x+h*f(x+h/2*f(x,T[i]),T[i]+h/2)
        X.append(x)
    return X
```

Appliquons-là à l'exponentielle. Le résultat est indiqué en figure 8.4. Déja pour 5 points, le résultat obtenu est très proche de la solution exacte. On peut montrer que l'ordre de la méthode est 2 sous de bonnes conditions, puisque la méthode du point milieu pour l'intégration est d'ordre 1 (exacte pour les fonctions polynomiales de degré au plus 1).

8.4.3 Méthode de Heun

Elle est basée sur la méthode des trapèzes. Puisque la méthode des trapèzes est elle-même d'ordre 1, cette méthode est d'ordre 2 sous de bonnes conditions. À partir de x_i, on peut estimer x_{i+1} comme avec la méthode d'Euler explicite : $\widetilde{x}_{i+1} = x_i + hf(x_i, t_i)$. La donnée de $\widetilde{x}_{i+1}$ permet d'estimer la dérivée de x au temps t_{i+1}, donnée par $f(\widetilde{x}_{i+1}, t_{i+1})$. On peut alors faire la moyenne comme dans la méthode des trapèzes, pour en déduire une nouvelle estimation de la solution au temps t_{i+1} : $x_{i+1} = x_i + \frac{h}{2}(f(x_i, t_i) + f(\widetilde{x}_{i+1}, t_{i+1}))$. Récapitulons :

$$x_{i+1} = x_i + h\frac{f(x_i, t_i) + f(x_i + hf(x_i, t_i), t_{i+1})}{2}$$

On en déduit le code Python suivant :

6. Ce qui est physiquement abérrant.

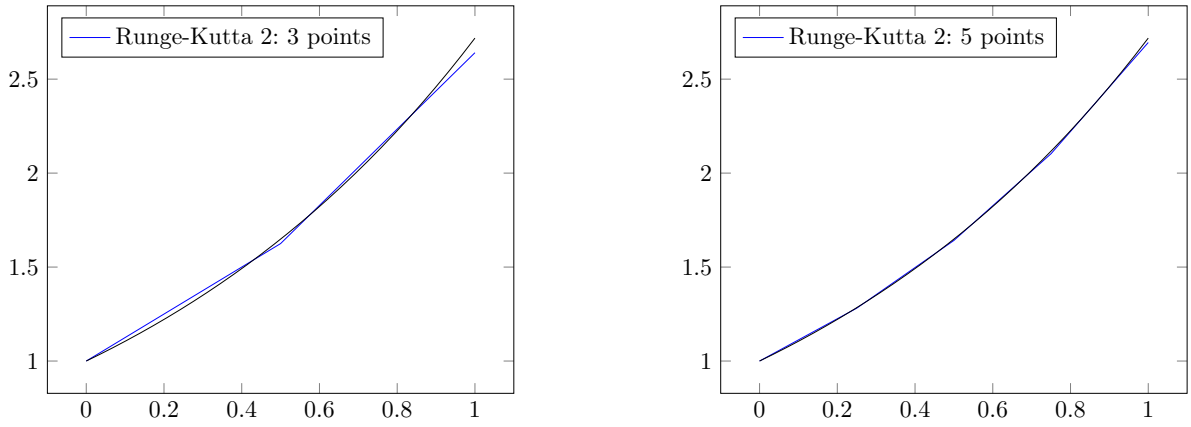


FIGURE 8.4 – Méthode Runge-Kutta 2 pour la fonction exponentielle.

```
def Heun(f,x0,T):
    x=x0
    X=[x]
    for i in range(len(T)-1):
        h=T[i+1]-T[i]
        k1=f(x,T[i]) #on stocke pour éviter un appel à f.
        x=x+h*(k1+f(x+h*k1,T[i+1]))/2
        X.append(x)
    return X
```

Le résultat est donné en figure 8.5. On remarque que le graphique est très semblable à celui de la méthode de Runge-

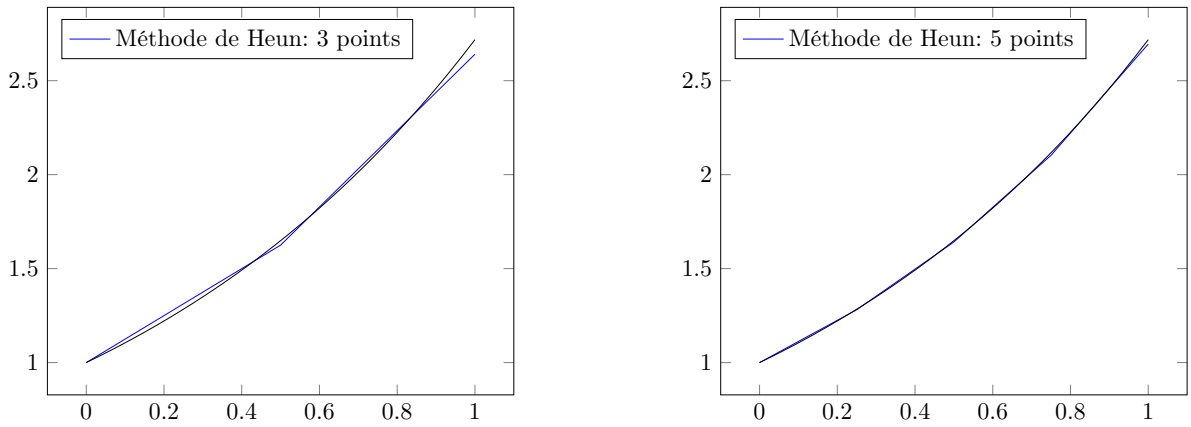


FIGURE 8.5 – Méthode de Heun pour la fonction exponentielle.

Kutta d'ordre 2 : en fait, pour le cas particulier de la fonction exponentielle, la fonction $f : (y, t) \mapsto y$ (linéaire en y) donne les mêmes itérations.

8.4.4 Méthode Runge-Kutta 4

Elle est basée sur la méthode de Simpson pour le calcul d'intégrale. Comme cette dernière est d'ordre 3, on en déduit que sous de bonnes conditions, la méthode Runge-Kutta 4 est d'ordre [7] 4.

Rappelons la méthode de Simpson, pour notre cas particulier (avec $t_{i+1/2} = (t_i + t_{i+1})/2$) :

$$\begin{aligned} x(t_{i+1}) &= x(t_i) + \int_{t_i}^{t_{i+1}} f(x(u), u)\,\mathrm{d}u \\ &\simeq x(t_i) + \frac{t_{i+1} - t_i}{6}\left(f(x(t_i), t_i) + 4f\left(x(t_{i+1/2}), t_{i+1/2}\right) + f(x(t_{i+1}), t_{i+1})\right) \end{aligned}$$

Ici, seule la valeur $x(t_i)$ est connue (de manière approchée, on connaît en fait x_i). L'idée est en quelque sorte de mélanger les méthodes précédentes :

7. C'était marqué dessus... !

— $f(x(t_i), t_i)$ s'approche par $k_1 = f(x_i, t_i)$

— on peut estimer $x(t_{i+1/2})$ à partir de x_i et $k_1 = f(x_i, t_i)$: cette approximation est similaire à celle des rectangles à gauche, on a donc $x(t_{i+1/2}) \simeq x(t_i) + \frac{h}{2}k_1$, et on pose donc $k_2 = f(x_i + \frac{h}{2}k_1, t_{i+1/2})$.

— on peut estimer $x(t_{i+1/2})$ d'une manière différente à l'aide de k_2 que l'on vient de calculer (un peu comme dans la méthode Runge-Kutta d'ordre 2) : $x(t_{i+1/2}) \simeq x(t_i) + \frac{h}{2}k_2$, et on pose $k_3 = f(x_i + \frac{h}{2}k_2, t_{i+1/2})$

— on estime $x(t_{i+1})$ et donc $f(x(t_{i+1}), t_{i+1})$ à l'aide de k_3 : on calcule $k_4 = f(x_i + hk_3, t_{i+1})$.

— pour estimer la somme ci-dessus, on utilise les termes que l'on vient de calculer. On a deux estimations différentes pour $f(x(t_{i+1/2}), t_{i+1/2})$, on prend comme dans la méthode de Heun la moyenne des deux. Ainsi l'approximation finale est :

$$\int_{t_i}^{t_{i+1}} f(x(u), u)\,\mathrm{d}u \simeq \frac{t_{i+1} - t_i}{6}(k_1 + 2k_2 + 2k_3 + k_4) \qquad \text{avec} \qquad \begin{cases} k_1 = f(t_i, x_i) \\ k_2 = f(t_i + \frac{h}{2}, x_i + \frac{h}{2}k_1) \\ k_3 = f(t_i + \frac{h}{2}, x_i + \frac{h}{2}k_2) \\ k_4 = f(t_i + h, x_i + hk_3) \end{cases}$$

Voici un code Python possible :

```
def RK4(f,x0,T):
    x=x0
    X=[x]
    for i in range(len(T)-1):
        h=T[i+1]-T[i]
        k1=f(x,T[i])
        k2=f(x+h/2*k1,T[i]+h/2)
        k3=f(x+h/2*k2,T[i]+h/2)
        k4=f(x+h*k3,T[i+1])
        x=x+h*(k1+2*k2+2*k3+k4)/6
        X.append(x)
    return X
```

Le résultat est donné en figure 8.6. L'approximation est très précise : les points où la fonction est calculée sont quasiment confondus avec la fonction elle-même.

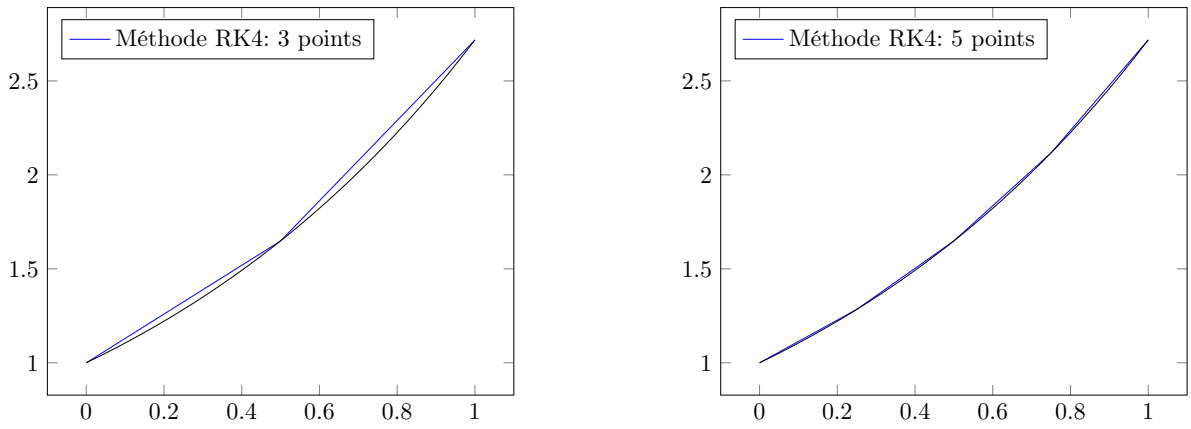
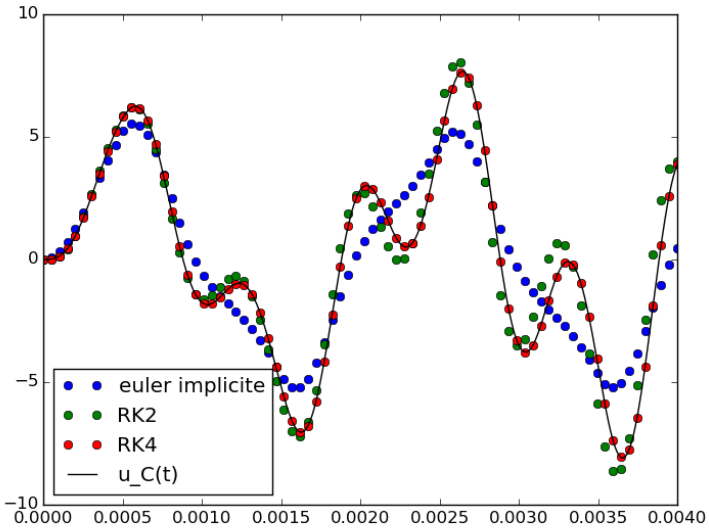


FIGURE 8.6 – Méthode de Runge-Kutta 4 pour la fonction exponentielle.

8.4.5 Comparaison avec la tension aux bornes du condensateur

À titre de comparaison, on conclut en observant les différentes méthodes pour la résolution de l'équation différentielle donnant la tension aux bornes du condensateur d'un circuit RLC soumis à une tension en triangles, sur un intervalle $[0, 2p]$, avec p la période du générateur. On ne prend que 80 points sur cet intervalle pour bien observer les différences de précision.



La conclusion est sans appel :

- la méthode d'Euler explicite explose complètement, et n'a pas été représentée ;
- la méthode d'Euler implicite, bien que stable, ne donne pas une approximation très pertinente ;
- la méthode RK2 (et la méthode de Heun, non représentée car ses résultats sont très proches de la méthode RK2) donne une approximation bien plus précise ;
- la méthode RK4 donne une approximation indiscernable de la solution proposée par `odeint` avec 10000 points (la courbe $u_C(t)$, qu'on considère exacte).

Ainsi, utiliser une méthode de résolution avec un ordre élevé est pertinent pour obtenir une meilleure approximation. Évidemment, la méthode RK4 est un peu plus coûteuse car elle demande plus d'appels à la fonction f. Mais toutes les méthodes proposées ont une complexité en $O(1/h)$ avec h le pas, et pour avoir une solution aussi précise avec les autres méthodes, il faut considérablement réduire le pas : le surcoût est ainsi petit par rapport à la précision gagnée.

Troisième partie

Bases de données

Chapitre 9

Requêtes dans les bases de données

9.1 Introduction : limite des structures de données plates pour la recherche d'informations

On souhaite représenter l'ensemble des élèves de première 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 (MPSI ou PCSI) suivie ;

— le numéro (1 ou 2).

Prenons pour simplifier un ensemble d'élèves réduit :

Prénom	Nom	Filière	Numéro	Lycée d'origine
Mathilde	Dufour	PCSI	2	Calmette
Léa	Dupond	MPSI	2	Massena
Paul	Dugommier	PCSI	1	Massena
Mathilde	Dugommier	MPSI	1	Calmette
Clément	Durand	PCSI	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_sup=[
[
[['Mathilde', 'Dugommier', 'Calmette']],
[['Léa', 'Dupond', 'Massena']]
], [
[['Paul', 'Dugommier', 'Massena'], ['Clément', 'Durand', 'Parc Imperial']],
[['Mathilde', 'Dufour', 'Calmette']]
]
]
```

Ici, `classes_de_sup[1]` fournit par exemple les deux classes de PCSI, alors que `classes_de_sup[0][0]` donne la liste des élèves de MPSI 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_sup[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_sup:
    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 MPSI 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...)

9.2 Présentation succincte des bases de données

9.2.1 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 ;
— ...

9.2.2 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	834	1
Léa	Dupond	832	2
Paul	Dugommier	833	2
Mathilde	Dugommier	831	1
Clément	Durand	833	3

Table lycee

id_lycee	nom
1	Calmette
2	Massena
3	Parc Imperial

Table classe

id_classe	filiere	numero
831	MPSI	1
832	MPSI	2
833	PCSI	1
834	PCSI	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 MPSI 1, et les classer par ordre alphabétique

```
SELECT nom,prenom
FROM eleves
WHERE id_classe = 831
ORDER BY nom
```

Ici, on a supposé que l'on connaissait le code de la MPSI 1, si ce n'est pas le cas, il faut croiser les tables :

sélectionner les élèves de PCSI 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="PCSI" 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 MPSI :

```
SELECT COUNT (*)
FROM eleve
JOIN classe ON eleve.id_classe=classes.id_classe
WHERE filiere="MPSI" ;
```

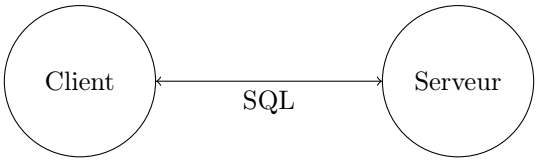
Le programme officiel impose uniquement l'enseignement des requêtes de recherche dans une base de données. Voici toutefois un exemple d'insertion dans une base :

```
INSERT INTO eleve (nom, prenom, id_classe)
VALUES (Ducourneau, Guillaume, 833) ;
```

(Il n'est pas obligatoire de renseigner tous les champs, mais ceci sort assez largement du programme).

9.2.3 Architecture client-serveur

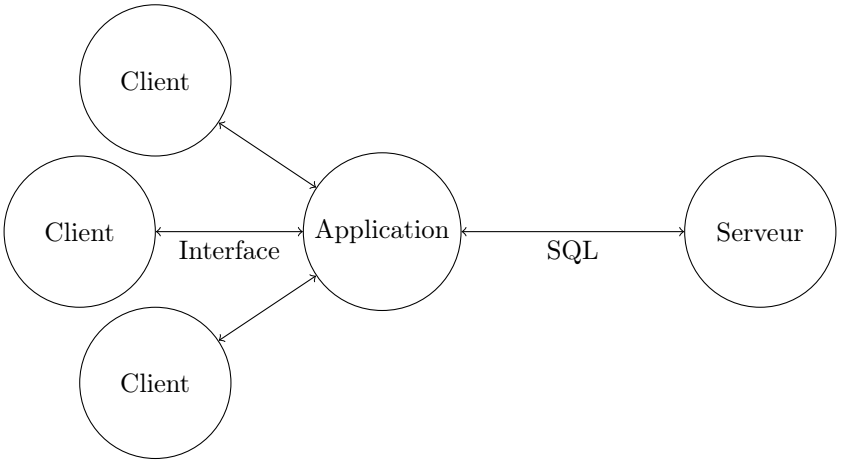
Les bases de données sont articulées sous la forme client-serveur : l'utilisateur travaille sur un poste (client) qui peut être éloigné de l'ordinateur qui gère les données (serveur). Il est nécessaire de permettre un dialogue entre ces deux parties.



