Introduction à l'informatique

Benjamin Monmege

2022/2023

Manuscrit relu par Sébastien Delecraz et les étudiants du cours à distance depuis 2018.
Cours enseigné en présentiel par Antonio E. Porreca, Sylvain Sené et Alessia Milani.

Licence CC-BY-NC-SA : Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions

Préface

Description de l'enseignement

L'objectif de cette unité d'enseignement est de découvrir différents aspects de la science informatique, tant dans ces aspects de traitement de l'information que de l'étude du calcul. Le cours sera illustré par l'étude de problèmes complexes concrets qu'on décompose en tâches plus simples.

Compétences à acquérir

— Se servir aisément des bases de la logique pour valider ou réfuter un raisonnement.
— Être familiarisé avec les concepts fondamentaux de complexité et calculabilité.
— Utiliser les concepts fondamentaux de l'informatique (langages formels, logique, et graphes) pour la programmation et la modélisation.
— Évaluer la complexité et la correction d'une solution algorithmique.
— Modéliser un problème concret sous la forme d'un problème algorithmique connu.
— Concevoir le traitement informatisé d'informations de différentes natures, telles que du texte, des images et des nombres.

Modalité de contrôle des connaissances

Aux deux sessions de ce cours en formation à distance, la note est calculée via le formule

$$\max(\text{Examen}, \quad 0.2 \times \text{Contrôle Continu} + 0.8 \times \text{Examen})$$

avec examen terminal de deux heures, sans document ni calculatrice. La note de contrôle continu contient le résultat de deux devoirs à la maison :
— devoir 1 : envoi le 6 décembre 2021, à rendre avant le 11 février 2022
— devoir 2 : envoi le 14 février 2022, à rendre avant le 15 avril 2022

Table des matières

1 Introduction **7**
1.1 Mais la « science informatique », c'est quoi ? . . . 9
1.2 Codage de l'information . . . 11
1.3 La science du calcul . . . 23
1.4 Algorithmes... ou algo-rythmes ? . . . 27

2 Description des algorithmes **33**
2.1 Structures de contrôle : une introduction en Scratch . . . 33
2.2 Variables . . . 37
2.3 Fonctions . . . 39
2.4 Conditionnelles . . . 40
2.5 Itérations . . . 41
2.6 Lecture et écriture . . . 43

3 Algorithmes sur les structures linéaires **47**
3.1 Définition d'un tableau . . . 47
3.2 Tableaux et chaînes de caractères : application à la cryptologie . . . 49
3.3 Rechercher dans un tableau . . . 55
3.4 Tri d'un tableau . . . 61

4 Algorithmes sur les entiers et les flottants **71**
4.1 Addition d'entiers . . . 71
4.2 Divisibilité . . . 74
4.3 Exponentiation . . . 79
4.4 Recherche d'un zéro d'une fonction . . . 81

5 Graphes : modélisation et parcours **87**
5.1 Les graphes sont partout . . . 87
5.2 Graphes : application au diamètre des réseaux sociaux . . . 87
5.3 Graphes orientés : application aux graphes de configuration . . . 92
5.4 Codage d'un graphe . . . 95
5.5 Parcours de graphe . . . 96
5.6 Plus courts chemins dans des graphes pondérés : algorithme de Dijkstra . . . 100

6 Théorie des graphes : chemins eulériens et coloration **107**
6.1 Graphes eulériens : le problème des sept ponts de Königsberg . . . 107
6.2 Coloration de graphes et des cartes . . . 114

7 Arbres **121**
7.1 Exemples d'arbres déjà rencontrés . . . 121
7.2 Définitions . . . 123
7.3 Arbres binaires . . . 127
7.4 Affichage d'une arborescence de fichiers : parcours préfixe . . . 128
7.5 Expressions arithmétiques et parcours postfixe . . . 132
7.6 Parcours infixe : affichage d'une expression . . . 138
7.7 Arbres binaires de recherche . . . 139

8 Calculabilité **145**
8.1 Arbres de décision . . . 145
8.2 Automates finis . . . 150
8.3 Applications des automates finis . . . 154
8.4 Langage non accepté par un automate fini . . . 157
8.5 Des automates vers les machines . . . 158
8.6 Machines de Turing . . . 163
8.7 Lien entre machines de Turing et pseudo-code . . . 170
8.8 Peut-on tout calculer ? . . . 171

9 Conclusion **177**

Chapitre 1

Introduction

Ce cours est l'occasion de mieux comprendre ce qu'est la science informatique. Vous y verrez comment concevoir le traitement informatisé d'informations de différentes natures, telles que du texte, des images et des nombres. Nous y modéliserons des problèmes concrets sous la forme d'un problème algorithmique connu, puis nous évaluerons la complexité et la correction d'une solution algorithmique. Vous serez également familiarisé avec les concepts fondamentaux de complexité et de calculabilité.

Par contre, ce cours n'est pas un cours d'introduction aux traitements de texte, ni une Install Party de Linux ; encore moins un cours sur la résolution du PacMan ou sur les graphismes du dernier Tomb Raider. Malheureusement, ce n'est pas dans ce cours non plus que vous apprendrez à hacker la NSA...

Cette unité se déroule en parallèle d'une autre unité, *Mise en œuvre informatique* (3 crédits ECTS) [1], qui complémente ce cours d'une mise en pratique dans le langage Python des idées et algorithmes que nous développerons au cours de ce cours. Il s'agit donc de deux unités différentes (qui se dérouleront de manière indépendante) traitant du même sujet de façon complémentaire :

la ≪ science informatique ≫
la ≪ pensée calculatoire ≫ (ou ≪ computational thinking ≫ en anglais)

Afin de clarifier l'objectif du cours, commençons par noter que l'informatique c'est à la fois une *technologie* et une *discipline scientifique*. C'est une technologie en ce sens qu'elle abrite le développement de logiciels (conception, développement, test...), d'ordinateurs (architecture, stockage...), le tout rendu possible grâce à des matériaux (silicium) et beaucoup d'électronique (micro-composants). Mais c'est aussi une discipline scientifique à part entière : c'est la science de l'*information* et du *calcul*. C'est une discipline proche, mais différente, des mathématiques. Voici une image pour mieux appréhender la distinction entre la technologie et la discipline scientifique :

> ≪ Demander à un·e chercheur·se en informatique de réparer la souris d'un ordinateur, c'est comme demander à un·e chercheur·se en mécanique des fluides de réparer des toilettes bouchées. ≫

Comme toute discipline scientifique, l'informatique a donc des savant·e·s clés : quelques-un·e·s d'entre eux·elles sont représenté·e·s en Figure 1.1. En haut à gauche, George Boole

1. dont je suis également l'enseignant responsable, à partir de la rentrée 2022

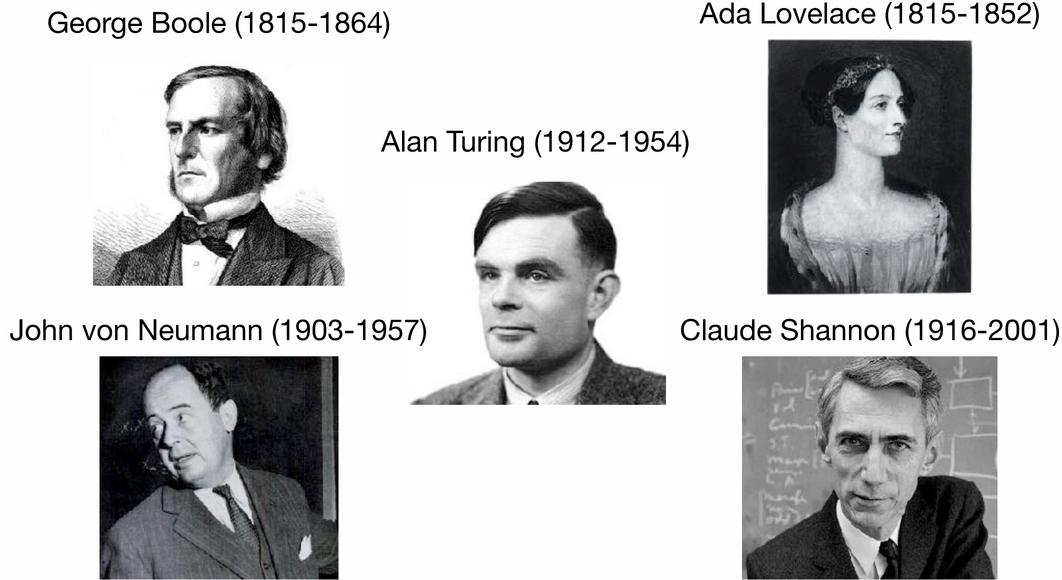


FIGURE 1.1 – Quelques savants clés de la discipline scientifique informatique

est le père de la logique binaire (avec deux valeurs 0/faux et 1/vrai), à la base du codage informatique. En haut à droite, Ada Lovelace est souvent reconnue comme l'auteure du premier programme informatique lors de son travail sur l'ancêtre de l'ordinateur, la machine analytique de Charles Babbage. Au milieu, Alan Turing a posé les bases de la calculabilité à travers la mise en place d'un modèle théorique de machines, ensuite appelés la machine de Turing : il a aussi joué un rôle majeur dans la cryptanalyse de la machine Enigma pendant la seconde guerre mondiale. En bas à gauche, John Von Neumann a travaillé à l'axiomatisation logique des mathématiques, mais il a aussi donné son nom à une architecture (unité de contrôle, mémoire, unité arithmétique et logique) utilisée dans la quasi-totalité des ordinateurs modernes. Il est également à l'origine du concept d'automates cellulaires, un modèle de calcul inspiré par la biologie. En bas à droite, Claude Shannon est le père fondateur de la théorie de l'information qui régit la communication d'information entre deux machines : il a popularisé l'usage du mot *bit* comme mesure élémentaire de l'information numérique.

Les premiers ordinateurs modernes datent !è de la seconde guerre mondiale et de ses besoins accrus en calcul (cryptanalyse, calculs de trajectoires...). Trois ordinateurs sont ainsi représentés en Figure 1.2 : observez la taille importante de ces ordinateurs qui remplissaient alors des pièces entières. Notez également la présence de femmes sur les photographies, qui rappelle que les premières à programmer des ordinateurs étaient des femmes, ce qui s'est malheureusement perdu au fil du temps. Le film *Hidden Figures* (en français, *Les figures de l'ombre*) de Theodore Melfi illustre bien ce fait historique au travers de l'histoire de la mathématicienne et ingénieure spatiale américaine Katherine Johnson, ayant permis d'automatiser les calculs de trajectoires à l'aide de machines.

Les fondements de la science informatique sont donc bien antérieurs au développement de machines.

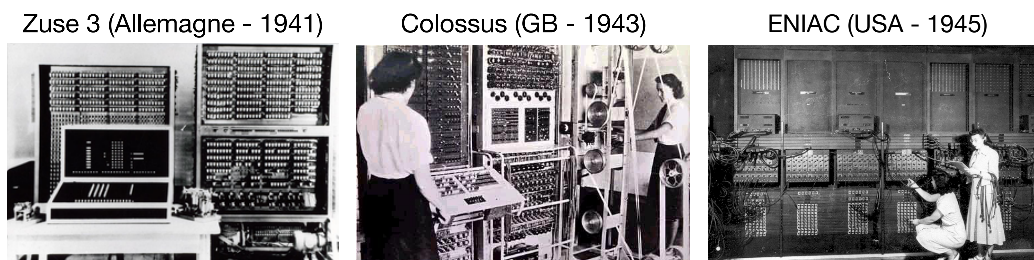


Figure 1.2 – Les premiers ordinateurs modernes

1.1 Mais la « science informatique », c'est quoi ?

Une des nombreuses façons de décrire la science informatique, c'est de comprendre ce qu'on peut faire avec de l'*information* (cf Figure 1.3) :

— on peut la recevoir, par exemple lorsqu'on charge une page web dans un navigateur ;
— on peut l'émettre, par exemple lorsqu'on envoie un sms ;
— on peut la traiter, par exemple notre cerveau traite de millions d'informations par seconde, et l'algorithme d'Euclide est une façon de traiter deux entiers en calculant leur plus grand commun diviseur ;
— on peut la stocker, par exemple dans des clés USB ou des bases de données.

Une spécificité de l'informatique est donc de manipuler de l'information, comme notre cerveau. Là où la discipline se distingue, c'est en essayant d'automatiser cette manipulation d'information. C'est le point commun entre les images de la Figure 1.4 :

— en haut à gauche, on y voit la Pascaline, la première machine à calculer inventée par Pascal au XVIIème siècle : elle permettait de réaliser des additions de manière autonome, grâce à des mécanismes internes ;
— en haut à droite, une chaîne de montage industrielle donne à voir des bras robotisés qui aident l'humain à assembler un produit ;
— en bas à gauche, une utopie des voitures autonomes qui sont sur le point de déferler dans nos vies : ces voitures doivent traiter des tonnes d'informations, visuelles en particulier (les piétons, les autres voitures, la signalisation), afin de prendre les bonnes décisions ;
— en bas à droite, un robot aide les consommateurs dans un grand magasin de manière ludique ;
— au centre, le canard automatique de Jacques Vaucanson, au XVIIIème siècle, qui donnait l'illusion de manger, digérer et bouger comme un canard : c'est l'un des automates produits par Vaucanson.