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.



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...

9.2.4 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 [1], bien qu'elle n'en soit qu'une abstraction. Dans ce cours, nous verrons les requêtes dans des bases de données à la fois d'un point de vue pratique (SQL) et d'un point de vue théorique (opérations en algèbre relationnelle).

9.3 Vocabulaire des bases de données

9.3.1 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 :

— 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	PCSI	2	Calmette	18
Léa	Dupond	MPSI	2	Massena	14
Paul	Dugommier	PCSI	1	Massena	12
Mathilde	Dugommier	MPSI	1	Calmette	15
Clément	Durand	PCSI	1	Parc Imperial	13

9.3.2 Vocabulaire

Attributs. Les différents titres de colonnes sont appelés *attributs*. 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, mais on peut imaginer que le domaine des classes est réduit à l'ensemble des sigles dénotant une filière de classe préparatoire (MPSI,PCSI, BCPST, PSI...), le numéro est un entier, et la note du bac un flottant (qu'on peut imposer dans l'intervalle $[0, 20]$ avec éventuellement un certain nombre de chiffres significatifs...). Tout cela a été fixé à la création de la base, et ne va donc pas nous concerner dans nos requêtes de recherche.

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.

Tuple. Chaque ligne s'appelle un p-uplet (ou tuple en anglais). 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 du tuple (Mathilde, Dufour, PCSI, 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.

1. Un modèle équivalent est le calcul relationnel, que nous n'étudierons pas

Relation. On appelle *relation* (ou table) un ensemble fini de p-uplets de $\mathrm{Dom}(A_1) \times \cdots \cup \times \mathrm{Dom}(A_p)$, qu'on notera souvent $\mathcal{R}$ dans ce cours. Pour préciser que la relation est associée au schéma relationnel S, on notera $\mathcal{R}(S)$. Les éléments de $\mathcal{R}(S)$ (les lignes dans le tableau ci-dessus) sont appelés *valeurs* ou *enregistrements*.

9.3.3 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 $X = (B_1, \ldots, B_n) \subseteq \mathcal{S}$, on note $t[X]$ le n-uplet $(t[B_1], \ldots, t[B_n])$

Définition 9.1. *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 $S = \{A_1, \ldots, A_p\}$, si $t[A_1, \ldots, A_p] = t'[A_1, \ldots, A_p]$ 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.

9.3.4 Clés

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 l'utilisation de relations multiples.

Définition 9.2. *Une super-clé d'une relation $\mathcal{R}$ est un ensemble X d'attributs tel que*

$$\forall t, t' \in R \qquad t[X] = t'[X] \Longrightarrow (t = t')$$

Ainsi, si les deux enregistrements sont égaux sur les attributs de X alors ils sont égaux partout. En raison de la contrainte d'ensemble pour les enregistrements, l'ensemble de tous les attributs est toujours une super-clé. Dans l'exemple ci-dessus {prenom} et {nom} ne sont pas des super-clés, contrairement à {nom, prenom}, par exemple.

Définition 9.3. *Une clé candidate d'une relation r est une super-clé minimale (pour l'inclusion).*

K est donc une clé candidate si

— K est une super-clé.

— Pour tout K' inclus dans K, K' n'est pas une super-clé.

Dans l'exemple ci-dessus, {nom, prenom} est une clé candidate.

Proposition 9.4. *Une super-clé qui ne contient qu'un seul attribut est toujours une clé candidate.*

Par contre une telle super-clé n'existe pas toujours, il n'y en a pas dans le tableau Éleves ci-dessus. À Masséna, le numéro d'une classe (831, 833...) est une super-clé :

Table Classes		
classe	filière	numéro
831	MPSI	1
832	MPSI	2
833	PCSI	1
834	PCSI	2

Définition 9.5. *Parmi les clés candidates on en choisit une : c'est la clé primaire.*

Pour indiquer la clé primaire on souligne les attributs correspondants dans la table. Indiquer au système une clé primaire pour chaque table permet une indexation des données à l'aide de cette clé, ce qui renforce l'efficacité des procédures d'interrogation de la table. Cette indication doit se faire à la création de la table.

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} est une clé candidate, mais rien ne garantit qu'elle puisse le rester !

On introduit souvent un attribut supplémentaire (commençant souvent par id), dans la définition de la relation qui sera un entier qu'on incrémentera à chaque ajout d'un tuple. 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	PCSI	2	Calmette	18
2	Léa	Dupond	MPSI	2	Massena	14
3	Paul	Dugommier	PCSI	1	Massena	12
4	Mathilde	Dugommier	MPSI	1	Calmette	15
5	Clément	Durand	PCSI	1	Parc Imperial	13

On ne le fera pas systématiquement, notamment lorsque dans la relation se trouve des *clés étrangères*, concept que l'on détaillera plus tard.

9.4 Algèbre relationnelle simple

L'algèbre relationnelle sert à formaliser les questions que l'on peut poser à une relation ou, comme plus tard, à un ensemble de relation. Elle exprime ce que l'on fait ; le calcul relationnel exprime ce que l'on veut. Le langage SQL est l'expression concrète des opérations de l'algèbre relationnelle.

9.4.1 Définition

L'algèbre relationnelle est un ensemble d'opérateurs que l'on peut appliquer à des relations, et dont le résultat est une relation. Comme le résultat est toujours une relation on pourra combiner les opérateurs : on forme ainsi, à partir d'opérateurs élémentaires, des requêtes élaborées. L'objectif est de pouvoir exprimer toute manipulation raisonnable par une expression algébrique des opérateurs élémentaires.

Voici des exemples de questions qu'on peut demander au serveur de bases de données.

— Quels sont les élèves venant du lycée Calmette ? Ceux de MPSI 1 ?

— Quelle est la moyenne au baccalauréat des élèves de Masséna ? Qui sont ceux qui ont obtenu une mention ?

— Étant donnés deux relations pour les élèves de Masséna en 2013-2014 et 2014-2015, qui sont les 5/2 ?

La troisième opération demande d'avoir deux relations, mais on va quand même y répondre ici.

9.4.2 Opérateurs ensemblistes

Un premier type d'opérateur combine 2 relations qui ont le même schéma ; on les utilisera surtout pour assembler les résultats d'autres requêtes. $\mathcal{R}$ et $\mathcal{R}'$ sont deux relations ayant le même schéma (c'est-à-dire les mêmes attributs).

— $\mathcal{R} \cup \mathcal{R}'$ 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 $\mathcal{R}'$.

Il suffit donc deux faire l'intersection de deux relations pour avoir les élèves ayant fait 5/2 en 2014-2015.

9.4.3 Opérateurs spécifiques

Les opérateurs étudiés ici sont ceux qui utilisent la structure des relations. Par exemple la question « Quels sont les élèves en MPSI 1 ? » se traduit par : donner les noms et prénoms (projection) des élèves de Masséna étant dans la classe 831 (sélection). On reprend notre table suivante ici :

Table eleve

prenom	nom	filiere	numero	lycee_origine	note_bac
Mathilde	Dufour	PCSI	2	Calmette	18
Léa	Dupond	MPSI	2	Massena	14
Paul	Dugommier	PCSI	1	Massena	12
Mathilde	Dugommier	MPSI	1	Calmette	15
Clément	Durand	PCSI	1	Parc Imperial	13

Projection. La projection consiste à ne garder qu'une partie des attributs. Pour $X \subseteq S$ un ensemble d'attributs, On note $\pi_X(r)$ la relation extraite de $\mathcal{R}$ avec les mêmes tuples restreints à l'ensemble X.

$$\pi_X(\mathcal{R}) = \{t[X] \mid t \in \mathcal{R}\}$$

Par exemple, $\pi_{nom,prenom}(eleve)$ est la table suivante :

$\pi_{nom,prenom}(eleve)$	
prenom	nom
Mathilde	Dufour
Léa	Dupond
Paul	Dugommier
Mathilde	Dugommier
Clément	Durand

Rappelons que dans une relation, les tuples forment un ensemble. Il se peut donc que $\pi_X(\mathcal{R})$ ait moins d'éléments que la table initiale. (En fait le nombre d'éléments est conservé si et seulement si l'ensemble d'attributs X est une super-clé de la relation).

$\pi_{prenom}(eleve)$
prenom
Mathilde
Léa
Paul
Clément

Sélection. La sélection consiste à ne garder que les tuples qui vérifie une propriété. On note $\sigma_{\mathcal{P}}(\mathcal{R})$ la relation extraite de $\mathcal{R}$ avec les mêmes attributs dont les tuples vérifient la propriété $\mathcal{P}$.

$$\sigma_{\mathcal{P}}(\mathcal{R}) = \{t \in \mathcal{R} \mid \mathcal{P}(t)\}$$

Les propriétés élémentaires sont de la forme E op E' où op un opérateur de comparaison $(=, <, \leq, >, \geq)$ et E et E' des expression construites à partir des attributs et des constantes avec des fonctions usuelles. Une propriété composée avec des connecteurs logiques (ET, OU et NON, aussi notés $\wedge$, $\vee$ et $\neg$) correspond à des unions et intersections et peut être écrite directement. Par exemple $\sigma_{\mathcal{P}\vee\mathcal{P}'}(\mathcal{R}) = \sigma_{\mathcal{P}}(\mathcal{R}) \cup \sigma_{\mathcal{P}'}(\mathcal{R})$. On utilisera donc plutôt des conditions composées, qui ont une écriture plus courte. Par exemple, les élèves de MPSI dans la table ci-dessus est :

$\sigma_{filiere="MPSI"OUprenom="Mathilde"}(eleve)$					
prenom	nom	filiere	numero	lycee_origine	note_bac
Mathilde	Dufour	PCSI	2	Calmette	18
Léa	Dupond	MPSI	2	Massena	14
Mathilde	Dugommier	MPSI	1	Calmette	15

Sélection et projection vont souvent ensemble : si on s'intéresse aux noms et prénoms des élèves de MPSI, on obtient :

$\pi_{nom,prenom}(\sigma_{filiere="MPSI"}(eleve))$	
prenom	nom
Léa	Dupond
Mathilde	Dugommier

Renommage. Le renommage consiste à renommer un attribut. Ce sera utile lors de produits de tables lorsque deux tables ont des attributs portant le même nom mais correspondent à des données différentes. Si $A \in S$ où S est le schéma de $\mathcal{R}(S)$ et $B \notin S$ on peut renommer A en B, pour obtenir un élément de $\mathcal{R}'(S')$ de schéma $S' = (\mathcal{S}\backslash\{A\}) \cup B$:

$$\rho_{A\to B}(\mathcal{R}) = \{t \mid \exists r \in \mathcal{R}, \quad t[B] = r[A] \qquad \text{et} \qquad \forall C \in S\backslash\{A\}, \quad t[C] = s[C]\}$$

Par extension, on notera $\rho_{A_1,\dots,A_p\to B_1,\dots,B_p}$ pour $\rho_{A_1\to B_1} \circ \cdots \circ \rho_{A_p\to B_p}$ (en supposant les A_i et B_j tous distincts)

Table classe		
id_classe	filière	numéro
831	MPSI	1
832	MPSI	2
833	PCSI	1
834	PCSI	2

Table $\rho_{filiere\to sec,numero\to nb}$(classe)		
id_classe	sec	nb
831	MPSI	1
832	MPSI	2
833	PCSI	1
834	PCSI	2

9.5 SQL

9.5.1 Introduction

La présentation des bases de données via l'algèbre relationnelle correspond à une abstraction : dans la pratique chaque éditeur d'un logiciel de gestion de base de données organise les données afin d'optimiser leur accès, mais ceci est transparent pour l'utilisateur. En pratique l'utilisateur envoie des requêtes (via le serveur applicatif) : c'est l'architecture trois tiers déja évoquée.

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*. Ce langage reprend la structure de l'algèbre relationnelle en y ajoutant des moyens de calculs et autres améliorations (ordonnancement des résultats, par exemple). Ce langage est très proche du langage humain (l'anglais).

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.

9.5.2 Syntaxe

Les représentations des relations se font avec le modèle du tableau. En SQL on parlera de

— tables à la places de relations ;

— colonnes à la places d'attributs ;

— lignes à la places de tuples.

La forme générale d'une requête en SQL est

```
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 dans une base de données est SELECT, qu'on retrouvera en tête de toutes nos requêtes SQL. Les requêtes se terminent par un point-virgule.

Projection. Basiquement, SELECT permet de faire une projection (attention à ne pas confondre avec la sélection...). Il suffit d'indiquer les attributs que l'on veut garder juste après SELECT :

```
SELECT A1,...,Ap FROM table ;
```

Les attributs (colonnes) à garder sont énumérés et séparés par une virgule. 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 ;
```

Le mot-clef FROM permet de spécifier le nom de la table à utiliser. Attention : en SQL, les résultats d'une requête ne forment pas une table car 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"
```

Sélection. La sélection se fait avec le mot-clef WHERE, placé après le nom de la table. 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) qu'on notera aussi $\wedge$, $\vee$ et $\neg$.

Renommage. On peut renommer un ou plusieurs attributs avec AS ou même en juxtaposant le nouveau nom à droite de l'ancien. Ceci sera particulièrement utile lorsqu'on utilisera plusieurs tables dont les noms d'attributs sont les mêmes, ou lorsqu'on fera des sous-requêtes. La syntaxe générale est :

```
SELECT A1 AS B1,..,Ai AS Bi, C1,..,Cj FROM table ;
```

Le `AS` est facultatif, mais c'est plus lisible.
Par exemple : `SELECT prenom p, nom n, notebac/2 note_sur_10 FROM eleve ;` produit la table suivante :

Table $\rho_{note\_bac \to note\_sur\_10}(\pi_{nom,prenom,note\_bac/2}(eleve))$

p	n	note_sur_10
Mathilde	Dufour	9
Léa	Dupond	7
Paul	Dugommier	6
Mathilde	Dugommier	7
Clément	Durand	6

La projection est un peu abusive ici car on fait une opération arithmétique en plus.

Opérations ensemblistes. Si on veut combiner plusieurs requêtes on peut les assembler avec UNION, INTERSECT ou EXCEPT, qui réalise l'union, l'intersection et la différence. Ceci dit, lorsqu'on a qu'une seule table il est plus simple (et préférable) de combiner les conditions. Voici un exemple : imaginons que l'on ait deux tables eleves_14_15 et eleves_15_16 qui donnent les élèves de deuxième année du lycée de deux années successives, alors ceux ayant fait 5/2 (dans la même classe...) en 2015-2016 sont :

```
SELECT * FROM eleves_14_15
INTERSECT
SELECT * FROM eleves_15_16
```

Opérations entre attributs. Il est tout à fait possible de réaliser des opérations entre attributs : par exemple la requête

```
SELECT a+b+c FROM ...
```

fait la somme de trois attributs a, b et c d'une table.

9.6 Agrégats et fonctions d'agrégation

9.6.1 Agrégats

Définition 9.6. *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 ne change pas lorsqu'on permute les arguments.

Définition 9.7. *On appelle fonction d'agrégation $E \to F$ une suite $(f_n)_{n\geq 1}$ de fonctions symétriques de $E^n \to F$.*

Ça paraît un peu étrange, mais c'est légitime. Voila des exemples de fonctions d'agrégation : la somme (on a $f_n(x_1, \dots, x_n) = \sum x_i$), le produit, le max, le min, le cardinal (dans ce cas f_n est la fonction constante égale à n), la moyenne (sur des ensembles où ces opérations sont possibles)...

On peut utiliser de telles fonctions en regroupant les données par paquets pour obtenir une nouvelle table. 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 f appliquée à l'attribut A pour la relation $\sigma_{t[X]=t_0}(\mathcal{R})$ c'est-à-dire appliquée à l'ensemble des tuples qui prennent les valeurs 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 regroupe la relation selon les valeurs des attributs de X et on calcule $f(A)$ pour tous les ensembles de lignes définies par ces regroupements.

Exemple :

— ${}_{filiere}\gamma_{MOYENNE(notebac)}(eleve)$ est une relation regroupant la moyenne des notes au bac suivant les filières.

— ${}_{\emptyset}\gamma_{MOYENNE(notebac)}(eleve)$ est une relation avec un unique tuple de taille 1, donnant la moyenne des notes au bac des élèves. 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.

9.6.2 SQL

Les résultats d'une requête seront souvent utilisés ensuite, par exemple à des fins statistiques. SQL contient la possibilité de faire quelques uns de ces calculs. Les fonctions disponibles (de base) sont

— 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

Le résultat est une table avec une unique ligne et une unique colonne, que l'on peut utiliser comme valeur. Si f est une de ces fonctions on l'emploie sous la forme

```
SELECT f(attribut) FROM table WHERE condition ;
```

Par exemple, avec la relation eleve, `SELECT AVG(notebac) FROM eleves` produit 14.4. Le mot-clef GROUP BY sert à indiquer sur quels attributs sont effectués les regroupements.

Par exemple :

```
SELECT AVG(notebac) FROM eleves GROUP BY filiere ;
```

produit une table ayant une unique colonne :

AVG(notebac)
14.5
14.333333333333334

Ici on a en fait fait une sélection supplémentaire, il est ici a priori plus intéressant d'afficher aussi la filière

```
SELECT filiere,AVG(notebac) FROM eleves GROUP BY filiere ;
```

qui produit :

filiere	AVG(notebac)
MPSI	14.5
PCSI	14.333333333333334

9.6.3 Sélection après agrégation

Pour faire une sélection après une opération d'agrégation, on utilise HAVING. 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 $\sigma_{MOYENNE(notebac)>14.4} \circ {}_{filiere}\gamma_{MOYENNE(notebac)}(eleve)$. En SQL, on fait la distinction : on utilise le mot-clef HAVING pour sélectionnier *après* une agrégation. 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)
MPSI	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
MPSI	14.5

Ceci correspond en algèbre relationnelle à la composition

$$\sigma_{moy>14.4} \circ \rho_{MOYENNE(notebac)\to moy} \circ {}_{filiere}\gamma_{MOYENNE(notebac)}(eleve)$$

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. Lorsqu'on a le choix, il vaut mieux sélectionner en amont, pour n'effectuer l'agrégation que sur un nombre minimal de lignes. Évidemment dans l'exemple ci-dessus, la sélection en aval est tout à fait légitime car on ne pouvait pas faire de sélection en amont.

9.7 Affichage des résultats

Cette section n'est pas vraiment de l'algèbre relationnelle puisqu'on ne raisonne plus sur une relation en temps que simple ensemble, mais comme ensemble ordonné.

9.7.1 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	PCSI	1	Massena	12
Clément	Durand	PCSI	1	Parc Imperial	13
Léa	Dupond	MPSI	2	Massena	14
Mathilde	Dugommier	MPSI	1	Calmette	15
Mathilde	Dufour	PCSI	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.

9.7.2 Limiter l'affichage avec LIMIT et OFFSET

Ce point est hors programme, mais n'est pas dur à comprendre. Il est précisé pour la culture. 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 : 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, 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.

9.8 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 composition

$$\sigma_{moy>14.4} \circ \rho_{MOYENNE(notebac)\to moy} \circ {}_{filiere}\gamma_{MOYENNE(notebac)}(eleve)$$

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. 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 PCSI qui ont même prénom qu'un élève de MPSI ? », on peut répondre avec la requête :

```
SELECT * FROM eleve WHERE filiere="PCSI" AND prenom IN (SELECT prenom FROM eleve WHERE filiere="MPSI")
```

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 (SELECT ...) AS r SELECT .. FROM r ...
```

9.9 Produit cartésien et jointure

9.9.1 Introduction

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). Les principes guidant cette organisation sont plutôt hors programme, on va simplement étudier ici l'utilisation simultanée de plusieurs tables, comme celles-ci dessous.

Table eleve

prenom	nom	id_classe	id_lycee
Mathilde	Dufour	834	1
Léa	Dupond	832	2
Paul	Dugommier	833	2
Mathilde	Dugommier	831	1
Clément	Durand	833	3

Table lycee

id_lycee	nom_lycee
1	Calmette
2	Massena
3	Parc Imperial

Table classe

id_classe	filiere	numero
831	MPSI	1
832	MPSI	2
833	PCSI	1
834	PCSI	2

9.9.2 Notion de clé étrangère

Une clé étrangère d'une relation $\mathcal{R}$ est un attribut qui est lié à une clé primaire d'une autre relation $\mathcal{R}'$. Notamment, les valeurs que peut prendre cet attribut dans $\mathcal{R}$ est limité aux valeurs de la clé primaire dans $\mathcal{R}'$. Dans l'ensemble de relations ci-dessus, les deux attributs `id_classe` et `id_lycee` de la table `eleve` peuvent naturellement être considérés comme des clés étrangères vers les attributs du même nom dans les tables `classe` et `lycee` (il n'y a aucune obligation que les noms soient les mêmes, d'ailleurs les clés primaires s'appellent souvent `id` en pratique !).

Tout comme la déclaration d'une clé primaire, la déclaration d'une clé étrangère se fait à la création de la base. L'utilité d'une telle contrainte n'apparaît pas vraiment lorsqu'on fait des recherches dans une base (la seule chose au programme), mais néanmoins il est facile d'en percevoir l'intérêt : si on voulait rajouter un éléve dans la classe 8340, le serveur nous dirait que cette classe n'existe pas. Ceci dit, les clés étrangères sont très utiles pour faire des jointures sur les tables, ce qu'on va voir bientôt.

9.9.3 Produit

Le premier moyen d'assembler deux relations est d'en faire le produit cartésien.

Définition 9.8 (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					
prenom	nom	id_classe	id_lycee	id_lycee	nom_lycee
Mathilde	Dufour	834	1	1	Calmette
Mathilde	Dufour	834	1	2	Massena
Mathilde	Dufour	834	1	3	Parc Imperial
Léa	Dupond	832	2	1	Calmette
Léa	Dupond	832	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).

9.9.4 Division

Il existe une fonction réciproque du produit, la division, dans l'algèbre relationnelle. Elle est l'analogue de la division euclidienne vis à vis de la multiplication sur les entiers. La division de a par b est le plus grand entier q tel que $bq \leq a$. C'est la même chose avec des relations, en remplaçant la relation d'ordre $\leq$ par l'inclusion $\subseteq$.

Définition 9.9 (Division de deux relations)**.** *Si $\mathcal{R}$ est une relation de schéma S et $\mathcal{R}'$ une relation de schéma S' avec $S' \subsetneq S$, alors la division de $\mathcal{R}$ par $\mathcal{R}'$ est la relation $\mathcal{R} \div \mathcal{R}'$ de schéma $S \backslash S'$ définie par*

$$\mathcal{R} \div \mathcal{R}' = \{u \mid \forall t \in \mathcal{R}', (u,t) \in \mathcal{R}\}$$

De manière équivalente, on peut aussi définir $\mathcal{R} \div \mathcal{R}'$ comme la plus grande relation $\mathcal{R}''$ de schéma $S \backslash S'$ telle que $\mathcal{R}' \times \mathcal{R}'' \subseteq \mathcal{R}$. Avec cette définition alternative, le lien avec la division euclidienne apparaît plus clairement.

Par exemple, la division de la relation $\pi_{id\_classe,id\_lycee}(eleve)$ par $\pi_{id\_lycee}(lycee)$ donne l'ensemble des classes possédant un élève en provenance de chacun des lycées de la table lycee. Avec les petites tables ci-dessus cette division donne une relation vide, mais il suffirait d'ajouter un élève en 833 provenant de Calmette pour que la 833 se retrouve dans $\pi_{id\_classe,id\_lycee}(eleve) \div \pi_{id\_lycee}(lycee)$

9.9.5 Jointure naturelle

Visiblement, dans l'exemple de produit cartésien ci-dessus, un bon nombre de lignes n'ont pas d'intérêt réel : ce sont celles pour lesquelles eleve.id_lycee est différent de lycee.id_lycee. Il est ici naturel lors du produit d'identifier les colonnes des attributs qui portent le même nom : on parle de jointure naturelle.

Définition 9.10 (Jointure naturelle)**.** *Pour $\mathcal{R}$ et $\mathcal{R}'$ deux relations de schémas S et S' avec $S \cap S' = S''$, on définit la jointure naturelle de $\mathcal{R}$ et $\mathcal{R}'$ comme la relation $\mathcal{R} \bowtie \mathcal{R}'$ de schéma $S \cup S'$ définie par*

$$\mathcal{R} \bowtie \mathcal{R}' = \{u \mid u[S] \in \mathcal{R} \text{ et } u[S'] \in \mathcal{R}'\}$$

On a déja vu deux cas particuliers de cette notion :

— si $S \cap S' = \emptyset$ on retrouve le produit de $\mathcal{R}$ et $\mathcal{R}'$.

— si $S = S'$ on retrouve l'intersection de de $\mathcal{R}$ et $\mathcal{R}'$.

Voici un exemple de jointure naturelle :

Table eleve⋈lycee				
prenom	nom	id_classe	id_lycee	nom_lycee
Mathilde	Dufour	834	1	Calmette
Léa	Dupond	832	2	Massena
Paul	Dugommier	833	2	Massena
Mathilde	Dugommier	831	1	Calmette
Clément	Durand	833	3	Parc Impérial

Toutefois, cette jointure n'est pas explicitement au programme : la seule est la jointure symétrique qu'on va définir ci-dessous.

9.9.6 Jointure

La jointure n'est, d'un point de vue algébrique, qu'une abréviation : elle consiste à sélectionner dans le produit cartésien.

Définition 9.11 (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*

$$\mathcal{R} \bowtie_{\mathcal{C}} \mathcal{R}' = \sigma_{\mathcal{C}}(\mathcal{R} \times \mathcal{R}')$$

Si $\mathcal{C}$ est la condition triviale (toujours vraie) on retrouve le produit cartésien standard. Si on considère que les noms des attributs d'une relation $\mathcal{R}$ sont $\mathcal{R}.A$ au lieu de A alors la jointure naturelle ressemble à une jointure avec comme condition la conjonction des égalités $\mathcal{R}.A_i = \mathcal{R}'.A_i$ pour les attributs A_i appartenant à $R \cap R'$. La différence est qu'alors les colonnes égales sont répétées : la jointure naturelle est plus... naturelle. Voici un exemple :

Table eleve $\bowtie_{\text{eleve.id\_lycee=lycee.id\_lycee}}$ lycee					
prenom	nom	id_classe	id_lycee	id_lycee	nom_lycee
Mathilde	Dufour	834	1	1	Calmette
Léa	Dupond	832	2	2	Massena
Paul	Dugommier	833	2	2	Massena
Mathilde	Dugommier	831	1	1	Calmette
Clément	Durand	833	3	3	Parc Impérial

On parle de jointure symétrique lorsque la condition de jointure est l'égalité de deux attributs (comme précédemment) : c'est la seule au programme.

9.10 Tables multiples dans SQL

9.10.1 Produit cartésien

La traduction en SQL du produit cartésien est très simple : il suffit d'énumérer les différentes tables dans la clause FROM. SQL considère les noms des colonnes de la table T sous la forme T.nom, il n'y a pas de noms identiques. Cependant, lors de l'affichage, le titre de la colonne ne sera que le nom de l'attribut et on peut croire que la contrainte d'ensemble pour les attributs n'est pas respectée. Lorsqu'on aura des noms d'attributs égaux, il faudra donc spécifier la table lors de l'écriture des requêtes pour lever les ambiguïtés, ce dont on peut se passer si les noms d'attributs sont distincts. Il est courant de procéder à un renommage de la table elle-même pour écourter l'écriture des requêtes.

Voici trois exemples de requêtes simples avec des produits cartésiens :

```
SELECT * FROM eleve,classe,lycee ;
SELECT e.nom,e.prenom FROM eleve e, lycee L WHERE e.id_lycee=L.id_lycee AND L.id_lycee="Calmette" ;
SELECT e.nom,e.prenom FROM eleve e, lycee L, classe c WHERE e.id_lycee=L.id_lycee AND
      c.id_classe=e.id_classe AND L.nom_lycee="Calmette" AND c.filiere="MPSI" AND c.numero="1" ;
```

La première fait simplement le produit des trois tables (ce qui donne une grosse table !). La deuxième fait une sélection après un produit cartésien : en fait on fait d'abord une jointure sans le dire, pour sélectionner les élèves venant du lycée Calmette. La troisième sélectionne également ceux qui sont en MPSI 1, avec une jointure supplémentaire, toujours sans le dire.

9.10.2 Jointure naturelle

Il suffit d'employer NATURAL JOIN pour effectuer une jointure naturelle entre deux tables. Voici un exemple, on reprend la requête précédente, rendue (un peu) plus courte avec NATURAL JOIN.

```
SELECT nom,prenom FROM eleve NATURAL JOIN lycee NATURAL JOIN classe WHERE
    id_lycee="Calmette" AND filiere="MPSI" AND numero="1" ;
```

9.10.3 Division

Elle n'existe pas en SQL, et existe en algèbre relationnelle pour garantir une certaine complétude des opérations. On peut par exemple réaliser une division avec des sous-requêtes, des jointures et la fonction d'agrégation de comptage, mais ce n'est pas simple et sort du cadre de ce cours.

9.10.4 Jointure

La traduction de la jointure en SQL est JOIN. On spécifie la condition de jointure avec ON :

```
table1 JOIN table2 ON condition
```

Bien que cela soit possible il n'est pas recommandé de placer toutes les sélections dans la clause ON à la place de la clause WHERE. Typiquement pour une *équijointure* (jointure avec condition d'égalité entre deux attributs), on placera la condition d'égalité de deux colonnes dans la clause ON, et les autres conditions de sélection dans la clause WHERE. L'intérêt d'une jointures est de structurer plus clairement les requêtes, bien qu'on puisse s'en passer en faisant des produits cartésiens et en sélectionnant, comme on l'a fait plus haut. Voici encore une variante de la requête qu'on a vu plus haut avec produit cartésien et jointure naturelle :

```
SELECT nom,prenom FROM eleve e JOIN lycee L ON e.id_lycee=L.id_lycee JOIN classe c ON
    e.id_classe=c.id_classe WHERE e.nom_lycee="Calmette" AND c.filiere="MPSI" AND c.numero="1" ;
```

Petite remarque : il n'y a pas d'ordre dans les attributs en SQL. Rien n'empêche de donner d'abord les tables que l'on va utiliser (séparées par JOIN) pour donner ensuite les conditions de jointures (séparées par ON).