Un objectif de l'informatique consiste donc à *automatiser*, à faire exécuter de manière *répétée* des tâches (possiblement complexes comme conduire une voiture, ou très simple comme effectuer une addition...). La question devient donc : « que peut-on *calculer* de manière autonome et comment ? et efficacement si possible ? »

Mais calculer, ça veut dire quoi ? Pour l'instant, disons que cela signifie qu'on a un problème avec un certain nombre d'entrées, desquelles on veut produire une ou plusieurs sorties. Par exemple,

— on se donne la suite de symboles « 3+2 » et on souhaite obtenir en sortie l'entier « 5 » ;

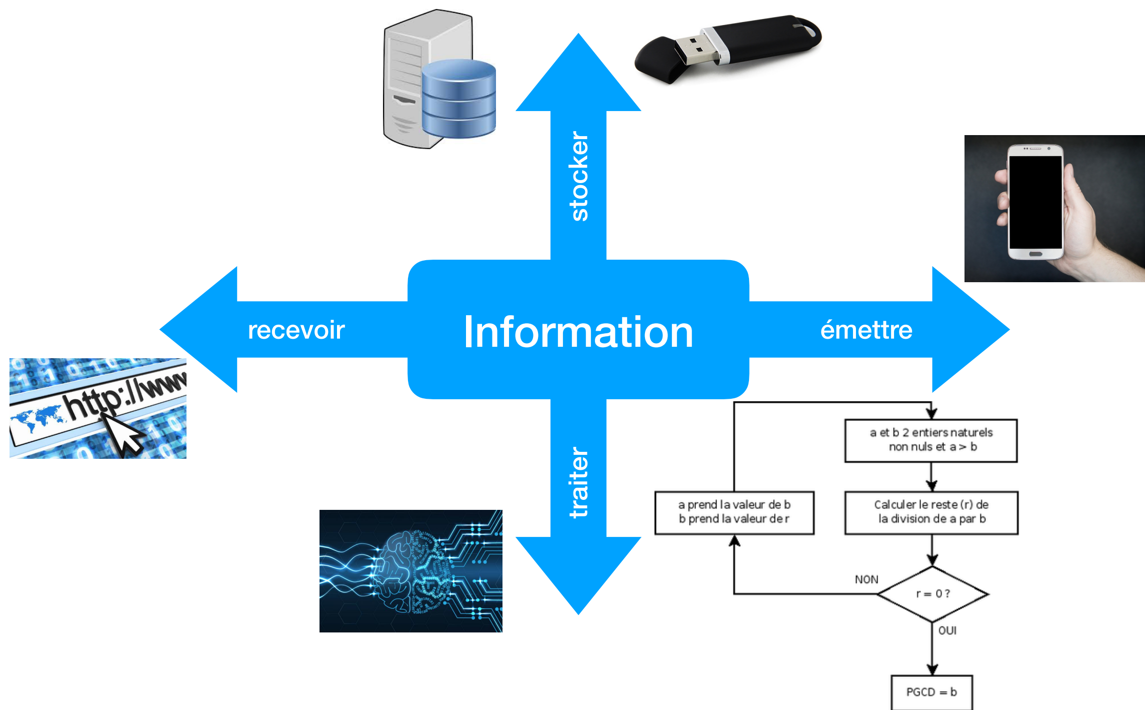


FIGURE 1.3 – La science de l'information, telle que vue par Michel Serres (cf, par exemple, `http://www.academie-francaise.fr/actualites/communication-de-m-michel-serres`)

FIGURE 1.4 – Un point commun : l'automatisation et la répétabilité

— on dit à son téléphone « Dis Siri, définis calculer » et on veut que le téléphone aille chercher dans un dictionnaire la définition du verbe « calculer » pour nous l'imprimer sur l'écran, ou mieux, nous la lise à haute voix ;
— on tape « restaurant Marseille » dans la barre de Google et on s'attend à voir s'afficher une carte avec les meilleurs restaurants de la cité Phocéenne.

Nous verrons plus tard une définition plus détaillée de calcul, mais pour l'instant, cela nous suffira. En tout cas, cela nous permet de nous poser la question suffisante : mais comment un ordinateur sait ce que « vingt-trois » veut dire ?

1.2 Codage de l'information

Plus généralement, comment encode-t-on de l'information dans une machine, afin de calculer dessus par la suite ? Il est crucial de se mettre d'accord sur une façon d'encoder les informations puisque nous allons ensuite potentiellement devoir échanger de l'information entre nous. Il existe donc des standards internationaux. Mais la notion principale permettant d'établir ces standards est l'*abstraction* : nous allons construire petit à petit les encodages d'informations complexes, à partir d'informations plus simples.

1.2.1 Codage des entiers naturels

Commençons donc par coder des entiers naturels : 0, 1, 2, ... Prenons l'entier 24 par exemple. Hormis cette représentation d'un caractère « 2 » suivi d'un caractère « 4 », on peut opter pour la représentation latine « XXIV » ou même la représentation en français « vingt-quatre ». Pour nous aider à choisir une représentation, il faut savoir ce qu'on souhaite faire avec cette représentation. Ce qui fait privilégier l'écriture décimale « 24 » aux deux autres, c'est la simplicité de calculer avec cette représentation : on a ainsi bien du mal à ajouter « deux cent trente-quatre » et « deux cent quatre-vingt-un » si on pose l'addition :

$$\begin{array}{rr} & \texttt{deux cent trente-quatre} \\ + & \texttt{deux cent quatre-vingt-un} \\ \hline & ??? \end{array}$$

La représentation décimale est pratique pour nous, puisque nous avons 10 doigts : nous pouvons donc nous aider de nos mains pour compter. Mais d'ailleurs, jusqu'à combien peut-on compter sur une main ? Une réponse naturelle consiste à répondre 5. Une autre réponse, a priori moins naturelle, consiste à préférer 31 en utilisant davantage de combinaisons de nos doigts levés ou pas... Pour cela, on doit passer d'une représentation décimale des entiers, à une représentation *binaire*, c'est-à-dire en base 2. Suivons la figure 1.5 pour compter de 0 à 31 :
— naturellement, on part de 0 qu'on représente avec la main fermée ;
— tout aussi naturellement, on représente 1 en levant le pouce ;
— un peu d'originalité : représentons 2 en abaissant le pouce, mais en relevant l'index ;
— puis 3 est obtenu en relevant le pouce ;
— la représentation de 4 n'est pas des plus élégantes, avec le seul majeur levé ;
— et 5 s'obtient en levant à nouveau le pouce.

On continue ainsi en passant d'un entier à l'autre
— en levant le pouce s'il est baissé,

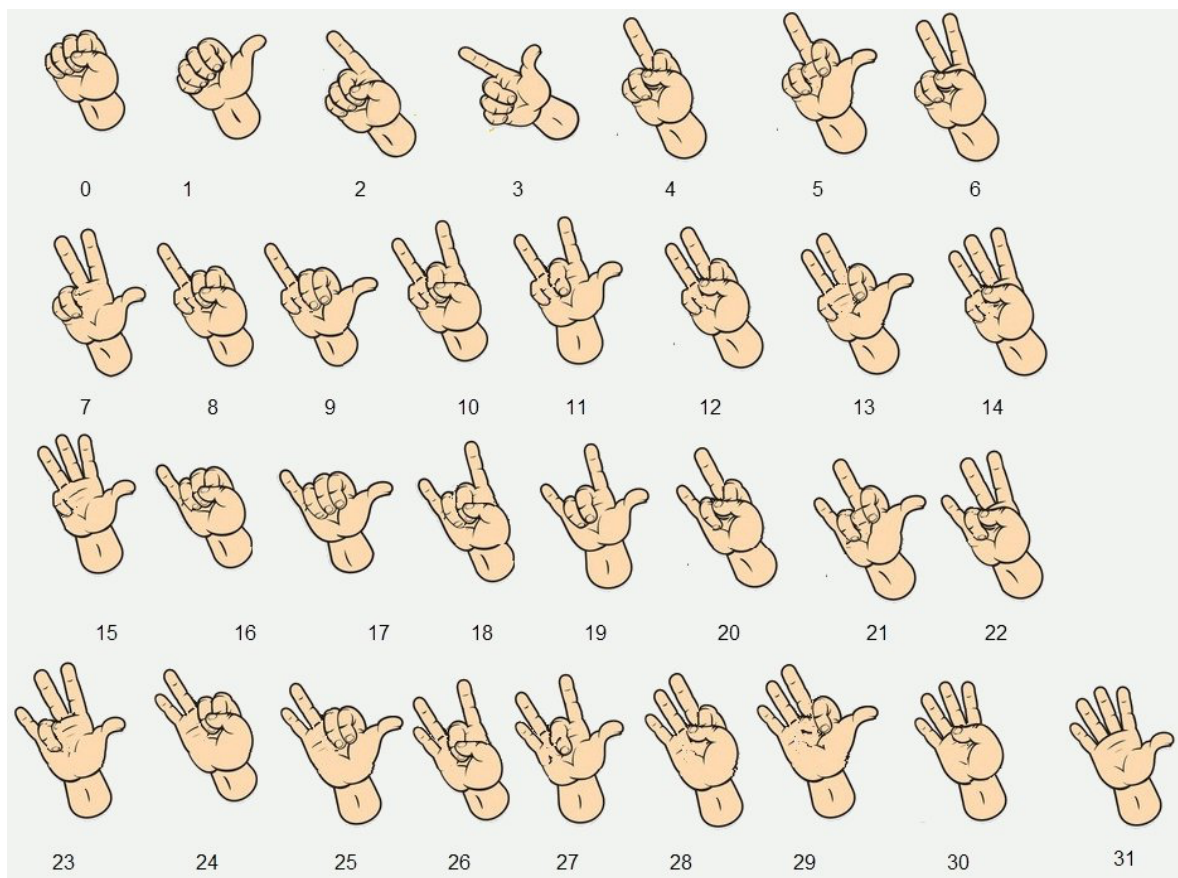


FIGURE 1.5 – Compter sur ses doigts en binaire

— ou alors en baissant le pouce et levant l'index s'il est baissé,
— ou alors en baissant le pouce et l'index et levant le majeur s'il est baissé...

L'entier 5 se représente donc par 101, alors que l'entier 6 est codé par 110 : 1 code un doigt levé, et 0 un doigt baissé. Pour mieux comprendre cette représentation en binaire, rappelons-nous ce que signifie la représentation décimale des entiers. Ainsi, l'entier 317 signifie qu'on a 3 centaines, 1 dizaine et 7 unités, représentant donc :

$$\begin{aligned} 317 &= 300 + 10 + 7 \\ &= 3 \times 10^2 + 1 \times 10^1 + 7 \times 10^0 \end{aligned}$$

C'est donc tout naturellement que la séquence 100111101 représente en binaire l'entier :

$$\begin{aligned} & 1 \times 2^8 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^0 \\ = {} & 256 + 32 + 16 + 8 + 4 + 1 \\ = {} & 317 \end{aligned}$$

Définition 1. Un bit est l'unité d'information la plus simple, pouvant prendre deux valeurs communément notée 0 et 1. On représente l'entier naturel 0 avec le bit 0. On représente un entier naturel non nul $a \in \mathbb{N}^* = \{1,2,3,\ldots\}$ par une *suite de bits* $a_{n-1}a_{n-2}\cdots a_1a_0$, avec $a_0,a_1,\ldots,a_{n-2},a_{n-1} \in \{0,1\}$, telle que $a_{n-1} = 1$ et

$$a = a_{n-1} \times 2^{n-1} + a_{n-2} \times 2^{n-2} + \cdots + a_1 \times 2^1 + a_0 \times 2^0 = \sum_{i=0}^{n-1} a_i 2^i$$

Cette suite est unique, grâce à la condition sur a_{n-1} : on l'appelle la *représentation binaire* de l'entier a.

Par exemple, la représentation binaire de 35 est 100011 car $35 = 2^5 + 2^1 + 2^0$ et celle de 16 est 10000 car $16 = 2^4$.

Il est donc important de connaitre, ou au moins pouvoir retrouver rapidement, les valeurs de 2^n pour n un petit entier naturel :

n	0	1	2	3	4	5	6	7	8	9	10
2^n	1	2	4	8	16	32	64	128	256	512	1024

Lorsqu'on fait des calculs en puissance de 2, on approche donc souvent 2^{10} avec 10^3.

Exercice 1

1. Donner la représentation binaire des entiers 13, 85, 128 et 127.
2. Quels sont les entiers dont les représentations binaires sont 10001, 110101 et 11111111 ?

Exercice 2

Un ordinateur sait calculer des opérations arithmétiques sur les représentations binaires d'entiers.

1. Saurez-vous additionner les représentations binaires 101001 et 1111010 ? Vérifier votre calcul en convertissant les représentations en entiers.
2. Poser de même la multiplication des représentations binaires 10110 et 1011.
3. Effectuer à l'aide des représentations binaires le calcul de 15×15.

Exercice 3

Soit n un entier dont la représentation binaire est de la forme $100 \cdots 001$ (des bits 0 encadrés par deux bits 1). Quelles sont les représentations binaires des entiers n^2 et n^3 ?

Exercice 4

Incrémenter, c'est ajouter un à un compteur. Par exemple, lorsqu'on incrémente un compteur dont la valeur est 13, on obtient la valeur 14. Lorsque le compteur est représenté en binaire, on passe ainsi de 1101 à 1110. Il existe une méthode infaillible pour incrémenter la représentation binaire d'un compteur :

(i) commencer par le bit de poids faible (celui qui est le plus à droite) ;

(ii) inverser le bit ;

(iii) tant que ce bit est à zéro, recommencer l'étape (ii) avec le bit situé à sa gauche ;

(iv) si on arrive au bout de la représentation binaire, ajouter un bit 1.

1. Exécuter cette méthode sur les représentations binaires 11011, 1000 et 11111.
2. Sachant que *décrémenter*, c'est retirer un d'un compteur ayant une valeur strictement positive, décrire une méthode qui réalise cette opération.
3. Décrire de même une méthode pour multiplier par deux la valeur d'un compteur.

1.2.2 Codage des entiers relatifs

Au-delà des entiers naturels, il est important de pouvoir coder des entiers relatifs :

$$\ldots -3, -2, -1, 0, 1, 2, 3 \ldots$$

Il s'agit donc d'un entier *signé*. Une représentation naturelle d'un entier relatif $n \in \mathbb{Z}$ consiste donc à ajouter un *bit de signe* à gauche de la représentation en binaire de l'entier naturel $|n|$ correspondant à la valeur absolue de n : le bit de signe vaut 0 si $n \geq 0$ et 1 si $n < 0$. Ainsi, lorsqu'on code des entiers relatifs, le codage de 7 est 0111 et celui de -7 est 1111.

Notons qu'il ne s'agit pas de la représentation privilégiée dans les ordinateurs modernes : on utilise plutôt la représentation par *complément à deux*, mais nous n'en parlerons pas dans ce cours.