9.11 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
```

Quatrième partie

Algorithmique « avancée »

Chapitre 10

Algorithmes naïfs de tri

10.1 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 10.1. — *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 10.2. *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 10.3 : 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.

10.2 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.

10.3 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.

10.4 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 :

$$\text{Inv : Pour tout } k \text{ tel que } j < k \leq i, \texttt{L[k]>x}.$$

En sortie de boucle `while`, la condition `j>0 and L[j-1]>x` est fausse, ainsi la boucle `for` possède l'invariant suivant :

$$\text{Inv}_i : \text{la liste } \texttt{L[0:i]} \text{ est triée.}$$

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`.

10.5 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.

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 10.1 – 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.

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

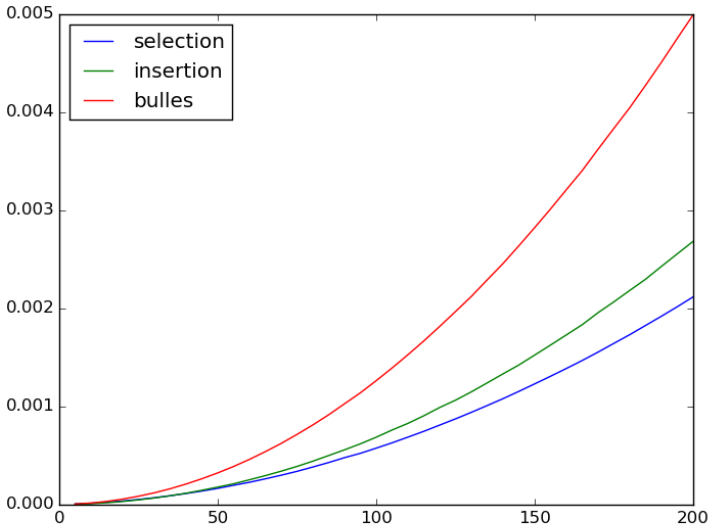


FIGURE 10.1 – Temps CPU pour trier une liste de taille n constituée d'entiers tirés aléatoirement entre 0 et 10000 (moyenne sur 1000 tests), avec $5 \leq n \leq 200$.

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 10.1 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.

10.6 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 $k = o(n \log n)$.

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 11

Structures de données linéaires : piles (et files)

Introduction

Ce chapitre décrit pour la première fois une structure de donnée abstraite : la pile. C'est une des structure les plus élémentaires mais aussi l'une des plus utiles. Donnons tout de suite une métaphore : de très lourdes assiettes sont empilées les unes sur les autres. Les opérations que l'on peut réaliser sont les suivantes :

— enlever une assiette de la pile (celle du haut), si bien sûr il reste des assiettes dans la pile ;

— ajouter une assiette en haut de la pile.

Et c'est tout ! On suppose qu'une assiette ailleurs qu'au sommet n'est pas accessible, à moins de *dépiler* toutes les assiettes situées au dessus, comme montré en figure 11.1.

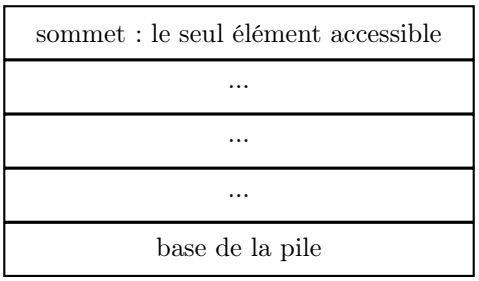


FIGURE 11.1 – Une pile

Outre les opérations ci-dessus, on voudrait aussi savoir si notre pile est vide, créer éventuellement une nouvelle pile, et savoir si notre pile est pleine (dans le cas où elle a une capacité finie), ce genre de choses. Donnons tout de suite quelques exemples d'application des piles :

— lorsqu'en programmation, on appelle une fonction, cet appel (avec ces paramètres) est empilé sur la pile d'appel. Peu importe qui se situe ailleurs qu'au sommet dans la pile d'appel : seul le dernier appel est traité et doit être résolu avant de revenir à la fonction précédente. On verra qu'en cas d'appels récursifs, ces appels sont *empilés* successivement sur la pile d'appel, jusqu'à arriver à un cas terminal puis sont dépilés ensuite.

— dans votre navigateur internet, vous avez deux boutons « page suivante » et « page précédente ». Chacun utilise une pile !

— si vous avez une vieille calculatrice qui utilise la notation « polonaise inverse » (par exemple, l'opération $(3+4)\times(5-7)$ se note $3\ 4+5\ 7-\times$, les parenthèses sont inutiles !), celle-ci est basée sur une pile. On détaillera cet exemple en fin de chapitre.

11.1 Les Piles : une classe abstraite

Commençons par donner la définition d'une classe abstraite.

Définition 11.1. *Une classe abstraite est un type, muni d'opérations.*

Cette définition est un peu vague, mais elle a le mérite de fixer les idées : pour définir une pile, on a essentiellement besoin de définir les opérations que l'on veut effectuer. L'implémentation effective d'une pile est indépendante de sa définition en tant que classe abstraite.

Définition 11.2. *Le type pile est une structure de données abstraite, munie des opérations suivantes :*

- `creer_pile(c)` *: construire une pile vide, de capacité* `c` *: la pile peut contenir* `c` *éléments.*
- `est_vide(P)` *: renvoie un booléen suivant si la pile* `P` *est vide ou non.*
- `sommet(P)` *: renvoie le sommet de la pile* `P` *si celle-ci est non vide.*
- `depiler(P)` *: enlève l'élément au sommet de la pile* `P` *et le renvoie. Si la pile est vide, renvoie une erreur.*
- `est_pleine(P)` *: suivant la réalisation de la pile, renvoie un booléen suivant si la pile* `P` *est pleine ou non. Si elle est pleine, on ne peut plus ajouter d'éléments.*
- `empiler(P,x)` *: ajoute l'élément* `x` *à la pile* `P` *si la pile n'est pas pleine, sinon, renvoie une erreur.*

Maintenant que l'on a défini la classe abstraite pile, on peut déja décrire des algorithmes qui vont utiliser la structure, avant même d'avoir proposé une réalisation concrète de la classe.

Donnons tout de suite un exemple très classique mais aussi très instructif : déterminer si un mot est bien parenthésé et indiquer (dans une liste, par exemple) les couples de positions des parenthèses ouvrantes et fermantes.

Définition 11.3. *Un mot parenthésé est une chaîne de caractères constituée uniquement des caractères* `'('` *et* `')'`*. Le mot est bien parenthésé si il contient autant de parenthèses ouvrantes que fermantes et si tout* préfixe *du mot contient au moins autant de parenthèses ouvrantes que fermantes.*

Voici quelques exemples :

- Le mot vide `''` est bien parenthésé.
- Le mot `'(()'` n'est pas bien parenthésé car il possède plus de parenthèses ouvrantes que fermantes.
- Le mot `'(())()'` est bien parenthésé : ses préfixes sont au nombre de 7 :

  `''` `'('` `'(('` `'(()'` `'(())'` `'(())('` `'(())()'`

  et contiennent chacun plus de parenthèses ouvrantes que fermantes, avec égalité éventuelle.
- Le mot `'(()()))(()'` contient autant de parenthèses ouvrantes que fermantes, mais n'est pas bien parenthésé : il contient en particulier le préfixe `'(()()))'` qui possède 4 parenthèses fermantes mais seulement 3 ouvrantes.

Proposition 11.4. *Soit m un mot bien parenthésé. Alors soit m est le mot vide, soit il se décompose de façon unique comme $m = (u)v$ où u et v sont deux mots bien parenthésés, éventuellement vides.*

Démonstration. Supposons m non vide. On peut considérer le plus petit préfixe non vide p de m ayant autant de parenthèses ouvrantes que fermantes. Celui-ci existe car l'ensemble des préfixes ayant cette propriété est non vide : m convient. Alors p commence par une parenthèse ouvrante (sinon m n'est pas bien parenthésé), et termine par une parenthèse fermante (même chose), et p s'écrit $p = (u)$ et $m = pv = (u)v$, avec u et v deux mots. Par minimalité de p, u est un mot bien parenthésé, et puisque m est bien parenthésé, v aussi. D'où l'existence de la décomposition. Montrons maintenant l'unicité : si $m = (u')v'$ avec u' et v' deux mots bien parenthésés, alors u' a pour préfixe u (par minimalité de u), mais ne peut contenir le caractère suivant (une parenthèse fermante) sinon m aurait un préfixe contenant plus de parenthèses fermantes qu'ouvrantes. Donc $u = u'$, et $v = v'$. □

Sur l'exemple du mot vide `''`, notre algorithme devrait renvoyer une liste vide car il n'y a pas de parenthèses. Sur le mot `'(())()'`, notre algorithme devrait renvoyer la liste d'indices `[(1,2),(0,3),(4,5)]` (on verra que l'ordre croissant suivant la deuxième composante est naturel).

L'idée de l'algorithme est assez simple : on crée une pile (initialement vide), et on parcourt la chaîne de caractères passée en entrée de gauche à droite. Lorsqu'on examine une parenthèse ouvrante, on empile la position (l'indice dans la chaîne) de cette parenthèse. Lorsqu'on examine une parenthèse fermante, deux cas peuvent se produire :

- la pile est vide : alors on a trouvé un préfixe qui contient plus de parenthèses fermantes qu'ouvrantes, ce qui signifie que le mot n'est pas bien parenthésé.
- la pile est non vide : on dépile alors l'élément en haut de la pile (qui est l'indice d'une parenthèse ouvrante), et on ajoute le couple (indice ouvrant, indice fermant) à la liste des couples en construction.

À la fin du traitement de la chaîne, on vérifie que la pile est vide. Si ce n'est pas le cas, le mot possède plus de parenthèses ouvrantes que fermantes et n'est donc pas bien parenthésé.

Le code Python correspondant est donc le suivant :

Traitement d'une expression parenthésée

```
def parentheses(s):
    """ Prend une chaîne de caractères et détermine si l'expression est bien parenthésée.

    s: chaîne composée de '(' et ')'
    Renvoie une liste donnant les couples de positions des parenthèses ouvrantes fermantes qui
    correspondent, ou affiche une erreur si le mot n'est pas bien parenthésé."""

    P=creer_pile(len(s)) #on pourrait optimiser car théoriquement, len(s)/2 éléments suffisent.
    L=[]
    for i in range(len(s)):
        c=s[i]
        assert c=='(' or c==')', "Le mot n'est pas uniquement constitué de parenthèses."
        if c=='(':
            empiler(P,i)
        else:
            if est_vide(P):
                print("Le mot n'est pas bien parenthésé.")
                return
            else:
                x=depiler(P)
                L.append((x,i))
    if not est_vide(P):
        print("Le mot n'est pas bien parenthésé.")
    else:
        return L
```

Donnons tout de suite quelques exemples d'exécution de la fonction

Exemple d'exécution

```
>>> parentheses('(())()')
[(1, 2), (0, 3), (4, 5)]
>>> parentheses('')
[]
>>> parentheses('(()')
Le mot n'est pas bien parenthésé
>>> parentheses('(()()))(()')
Le mot n'est pas bien parenthésé
```

Évidemment, il a bien fallu implémenter le type Pile pour faire fonctionner cet exemple. C'est le but de la section suivante.

11.2 Implémentation d'une pile de capacité finie

Pour une classe donnée, il existe plusieurs implémentations possibles : ce qui est important n'est pas réellement l'implémentation mais que les actions possibles sur les objets de la classe puisse s'effectuer. On donne ici une implémentation possible à l'aide de listes. Lorsque la pile est créée, on lui adjoint un nombre fixé qui est le nombre maximal d'éléments qu'elle peut contenir, sa *capacité*.

On choisit l'implémentation suivante :

— À la création d'une pile de capacité `c` on crée un liste de taille `c+1`. Le premir élément marquera toujours le nombre d'éléments effectivement présents dans la pile, et est donc positionné à 0 initialement. Les autres éléments de la liste peuvent être quelconque, par exemple `None` au début.

— Pour tester si une pile est vide, on vérifie si le premier élément vaut 0 ou non.

— Pour tester si une pile est pleine, on vérifie si `P[0]` est égal à `len(P)-1`, qui est la capacité de la pile.

— Pour dépiler `P`, il suffit de renvoyer l'élément d'indice `P[0]`, sans oublier de décrémenter `P[0]` (avant).

— Pour ajouter un élément `x` à la pile `P`, on vérifie d'abord que `P[0]` n'est pas égal à la capacité de la pile (égale à `len(P)-1`), sinon la pile est pleine. Si c'est bien le cas, on incrémente `P[0]` et on met `x` dans `P` à l'indice `P[0]`.

Le code Python correspondant est donné ci-dessous.

Implémentation du type Pile (piles bornées)

```
def creer_pile(c):
    P=[None]*(c+1)
    P[0]=0
    return P

def est_vide(P):
    return P[0]==0

def est_pleine(P):
    return P[0]==len(P)-1

def sommet(P):
    assert not est_vide(P), "la pile est vide"
    return P[P[0]]

def depiler(P):
    assert not est_vide(P), "la pile est vide"
    x=P[P[0]]
    P[0]-=1
    return x

def empiler(P,x):
    assert not est_pleine(P), "la pile est pleine"
    P[0]+=1
    P[P[0]]=x
```

Chaque opération se fait en complexité constante ($O(1)$), à part la création de pile qui a une complexité proportionnelle à la taille de la pile ($O(c)$).

11.3 Implémentation d'une pile de capacité infinie

Une autre implémentation possible d'une pile, là aussi à l'aide de listes, utilise la méthode `append` qui permet de rajouter un élément à une liste, et `pop` qui permet d'en retirer. Elles s'utilisent ainsi :

— Avec `L` une liste et `x` un élément, `L.append(x)` ajoute `x` à la fin de la liste `L`.

— Avec `L` une liste de longueur n et `i` un indice entre 0 et $n-1$, `L.pop(i)` supprime l'élément d'indice `i` de la liste et le renvoie, les éléments d'indice supérieur étant décalés d'un cran vers la gauche. Si `i` n'est pas spécifié, c'est le dernier élément qui est considéré, et c'est tout ce qui nous servira pour implémenter une pile.

L'implémentation à l'aide de ces opérations est la suivante. La fonction `creer_pile` ne prend plus de paramètre. Avec cette implémentation, la pile n'est jamais pleine. Bien sûr, la pile n'est pas vraiment de capacité infinie car on est limité par la mémoire.

— Pour créer une pile, on crée simplement une liste vide `[]`.

— Pour tester si une pile `P` est vide, on vérifie simplement si `P` est égale à `[]`.

— Pour dépiler `P`, il suffit d'utiliser `pop`.

— Pour ajouter un élément `x` à la pile `P`, on utilise `append`.

Implémentation du type Pile (piles non bornées)

```
def creer_pile():
    return []

def est_vide(P):
    return P==[]

def est_pleine(P):
    return False

def sommet(P):
    assert not est_vide(P), "la pile est vide"
    return P[-1]

def depiler(P):
    assert not est_vide(P), "la pile est vide"
    return P.pop()

def est_pleine(P):
```

```
    return False

def empiler(P,x):
    P.append(x)
```

La formulation `return P.pop()` pour dépiler une pile peut-être intriguante, mais ce qu'il se passe est ceci : l'instruction `P.pop()` est évaluée, elle a pour effet d'enlever un élément à `P` et de le renvoyer. Le résultat de l'expression `P.pop()` et donc cet élément, qu'on renvoie à l'aide de `return`. Si ça vous gène, vous pouvez procéder en deux étapes : `x=P.pop()` puis `return x`.

La complexité des trois premières opérations est constante, celle des deux dernières est un peu plus dure à déterminer car elle demande de comprendre précisément comment marchent les méthodes `append` et `pop` de Python.

Une liste en mémoire est stockée dans un espace mémoire fixé. Lorsqu'on veut ajouter un élément `x` à une liste, il se peut que cet espace ne soit pas suffisant. Dans ce cas, Python créé une nouvelle liste de taille supérieure, recopie les éléments dans cette nouvelle liste et rajoute `x` à la fin. Donc, la complexité d'ajouter un élément à la fin d'une liste est dans le pire cas proportionnelle à la taille de la liste. Ceci dit, si la liste n'est pas pleine, la complexité est constante. Python fait les choses bien, lorsqu'une liste de taille n est pleine, il crée une nouvelle liste de taille 2 fois supérieure [1]. On pourra donc réaliser ensuite n ajouts en temps constant. Ainsi, en moyenne, un ajout n'a coûté qu'un temps constant. De même, pour garder un espace en mémoire proportionnel au nombre d'éléments de la liste, lorsque la proportion d'éléments effectivement stockés en mémoire devient inférieur à un quart [2] de la capacité de la liste en mémoire, Python la réduit à un demi en libérant la moitié de l'espace inutilisé.

En résumé, on peut montrer que n'importe quelle suite de n opérations `L.append()` et `L.pop()` à partir d'une liste vide se fait en temps linéaire en n, et la liste n'occupe qu'une place en mémoire linéaire en le nombre d'éléments stockés. On parle de complexité *amortie* constante. Les opérations `ajouter` et `depiler` ont donc une complexité amortie constante.

La complexité amortie n'est pas au programme. On considère donc que toutes les opérations sur les piles de capacité infinie se font en temps constant.

11.4 Application à l'évaluation d'une expression postfixe

Prenons maintenant un exemple un peu plus complexe que le traitement des expressions bien parenthésées. On considère une expression arithmétique, qui pour simplifier ne sera composées que de nombres positifs et des opérateurs $+$, $-$, $\times$ et $/$.

Une telle expression peut se représenter sous la forme d'un arbre binaire, les feuilles sont les opérandes (les nombres), et les noeuds internes sont les opérateurs. Les parenthèses dans une expression indiquent quels opérations sont à effectuer en premier.

Par exemple, $(3+4)\times(5-(2\times 6))$ se représente ainsi comme en figure 11.2.

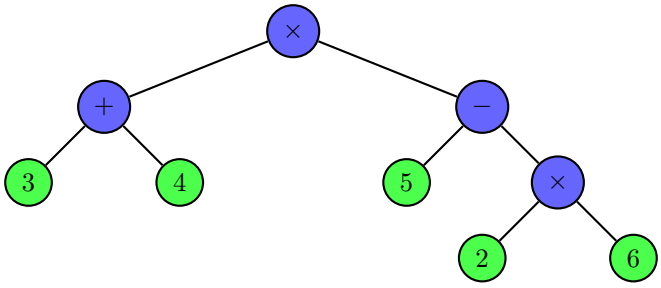


FIGURE 11.2 – L'arbre associé à l'expression arithmétique $(3+4)\times(5-(2\times 6))$.

Différentes énumérations d'un tel arbre sont possibles. Celle que l'on utilise habituellement en mathématiques est l'énumération infixe. Pour énumérer un arbre, on procède de manière récursive comme suit :

— si l'arbre est une feuille, on énumère l'étiquette et il n'y a plus rien à faire.

— sinon, on énumère d'abord le sous-arbre gauche, puis l'étiquette de la racine, et on énumère le sous-arbre droit.

1. J'ai un petit doute sur le 4, c'est marqué quelque part dans la doc. Ça n'a pas vraiment d'importance, ce qui compte c'est le principe.
2. Là aussi, le 1/4 est à vérifier...

L'énumération infixe de l'expression représentée en figure 11.2 est donc : $3 + 4 \times 5 - 2 \times 6$. Le problème de cette énumération est que les parenthèses sont nécessaires pour savoir les opérations à effectuer en premier (on en enlève certaines en convenant que $\times$ et / sont prioritaires sur $+$ et $-$ et que les opérations de même priorité s'effectuent de gauche à droite, mais ce n'est qu'une convention !). En effet, l'arbre suivant a la même énumération infixe :

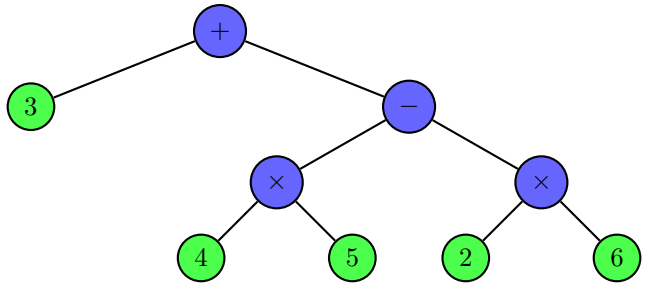


FIGURE 11.3 – Un autre arbre dont l'énumération infixe est $3 + 4 \times 5 - 2 \times 6$.

D'autres énumérations que l'énumération infixe sont possibles :

— l'énumération préfixe : racine, sous-arbre gauche, sous-arbre droit. Pour l'arbre de la figure 11.2, l'énumération est donc $\times + 3\ \ 4 - 5 \times\ \ 2\ \ 6$ et celle de la figure 11.3 est $+\ 3 - \times\ 4\ \ 5 \times 2\ \ 6$

— l'énumération postfixe : sous-arbre gauche, sous-arbre droit, racine. Pour l'arbre de la figure 11.2, on obtient $3\ \ 4\ +\ 5\ \ 2\ \ 6\ \times - \times$ et pour celui de la figure 11.3 : $3\ \ 4\ \ 5\ \times\ 2\ \ 6\ \times\ -\ +$.

L'intérêt de ces deux énumérations par rapport à l'énumération infixe est qu'il n'y a pas ambiguïté [3] et que les parenthèses sont inutiles. Les calculatrices en notation polonaise inversée utilisent [4] l'énumération postfixe et l'évaluation de l'expression se fait avec une pile, en examinant un par un les éléments de l'expression :

— si c'est un opérande, on l'empile sur la pile.

— si c'est un opérateur, on dépile les deux éléments au sommet de la pile, on réalise l'opération et on empile le résultat.

On part bien entendu d'une pile vide, et si l'expression est correcte à la fin du traitement il n'y a qu'un seul élément dans la pile : le résultat de l'évaluation. Le code Python est donné ci-dessous. On suppose que le paramètre passé en entrée de la fonction est une expression postfixe correcte, donnée sous la forme d'une liste. Pour simplifier, on se limite à des nombres positifs, et les opérateurs sont restreints aux 4 opérations habituelles. On les suppose donnés sous forme de caractères.

Évaluation d'une évaluation postfixe

```
def evaluation_postfixe(E):
    P=creer_pile(len(E))
    for x in E:
        if not type(x)==str:
            empiler(P,x)
        else:
            b=depiler(P)
            a=depiler(P)
            if x=='+':
                r=a+b
            elif x=='-':
                r=a-b
            elif x=='*':
                r=a*b
            else:
                r=a/b
            empiler(P,r)
    return P[0]
```

Sur les exemples ci-dessus, on obtient :

```
>>> evaluation_postfixe([3, 4, '+', 5, 2, 6, '*', '-', '*'])
-49
>>> evaluation_postfixe([3, 4, 5, '*', 2, 6, '*', '-', '+'])
11
```

3. On le montre par récurrence.
4. utilisaient... Il n'y en a plus tellement !

ce qui correspond bien aux expressions. Il est bien sûr possible d'améliorer grandement la fonction d'evaluation ci-dessus, pour tester si l'expression est bien une expression postfixe correcte, intégrer les nombres négatifs, les opérateurs unaires (carré, fonctions usuelles)... Mais tout cela se fait suivant la même idée, à l'aide d'une pile.

11.5 Introduction à la programmation orientée objet

La programmation orientée objet (POO) permet de créer des entités (*objets*) que l'on peut manipuler. En bref, elle permet de réaliser une classe abstraite. Les listes Python sont par exemple des objets, de la classe « list ». En effet :

```
>>> L=[0,1,2]
>>> type(L)
<class 'list'>
```

Une classe regroupe les fonctions et les attributs définissant un objet. Les fonctions associées à une classe sont appelées des « méthodes » : on a déja parlé des méthodes `pop` et `append` sur les listes, pour ne citer qu'elles. On va voir maintenant comment créer nous-mêmes des classes et définir des méthodes sur les objets de la classe.

11.5.1 Bases de la POO

Un exemple vaut mieux qu'un long discours :

```
class personne:
    def __init__(self,n,p):
        self.nom=n
        self.prenom=p
        self.mail=None

    def fixe_mail(self,s):
        self.mail=s

    def nom_personne(self):
        return self.nom

    def prenom_personne(self):
        return self.prenom
```

On définit ici une classe `personne`, avec 4 méthodes. Une instance de la classe `personne` est munie de 4 attributs : `nom`, `prenom` et `mail`. Si `a` est un tel objet, on accède à ses attributs à l'aide de `a.nom`, `a.prenom` et `a.mail`. Les deux méthodes `nom_personne` et `prenom_personne` permette d'accéder à deux de ces attributs, la méthode `fixe_mail` permet de modifier l'attribut `mail`. La méthode `__init__` est tout à fait spéciale : elle permet la création d'un objet associé à la classe `personne`. Toutes les méthodes ont en paramètre une variable `self` : il s'agit de l'objet sur lequel on travaille. Si `a` est un objet de la classe `personne` et `s` une chaîne de caractère, `personne.fixe_mail(a,s)` remplace la valeur de l'attribut `mail` de `a` par `s`. Mais tout comme les méthodes sur les listes, on emploiera plutôt la syntaxe `a.fixe_mail(s)` qui est équivalente.

Revenons à `__init__`, qui s'utilise ainsi pour créer (et renvoyer) un nouvel objet de la classe : `personne(n,p)` avec `n` et `p` contenant *a priori* deux chaînes de caractères. Petit exemple :

```
>>> a=personne("Dupont","Martin")
>>> a.prenom
'Martin'
>>> a.fixe_mail("truc@bidule.machin")
>>> a.mail
'truc@bidule.machin'
>>> a.nom_personne()
'Dupont'
```

11.5.2 Implémentation d'une pile de capacité finie avec une liste

En reprenant l'implémentation vue plus haut, sous forme de classe, on obtient la classe `pile_bornee` suivante :

```
class Pile_bornee:
    def __init__(self,n):
        self.nb=0
```

```
        self.capacite=n
        self.contenu=[None]*n

    def est_vide_pile(self):
        return self.nb==0

    def est_pleine(self):
        return self.nb==self.capacite

    def empiler(self,x):
        assert not self.est_pleine(), "la pile est pleine"
        self.contenu[self.nb]=x
        self.nb+=1

    def sommet(self):
        assert not self.est_vide_pile(), "la pile est vide"
        return self.contenu[self.nb-1]

    def depiler(self):
        assert not self.est_vide_pile(), "la pile est vide"
        self.nb-=1
        return self.contenu[self.nb]
```

11.5.3 Implémentation d'une pile non bornée avec une liste Python (vue comme une liste chaînée)

De même pour les piles de capacité infinie, à l'aide des listes Python :

```
class Pile_liste:
    def __init__(self):
        self.contenu=[]

    def est_vide_pile(self):
        return self.contenu==[]

    def empiler(self,x):
        self.contenu.append(x)

    def sommet(self):
        assert not self.est_vide_pile(), "la pile est vide"
        return self.contenu[-1]

    def depiler(self):
        assert not self.est_vide_pile(), "la pile est vide"
        return self.contenu.pop()
```

11.5.4 Implémentation personnalisée d'une pile non bornée (liste chaînée)

Enfin, on donne une nouvelle implémentation qui exploite vraiment la structure de classe. Un objet de la classe `Pile_perso` définie ici peut être vu comme une *liste chaînée*. De manière abstraite, une liste chaînée est définie récursivement :

— la liste vide est une liste chaînée ;

— si elle n'est pas vide, une liste chaînée est constituée d'un élément, suivi d'une liste chaînée.

Par exemple, voici une liste constituée des éléments 12, 99, 37 et 8. La liste est constituée de l'élément 12, puis de la liste constituée des éléments 99, 37 et 8, etc...

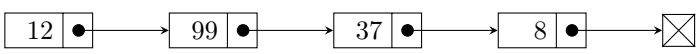


FIGURE 11.4 – La représentation interne d'une liste chaînée

Pour implémenter une classe associée à cette structure, on va d'abord définir une classe `Cellule`, comportant une seule méthode (de création). Une instance de cette classe comporte deux attributs, `element` (l'élément qu'elle contient) et `suiv` (une cellule vers laquelle elle « pointe »). À la création une cellule ne « pointe » sur rien. Une pile (liste chaînée), instance de la classe `Pile_perso` suivante, ne comporte qu'un seul attribut : `tete`. En suivant la définition abstraite,

soit `tete` vaut `None` (ceci est associé à une pile vide), soit elle pointe vers une cellule (le sommet de la pile). Les méthodes de la classes (les opérations de pile) sont naturelles et vont modifier les champs `suiv` des cellules.

```
class Cellule:
    def __init__(self,x):
        self.element=x
        self.suiv=None

class Pile_perso:
    def __init__(self):
        self.tete=None

    def est_vide_pile(self):
        return self.tete==None

    def empiler(self,x):
        c=Cellule(x)
        c.suiv=self.tete
        self.tete=c

    def sommet(self):
        assert not self.est_vide_pile(), "la pile est vide"
        return self.tete.element

    def depiler(self):
        assert not self.est_vide_pile(), "la pile est vide"
        c=self.tete
        self.tete=c.suiv
        return c.element
```

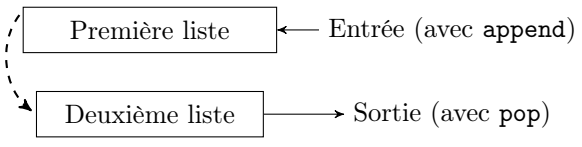
Un exemple d'utilisation

```
>>> P=Pile_perso()
>>> P.empiler(4)
>>> P.empiler(0)
>>> P.depiler()
0
>>> P.empiler(1)
>>> P.tete
<__main__.Cellule object at 0x7f2c585a1710>
>>> P.tete.element
1
>>> P.sommet()
1
```

Dans cette dernière implémentation, toutes les opérations ont une complexité $O(1)$ (pas seulement amortie).

11.6 Structure de file

Dans cette section, on aborde une autre classe abstraite que celle de pile. La pile fonctionne sur le principe LIFO (last-in, first out), la file fonctionne suivant le principe FIFO (first-in, first-out). Il y a plusieurs réalisations possibles, on en donne une qui utilise deux listes. L'ajout d'un élément à la file se fait uniquement avec `append` dans la première liste, la suppression se fait avec `pop` dans la deuxième liste.



Lorsque la deuxième liste est vide et qu'on veut défiler (sortir un élément de la file), il est nécessaire de transférer les éléments de la première liste dans la deuxième. Pour conserver l'ordre dans la file, il est nécessaire d'inverser l'ordre des éléments lors du transfert. Voici les opérations de file à écrire :

— `creer_file()` : construire une file vide.

— `est_vide(F)` : renvoie un booléen suivant si la file `F` est vide ou non.

— `defiler(F)` : défile `F` en sortant l'élément en tête de file, et le renvoie. Si la file est vide, renvoie une erreur.

— `enfiler(F,x)` : ajoute l'élément `x` à la queue de la file `F`.

Et voici une implémentation :

```
def creer_file():
    return [],[] #couple de listes

def est_vide_file(F):
    return F[0]==[] and F[1]==[]

def enfiler(F,x):
    F[0].append(x)

def defiler(F):
    assert not est_vide_file(F), "file vide"
    if F[1]==[]:
        F[1][:]=F[0][::-1] # contenu de la deuxième liste remplacé par celui de la première, à l'envers.
        F[0][:]=[] # première liste vidée.
    return F[1].pop()
```

Bien sûr, une classe serait tout indiquée. En terme de complexité, on peut montrer que toutes les opérations se font en temps constant (amorti).

Chapitre 12

Récursivité

12.1 Principes de la récursivité

12.1.1 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`.

12.1.2 La pile d'exécution

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 contenu dans la pile d'appel, l'exécution de f peut reprendre juste après l'endroit où g a été appelée.

Une fonction récursive est essentiellement une fonction qui s'appelle elle-même, ainsi les appels successifs à f s'empile dans la pile d'appels (voir figure 12.1).

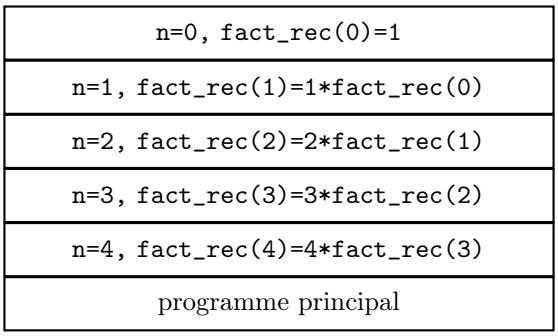


FIGURE 12.1 – La pile d'exécution lors de l'appel de `fact_rec(4)`.

Une fois arrivé à un cas terminal (ne produisant pas d'appel récursif), le nombre d'éléments de la pile d'appels se réduit. Dans le cas de la fonction factorielle, comme celle-ci ne rappelle qu'une fois, dés lors qu'on a commencé à dépiler on ne s'arrête plus (voir la figure 12.2)

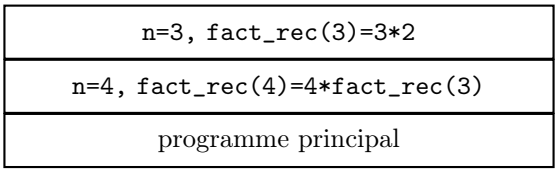


FIGURE 12.2 – La pile d'exécution lors de l'appel de `fact_rec(4)` : on a dépilé les appels relatifs à $n = 0, n = 1$ et $n = 2$.

Enfin, la récursion s'arrête lorsqu'on dépile l'élément correspondant au premier appel de la fonction (figure 12.3).

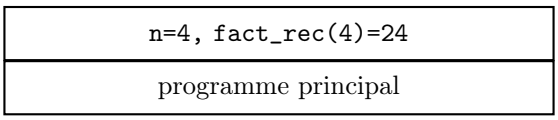


FIGURE 12.3 – La pile d'exécution lors de l'appel de `fact_rec(4)` : juste avant dépilage de l'appel initial.

On voit que ce 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, on en reparle dans la sous-section qui suit.

12.1.3 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 vu en première année 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
```

12.1.4 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 :

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

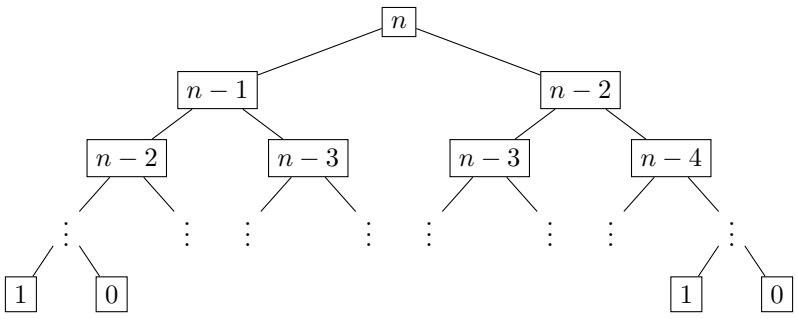


FIGURE 12.4 – Un arbre d'appels récursifs

Le problème de l'algorithme précédent est le nombre d'appels récursifs effectués. Notons A_n le nombre d'appels récursifs nécessaires pour le calcul de F_n. Alors, la suite (A_n) vérifie la relation de récurrence $A_0 = A_1 = 0$ et pour tout $n \geq 2$, $A_n = 2 + A_{n-1} + A_{n-2}$, soit $A_n + 2 = (A_{n-2} + 2) + (A_{n-1} + 2)$. Autrement dit, la suite $(A_n + 2)_{n\in\mathbb{N}}$ coïncide avec la suite $(2F_n)_{n\in\mathbb{N}}$. On peut donner une expression explicite de A_n, à savoir :

$$\text{Pour tout entier naturel } n, \qquad A_n = \frac{2}{\sqrt{5}}\left[\left(\frac{\sqrt{5}+1}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right] - 2$$

mais ce qui importe, c'est que $A_n \underset{n\to+\infty}{\sim} C \times \left(\frac{\sqrt{5}+1}{2}\right)^n$ où C est une constante strictement positive. Un appel récursif n'est jamais gratuit (et possède un coût minoré par une constante strictement positive), ainsi le calcul de F_n par cette

méthode est de complexité exponentielle, puisque $\frac{\sqrt{5}+1}{2} > 1$. Il vaut mieux utiliser une méthode itérative dans ce cas, comme la suivante [1] :

```
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 [2] en n. En conclusion, il faut faire attention à ne pas faire des appels récursifs qui se recoupent, sous peine de voir la complexité exploser !

Taille de la pile (Python). Supposons que l'on souhaite calculer $n!$, pour n relativement grand [3], 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 [4] mais elle est là pour éviter un dépassement de capacité de la pile d'appels : le fameux « stack overflow » [5].

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.

12.1.5 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 12.5.

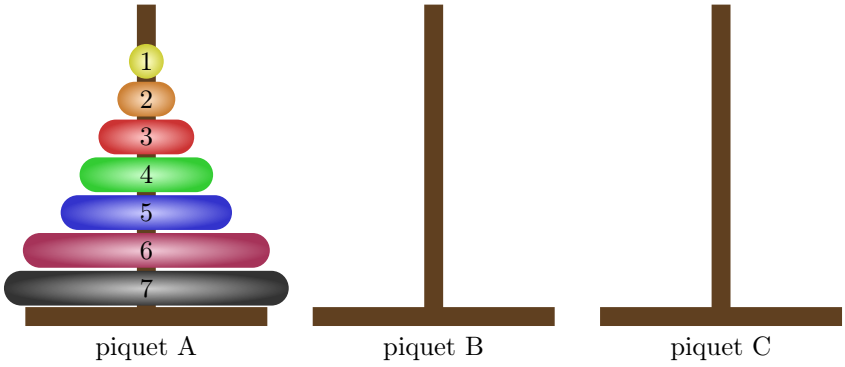


FIGURE 12.5 – Le jeu de Hanoï : comment déplacer les 7 disques du piquet A au piquet C, en suivant les règles ?

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 ;

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

La figure 12.6 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).

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 :

1. on peut cependant s'en sortir avec une version récursive ayant un coût linéaire.
2. 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 !
3. Rappelons à toute fin utile que les entiers ne sont pas bornés en Python.
4. À l'aide de la fonction `setrecursionlimit` du module `sys`.
5. Sur mon ordinateur personnel, le calcul de $n!$ avec la méthode récursive ne passe plus à partir de 20000 environ.

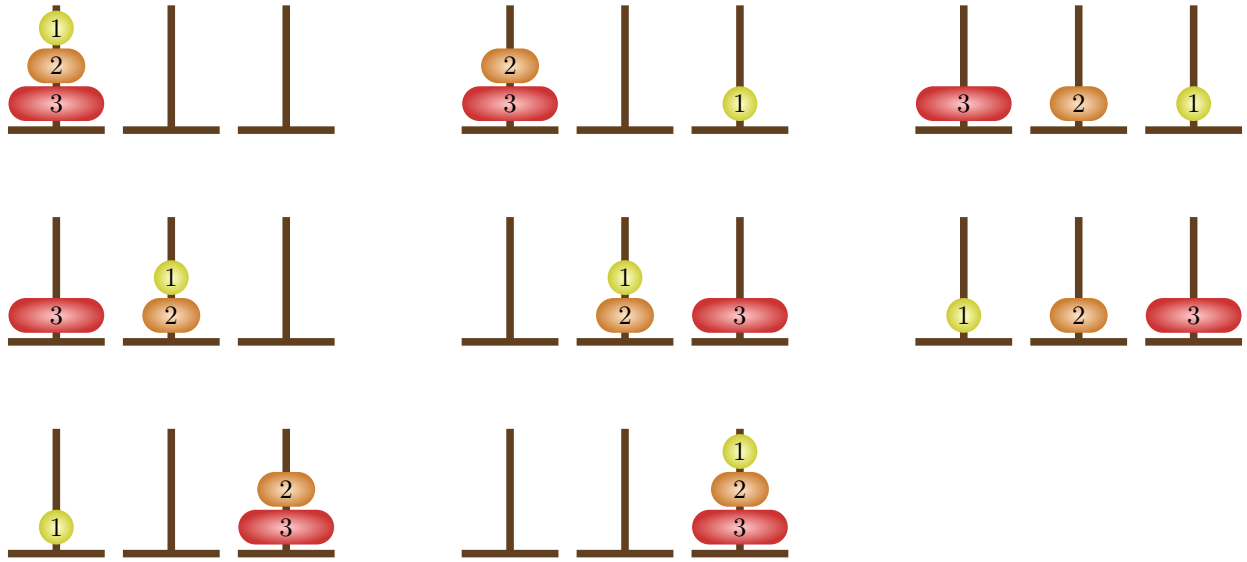


FIGURE 12.6 – Résolution du jeu de Hanoi pour $n = 3$

— 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 12.6. Il est facile de montrer par récurrence que le nombre de mouvements produits est $2^n - 1$: c'est optimal [6].

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.

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

12.2 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 (temporelles et spatiales).

12.2.1 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 ;

— 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.

Il n'est pas toujours évident d'exhiber une quantité qui décroît dans une fonction récursive. Le faire pour la fonction de Syracuse suivante permettrait de résoudre un problème ouvert.