1.2.3 Codage flottant

Il n'y a pas que des entiers à savoir coder. On peut aussi vouloir coder des « nombres à virgule » en binaire : il existe une zoologie importante d'ensemble de nombres, au-delà des

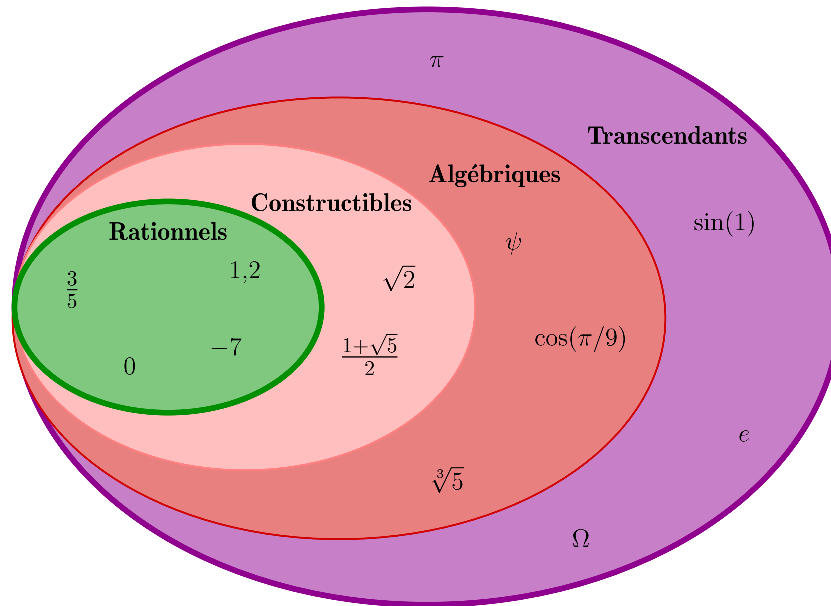


FIGURE 1.6 – Au-delà des entiers relatifs...

entiers relatifs (cf Figure 1.6). Coder un nombre rationnel $r \in \mathbb{Q}$ n'est pas très compliqué, une fois qu'on se rappelle qu'il s'agit d'un quotient $r = a/b$ avec $a \in \mathbb{Z}$ un entier relatif et $b \in \mathbb{N}^*$ un entier naturel non nul. On a donc vu comment représenter a et b précédemment : fort de cette abstraction, on peut donc représenter le rationnel r comme une *paire* (a,b).

Mais on utilise finalement assez peu les nombres rationnels, et on préfèrerait pouvoir écrire des « nombres à virgule » plus généraux en binaire. Par exemple, de même que le nombre 3,14 vaut $3+\frac{1}{10^1}+\frac{4}{10^2}$, en binaire on peut écrire le nombre à virgule 1,011 valant $1+\frac{0}{2^1}+\frac{1}{2^2}+\frac{1}{2^3}$. On est cependant encore limité. Par exemple, les physiciens ont souvent besoin de pouvoir encoder des grands entiers ou des nombres rationnels sous la forme de leur notation scientifique, tel que le nombre d'Avogadro par exemple qui décrit le nombre d'entités élémentaires (atomes, molécules ou ions) dans une *mole* de matières :

$$N_A = 6{,}022\ 140\ 857 \times 10^{23}\,\mathrm{mol}^{-1}$$

ou la constante universelle de gravitation apparaissant dans la loi universelle de la gravitation de Newton :

$$G = 6{,}674\ 08 \times 10^{-11}\,\mathrm{m}^3\,\mathrm{kg}^{-1}\,\mathrm{s}^{-2}$$

Pour stocker de tels nombres, on utilise la notation scientifique en binaire, appelée *représentation flottante*, pour approcher des nombres réel.

Définition 2. On considère donc des réels pouvant s'écrire sous la forme

$$s\ m \times 2^k$$

avec s le signe (+ ou -) du réel, m un nombre à virgule compris entre 1 inclus et 2 exclus, et k son exposant. En *simple précision* (c'est-à-dire quand on se réserve 32 bits pour stocker le nombre flottant),

— 1 bit est utilisé pour représenté le signe s (0 pour le signe +, 1 pour le signe -) ;

— 8 bits pour l'exposant k : l'exposant est un entier relatif entre -126 et 127 qu'on représente par l'entier naturel $k+127$ qui est donc compris entre 1 et 254 (avec 8 bits, on peut aussi représenter les deux exposants 0 et 255, qui sont cependant réservés pour des situations exceptionnelles telles que $+\infty$, $-\infty$, etc.) ;
— et 23 bits pour le nombre m qu'on appelle *mantisse* : puisqu'on représente m en binaire et qu'on le choisit dans l'intervalle $[1;2[$, le nombre avant la virgule vaut toujours 1 et on ne le stocke donc pas en machine, utilisant ainsi les 23 bits pour les chiffres après la virgule.

Par exemple, la séquence de bits

$$10101001111001000110000000000000$$

est la représentation flottante du réel $-\frac{1827}{1024} \times 2^{-44} \approx -1{,}01 \times 10^{-13}$ puisque :
— le signe est encodé par le premier 1, et est donc - ;
— l'exposant est encodé par les huit bits suivants, 01010011, représentation binaire de l'entier 83, impliquant que l'exposant vaut $83 - 127 = -44$;
— la représentation en binaire de la mantisse est 1,11001000110000000000000 qui vaut :

$$1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^5} + \frac{1}{2^9} + \frac{1}{2^{10}} = \frac{2^{10} + 2^9 + 2^8 + 2^5 + 2 + 1}{2^{10}} = \frac{1827}{1024}$$

Mais pourquoi fait-on ce décalage de +127 pour calculer l'exposant ? On a vu que l'exposant est codé sur 8 bits : on a donc toutes les possibilités entre 00000000 et 11111111, c'est-à-dire entre 0 et 255. On a ainsi 256 codages possibles pour l'exposant. Puisqu'on veut pouvoir avoir des exposants positifs et d'autres négatifs, on répartit ces 256 codages possibles entre les négatifs et les positifs : on choisit arbitrairement d'aller de -127 à 128. Oui mais voilà, la convention IEEE-754, qui a en charge de mettre tout le monde d'accord sur le format des nombres flottants, en a décidé autrement : elle s'est réservée les deux extrémités (-127 et 128) pour des usages exceptionnels, en particulier pour écrire le nombre 0,0 (on décide alors de l'encoder avec 32 bits à 0), des flottants qui ne sont pas des nombres (lorsqu'on vient de faire une division par zéro interdite par exemple) et + ou $-$ l'infini... Ainsi, on se restreint à l'intervalle $[-126;127]$. Maintenant, il faut représenter ça sur 8 bits : pour faire simple, on décale tout de 127, pour arriver à l'intervalle [1,254] qui est donc codé par les codages binaires de 00000001 à 11111110 (réservant les codes extrémaux 00000000 et 11111111 pour les usages exceptionnels).

Exercice 5

Trouver le réel représenté par la séquence 01001110001100110100000000000000.

Exercice 6

Comment représente-t-on en binaire le réel 2^{-126} (qui est proche de $1{,}18 \times 10^{-38}$) sur 32 bits ? Et l'entier 7 ? Et le réel 7,0 sur 32 bits ?

Attention, choisir de représenter un nombre à l'aide d'un codage à virgule (flottante) induit nécessairement des imprécisions qui peuvent être préjudiciables si on n'y prend pas garde. Par exemple, considérons le nombre 0,1. Comment s'écrit-il en binaire ? Il faut qu'on essaie de l'écrire avec un nombre à virgule, c'est-à-dire comme une somme de puissance de 2. Puisque $0{,}1 \in [0{,}0625; 0{,}125] = [\frac{1}{2^4}; \frac{1}{2^3}]$, le premier 1 dans l'écriture à virgule est à la quatrième

position : l'écriture commence donc par 0,0001... et l'imprécision restante est de $0{,}1-0{,}0625 = 0{,}0375$. Puisque $\frac{1}{2^5} = 0{,}03125$, la prochaine puissance de 2 doit être prise : l'écriture binaire commence donc par 0,00011... et il reste $0{,}0375 - 0{,}03125 = 0{,}00625 = 0{,}1 \times \frac{1}{2^4}$. On voit réapparaître le nombre 0,1 duquel on était parti, signe qu'on s'apprête à écrire un nombre infini de chiffres après la virgule (comme lorsqu'on essaie d'écrire le rationnel 1/3 comme un chiffre à virgule en décimal...). Pour être plus précis, résumons nos calculs précédents avec l'équation

$$0{,}1 = \frac{1}{2^4} + \frac{1}{2^5} + 0{,}1 \times \frac{1}{2^4}$$

dans laquelle on peut remplacer le 0,1 à droite par cette même écriture

$$0{,}1 = \frac{1}{2^4} + \frac{1}{2^5} + \left(\frac{1}{2^4} + \frac{1}{2^5} + 0{,}1 \times \frac{1}{2^4}\right) \times \frac{1}{2^4} = \frac{1}{2^4} + \frac{1}{2^5} + \frac{1}{2^8} + \frac{1}{2^9} + 0{,}1 \times \frac{1}{2^8}$$

et ainsi de suite, de sorte qu'on trouve finalement le nombre binaire à virgule :

$$0{,}00011001100110011\ldots$$

On peut en déduire l'écriture en virgule flottante en faisant glisser le premier 1 avant la virgule, de sorte que 0,1 vaut, en binaire,

$$1{,}1001100110011001100110011\ldots \times 2^{-4}$$

En simple précision, on a donc :
- un bit de signe à 0 puisque le nombre 0,1 est positif ;
- l'exposant qui vaut $-4 + 127 = 123$, encodé en binaire par les huit bits 01111011 ;
- les 23 bits de mantisse étant le début de ce qui suit la virgule ci-dessous, à savoir 10011001100110011001100.

La représentation en virgule flottante de 0,1 est donc

0 01111011 10011001100110011001100

En faisant cela, on a introduit une imprécision de l'ordre de $2^{-27} \approx 7 \times 10^{-9}$. En particulier, si on fait exécuter par une machine (Python par exemple) le calcul $0.1 + 0.1 + 0.1$, on ne trouvera pas la même chose que 0.3 : un test d'égalité entre ces deux valeurs renverra donc faux, à la surprise des programmeurs en herbe ! Rappelez-vous donc qu'on ne fait jamais de test d'égalité entre deux nombres à virgule flottante : à la place, on préfère tester si la différence entre ces deux nombres est suffisamment petite vis-à-vis de la précision voulue.

Exercice 7

1. Quel est le plus grand entier qu'on peut représenter en binaire sur 32 bits ?
2. Quel est le plus grand réel qu'on peut représenter sur 32 bits ? Et le plus petit (négatif) ?
3. Quel est le plus petit réel strictement positif qu'on peut représenter sur 32 bits ?

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

FIGURE 1.7 – Table ASCII

1.2.4 Codage de texte

La prochaine étape, une fois qu'on sait stocker en machine des nombres, consiste à pouvoir stocker du texte, que ce soit le contenu d'un livre, un e-mail ou bien un mot de passe. Pour cela, faisons appel à l'abstraction : pour ne pas tout reprendre depuis le début, on peut utiliser le fait que nous savons désormais coder des entiers. Ainsi, on représente chaque caractère imprimable (« A », « h », « $ », mais aussi « 7 » ou « + ») par un entier. Il s'agit donc de se mettre d'accord afin que tout le monde utilise un codage que les autres peuvent comprendre. Le codage le plus simple consiste à utiliser la table ASCII (c'est l'acronyme de *American Standard Code for Information Interchange*). Cette table contient 128 caractères chacun associé avec un code (décimal) entre 0 et 127 : son contenu est représenté dans la Figure 1.7. Par exemple, la lettre « A » est codée par l'entier 65. Dans la machine, cet entier est évidemment stocké en binaire : son code binaire est 1000001. Le symbole « # » est représenté par l'entier 35. La table ASCII contient également le code d'éléments utiles pour coder du texte ou des touches du clavier (la touche « escape », le « retour à la ligne » par exemple ou l'« espace », de codes ASCII respectifs 27, 13 et 32).

Puisqu'on sait coder les caractères, on sait aussi coder une *chaîne de caractères*, c'est-à-dire du texte. Ainsi la phrase « Dessine-moi un mouton. » est codée par la séquence d'entiers :

68 101 115 115 105 110 101 45 109 111 105 32 117 110 32 109 111 117 116 111 110 46

En réalité, évidemment, on ne stocke pas des entiers en décimal, mais bien des entiers codés en binaire, chacun sur 7 bits : par exemple, le tiret « - » est donc codé par la séquence de bits 0101101, où on a donc ajouté un 0 en premier pour utiliser les 7 bits à disposition. Puisque chaque caractère est codé sur 7 bits, on a donc pas besoin de « séparer » les codages de chaque caractère (comme ci-dessus).

Il existe d'autre façons de coder des caractères : citons par exemple d'autres tables, telles que l'UTF-8 (ou UTF-16, ou UTF-32) ou sa généralisation, l'Unicode, permettant de coder bien davantage de caractères (les caractères accentués ou les caractères d'autres alphabets que l'alphabet latin).

lettre	codage variable
a	1010
b	0010011
c	01001
d	01110
e	110
f	0111100
g	0111110
h	0010010
i	1000
j	011111110
k	011111111001
l	0001
m	00101
n	1001
o	0000
p	01000
q	0111101
r	0101
s	1011
t	0110
u	0011
v	001000
w	011111111000
x	01111110
y	0111111111
z	01111111101
espace	111

lettre	codage fixe
a	00000
b	00001
c	00010
d	00011
e	00100
f	00101
g	00110
h	00111
i	01000
j	01001
k	01010
l	01011
m	01100
n	01101
o	01110
p	01111
q	10000
r	10001
s	10010
t	10011
u	10100
v	10101
w	10110
x	10111
y	11000
z	11001
espace	11010

FIGURE 1.8 – Deux codages possibles des lettres de l'alphabet

Exercice 8

Dans cet exercice, nous allons considérer les deux codages de la Figure 1.8 pour les 27 symboles d'un texte (les 26 lettres de l'alphabet plus l'espace) qui associent à chaque symbole un mot binaire. Par exemple le mot « patate » se code 01000 1010 0110 1010 0110 110 avec le codage variable et 01111 00000 10011 00000 10011 00100 avec le codage fixe. Il est à noter que les espaces entre les codes des différentes lettres sont présents uniquement pour la lisibilité et ne font pas partie du code.