```
def Syracuse(n):
    assert n>=1
    print(n)
    if n==1:
        return(1)
    elif n%2==0:
        return Syracuse(n//2)
    else:
        return Syracuse(3*n+1)
```

Une remarque pour terminer : la quantité qui décroît n'est pas nécessairement à chercher dans $\mathbb{N}$, un ensemble ordonné n'ayant pas de suite infinie strictement décroissante suffit. C'est par exemple le cas de $\mathbb{N}^2$ ordonné par l'ordre $\preceq$ (dit lexicographique) défini comme suit :

$$(x_0, y_0) \prec (x_1, y_1) \qquad \Longleftrightarrow \qquad x_0 < x_1 \quad \text{ou} \quad x_0 = x_1 \ \text{ et } \ y_0 < y_1$$

12.2.2 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.

12.2.3 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 [7].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).

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.

7. 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.

Récurrences « diviser pour régner ». Un cas particulier que l'on retrouve souvent dans l'examen de la complexité des fonctions récursives est celui des récurrences de la forme [8] :

$$C(n) = a \cdot C(\lfloor n/2 \rfloor) + b \cdot C(\lceil n/2 \rceil) + O(n^\alpha)$$

avec a et b deux entiers positifs non tous deux nuls, et $\alpha \geq 0$. Indiquons comment traiter ces récurrences, dans le cas où n est une puissance de 2 : en notant $\gamma = a + b$, et $n = 2^k$, on a donc $C(2^k) = \gamma C(2^{k-1}) + O(2^{\alpha k})$. Ainsi :

$$\frac{C(2^k)}{\gamma^k} = \frac{C(2^{k-1})}{\gamma^{k-1}} + O\left((\frac{2^\alpha}{\gamma})^k\right)$$

Comme $\frac{C(2^k)}{\gamma^k} = \sum_{i=1}^{k} \left[\frac{C(2^i)}{\gamma^i} - \frac{C(2^{i-1})}{\gamma^{i-1}}\right] + O(1)$, on est ramené à la sommation des $(\frac{2^\alpha}{\gamma})^i$. Il y a donc 3 cas :

- si $2^\alpha < \gamma$, $\sum_{i=1}^{k}(\frac{2^\alpha}{\gamma})^i = O(1)$ et donc $C(2^k) = O(\gamma^k)$.
- si $2^\alpha > \gamma$, $\sum_{i=1}^{k}(\frac{2^\alpha}{\gamma})^i = O((\frac{2^\alpha}{\gamma})^k)$ et donc $C(2^k) = O(2^{k\alpha}) = O(n^\alpha)$.
- si $2^\alpha = \gamma$, $\sum_{i=1}^{k}(\frac{2^\alpha}{\gamma})^i = O(k)$ et donc $C(2^k) = O(k\gamma^k)$.

Revenons à l'entier n : comme $2^k = n$, on a $k = \log_2(n)$ et $\gamma^k = 2^{\log_2(\gamma^k)}$ car $\log_2$ est la réciproque de $x \mapsto 2^x$. Or $2^{\log_2(\gamma^k)} = 2^{k\log_2(\gamma)} = n^{\log_2(\gamma)}$. En admettant que ce résultat est encore vrai si n n'est pas une puissance de 2, on a montré le théorème suivant :

Théorème 12.1 (Complexité des stratégies « diviser pour régner »)**.** *Supposons qu'il existe deux entiers naturels a et b non tous deux nuls et un réel $\alpha \geq 0$ tels que $C(n)$ vérifie :*

$$C(n) = a \cdot C(\lfloor n/2 \rfloor) + b \cdot C(\lceil n/2 \rceil) + O(n^\alpha)$$

Alors, en notant $\gamma = a + b$, trois cas peuvent se présenter :

— *Si $\gamma > 2^\alpha$, alors $C(n) = O(n^{\log_2(\gamma)})$.*

— *Si $\gamma < 2^\alpha$, alors $C(n) = O(n^\alpha)$.*

— *Si $\gamma = 2^\alpha$, alors $C(n) = O(n^\alpha \cdot \log(n))$.*

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 [9] $C(p) = C(\lfloor p/2 \rfloor) + O(1)$. Le théorème précédent montre directement que $C(p) = O(\log_2(p))$, on retrouve donc bien une complexité logarithmique pour la recherche dichotomique récursive. Remarquez que le code est très semblable à la version classique de recherche dichotomique avec une boucle `while`, ce qui n'est pas surprenant.

8. On rappelle que $\lfloor . \rfloor$ est la partie entière usuelle (inférieure) et $\lceil . \rceil$ la partie entière supérieure.

9. 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.

12.3 Le paradigme « diviser pour régner » : un exemple avec l'algorithme de Karatsuba

On donne un autre exemple que le problème des tours de Hanoï faisant intervenir plusieurs appels récursifs à chaque étape.

Le problème. Soit n un entier positif. Considérons $P = \sum_{k=0}^{n-1} p_k X^k$ et $Q = \sum_{k=0}^{n-1} q_k X^k$ deux polynômes de même taille n (la taille est égale au degré, plus 1). Les coefficients sont dans un anneau commutatif $\mathbb{A}$ quelconque, qu'on pourra considérer comme étant l'ensemble des entiers $\mathbb{Z}$: une opération dans $\mathbb{A}$ (multiplication, addition, soustraction) est considérée comme élémentaire. On représente les polynômes P et Q par les listes $[p_0, p_1, \dots, p_{n-1}]$ et $[q_0, q_1, \dots, q_{n-1}]$. Le produit $P \times Q$ est un polynôme de degré $2n-2$. Comment obtenir efficacement la liste de ses $2n-1$ coefficients ?

Méthode naïve. On utilise simplement la relation $PQ = \sum_{k=0}^{n-1} \sum_{j=0}^{n-1} p_k q_j X^{k+j}$. On en déduit un algorithme de complexité $O(n^2)$ répondant au problème (la transcription en Python est laissée en exercice) :

Algorithme 12.2 : Multiplication naïve

Entrées : Deux polynômes P et Q de même taille n.
Sortie : Le produit $P \times Q$.
$T \leftarrow$ une liste de $2n-1$ zéros;
pour k *entre 0 et* $n-1$ **faire**
  pour j *entre 0 et* $n-1$ **faire**
    $T[j+k] \leftarrow T[j+k] + p_k \times q_j$
retourner T

Paradigme « diviser pour régner ». On va introduire un paradigme de résolution de problèmes, qui va s'appliquer à celui de la multiplication de polynômes, le paradigme « diviser pour régner ». Il consiste à :

— découper le problème principal en sous-problèmes de tailles plus petites (division) ;

— calculer récursivement une solution à ces sous-problèmes ;

— recomposer la solution du problème principal à l'aide des solutions des sous-problèmes (règne).

L'idée de l'algorithme de Karatsuba [10] : $n = 2$. Considérons pour commencer le cas $n = 2$: les polynômes P et Q sont tous deux de degré 1, et on a $PQ = p_0q_0 + (p_0q_1 + p_1q_0)X + p_1q_1X^2$. On remarque qu'on peut facilement abaisser le nombre de multiplications lorsque avec l'idée suivante : on considère les produits

$$\begin{cases} t_0 &= p_0q_0 \\ t_1 &= p_1q_1 \\ t_2 &= (p_0+p_1)\times(q_0+q_1) \end{cases}$$

Alors le terme $p_0q_1 + p_1q_0$ se déduit des produits ci-dessus car $c_1 = t_2 - t_1 - t_0$. Ainsi, seulement 3 multiplications sont nécessaires lorsque $n = 2$, au lieu de 4 avec la méthode naïve. Certes, le nombre d'additions/soustractions est passé de 1 à 4, ce qui fait passer le nombre d'opérations élémentaires de 5 à 7. Mais la récursivité va nous permettre d'obtenir un gain substantiel avec cette idée.

L'algorithme de Karatsuba. Si $n = 1$ (les deux polynômes sont des constantes), le produit est simplement le produit des deux constantes. Sinon, posons $m = \lfloor \frac{n}{2} \rfloor$, et découpons nos polynômes en 2. On écrit donc $P_0 = \sum_{k=0}^{m-1} p_k X^k$ et $P_1 = \sum_{k=m}^{n-1} p_k X^{k-m}$ de sorte que $P = P_0 + X^m P_1$, et de même $Q = Q_0 + X^m Q_1$. Posons ensuite $T_0 = P_0Q_0$, $T_1 = P_1Q_1$ et $T_2 = (P_0 + P_1) \times (Q_0 + Q_1)$.

Comme $P \times Q = (P_0 + X^m P_1)(Q_0 + X^m Q_1) = P_0Q_0 + X^m(P_1Q_0 + P_0Q_0) + X^{2m}P_1Q_1$, on obtient le produit en combinant les facteurs (T_i) de la façon suivante : $P \times Q = T_0 + X^m(T_2 - T_1 - T_0) + X^{2m}T_1$. Les produits (T_i) sont eux-même calculés récursivement en exploitant cette idée. L'algorithme, en pseudo-code, est le suivant :

10. Anatoli Alekseïevitch Karatsuba, mathématicien russe (1937-2008).

Algorithme 12.3 : L'algorithme de Karatsuba

Entrées : Deux polynômes P et Q de même taille n.
Sortie : Le produit $P \times Q$.
si $n = 1$ **alors**
  retourner $([p_0 q_0])$
sinon
  $m = \lfloor n/2 \rfloor$;
  Décomposer $P = P_0 + X^m P_1, Q = Q_0 + X^m Q_1$;
  T_0 =Karatsuba(P_0, Q_0);
  T_1 =Karatsuba(P_1, Q_1);
  T_2 =Karatsuba$(P_0 + P_1, Q_0 + Q_1)$;
  retourner $T_0 + X^m(T_2 - T_1 - T_0) + X^{2m} T_1$

Étude de la complexité de l'algorithme de Karatsuba. Supposons pour simplifier que n est une puissance de 2, donc s'écrit 2^k. Dans chaque appel récursif, les instances ont des tailles divisées exactement par 2. Pour déterminer la complexité globale, il est essentiel d'estimer la complexité des deux étapes diviser et régner. Ici, on suppose qu'on ne compte que les opérations arithmétiques dans l'anneau $\mathbb{A}$ (additions, multiplications, soustractions).

— Pour diviser, il faut créer les tableaux associés aux polynômes P_0 et P_1, et calculer les polynômes $P_0 + P_1$ et $Q_0 + Q_1$. Ceci se fait en temps linéaire en n.

— Pour régner, il faut créer une liste T de taille $(2n - 1)$, et combiner les trois listes associées à T_0, T_1 et T_2 pour obtenir la liste T. Ceci se fait également en temps linéaire en n puisqu'il suffit de parcourir les listes T_0, T_1 et T_2 et de modifier un ou deux éléments de T pour chacun des éléments des trois listes.

Pour le calcul du produit, on fait trois appels récursifs pour résoudre des problèmes de taille divisée par 2. Ainsi, la complexité $C(n)$ de l'algorithme de Karatsuba satisfait à l'équation : $C(n) = 3 \times C(n/2) + O(n)$ dont la solution est $C(n) = O\left(n^{\log_2(3)}\right)$. On admet que cette solution est valable pour n quelconque. C'est un cas particulier du théorème sur la résolution des récurrences « diviser pour régner », on a en fait ici

$$C(n) = C(\lfloor n/2 \rfloor) + 2 \cdot C(\lceil n/2 \rceil) + O(n)$$

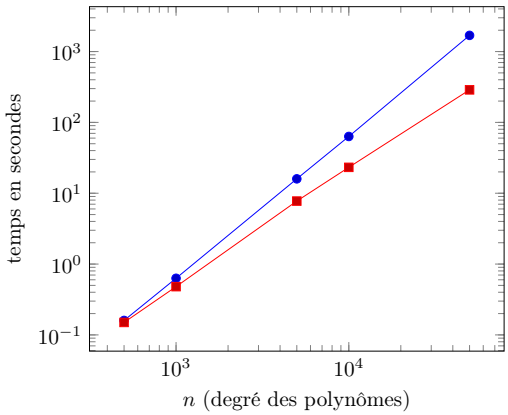
Code Python. Il faut faire attention, la taille de `P1` est la même que celle de `P0` si n est pair, un de plus si n est impair. De même pour `Q0` et `Q1`. Ainsi `T1` et `T2` ont la même taille, supérieure à celle de `T0` si n est impair.

```
def Karatsuba(P,Q):
    n=len(P)
    if n==1:
        return [P[0]*Q[0]]
    m=n//2
    P0=P[:m] ; P1=P[m:] ; Q0=Q[:m] ; Q1=Q[m:]
    T0=Karatsuba(P0,Q0) ; T1=Karatsuba(P1,Q1)
    for i in range(m):
        P1[i]+=P0[i] ; Q1[i]+=Q0[i]
    T2=Karatsuba(P1,Q1)
    T=[0]*(2*n-1)
    for i in range(len(T0)):
        T[i]+=T0[i] ; T[i+m]-=T0[i]
    for i in range(len(T1)):
        T[i+m]+=T2[i]-T1[i] ; T[i+2*m]+=T1[i]
    return T
```

Et en pratique ? Le tableau suivant montre les temps en secondes nécessaires en Python pour calculer sur mon ordinateur personnel (Pocket PC de 2012) le produit de deux polynômes P et Q avec de petits coefficients entiers (tirés aléatoirement dans l'intervalle $[\![-1000, 1000]\!]$), de degrés variables, avec un algorithme naïf et avec l'algorithme de Karatsuba.

$\deg(P) = \deg(Q) = n$	100	500	1000	5000	10000	50000
multiplication naïve	0.006	0.16	0.63	15.96	63.3	1692
Karatsuba	0.07	0.15	0.48	7.75	23.2	288

Pour mieux apercevoir les variations, on peut tracer le diagramme log-log de ces temps. Un complexité $C(n) = n^\alpha$ donne une droite car $\log(C(n)) = \alpha \cdot \log(n)$, ce qu'on observe sur le graphe suivant. Une régression linéaire fait apparaître des coefficients directeurs proches des valeurs théoriques 2 et $1.58 \simeq \log_2(3)$.



Pour de petits polynômes, l'algorithme de Karatsuba est sans intérêt, mais il devient assez vite intéressant : il est plus efficace que l'algorithme naïf pour des polynômes de degré au moins 500.

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 ».

Chapitre 13

Algorithmes de tri efficaces, Médiane

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 rappelle 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 $O(n\log n)$. On montre ensuite qu'on ne peut avoir mieux qu'une complexité $O(n\log n)$ pour le problème du tri, ce qui fait du tri fusion un tri optimal. Enfin, on s'intéresse au problème de déterminer la médiane d'une liste (ou plus généralement son k-ème plus petit élément). Avec une variante du tri rapide, on peut résoudre ce problème en temps $O(n)$ en moyenne.

13.1 Rappels sur la stratégie « Diviser pour régner »

On rappelle ici les principes de la stratégie « diviser pour régner » :

— Diviser le problème principal en sous-problèmes 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.

13.2 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. 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), mais on trouvera dans la feuille d'exercices une version n'utilisant qu'une liste auxiliaire de même taille que la liste initiale pour procéder aux fusions successives.

13.2.1 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.

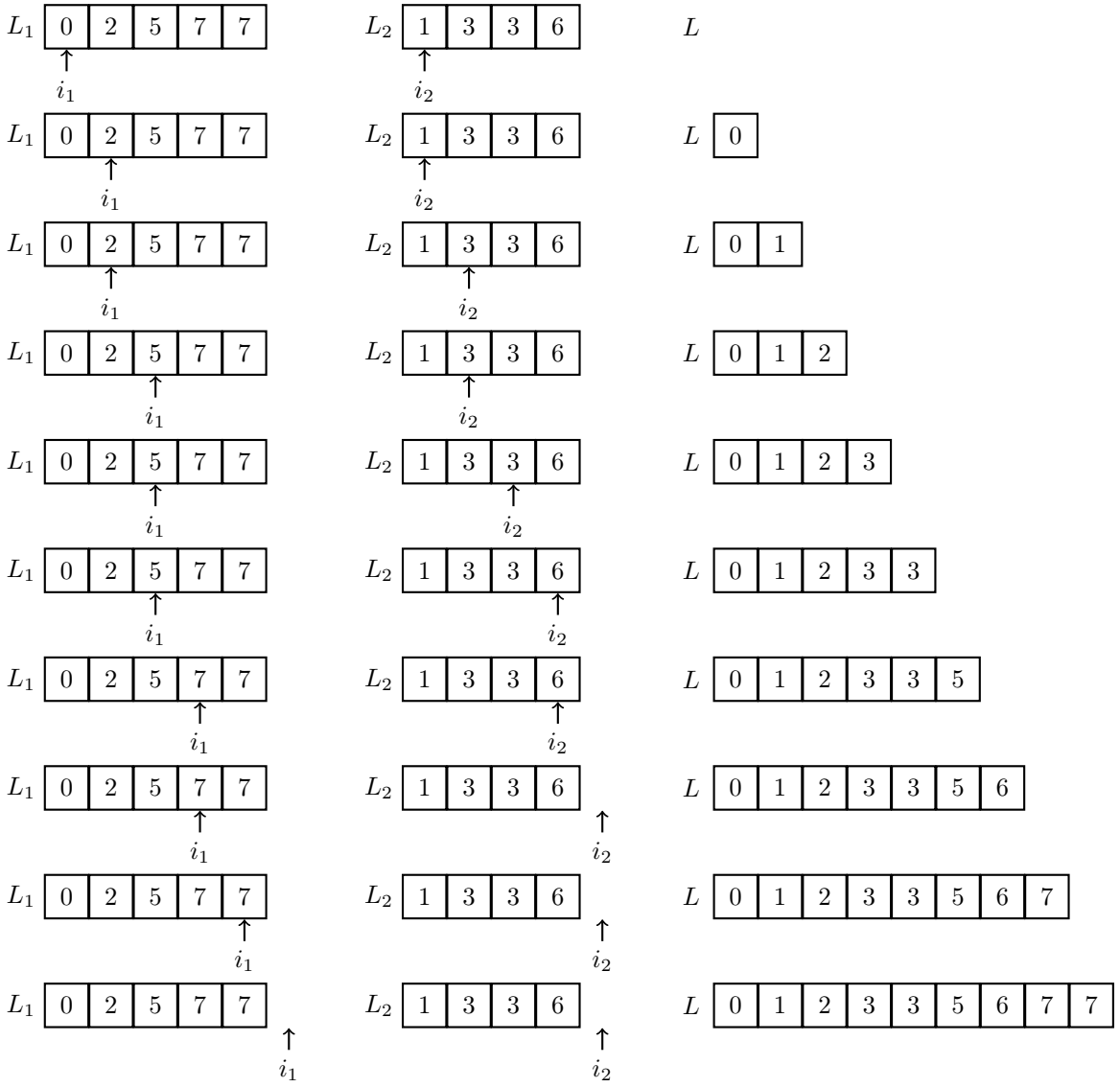


FIGURE 13.1 – Illustration de la fusion de deux listes triées

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 on 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 :

$$\text{Inv}(i) : \texttt{L} \text{ est triée dans l'ordre croissant et ses éléments sont ceux de } \texttt{L1[0:i1]} \text{ et } \texttt{L2[0:i2]}.$$

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. En terme de complexité, il est clair que le nombre d'affectations réalisées est exactement $n_1 + n_2$ (on compte 1 affectation pour l'utilisation d'`append`) puisqu'on en fait une à chaque tour de boucle. Le nombre de comparaisons dépend du déroulement de l'algorithme : on en fait plus si on a besoin de piocher alternativement dans `L1` et `L2`, alors que le nombre minimal va être atteint si les éléments de `L1` sont tous plus petits que les éléments de `L2` (si `L1` est de taille inférieure à `L2`, ce qui sera vérifiée dans la suite). Dans le pire cas, on fait donc $n_1 + n_2 - 1$ comparaisons, et dans le meilleur $\min(n_2, n_1)$.

13.2.2 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 $m = \left\lfloor \frac{n}{2} \right\rfloor$ 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 $m = \left\lfloor \frac{n}{2} \right\rfloor$, 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 (version rapide). 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 :

$$C(n) = C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C\left(\left\lceil \frac{n}{2} \right\rceil\right) + O(n)$$

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. Dans le cas où n est une puissance de 2 et s'écrit $n = 2^p$, on a $C(2^p) = 2C(2^{p-1}) + O(2^p)$. En divisant par 2^p, on obtient $\frac{C(2^p)}{2^p} = \frac{C(2^{p-1})}{2^{p-1}} + O(1)$. Ainsi,

$$\frac{C(2^p)}{2^p} = C(1) + \sum_{k=1}^{p} \left[\frac{C(2^k)}{2^k} - \frac{C(2^{k-1})}{2^{k-1}} \right] = O(p)$$

D'où $C(n) = O(n \log n)$ car $p = \log_2(n)$ ici. Ce résultat s'étend à n quelconque, en admettant la croissance de C et en encadrant n entre deux puissances de 2 successives. C'est un cas particulier du théorème sur les récurrences « diviser pour régner ». 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 :

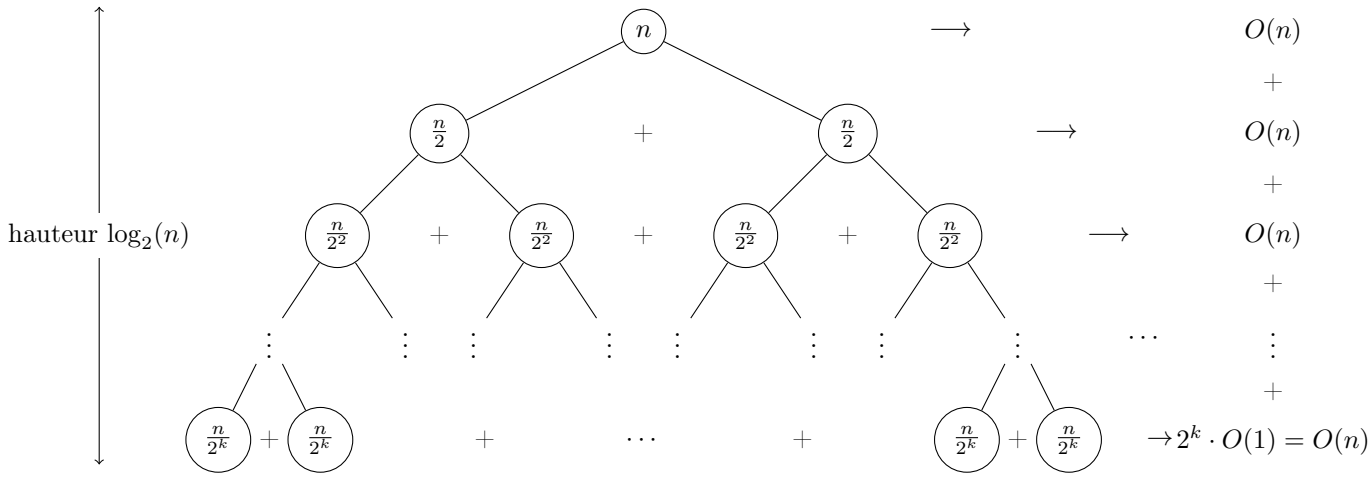


FIGURE 13.2 – Complexité dans le tri fusion sur une liste de taille $n = 2^k$.

Complexité (version longue). On peut essayer d'anayser plus précisément le nombre d'affectations et le nombre comparaisons.

Nombre de comparaisons. La fonction `fusion` fait $n - 1$ comparaisons pour fusionner deux listes de taille totale n dans le pire cas, et seulement la taille de la plus petite liste dans le meilleur cas. On obtient donc ici :

$$C_{\text{comp}}^{\text{pire}}(n) = \begin{cases} 0 & \text{si } n = 1. \\ C_{\text{comp}}^{\text{pire}}(\lfloor \frac{n}{2} \rfloor) + C_{\text{comp}}^{\text{pire}}(\lceil \frac{n}{2} \rceil) + n - 1 & \text{sinon.} \end{cases} \quad C_{\text{comp}}^{\text{meilleur}}(n) = \begin{cases} 0 & \text{si } n = 1. \\ C_{\text{comp}}^{\text{meilleur}}(\lfloor \frac{n}{2} \rfloor) + C_{\text{comp}}^{\text{meilleur}}(\lceil \frac{n}{2} \rceil) + \lfloor \frac{n}{2} \rfloor & \text{sinon.} \end{cases}$$

Dans le cas où n est une puissance de 2, on obtient $C_{\text{comp}}^{\text{pire}}(n) = n \log_2(n) - n + 1$ et $C_{\text{comp}}^{\text{meilleur}}(n) = \frac{n}{2} \log_2(n)$, en faisant essentiellement le même raisonnement que précédemment. On peut montrer que $C_{\text{comp}}^{\text{pire}}(n) \underset{n \to +\infty}{\sim} n \log_2(n)$ et $C_{\text{comp}}^{\text{meilleur}}(n) \underset{n \to +\infty}{\sim} \frac{n}{2} \log_2(n)$ pour n quelconque. [1].

1. Le lecteur suspicieux vérifiera que les solutions de ces récurrences sont $C_{\text{comp}}^{\text{pire}}(n) = n\,[\log_2(n) + f(\log_2(n))] - n + 1$ et $C_{\text{comp}}^{\text{meilleur}}(n) = \frac{n}{2}\,[\log_2(n) + f(\log_2(n))]$, avec f la fonction 1-périodique $f : x \mapsto 2 - \{x\} - 2^{1-\{x\}}$, où $\{x\} = x - \lfloor x \rfloor$ est la partie fractionnaire du réel x. Remarquez que f s'annule en les entiers, ce qui est cohérent avec le résultat vu si p est une puissance de 2.

Nombre d'affectations. Il est un peu plus délicat de parler du nombre d'affectations car ici on renvoie une copie triée de la liste, et il faut faire des découpages. Voir la feuille d'exercice pour un tri fusion « un peu plus en place », dont le nombre d'affectations vérifie :

$$C_{\text{aff}}(n) = \begin{cases} 0 & \text{si } n = 1. \\ C_{\text{aff}}(\lfloor \frac{n}{2} \rfloor) + C_{\text{aff}}(\lceil \frac{n}{2} \rceil) + n & \text{sinon.} \end{cases}$$

La solution de cette récurrence dans le cas où n est une puissance de 2 est $C_{\text{aff}}(n) = n \log_2(n)$. Ceci est également un équivalent [2] pour n non une puissance de 2.

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.

13.3 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.

13.3.1 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 [3]. 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 $g < d - 1$, 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.
- 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 :

2. De même que pour le nombre de comparaisons, il est possible dans ce cas de donner le nombre exact d'affectations effectuées : $C_{\text{aff}}(n) = n\,[\log_2(n) + f(\log_2(n))]$ avec f la fonction de la note précédente.
3. On laisse l'écriture d'un tel tri en exercice.

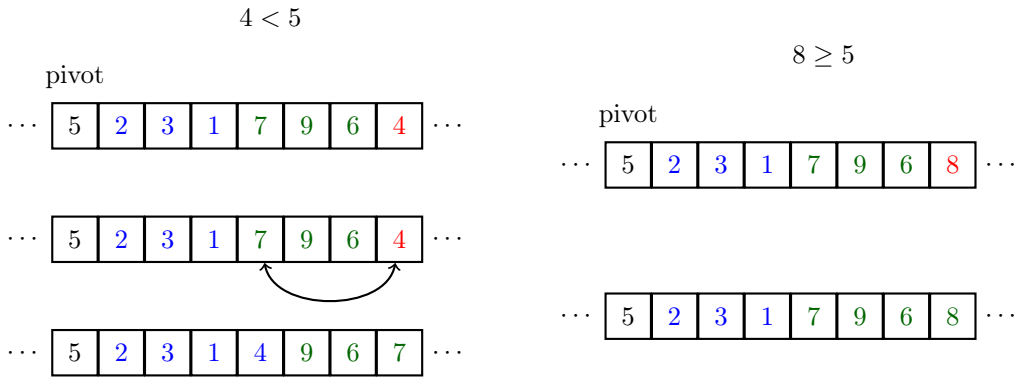


FIGURE 13.3 – Le pivot est 5. L'élément auquel on s'intéresse (en rouge) est soit strictement inférieur ou égal au pivot, auquel cas il est permuté avec l'élément supérieur ou égal le plus à gauche, soit supérieur, auquel cas on ne fait rien.

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
            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. En terme de complexité, il est clair que la fonction `partition` fait exactement $d-g-1$ comparaisons. En terme d'échanges, dans le « meilleur » cas, elle n'en fait aucun (ce qui correspond au cas ou les éléments sont tous strictement plus petits ou tous plus grands que le pivot) et dans le « pire cas », elle en fait $d-g-1$. Mais on va voir que cette notion de pire cas et de meilleur cas n'est pas très pertinente : ce qu'on espère surtout c'est que la partition soit équilibrée au maximum (dans ce cas, le nombre d'échanges est variable entre 1 et environ $\frac{d-g}{2}$).

13.3.2 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. 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, en terme de comparaisons :

$$C_{\text{comp}}^{\text{meilleur}}(n) = C_{\text{comp}}^{\text{meilleur}}\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) + C_{\text{comp}}^{\text{meilleur}}\left(\left\lceil \frac{n-1}{2} \right\rceil\right) + n - 1$$

On peut donner une formule exacte pour ce nombre de comparaisons lorsque n est de la forme $2^k - 1$. En effet dans ce cas, $C_{\text{comp}}^{\text{meilleur}}(n) = 2 + (n+1)(k-2) \underset{n\to+\infty}{\sim} n\log_2(n)$. On peut montrer que cet équivalent est toujours valable pour n quelconque[4]. Ainsi, dans le meilleur cas le tri par partition fait de l'ordre de $n\log_2(n)$ comparaisons, et environ $\frac{n}{2}\log_2(n)$ échanges (donc $n\log_2(n)$ affectations), ce qui est très bien. En fait, on est en général beaucoup plus proche du meilleur cas que du pire, comme on va le voir au paragraphe suivant.

Complexité : nombre moyen de comparaisons. Comme on l'a dit dans l'introduction du chapitre précédent sur les tris, pour l'étude de la complexité en moyenne, on suppose que les éléments de la liste à trier sont tous distincts et que les positions relatives des éléments sont équiprobables. On suppose donc 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.

4. Le lecteur vérifiera que la solution de cette récurrence est $C_{\text{comp}}^{\text{meilleur}}(n) = (n+1)\left[\log_2(n+1) - 2 + f(\log_2(n+1)-2)\right] + 2$, avec f la fonction 1-périodique $f : x \mapsto 2 - \{x\} - 2^{1-\{x\}}$ (la même que dans la note précédente).

— 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 :
$\mathbb{E}(X) \underset{n\to+\infty}{\sim} 2n\ln(n) = 2\ln(2)n\log_2(n)$. Comme $2\ln(2) \simeq 1.39$, on est très proche du meilleur cas !

Complexité : nombre moyen de comparaisons. Dans le code ci-dessus, il est facile de voir que le nombre d'échanges moyens réalisés est environ moitié moins que le nombre de comparaisons effectués (heuristiquement : on fait un échange si on tombe sur un élément inférieur au pivot, ce qui arrive en moyenne une fois sur deux), on trouve donc également $2n\ln(n)$ affectations en moyenne (deux affectations pour un échange).

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.

13.4 Une borne inférieure sur la complexité des tris par comparaisons

On a vu que le tri rapide a une complexité quasi-linéaire en moyenne, et le tri fusion une complexité quasi-linéaire tout court. Peut-on faire mieux ? La réponse est non. Pour le prouver, on va regarder le comportement d'un algorithme de tri par comparaisons quelconque sur les listes de la forme $[\sigma(0), \dots, \sigma(n-1)]$, où σ est une permutation de $\mathfrak{S}_n$.

À chaque algorithme de tri, on peut associer un *arbre binaire*, résumant le tri suivant les comparaisons effectuées, on part d'une liste $[\sigma(0), \dots, \sigma(n-1)]$ et on termine avec la liste $[0, 1, 2, \dots, n-1]$. Suivant le résultat d'une comparaison entre $\sigma(i)$ et $\sigma(j)$, une branche de l'arbre ou l'autre est suivie. Avec deux permutations distinctes, on se retrouve à deux *feuilles* distinctes de l'arbre, qui a donc au moins $n!$ feuilles.

La figure 13.4 montre un tel arbre associé au tri par sélection [5] sur une liste de 3 éléments $[a, b, c]$. Il a 8 feuilles, 6 d'entre elles sont associées à une permutation, 2 d'entres elles correspondent au résultat de deux comparaisons incompatibles avec une liste à entrées distinctes.

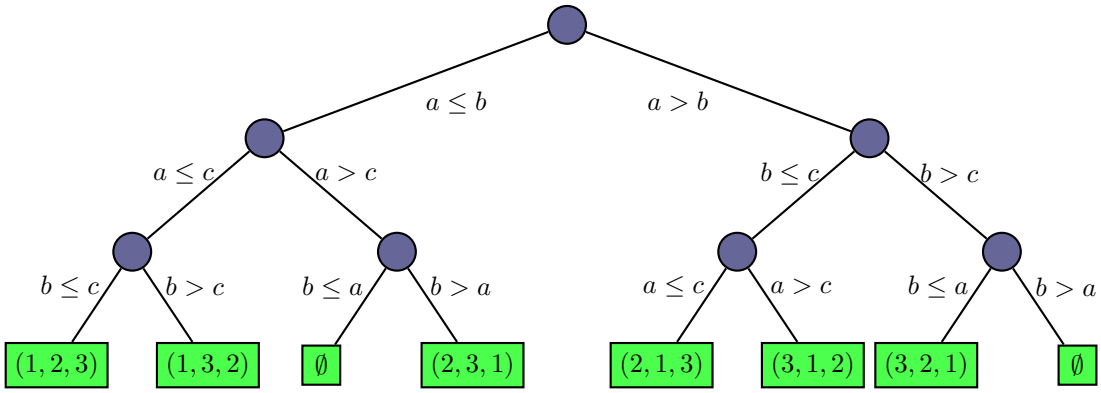


FIGURE 13.4 – Les comparaisons réalisées dans le tri par sélection sur une liste de taille 3.

Il est facile de voir qu'un arbre binaire de hauteur h a au plus 2^h feuilles. Par suite, un arbre ayant $n!$ feuilles a une hauteur d'au moins $\log_2(n!)$. D'après la formule de Stirling, on a $n! \underset{n\to+\infty}{\sim} n^n \, \mathrm{e}^{-n} \sqrt{2\pi n}$, donc $\log_2(n!) \underset{n\to+\infty}{\sim} n \log_2(n)$. Ainsi, la complexité dans le pire cas d'un algorithme de tri par comparaisons quelconque a une complexité d'au moins $\Theta(n \log_2(n))$.

Remarque : on peut montrer que la complexité en moyenne est aussi au moins $\Theta(n \log_2(n))$. Ainsi, le tri fusion est optimal en nombre de comparaisons effectuées !

13.5 Application : calcul de la médiane.

Le problème. On s'intéresse au calcul de la médiane d'une liste, ou plus généralement du k-ème plus petit élément d'une liste de taille n, avec $0 \leq k < n$. (Attention : le 0-ème plus petit élément est le plus petit élément de la liste).

Solution naïve. Évidemment, on peut pour déterminer le k-ème plus petit élément commencer par trier la liste et extraire l'élément à l'indice k. Cette approche a une complexité $O(n \log n)$ avec le tri fusion. On cherche à faire mieux !

Variante du tri par sélection. Une autre idée est par exemple de ne faire qu'une partie du tri par sélection. En effet, si on cherche le k-ème plus petit élément, en appliquant seulement les $k+1$ premières étapes du tri, les $k+1$ plus petits éléments se trouvent à leur position finale une fois la liste triée : on peut arrêter là le procédé pour extraire l'élément d'indice k. Cette approche a une complexité $O(nk)$ et est intéressante si k est petit. De même, si k est « grand » (proche de n), on peut faire une variante du tri par sélection qui s'intéresse d'abord au maximum, et ne faire que $n-k$ étapes. On obtient alors une complexité $O(n(n-k))$. Mais ni l'une ni l'autre de ces approches n'est efficace si k est proche de $n/2$: on obtient une complexité en $O(n^2)$, ce qui est moins bien que de trier entièrement la liste avec le tri fusion.

Variante du tri rapide. Observons par contre ce qui se passe lors d'une étape du tri rapide : on applique la fonction partition, qui retourne la position définitive m du pivot, et il n'y a plus qu'à trier les éléments situés à gauche et à droite. Si on s'intéresse simplement au k-ème plus petit élément, on a la distinction ci-dessous :

5. Il faut vraiment dérouler le tri pour construire l'arbre ! Essayez avec un autre tri.

— $m = k$: le pivot est notre élément cherché !

— $m < k$: le k-ème plus petit élément se trouve dans la partie à droite du pivot.

— $m > k$: le k-ème plus petit élément se trouve dans la partie à gauche du pivot.

Ainsi, pour trouver le k-ème plus petit élément de la liste, on peut procéder de manière similaire au tri rapide, la différence est qu'on ne fait qu'un seul appel récursif (au plus).

L'algorithme de calcul de la médiane