1. Coder votre nom avec les deux codages. Quel est le codage qui utilise le plus de bits ?

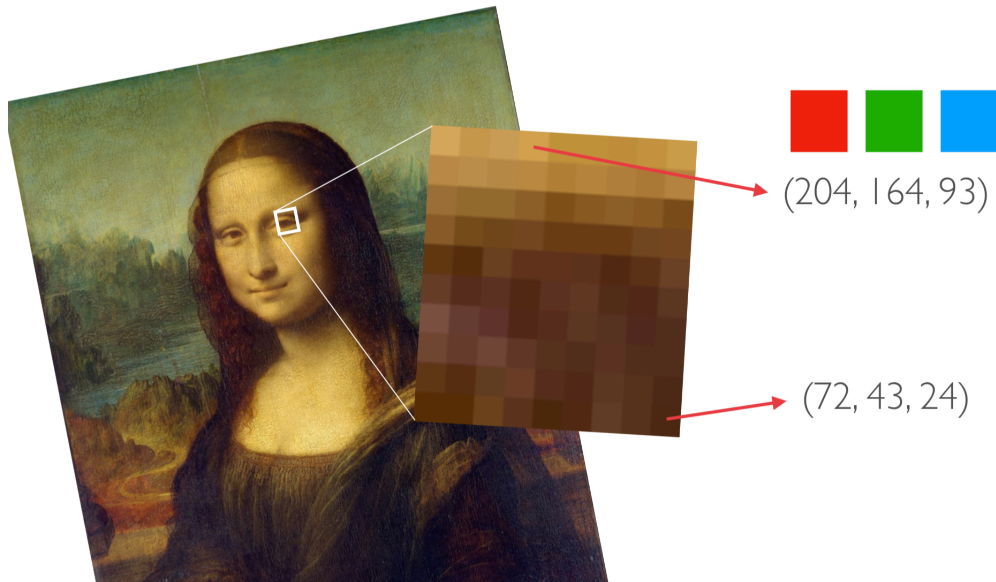


FIGURE 1.9 – Codage d'une image en format bitmap

2. Décoder le message 0111011001001000001110110 avec les deux codages.
3. Décoder le message 110100010001101010000011001100000001101011010 avec les deux codages.
4. Décoder le message 1000100101111000000001010010110100110100000001001 avec le codage variable.
5. Si on change le troisième bit du message de la question précédente que se passe-t-il ? Que devient le message si on change le deuxième bit ? le douzième bit ?
6. D'après vous, quels sont les avantages et les inconvénients du codage variable par rapport au codage fixe ?

1.2.5 Codage d'images

On sait désormais coder un roman, mais pas une bande dessinée : il nous manque la possibilité de coder des images. Un format simple (mais gourmand en espace) consiste à représenter une image comme un tableau bidimensionnel (une matrice) : chaque élément du tableau est alors appelé un *pixel*. Le codage d'un pixel consiste à représenter la couleur de la zone correspondante de l'image. Pour ce faire, on utilise généralement trois entiers qui représentent les quantités (entre 0 et 255) de rouge, de vert et de bleu dans la couleur. Par exemple, dans la Figure 1.9, une zone de l'œil de la Joconde est agrandie : cette zone comporte 9 colonnes et 10 lignes de pixels. Deux de ces pixels sont détaillés à droite. Celui du haut a une couleur marron pale qui est codée par le triplet (204, 164, 93) : la couleur est donc constituée de rouge à hauteur de $204/(204 + 164 + 93) = 44{,}25\%$. Ainsi, la couleur verte à 100% sera représentée par le triplet (0, 255, 0).

Notons que $255 = 2^8 - 1$. Ainsi, la représentation de 255 en binaire est 11111111. Pour

coder un entier entre 0 et 255, on a donc besoin de 8 bits. Cette quantité de bits se retrouve souvent en informatique :

Définition 3. Une séquence de 8 bits s'appelle un *octet*.

On utilise les octets pour rendre également plus lisible la représentation des entiers. Par exemple, l'entier 10^8 a pour représentation binaire 101111101011110000100000000. C'est bien difficile à lire ! De la même façon qu'on peut séparer les chiffres par groupes de 3 dans l'écriture décimale d'un nombre (par exemple, 100 000 000), on choisit de séparer les bits de la représentation binaire par groupes de 8 : 101 11110101 11100001 00000000. On utilise souvent l'octet comme unité de capacité mémoire de disques dans le commerce : 256 Go signifie ainsi 256 Giga octets, soit 256 milliards d'octets, ce qui représente donc $256 \times 8 = 2048$ milliards de bits.

Exercice 9

Environ 25 000 étudiants sont inscrits à l'UFR Sciences de l'université d'Aix-Marseille. Un numéro est attribué à chaque étudiant. Bien évidemment, deux étudiants différents ne doivent pas avoir le même numéro.

1. Combien de bits sont nécessaires pour coder un de ces numéros ?
2. Les logiciels utilisés ont l'octet comme unité de mémoire. Combien d'octets sont nécessaires pour coder un numéro d'étudiant ?
3. L'administration de l'université souhaite que les programmes mis au point pour gérer l'inscription et la scolarité des étudiants puissent servir 10 ans. Si l'on pense que la fréquentation de l'université peut augmenter au plus de 20% chaque année, combien d'octets faut-il réserver pour coder les numéros des étudiants ?

Exercice 10

Le cerveau humain a environ 100 milliards de neurones qui ont chacun en moyenne 10 000 synapses transmettant environ 100 impulsions binaires par seconde. Calculer la quantité d'information maximale transmise par seconde dans un cerveau, en octets.

Revenons au codage des images. Stocker une image de 1920 par 1200 pixels nécessite de représenter $1920 \times 1200 = 2\ 304\ 000$ pixels, chacun prenant 3 octets en mémoire : au total, cette image prend donc 6 912 000 octets, soit 55 296 000 bits, ce qui équivaut à 55 Mégabits. C'est beaucoup pour une simple image... En pratique, on ne stocke donc que rarement les images en format bitmap : on utilise plutôt des formats *compressés* qui essaient d'épargner de la mémoire en profitant de redondances dans l'image ou en supprimant des détails invisibles à l'œil nu.

Exercice 11

On souhaite effectuer des dessins sur une grille carrée comprenant 64x64 cases (ou pixels). Chaque point est repéré par un couple d'entiers (x,y) où x et y sont compris entre 0 et 63. On supposera que x et y sont écrits en binaire.

1. Combien faut-il de bits pour coder un point ?

   On suppose que les dessins seront composés de trois types d'éléments : des segments, des rectangles dont les côtés sont parallèles aux axes et qui pourront être pleins ou vides. Ainsi, le dessin de la Figure 1.10 est composé d'un rectangle vide $ADJI$, d'un rectangle plein $BCFE$ et de deux segments GK et KH.

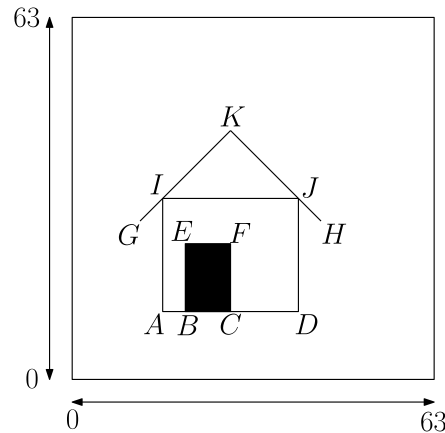


Figure 1.10 – Dessin d'une maison

— Un segment AB dont les extrémités ont pour coordonnées (x_A,y_A) et (x_B,y_B) sera représenté par 4 octets $01x_A$ $01y_A$ $01x_B$ $01y_B$.
— Un rectangle vide $ABCD$ dont deux sommets opposés sont A et C sera représenté par 4 octets $10x_A$ $10y_A$ $10x_C$ $10y_C$.
— Un rectangle plein $ABCD$ dont deux sommets opposés sont A et C sera représenté par les 4 octets $11x_A$ $11y_A$ $11x_C$ $11y_C$.
— Un dessin est représenté par la suite des représentations des éléments qui le compose.

On remarque donc qu'on peut savoir si un octet code un élément d'un segment s'il commence par 01, un élément d'un rectangle vide s'il commence par 10 et un élément d'un rectangle plein s'il commence par 11.

2. Combien faut-il de bits pour représenter le dessin de la figure ? Combien cela fait-il en octets ?
3. Indiquez quel dessin est représenté par le codage suivant 10001111 10001111 10101111 10101111 01001111 01001111 01101111 01101111 01001111 01101111 01101111 01001111

On suppose que les $64 \times 64 = 4096$ pixels de la grille sont numérotés selon un ordre conventionnel (par exemple de gauche à droite et de haut en bas). Dans une représentation *bitmap*, un dessin (en noir et blanc) est représenté par une suite de bits $b_1 \cdots b_n$ dont le i-ième bit b_i est égal à 0 si le i-ième pixel du dessin est blanc et 1 s'il est noir.

4. Combien faut-il d'octets pour représenter le dessin de la figure dans une représentation *bitmap* ?

On souhaite enrichir les codages possibles en représentant directement des lignes brisées $A_1A_2 \ldots A_k$ (c'est-à-dire des réunions de segments A_1A_2, A_2A_3, ..., $A_{k-1}A_k$).

5. Indiquez comment on pourrait étendre le codage ci-dessus pour représenter de telles lignes brisées ? (*Indication : les extrémités de la ligne pourront être*

repérées par 01 et les points intérieurs par 00. Combien faudrait-il alors de bits pour représenter le dessin de la figure ?)

1.2.6 Codage de vidéos

Toujours grâce au principe d'abstraction, on utilise le fait qu'on sait désormais coder une image pour pouvoir coder une vidéo comme une séquence d'images (environ 24 par secondes, par exemple, afin que l'œil ne puisse pas distinguer les images qui défilent). Là aussi, on n'utilise pas cette représentation naïve en pratique, mais des formats compressés permettent de gagner de la place en mémoire. Pour stocker une vidéo, il faut aussi pouvoir stocker du son ce qui est un problème plus complexe encore qu'on ne traitera pas dans ce cours.

Exercice 12

On cherche à graver sur un bluray un flux vidéo non compressée. On suppose que le flux vidéo est de 1920×1080 pixels, que chaque pixel est codé sur 3 octets (codage rgb), qu'il y a 30 images par secondes et qu'un bluray peut stocker 50 Go de données.

1. Quelle est la durée maximale de vidéo que l'on peut stocker sur le bluray ? Qu'en conclure ?
2. On souhaite transmettre le flux vidéo via une connexion wifi d'un débit de 100 Méga bits par seconde. Quel est le nombre d'images que l'on peut transmettre par seconde via cette connexion ?

1.2.7 Une étude de cas

Considérons la situation suivante que nous avons abordé plus tôt : supposons que vous venez visiter le centre de Marseille et que vous recherchez un restaurant pour pouvoir manger le midi. Vous utilisez alors un moteur de recherche tel que Google Maps qui vous permet de visualiser l'ensemble des restaurants proches de vous sur une carte (cf Figure 1.11).

Quelles données sont utilisées lors de cette recherche ? Il y en a beaucoup, en voilà quelques exemples :

— des entiers : un nombre d'étoiles pour chaque restaurant, ou une note attribuée par chaque utilisateur ;

— des flottants : les distances sur le plan affiché, ou une estimation du temps pour parcourir un certain chemin dans le cas où on demande ensuite un itinéraire ;

— du texte : le nom de chaque restaurant, mais aussi des rues de la ville ;

— des images : des photos de la rue ou des plats, ajoutées par les utilisateurs, ou des images satellites selon la vue choisie dans l'application de plans ;

— des sons : si on choisit de se laisser guider par le GPS de l'application, la voix des instructions à suivre.

1.3 La science du calcul

Maintenant que l'on sait comment stocker de l'information, grâce à des codes, il nous reste à comprendre ce qu'on va *calculer* sur ces données. En guise d'introduction, sortons nos règles et nos compas.

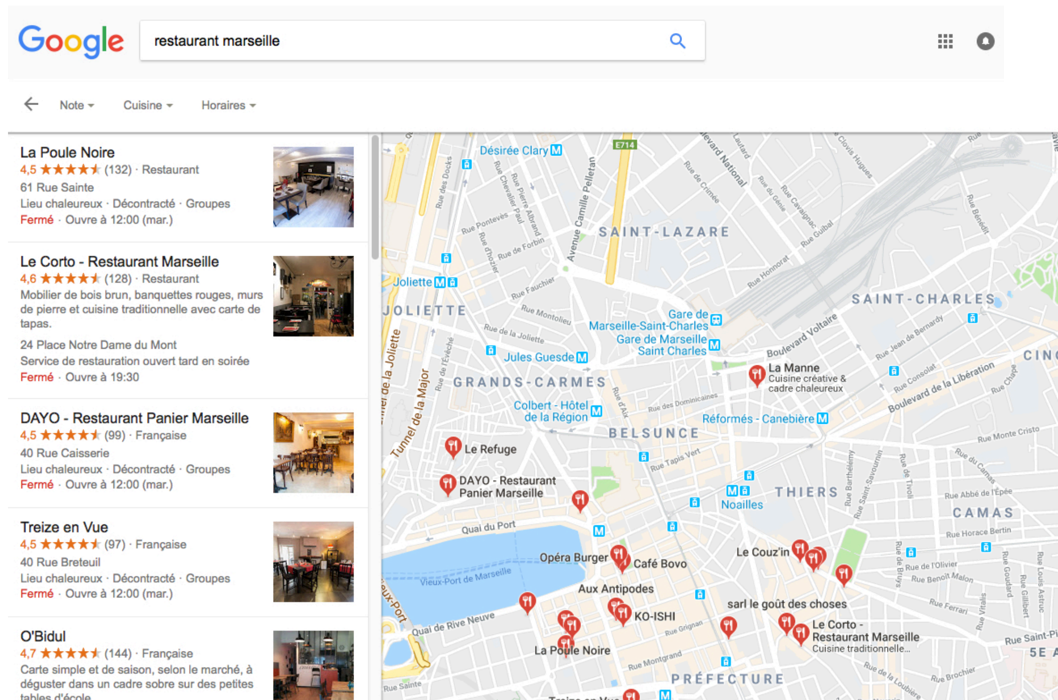


FIGURE 1.11 – Recherche d'un restaurant à Marseille dans Google Maps

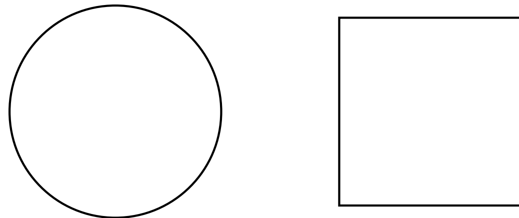


FIGURE 1.12 – Tracer un carré de même aire qu'un cercle donné

1.3.1 Pilier numéro 1 : la calculabilité

On se donne ainsi une règle et un compas. À l'aide de mon compas, je trace un cercle. Votre objectif est de tracer un carré ayant la même surface que mon cercle, uniquement à l'aide de votre règle (sans mesures) et de votre compas, tel que représenté en Figure 1.12.

Traduit dans le formalisme informatique, on a un problème qui se décompose de la manière suivante :

— l'entrée du problème est un cercle d'aire a fixé ;
— on se donne un ensemble d'*opérations élémentaires* autorisées, ici la règle et le compas ;
— la sortie attendue est un carré de même aire a.

La première question à se poser est de savoir si on peut résoudre ce problème. Après plusieurs centaines d'années de recherche sans succès, la solution a été trouvée en 1882 par Ferdinand von Lindemann. C'est le problème de la quadrature du cercle dont on sait désormais qu'il n'admet pas de solution : il est *impossible* de construire un carré de même aire qu'un cercle donné uniquement avec une règle et un compas. Cela illustre le premier pilier de la science informatique :

> « Étant donné un problème avec des entrées et des opérations élémentaires autorisées, **peut-on calculer** le résultat ? »

Il y a des résultats qu'on peut calculer... et d'autres non !

Mais y a-t-il vraiment des problèmes interessants qui sont *incalculables* (on dit aussi *indécidables*) ? La réponse est oui, et beaucoup même ! C'est ce qu'on verra à la fin de ce cours. Un exemple célèbre est celui du problème de l'arrêt d'une machine de Turing (le même Turing que celui dont nous avons parlé plus tôt) : on ne peut pas décider si une machine de Turing (ou un programme) termine ou tourne en boucle.

1.3.2 Pilier numéro 2 : la complexité temporelle

Modifions donc le problème pour qu'on ait plus de chance de pouvoir le résoudre. À l'aide de la règle et du compas, je trace un carré. Votre nouvel objectif est de tracer un carré qui a 4 fois la surface du carré précédent, toujours à l'aide uniquement d'une règle (sans mesures) et d'un compas. Le problème est illustré en Figure 1.13.

FIGURE 1.13 – Un carré et un carré 4 fois plus grand

Essayez de résoudre ce problème par vous-même.

La Figure 1.14 représente trois méthodes différentes (ce ne sont pas les seules !). Cela assure donc que le problème admet une solution : il est *calculable*. Par contre, cela pose question quant à la complexité des différentes solutions. Ainsi, la construction de gauche utilise 8 cercles et 4 segments, alors que celle du milieu n'utilise que 5 cercles, mais 6 segments : pour choisir entre les deux solutions, il faut donc savoir quel est le coût de chacune des deux opérations (tracer un trait à la règle ou tracer un cercle au compas). Par contre, il est facile de comparer les deux solutions les plus à droite : en effet, la construction de droite utilise 3 cercles et 6 segments et est donc meilleure que la solution du milieu à tout point de vue. Cela illustre le second pilier de la science informatique :

> « Étant donné un problème avec des entrées et des opérations élémentaires autorisées, **en combien de temps** peut-on calculer le résultat ? »

Il y a des résultats qu'on sait calculer efficacement... et d'autres non !

On peut se demander quel est l'intérêt de s'embêter avec ce pilier de complexité temporelle vu les machines toujours plus puissantes qui sont mises à notre disposition pour résoudre des problèmes. En pratique, pour des problèmes difficiles ou pour des grandes instances, la complexité sera déterminante pour savoir si on arrive un jour à obtenir le résultat ou pas. Par exemple, imaginons qu'on veuille trier un tableau d'entiers via le calcul de toutes ses permutations :

— si on veut trier un tableau de 20 entiers, il faut alors calculer $2{,}43 \times 10^{18}$ permutations dans le pire des cas, ce qui demande 26 ans environ sur un ordinateur de fréquence 3 GHz (c'est-à-dire qui réalise 3 milliards d'opérations par seconde) ;

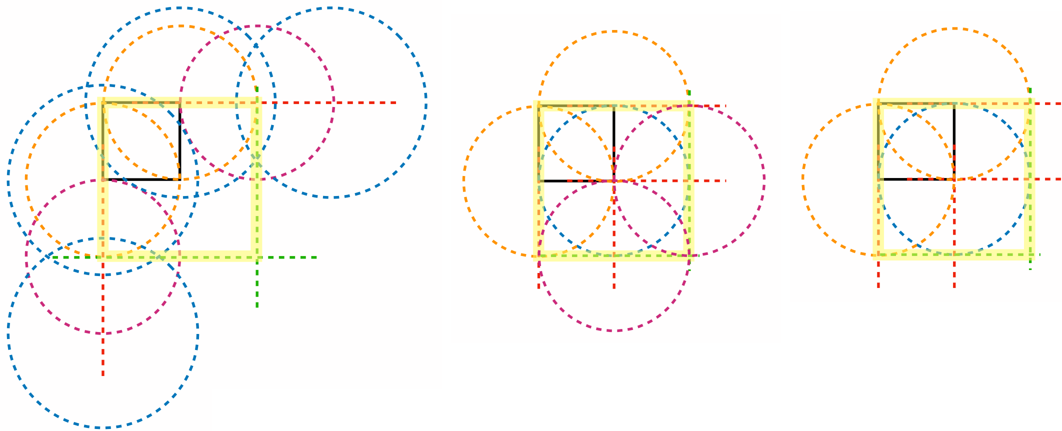


FIGURE 1.14 – Trois méthodes pour obtenir un carré 4 fois plus grand : le carré initial est en noir, le carré final est surligné en jaune

- si on veut trier un tableau de 30 entiers, il faut calculer $2{,}65 \times 10^{32}$ permutations dans le pire des cas, ce qui demande 3×10^{15} ans environ sur un ordinateur de fréquence 3 GHz, déjà plus long que l'âge de l'univers ;
- si on veut trier un tableau de 100 entiers (par exemple, si on veut trier les 100 restaurants à Marseille par distance croissante à notre position, lors de notre recherche sur internet), il faut calculer $9{,}33 \times 10^{157}$ permutations dans le pire des cas, ce qui demande 10^{141} ans environ sur un ordinateur de fréquence 3 GHz.

On voit bien sur cet exemple qu'il nous faut mesurer la complexité de notre solution afin d'en trouver une meilleure pour trier un tableau d'entiers : nous étudierons de meilleures méthodes pour cela, plus tard dans ce cours.

1.3.3 Pilier numéro 3 : la complexité spatiale

Dans la Figure 1.14, on peut aussi comparer les différentes constructions en fonction de la quantité de papier nécessaire pour tracer les dessins intermédiaires : la construction de gauche requiert d'avoir une feuille beaucoup plus grande que la construction de droite, par exemple. Cela illustre le troisième pilier de la science informatique :

> « Étant donné un problème avec des entrées et des opérations élémentaires autorisées, **combien d'espace** est utilisé pour calculer le résultat ? »

Il y a des résultats qu'on sait calculer avec peu d'espace (peu de mémoire)... et d'autres non !

1.3.4 Résolution d'une tâche complexe

L'informatique consiste ensuite à appliquer ces trois piliers (calculabilité, complexité temporelle, complexité spatiale) à la résolution de *tâches complexes*. On se base sur l'*abstraction* pour oublier des détails peu importants. On *décompose* alors le problème en tâches plus simples. Puis on essaie de résoudre ces tâches simples et ciblées avec des *algorithmes*. Une illustration naïve de ces trois temps peut être donnée par une image de brins de laine emmêlés (cf Figure 1.15). Avant d'avoir tricoté le pull tricolore pour le petit dernier, il faut d'abord

FIGURE 1.15 – Tricot d'un pull tricolore à partir de brins de laine emmêlés

démêler les brins pour ne considérer que des pelotes de laine de différentes couleurs : c'est la phase d'abstraction. On décompose ensuite la tâche complexe de tricot d'un pull bicolore en tâche plus simples : il faudra ainsi tricoter une manche bleue, un corps gris et une manche rouge, avant de coudre le tout ensemble.

Des tâches complexes plus réalistes, en terme d'informatique, sont décrites en Figure 1.16 et 1.17. On ajoute à la phase d'abstraction et à l'algorithme à proprement parler une phase de visualisation qui est le résultat observé par l'utilisateur : un plus court chemin du campus St Charles au Vieux Port, ou le tri des restaurants de Marseille par leur note moyenne décroissante.

1.4 Algorithmes... ou algo-rythmes ?

Il nous reste donc à savoir comment résoudre les tâches simples qui proviendront de la décomposition précédente. Résoudre, ou plutôt *faire résoudre* automatiquement par une machine. La description d'une telle méthode de résolution est appelée un *algorithme*. L'étymologie de ce mot provient d'al-Khuwarizmi (cf Figure 1.18) un savant ayant vécu au début du IXème siècle à Bagdad. Son ouvrage *Abrégé du calcul par la restauration et la comparaison* est à l'origine de l'algèbre : il y décrit et classifie des algorithmes tels que celui d'Euclide pour le calcul du plus grand commun diviseur de deux entiers ou pour la résolution d'équations du second degré.

Proposons alors une définition de ce qu'est un algorithme :

> « Un algorithme est la description *non ambiguë* d'une séquence *finie* d'instructions permettant de résoudre un problème (informatique) ou d'obtenir un résultat. »

Cinq conditions (ainsi décrites par Donald Knuth, un chercheur contemporain en informatique) sont à réunir pour être sûr qu'on a bien un algorithme :

1. **finitude** : un algorithme doit terminer après un nombre fini d'étapes ;
2. **définition précise** : un algorithme doit être décrit sans ambiguïté, c'est-à-dire sans laisser de place au doute ;
3. **entrées** : un algorithme peut avoir des données sur lesquelles il va travailler ;
4. **sorties** : un algorithme a généralement un résultat qu'on attend de lui ;

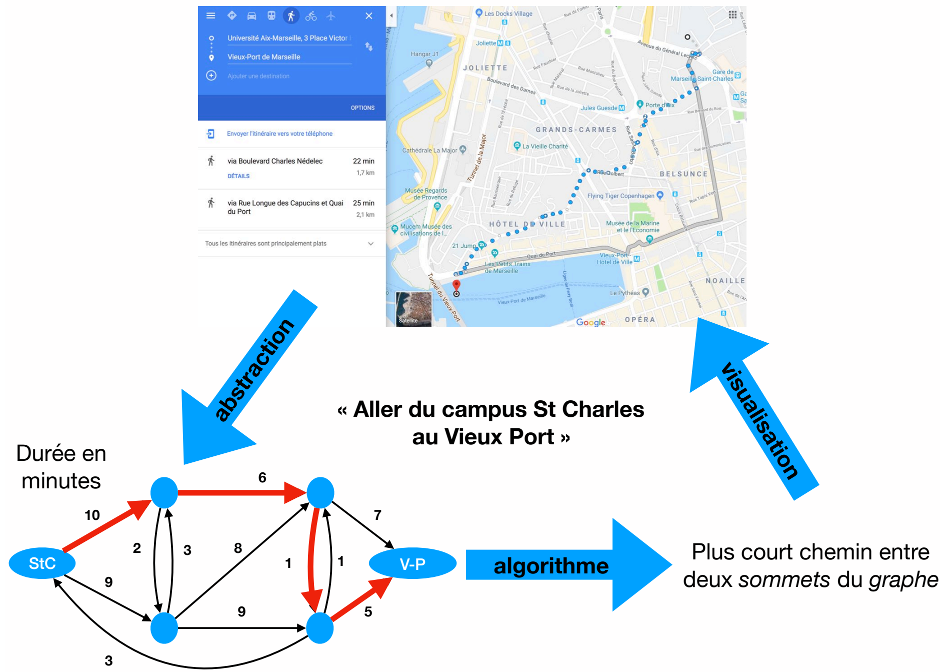


FIGURE 1.16 – Calcul du plus court chemin dans Google Maps : on abstrait le problème à l'aide d'un graphe avec des nœuds représentant les intersections de rue et des arcs reliant ces nœuds avec la durée en minutes du trajet correspondant

FIGURE 1.17 – Tri des restaurants par note moyenne décroissante : la seule information pertinente à conserver ici est le nom du restaurant et sa note moyenne. La visualisation consiste alors à réordonner effectivement les restaurants une fois qu'ils ont été triés

al-Khuwarizmi

أبو عبد الله محمد بن موسى الخوارزمي

FIGURE 1.18 – Le savant al-Khuwarizmi, premier auteur d'algorithmes

5. **rendement** : un algorithme doit utiliser des opérations basiques, généralement telles qu'un être humain puisse les exécuter (il n'est donc pas autorisé d'utiliser une opération qui consisterait à construire un carré de même aire qu'un cercle donné, à l'aide d'une règle et d'un compas !).

1.4.1 Dénombrement

Illustrons le concept d'algorithme à l'aide d'un problème concret, celui de compter le nombre de personnes dans une salle (par exemple une salle de concert ou un amphithéâtre à l'université). Il existe de multiples méthodes pour opérer ce dénombrement. Le plus simple est qu'une personne compte un par un les personnes de la salle. Il peut aussi choisir de compter les personnes deux par deux, ou même cinq par cinq. Le nombre d'*opérations élémentaires* qu'il effectue diffère dans ces trois cas :

— lorsqu'il compte un par un, il effectue autant d'additions qu'il y a de personnes dans la salle, disons n personnes ;
— lorsqu'il compte deux par deux, il effectue $n/2$ additions ;
— lorsqu'il compte cinq par cinq, il effectue $n/5$ additions.