```
def mediane(L,k):
    """ L liste non vide, k un entier entre 0 et len(L)-1. Renvoie le k-ième plus petit élément de L """
    T=L[:] #a priori, on ne veut pas modifier la liste L.
    def aux(g,d):
        m=partition(T,g,d)
        if m==k:
            return T[m]
        elif m<k:
            return aux(m+1,d)
        else:
            return aux(g,m)
    return aux(0,len(L))
```

Comme pour le tri rapide, la complexité dans le pire cas est toujours quadratique, ce qui est intéressant est là encore la complexité en moyenne. Analysons de la même manière quels sont les couples $i < j$ effectivement comparés, en fonction de k et de la taille n de la liste :

— si $i \leq k \leq j$, alors i et j sont comparés si et seulement si l'un des deux est pris comme pivot avant tout élément de $[\![i, j]\!]$, évènement qui a probabilité $\frac{2}{j-i+1}$.

— si $i < j < k$, alors i et j sont comparés si et seulement si l'un des deux est choisi comme pivot avant tout élément de $[\![i, k]\!]$, évènement qui a probabilité $\frac{2}{k-i+1}$.

— si $k < i < j$, alors i et j sont comparés si et seulement si l'un des deux est choisi comme pivot avant tout élément de $[\![k, j]\!]$, évènement qui a probabilité $\frac{2}{j-k+1}$.

Ainsi, le nombre moyen de comparaisons est donné par trois sommes à estimer. Allons-y.

— Tout d'abord :

$$\sum_{\substack{0\leq i<j\leq n-1\\ \text{avec } i\leq k\leq j}} \frac{2}{j-i+1} = \sum_{d=1}^{n-1} \sum_{\substack{(i,j)\\ i\leq k\leq j \text{ et } j=i+d}} \frac{2}{d+1} \leq \sum_{d=1}^{n-1} \frac{2}{d+1}(d+1) \leq 2n$$

En effet, les couples (i, j) vérifiant $0 \leq i \leq k \leq j \leq n-1$ et $j = i + d$ sont inclus dans l'ensemble $\{(k-d, k), (k-d+1, k+1), \ldots, (k, k+d)\}$, il y en a donc au plus $d + 1$.

— Passons à la somme suivante :

$$\sum_{\substack{0\leq i<j\leq n-1\\ \text{avec } i<j<k}} \frac{2}{k-i+1} = \sum_{i=0}^{k-2} \sum_{j=i+1}^{k-1} \frac{2}{k-i+1} \leq \sum_{i=1}^{k-2} \frac{2}{k-i+1}(k-i-1) \leq 2k$$

— De même,

$$\sum_{\substack{0\leq i<j\leq n-1\\ \text{avec } k<i<j}} \frac{2}{j-k+1} = \sum_{j=k+2}^{n-1} \sum_{i=k+1}^{j-1} \frac{2}{j-k+1} \leq \sum_{j=k+2}^{n-1} \frac{2}{j-k+1}(j-k-1) \leq 2(n-k)$$

Ainsi, le nombre moyen de comparaisons est majoré par $4n$. La complexité de calculer le k-ième plus petit terme de la liste à l'aide de l'algorithme `mediane` est donc linéaire en moyenne.

Remarques. • Là encore, on peut choisir le pivot au hasard dans l'algorithme de `partition`, et modifier l'algorithme `mediane` pour transformer l'appel récursif en boucle `while`.

• Il est en fait possible de calculer le k-ième plus petit élément d'une liste en temps linéaire même dans le pire cas. L'algorithme est plus complexe, et peut être utilisé pour obtenir une version quasi-linéaire du tri rapide même dans le pire cas : on choisit explicitement la médiane d'une liste avant d'en réaliser la partition. (Par contre, le résultat est un tri un peu moins rapide en pratique...)

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 13.1 – Complexité des tris efficaces (notation O)

13.6 Conclusion et comparaison des tris

Le tableau suivant récapitule les résultats que nous avons montré sur les deux tris efficaces de ce chapitre : Si l'on analyse le nombre d'opérations élémentaires des tris, on obtient plus précisément [6] :

Tri	Cas	Meilleur	Pire	Moyen
Tri fusion	Comparaisons	$\frac{n}{2}\log_2(n)$	$n\log_2(n)$	$n\log_2(n)$
	Affectations	$n\log_2(n)$	$n\log_2(n)$	$n\log_2(n)$
Tri rapide	Comparaisons	$n\log_2(n)$	$n^2/2$	$2n\ln(n)$
	Affectations	$O(n)$	$O(n)^*$	$2n\ln(n)$

TABLE 13.2 – Équivalent du nombre de comparaisons et d'affectations pour trier une liste de taille n avec l'un des tris efficaces, dans les meilleur, pire et moyen cas. Le pire cas pour le tri rapide en nombre de comparaisons donne peu d'affectations.

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.

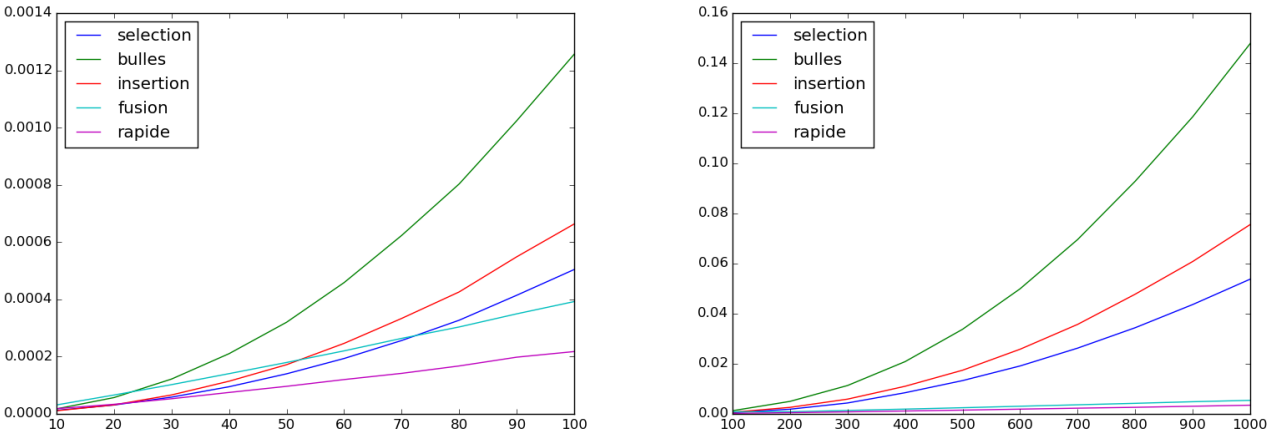


FIGURE 13.5 – Comparaisons des tris pour des tailles de 10 à 100 à gauche, et de 100 à 1000 à droite. Le temps est en secondes, on a pris une moyenne sur 100 listes différentes.

6. Pour le tri fusion, il s'agit d'une version optimisée, voir la feuille d'exercices.

Cinquième partie

Annexes

Chapitre 14

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.

14.1 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.

14.1.1 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`...

14.1.2 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.

14.2 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]])
```

14.3 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`.

14.3.1 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.

14.3.2 Quelques exemples

Un tracé basique. Deux courbes tracées sur le même graphique (voir figure 14.1) :

```
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 14.2

```
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
```

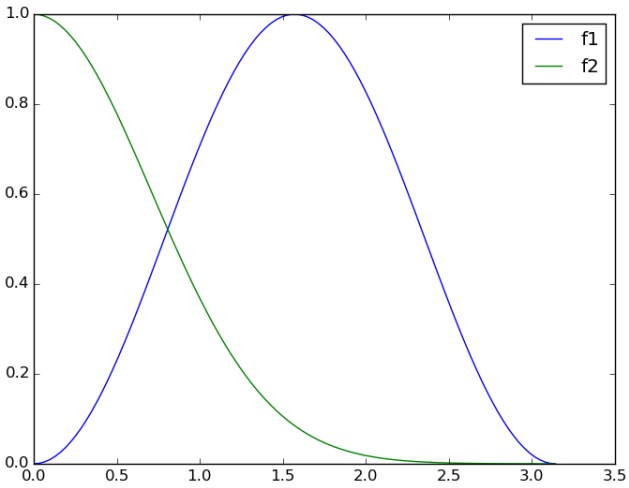


FIGURE 14.1 – Un tracé facile

```
                    # 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()
```

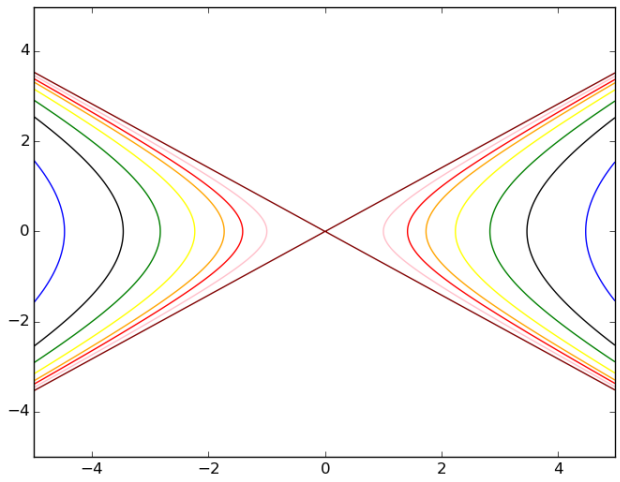


FIGURE 14.2 – Des lignes de niveau

Un petit exemple en 3D : la caténoïde. On va tracer une figure en 3D. La caténoïde de la figure 14.3 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()
```

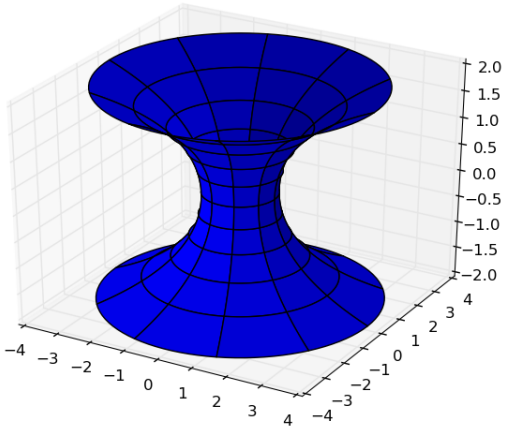


FIGURE 14.3 – Une caténoïde

Catalogue. On trouvera de nombreux exemples à l'adresse `http://matplotlib.org/examples/`.

14.4 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.

14.4.1 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])
```

14.4.2 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.

14.4.3 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

$$X'(t) = f(X(t), t), \quad X_0 = \binom{1}{5}, \quad \text{et} \quad f\left(\binom{u}{v}, t\right) = \binom{u(7-2v)}{-v(1-4u)}$$

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 14.4. 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.

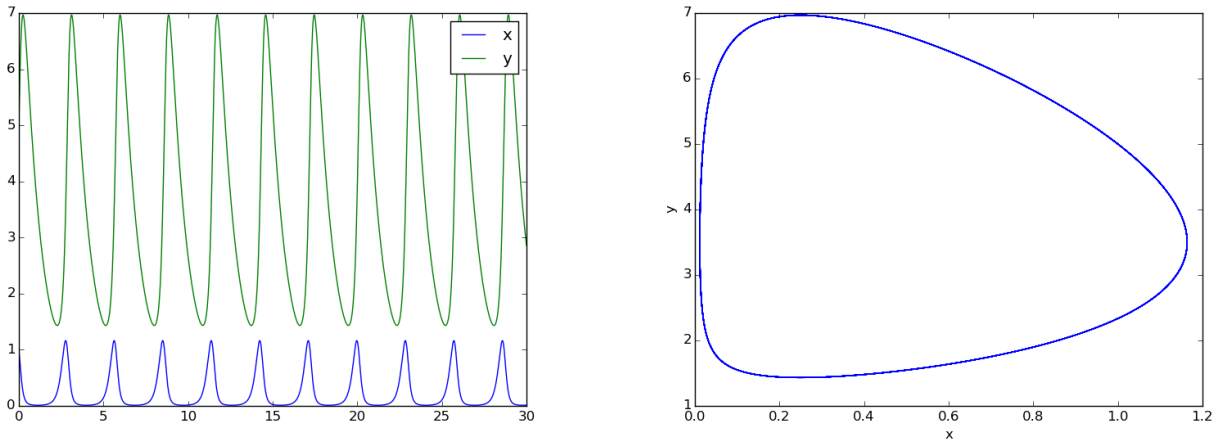


FIGURE 14.4 – Résolution de l'équation de Lokta-Volterra. (a) x et y en fonction du temps. (b) x en abscisse, y en ordonnée.

14.5 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/`