Au prix de savoir faire des additions deux par deux ou cinq par cinq rapidement, on voit bien qu'on effectue moins de calculs en comptant cinq par cinq qu'en comptant deux par deux, et que cette dernière technique effectue moins de calculs que celle qui consiste à compter un par un. Une représentation du nombre d'opérations en fonction de n peut être vue en Figure 1.19.

Une autre méthode de dénombrement consiste, si les personnes sont assises, à compter le nombre R de rangées de sièges et à estimer le nombre M moyen de personnes par rangée : il ne reste alors plus qu'à effectuer la multiplication $R \times M$ pour obtenir une estimation du nombre de personnes. Contrairement aux méthodes précédentes, celle-ci n'apporte qu'une réponse approximative : elle n'est pas (absolument) correcte. Par contre, elle est bien plus

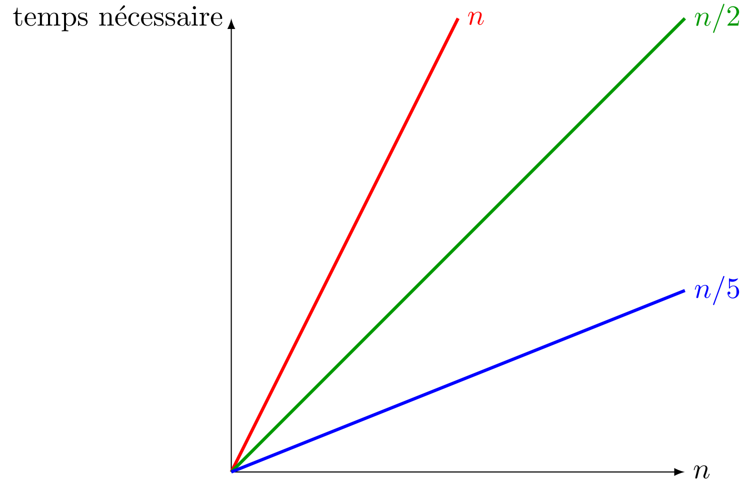


FIGURE 1.19 – Nombre d'opérations des différents algorithmes en fonction du nombre de personnes n dans la salle

rapide encore.

Dans les méthodes précédentes, plutôt que de tout faire tout seul, la personne en charge du comptage pourrait se faire aider : plus on est nombreux à travailler, moins de temps la tâche prendra. En informatique, c'est l'utilisation des multiples *cœurs* d'un processeur que nous pouvons utiliser en parallèle pour accélérer la résolution d'un problème.

Une autre façon de paralléliser, beaucoup plus massive, consiste à distribuer le calcul sur les personnes présentes dans la salle. Considérons ainsi l'algorithme suivant :

Entrée : des personnes debout dans une salle
- Chaque personne a en tête le nombre 1
- **Tant qu**'il reste au moins deux personnes debout :
  - chaque personne encore debout cherche du regard une autre personne debout
  - les deux personnes s'échangent le nombre qu'ils ont en tête (indépendamment des autres personnes)
  - l'une des deux personnes s'assoit ; l'autre additionne les deux nombres et reste debout

Sortie : la dernière personne debout crie son nombre

Cet algorithme est correct au sens où il se termine avec une seule personne encore debout, et que le nombre crié est effectivement le nombre de personnes dans la salle. Combien de temps prend-il pour terminer ? Cette fois-ci, plutôt que le nombre total d'additions effectuées, une meilleure estimation du temps d'exécution consiste à compter le nombre d'*itérations* effectuées par l'algorithme. Dis autrement, si on suppose que toutes les secondes, chaque personne encore debout en a trouvé une autre et que l'une des deux s'est assise, alors il s'agit de compter le nombre de secondes avant la fin de l'algorithme. Il n'est pas très difficile de se convaincre qu'à chaque seconde, la moitié environ des personnes encore debout s'assoit. Par conséquent, après k secondes, le nombre de personnes encore debout a été divisé par 2^k. Lorsque k devient supérieur à $\log_2(n)$, le nombre de personnes debout a été divisé par $2^{\log_2(n)} = n$: il ne reste alors plus qu'une seule personne encore debout. Ainsi, il faut $\log_2(n)$ secondes pour que cet

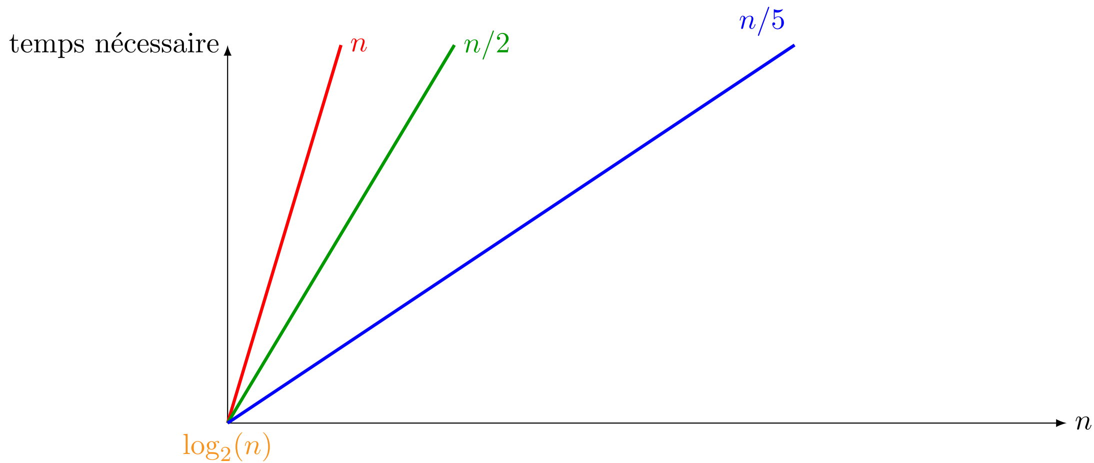


FIGURE 1.20 – Comparaison des fonctions linéaires (en rouge, vert et bleu) avec la fonction logarithme (en orange)

algorithme termine. La fonction logarithme croit beaucoup plus lentement que les fonctions linéaires trouvées précédemment. Dans la Figure 1.20, on peut observer que lorsque n est suffisamment grand, la courbe de la fonction $\log_2$ passe en dessous des courbes des fonctions $n \mapsto n$, $n \mapsto n/2$ et $n \mapsto n/5$. En particulier, s'il y a 200 personnes dans la salle et que chaque opération élémentaire (addition ou itération) nécessite une seconde :

— la méthode consistant à compter un par un nécessite 200 secondes, soit 3 minutes et 20 secondes ;
— la méthode consistant à compter deux par deux nécessite 100 secondes, soit 1 minutes et 40 secondes ;
— la méthode consistant à compter cinq par cinq nécessite 40 secondes ;
— la méthode distribuée nécessite $\log_2(200)$ secondes, soit 7 secondes environ : imbattable !

1.4.2 Comment (d)écrire des algorithmes ?

Maintenant que l'on sait ce que sont les algorithmes, il nous reste à savoir comment les décrire, tant à une personne qu'à un ordinateur. La méthode que nous avons utilisée plus haut pour la méthode de dénombrement distribuée consiste à une suite de phrases en langue naturelle. C'est proche de la façon dont nous écrirons des algorithmes tout au long de ce cours : on appelle cette description un *pseudo-code*. C'est tout à fait suffisant lorsqu'il s'agit de communiquer un algorithme à une autre personne. Cependant, cela ne suffit pas s'il faut le faire comprendre à une machine. Pour cela, on utilise des langages de programmation, tels que Python (que vous utiliserez dans l'UE *Mise en œuvre informatique*) ou Java (que vous utiliserez dans l'UE *Programmation 1*). Il existe aussi des langages de description graphique par blocs, tels que Scratch, permettant de manipuler le code à l'aide de blocs aimantés. D'autres descriptions graphiques existent et permettent de manipuler des algorithmes de manière visuellement pertinente. Un exemple parmi d'autres : les réseaux de tri, dont un exemple est décrit en Figure 1.21.

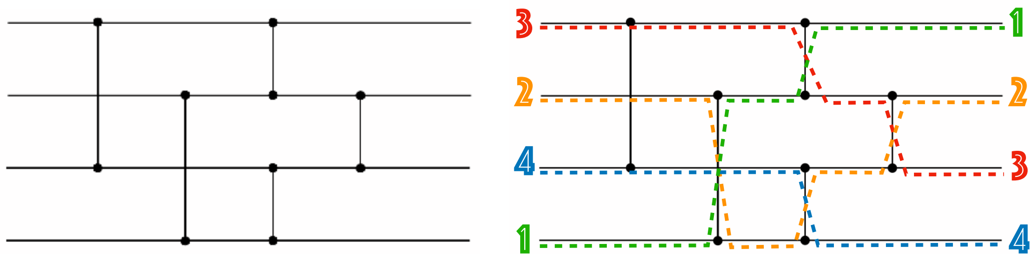


Figure 1.21 – Un réseau de tri (à gauche) et une des ses exécutions (à droite) : un réseau de tri est composé de fils horizontaux dans lesquels des entiers glissent de gauche à droite, et des fils verticaux, appelés comparateurs, qui permettent d'échanger les deux valeurs si jamais celle du haut est supérieure à celle du bas. Ce réseau est un réseau de *tri* dans le sens où tout quadruplet d'entiers placé à gauche ressort à droite du réseau de façon triée.

Chapitre 2

Description des algorithmes

Nous allons décrire les algorithmes à l'aide du langage Python dans la suite, sans être très strict, à l'écrit, sur une syntaxe parfaite. [1] Nous utiliserons des variables, chaque instruction élémentaire sera écrite sur une ligne séparée, le code sera indenté et nous découperons notre code en fonctions. Commençons par décrire dans ce chapitre les *structures de contrôle* permettant de structurer le pseudo-code.

2.1 Structures de contrôle : une introduction en Scratch

Nous allons décrire cinq types de structures de contrôle : itérations, fonctions, conditionnelles, écriture/lecture et variables. Pour motiver ce choix d'ingrédients de base, illustrons leur utilisation sur un petit exemple en Scratch : vous pouvez écrire et exécuter le code au fur et à mesure en utilisant l'éditeur Scratch en ligne, disponible dans l'onglet *Créer* du site `https://scratch.mit.edu`, et en chargeant le fichier donné en ligne sur le cours Ametice). L'objectif est de faire sortir un petit chat d'un labyrinthe très simple représenté en Figure 2.1, c'est-à-dire le faire atteindre la cible jaune en haut à droite du chemin blanc.

Après quelques essais, on trouve aisément une solution au problème, où la première ligne permet de dire à Scratch qu'on exécute le programme dès le début de l'exécution (c'est-à-dire

1. Jusqu'à l'an dernier, ce cours utilisait du pseudo-code pour écrire les algorithmes. Nous préférons changer à partir de maintenant, pour faciliter le lien avec l'UE de *Mise en œuvre informatique*.

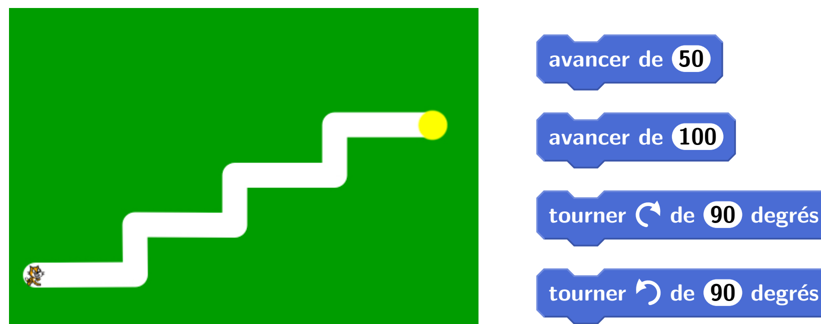


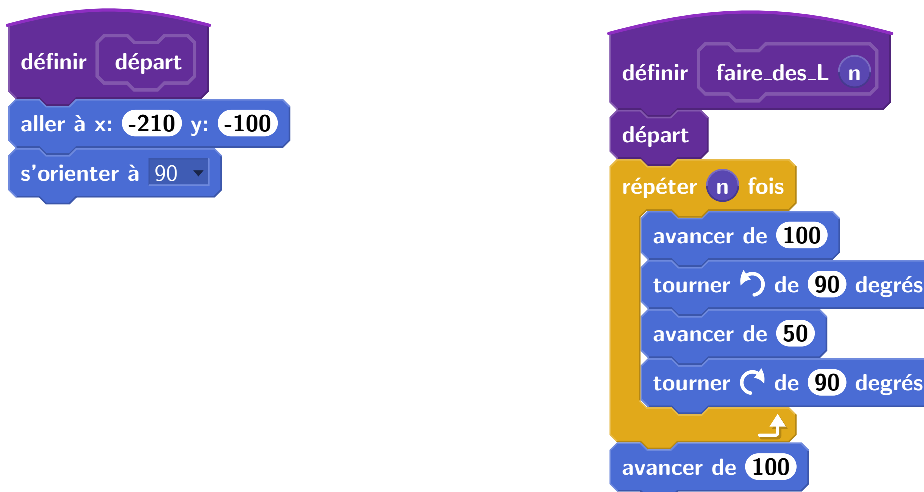
FIGURE 2.1 – Le labyrinthe d'où nous devons faire sortir le chat et les opérations élémentaires qu'on s'autorise

quand l'utilisateur appuie sur le bouton dans l'interface) :

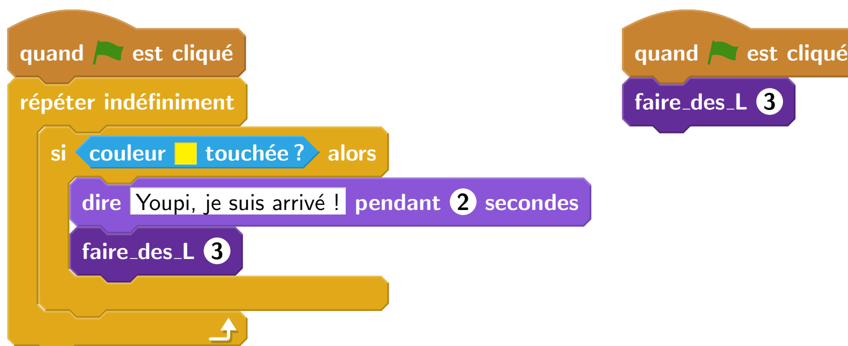
Ce code n'est que moyennement lisible et réutilise trois fois la même séquence d'opérations. On peut le simplifier grandement en utilisant une boucle permettant de répéter un certain nombre de fois la même séquence d'opérations :

Comment faire s'arrêter l'algorithme lorsqu'on atteint effectivement la cible de couleur jaune ? En Scratch, on peut décrire un autre bloc, indépendant du premier, qui exécutera en boucle un test, qu'on décrit à l'aide d'un bloc conditionnel :

Si, au contraire, on souhaite faire recommencer l'animation, il nous faut ré-exécuter le code précédent, plutôt que d'écrire le bloc d'arrêt. Pour éviter la recopie de code, on peut utiliser une fonction permettant de sauver un morceau de code qu'on peut ensuite réutiliser autant de fois que nécessaire en employant son nom. Ici, on a besoin de deux fonctions : une fonction qui remet le petit chat à sa position de départ (là encore, il faut quelques essais avant d'y parvenir...) et une fonction qui exécute la boucle précédente, en généralisant le nombre de « L » qu'il exécute en cas de modifications dans le futur.

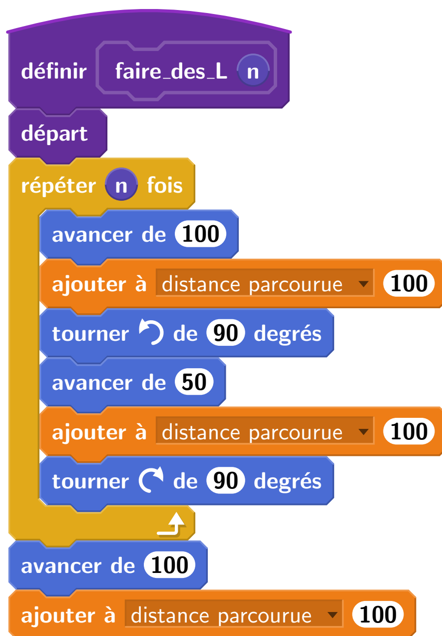


Le code principal se raccourcit alors beaucoup. Si de plus on fait *dire* au chat une petite phrase une fois qu'il a atteint la cible avant de repartir, le code devient

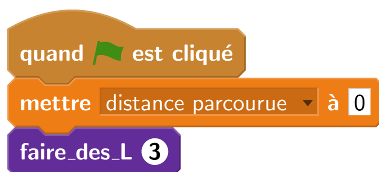


Il se peut que l'utilisateur ne veuille pas que ce code tourne indéfiniment, et donc avoir une possibilité de l'arrêter. Par exemple, on pourrait demander à l'utilisateur son avis, puis attendre sa réponse pour prendre une décision. En utilisant une conditionnelle supplémentaire pour tester sa réponse (qui est automatiquement stocké dans un bloc réponse) et réagir différemment si l'utilisateur souhaite continuer ou non, le code devient

Finalement, équipons le chat d'un podomètre, comptant le nombre de pas qu'il a effectué au total depuis le début de l'exécution du programme. Pour ce faire, il nous faut stocker ce nombre de pas dans ce qu'on appelle une *variable* : nommons-là ≪ distance parcourue ≫ pour clarifier sa signification. Une fois la variable créée, on peut la modifier et ajouter à son contenu une valeur entière, par exemple. Cela permet donc de modifier la fonction ≪ faire_des_L ≫ pour qu'elle enregistre dans la variable les modifications :



On n'oublie pas d'ajouter la remise à zéro de la variable dans le code principal :



Maintenant que nous avons vu l'utilité des différentes structures de contrôle en Scratch,

entrons dans le détail pour indiquer comment les décrire en Python.

2.2 Variables

L'utilisation de variables permet l'écriture de programmes stockant des données. Attention, le mot *variable* a deux sens, selon qu'on l'utilise dans son acception mathématique ou informatique :

— En mathématiques, une variables est une grandeur dont la valeur est (provisoirement) indéterminée, sur laquelle on effectue une combinaison d'opérations avec des constantes et d'autres variables. Par exemple, on peut considérer la variable x dans l'équation $x^2 - 3x + 2 = 0$. Elle n'a pas de valeurs, mais pourra en avoir une ou plusieurs une fois l'équation résolue : en l'occurrence, l'équation à deux solutions $x = 1$ ou $x = 2$.

— En informatique, une variable est un identifiant désignant un emplacement de la mémoire et son contenu peut donc évoluer au cours du temps. Si une variable n'est pas initialisée, sa valeur est temporairement non définie. Par exemple, on dénote par

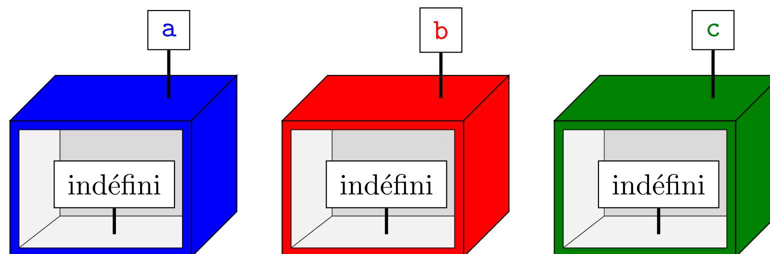
```
x = 1
```

l'affectation de la valeur 1 dans la variable `x`.

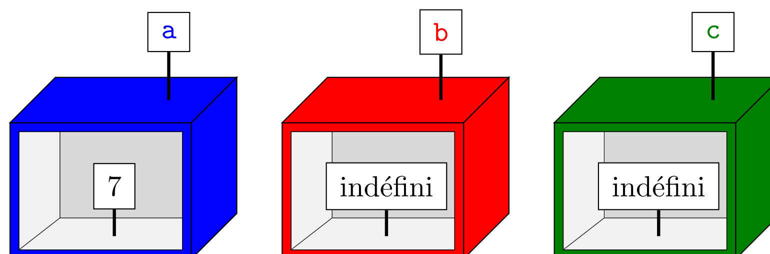
Considérons ainsi un programme qui enchaîne plusieurs affectations sur trois variables `a`, `b` et `c` :

```
a = 7
b = 3
c = b - a
a = a - c
```

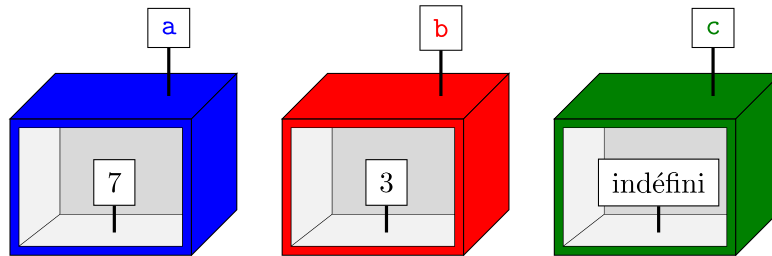
Au début de l'exécution du programme, le contenu de chaque variable est indéfini :



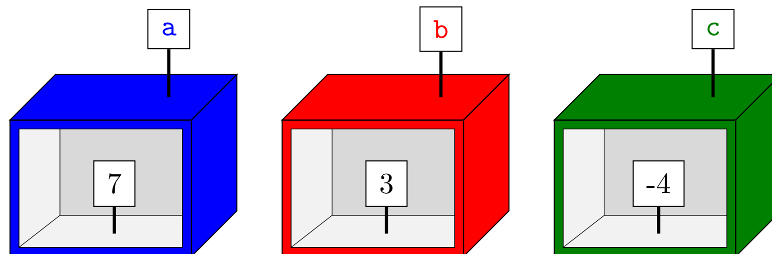
La première instruction `a = 7` s'exécute, modifiant le contenu de la variable `a` :



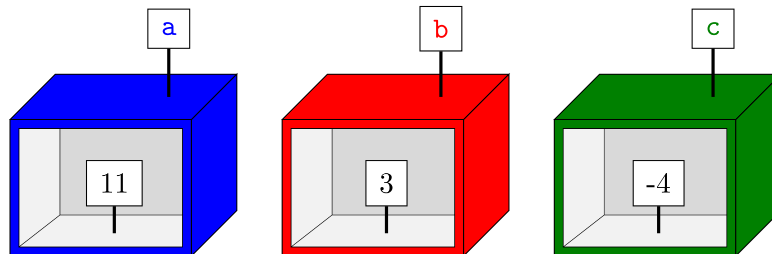
La seconde instruction `b = 3` s'exécute de même :



La troisième instruction `c = b − a` s'exécute alors en deux temps : d'abord les valeurs des variables `b` et `a` sont extraites, puis on effectue l'opération de soustraction avant de modifier le contenu de la variable `c` :



Finalement, la dernière instruction `a = a − c` commence par extraire les valeurs de `a` et `c`, calcule leur différence puis modifie le contenu de la variable `a` :



Notez que cela n'a donc rien à voir avec la résolution de l'équation mathématique

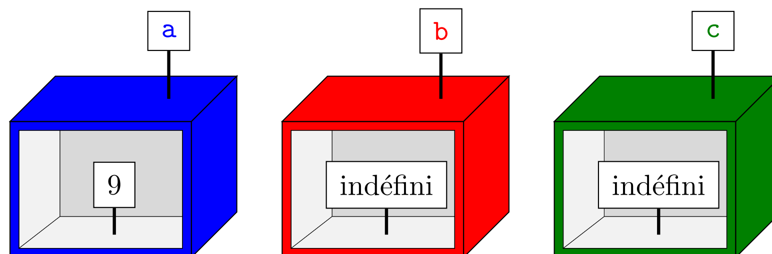
$$a = a - c$$

qui se résoudrait en $c = 0$...

Considérons un deuxième exemple de code :

```
a = 9
c = a + b
b = c
c = 2
```

Une fois la première ligne exécutée, on est dans la situation suivante :



Lorsqu'on essaie d'exécuter la second instruction `c = a + b`, on extrait les valeurs des variables `a` et `b` : on échoue alors puisque la variable `b` est indéfinie pour l'instant. Ce code n'est donc pas valide.

2.3 Fonctions

Considérons un autre exemple nécessitant l'usage de variables. On se donne ainsi les coordonnées GPS d'un restaurant et d'un client (en train de faire sa recherche Google Maps pour trouver un restaurant à Marseille...). Pour simplifier, supposons ici que ces coordonnées GPS sont données par une abscisse et une ordonnée dans un repère bidimensionnel orthonormé. On note ainsi `x_restaurant` et `y_restaurant` les coordonnées du restaurant, `x_client` et `y_client` celles du client. En se rappelant que la distance entre un point de coordonnées (x_1, y_1) et un point de coordonnées (x_2, y_2) est donné par la formule

$$d((x_1, y_1),(x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

on peut écrire le code suivant pour obtenir la distance entre le restaurant et le client :

```
dx = x_restaurant - x_client
dy = y_restaurant - y_client
distance_carrée = dx * dx + dy * dy
distance = sqrt(distance_carrée)
```

dans lequel on a utilisé les opérations arithmétiques `-`, `+`, `*` et `sqrt` pour la soustraction, l'addition, la multiplication et le calcul de racine carrée (accessible après l'import de la bibliothèque `math` en Python).

S'il y a 100 restaurants dont on veut connaître la distance au client, il faut donc répéter 100 fois ces mêmes quatre lignes, en modifiant les coordonnées du restaurant. Ce serait bien répétitif, source d'erreurs et difficilement maintenable si on décide désormais de changer de représentation pour les coordonnées GPS. À la place, il vaut mieux utiliser une fonction.

Attention, comme pour le mot variable, le mot *fonction* a deux sens selon qu'on l'utilise dans son acception mathématique ou informatique :

— En mathématiques, une fonction est une relation entre un ensemble d'entrées et un ensemble de sorties avec la propriété que chaque entrée est reliée à *au plus une sortie*. La fonction peut souvent être décrite par une expression utilisant des variables représentant les entrées. Par exemple, on peut considérer la fonction f qui a un entier x associe $f(x) = x^2 + 2x$.

— En informatique, une fonction est une portion de code représentant un sous-programme, qui effectue une tâche ou un calcul relativement indépendant du reste du programme. Une fonction peut avoir des arguments représentés par des variables que le code de la fonction peut utiliser, et peut renvoyer un résultat.

Ainsi, on peut écrire une fonction qui calcule la distance entre un restaurant et un client :

— il prend en entrée les coordonnées du restaurant et du client

— et produit en sortie la distance attendue.

En Python, on déclarera une fonction de la façon suivante :

```
def calcule_distance(x_restaurant, y_restaurant, x_client, y_client):
    dx = x_restaurant - x_client
    dy = y_restaurant - y_client
    distance_carrée = dx * dx + dy * dy
    distance = sqrt(distance_carrée)
    return distance
```

Le mot clé `def` est suivi du nom de la fonction qu'on définit, ainsi que ses arguments en parenthèses et deux points en fin de ligne : les arguments sont des variables dont la valeur est donnée lors de l'utilisation de la fonction et qu'on peut utiliser dans le corps de la fonction. On utilise le mot clé `return` pour renvoyer le résultat attendu. Cela arrête l'exécution de la fonction : on ne peut donc retourner qu'au plus une fois par fonction.

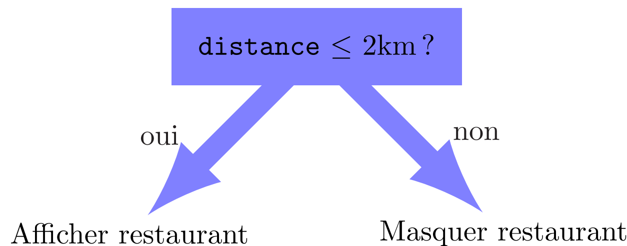
On peut alors calculer la distance entre plusieurs restaurants et plusieurs clients en appelant la fonction préalablement définie :

```
distance1 = calcule_distance (x_restaurant1 , y_restaurant1 ,
                              x_client , y_client)
distance2 = calcule_distance (x_restaurant2 , y_restaurant2 ,
                              x_client2 , y_client)
distance3 = calcule_distance (5.12 , 145.1 , 5.45 , 148.3)
```

Noter l'utilisation d'un point pour séparer la partie entière et décimale des flottants en Python, plutôt qu'une virgule dans la notation traditionnelle française.

2.4 Conditionnelles

Comment faire exécuter deux choses différentes à notre code selon qu'un restaurant est à moins de deux kilomètres d'un client ou pas ?



Il nous faut écrire une condition dans le pseudo-code, suivie de deux possibilités suivant que la condition est satisfaite ou non :

```
if distance <= 2:
    # afficher restaurant...
else:
    # masquer restaurant...
```

On a utilisé au-dessus des lignes commençant par le symbole `#` : il s'agit de *commentaires* permettant d'inscrire du texte qui ne sera pas utilisé lors de l'exécution du code. Ici, cela permet de cacher les détails d'implémentation permettant l'affichage ou le masquage d'un restaurant. Notez l'utilisation des deux points après le test et après le `else`, ainsi que l'indentation qui suit. Le test en lui-même est obtenu par une comparaison de la valeur d'une variable et de la constante entière 2. On utilise le symbole `<=` pour la comparaison « inférieur ou égal ».

Quelles sont les conditions que l'on peut tester ?

— Comparaisons : on peut comparer une variable avec une valeur ou une autre variable

```
distance <= 2
distance > 3
distance == 2.5
x != y
```

le troisième test étant un test d'égalité (notez l'utilisation du double-égal, puisque le symbole `=` est réservé pour l'affectation d'une variable) et le quatrième un test de non-égalité des deux contenus de variable.

— Divisibilité : on peut tester si le reste dans la division euclidienne d'un entier `n` par un entier `m` vaut `k` à l'aide de l'expression

```
n % m == k
```

En particulier, on peut tester la parité de l'entier `n` grâce au code

```
n % 2 == 0
```

— Combinaison de tests : on peut combiner les tests avec les opérateurs `and`, `or` et `not`. Si on cherche à n'afficher que les restaurants proches et ayant au moins 1 étoile, on exécute

```
(distance <= 2) and (nombre_étoiles >= 1)
```

Si on préfère ne voir que les restaurants qui sont très proches ou alors qui peuvent être plus éloignés mais ont au moins 2 étoiles, on utilisera plutôt

```
(distance < 1) or (nombre_étoiles >= 2)
```

Au passage, notons l'utilisation du symbole `%` qui permet de calculer le reste dans la division euclidienne. On peut l'utiliser en dehors d'une conditionnelle : par exemple, pour affecter dans une variable `a` le reste dans la division euclidienne du contenu de la variable `x` par 3, on écrira

```
a = x % 3
```

On peut aussi calculer des divisions flottantes (à l'aide du symbole `/`) ou des quotients dans une division euclidienne (à l'aide du symbole `//`).

2.5 Itérations

On l'a vu en Scratch, il est souvent utile de répéter une séquence d'opérations plusieurs fois. Plutôt que de copier-coller le morceau de code, on utilise des boucles permettant d'itérer ce morceau de code. Contrairement à l'exemple simpliste en Scratch, on a souvent besoin de connaître le nombre i d'itérations qui ont été déjà exécutées avant pour exécuter un code différent, dépendant de ce nombre i. Par exemple, essayons d'écrire le code calculant la somme des entiers de 1 à 1000. La façon naïve consiste à utiliser le code suivant (qu'on n'a pas écrit en entier...) :

```
somme = 0
somme = somme + 1
somme = somme + 2
somme = somme + 3
...
somme = somme + 1000
```

Le nombre d'*opérations élémentaires* (si on compte l'addition comme une opération élémentaire et l'affectation comme une autre opération élémentaire) effectuées par ce code est 2001, puisqu'il y a 1000 sommes et 1001 affectations. C'est long et pénible à écrire. À la place, on peut utiliser une boucle `for` qui exécute la même instruction pour toutes les valeurs de la variable décrite dans la boucle :

```
somme = 0
for n in range(1, 1001):
    somme = somme + n
```

La fonction `range` prend deux arguments `a` et `b` et permet de générer tous les entiers de `a` jusqu'à `b` -1 (attention ! on s'arrête un coup avant la borne). Il est possible de n'écrire qu'un seul argument si l'on souhaite générer plutôt les entiers de 0 à `b` -1 : `range(b)`.

Notez qu'on exécute exactement le même nombre d'opérations élémentaires, 2001 dans ce cas, mais ce code est bien plus court et lisible que le code précédent.

Évidemment il existe une solution bien plus simple pour réaliser ce calcul, puisqu'on connaît une formule mathématique pour calculer la somme des premiers termes d'une suite arithmétique de raison 1 et de premier terme 1 :

$$1 + 2 + \cdots + 1000 = \sum_{n=1}^{1000} n = \frac{1000 \times 1001}{2} = 500 \times 1001 = 500500$$

Cependant, la boucle s'avère indispensable lorsqu'on ne connaît pas de telles formules. Par exemple, si on souhaite calculer la somme des entiers de 1 à 1000 qui sont divisibles par 3 ou par 5, on pourra utiliser le code

```
somme = 0
for n in range(1, 1001):
    if (n % 3 == 0) or (n % 5 == 0):
        somme = somme + n
```

Notez l'absence de `else` dans la condition précédente : puisqu'on a rien à faire dans ce cas, Python nous permet de ne pas écrire le mot-clé.

Le nombre d'opérations élémentaires est un peu plus important dans ce cas. Pour chaque itération de la boucle `for`, on exécute 2 tests sur `n` suivi d'une disjonction (`or`), suivi, dans le pire des cas, d'une somme et d'une affectation : au total, chaque itération exécute donc au plus 5 opérations élémentaires. Puisqu'il y a 1000 itérations dans la boucle, le nombre total d'itérations est de 5000, auquel on ajoute la toute première affectation.

Exercice 13

On cherche à calculer la somme des n premières puissances de 2. Par exemple, si $n = 6$, le résultat est $2^0 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 = 1 + 2 + 4 + 8 + 16 + 32 = 63$.

1. Écrire une première fonction réalisant ce calcul, en s'autorisant le calcul des puissances de 2 comme opération élémentaire (en Python, on écrit par exemple `2**i` pour calculer 2^i), écrire un algorithme calculant et affichant la somme des n premières puissances de 2.
2. Combien d'opérations élémentaires effectue votre fonction lorsqu'elle est appelée avec un entier n en argument (en fonction de n) ?
3. Comment modifier votre code si on ne s'autorise plus l'utilisation de `2**i` comme opération élémentaire ? Calculer à nouveau le nombre d'opérations en fonction de n.
4. En calculant explicitement la valeur de $S = 2^0 + 2^1 + \cdots + 2^{n-1} = \sum_{i=0}^{n-1} 2^i$ en déduire une façon de calculer S avec un seul calcul de puissance et un nombre constant d'opérations élémentaires (additions, soustractions, multiplications...) supplémentaires.

On a parfois besoin d'écrire des boucles `for` qui égrène les éléments en sens inverse, ou qui saute d'un pas de plus de 1. Par exemple, si on veut réaliser des opérations pour tous les entiers entre 1 et 100 en commençant par le plus grand, on utilisera :

```
for n in range(100, 0, -1):
    ...
```

On utilise donc toujours la même fonction `range` mais avec trois arguments : le premier est le début de l'énumération, le second est l'entier suivant la fin de l'énumération voulue, et le troisième est le pas. Dans le cas d'un pas p négatif, `range(a, b, p)` permet donc de générer les entiers `a`, $\texttt{a} - p$, $\texttt{a} - 2p$, jusqu'au dernier entier $\texttt{a} - kp$ qui est strictement supérieur à `b`.

Si on souhaite ne visiter que les entiers pairs entre 2 et 200 (c'est-à-dire $2, 4, 6, \dots, 198, 200$), on utilisera :

```
for n in range(2, 201, 2):
    ...
```

On ne peut cependant pas utiliser directement ce genre de boucles lorsqu'on ne connaît pas à l'avance le nombre de tours de boucles à exécuter [2]. Un exemple typique est illustré par le calcul du nombre d'étapes avant de tomber sur la face 5, lors de tirages répétés d'un dé. En supposant qu'on dispose d'une fonction `lancer_dé()` ne prenant aucun argument (d'où les parenthèses vides) et renvoyant une face entre 1 et 6 tirée de manière aléatoire, on peut trouver le nombre d'étapes attendues avec le code suivant :

```
nombre_étapes = 0
face = lancer_dé()
while face != 5:
    face := lancer_dé()
    nombre_étapes = nombre_étapes + 1
```

On utilise donc une boucle `while` qui continue à exécuter le contenu de la boucle *tant que* la condition entre parenthèse reste vérifiée : ici, on continue tant qu'on n'est pas tombé sur la face 5. De manière générale, on sort donc de la boucle lorsque la condition n'est pas satisfaite au début d'une itération de la boucle.

2.6 Lecture et écriture

Pour finir, comme en Scratch, il nous sera parfois utile d'interagir avec l'utilisateur en imprimant un message ou le contenu d'une variable, ou bien en posant une question à l'utilisateur et attendre sa réponse. On utilisera deux fonctions `print` et `input` pour ces deux opérations, comme illustré par l'exemple suivant :

```
print("Êtes-vous sûr de vouloir quitter ?")
réponse = input()
if réponse == "oui":
    # quitter page...
```

La commande `input` renvoie l'information rentrée par l'utilisateur sous forme d'une chaîne de caractères. Si l'on souhaite en faire un entier par exemple, on peut utiliser la fonction `int` qui transforme une chaîne de caractères en entier. Ainsi, le code

```
print("Rentrez un nombre entier :")
x = int(input())
```

2. sauf à utiliser des mots-clés pour l'interruption prématurée de boucles, tels que `break` en Python, mais nous éviterons ce mot-clé dans ce cours

FIGURE 2.2 – Syntaxe pour les structures de contrôle, dans les pseudo-codes de ce cours

```
print("Le résultat de l'ajout de 2 est", x+2)
```

permet d'afficher la consigne, d'attendre que l'utilisateur écrive un entier, et on affiche alors l'entier auquel on a ajouté 2 en expliquant auparavant à l'utilisateur ce qu'on va afficher. Si l'utilisateur entre l'entier 8, il verra alors s'afficher le message :

```
Le résultat de l'ajout de 2 est 10
```

En résumé, les structures de contrôle que nous utiliserons dans ce cours sont rappelées en Figure 2.2.

Exercice 14

Écrivons du code Python pour jouer au jeu du juste prix.

1. Écrire un algorithme qui :
   — choisit aléatoirement un nombre entier entre 0 et 1000 de manière parfaitement opaque pour l'utilisateur, à l'aide de la fonction `randint` (qu'on suppose inclus en Python depuis la bibliothèque `random`) prenant deux arguments entiers a et b et renvoyant un nombre aléatoire dans l'intervalle $[a,b]$;
   — demande à l'utilisateur d'entrer des nombres entiers jusqu'à ce que ce dernier trouve le nombre préalablement choisi, en indiquant à chaque tentative

« trop haut » ou « trop bas » selon le nombre saisi au clavier.

2. Comment modifier votre algorithme pour qu'il affiche à l'utilisateur le nombre de tentatives qu'il a utilisées pour entrer le bon nombre ?
3. Dans le pire des cas, en combien d'étapes pouvez-vous être sûr que vous aurez trouvé le nombre mystère ? Quel est ce nombre d'étapes lorsqu'on cherche un nombre mystère entre 0 et un entier naturel n de la forme $2^p - 1$? Et pour un entier naturel n quelconque ?

Exercice 15

Rappelons-nous qu'un nombre premier est un nombre entier supérieur ou égal à 2 qui n'est divisible que par 1 et par lui-même.

1. Écrire une fonction `est_premier` prenant en argument un entier n et qui renvoie un booléen qui est `True` si n est premier, `False` sinon. *Si besoin, la partie entière d'un entier peut être calculée à l'aide de la fonction `floor` en Python.*
2. Écrire une fonction qui prend un entier i en argument et retourne le i-ième nombre premier, qu'on appellera p_i dans la suite, en s'aidant de la fonction précédente (sachant que 2 est le premier nombre premier, que 3 est le second, etc.) : lorsque $i > 1$, vous utiliserez une boucle qui énumère tous les entiers impairs (en effet, inutile d'essayer les entiers pairs, puisque le seul premier pair est 2).
3. Jusqu'en 1536, on croyait que les nombres de Mersenne, c'est-à-dire les nombres de la forme $2^p - 1$, avec p premier, étaient premiers. À partir des fonctions écrites en réponse aux deux questions ci-dessus, écrire un algorithme qui montre l'invalidité de la conjecture d'avant 1536. L'algorithme devra écrire :
   — à partir de quel nombre premier p_m la conjecture est fausse (c'est-à-dire tel que $2^{p_m} - 1$ n'est pas premier) ;
   — à quel rang m se trouve p_m dans la liste des nombres premiers.

Chapitre 3

Algorithmes sur les structures linéaires

Pour écrire des algorithmes, il faut des *structures de contrôle* telles que définies dans le chapitre précédent et des *structures de données* permettant de stocker des données de manière lisible. Illustrons ce concept au travers de quatre exemples représentés en Figure 3.1 :

— si les données à stocker sont des cartes à jouer, on préfèrera par exemple les ranger dans un certain ordre dans sa main : on utilise alors un tableau (comme dans un tableau Excel) pour les stocker de gauche à droite ;

— si les données sont des clients attendant d'être servis, on préfèrera les placer dans une file d'attente : c'est aussi une structure de tableau comme précédemment, mais avec la propriété fonctionnelle supplémentaire que le premier arrivé dans la file sera le premier servi ;

— si les données sont des assiettes dans un meuble de cuisine, on préfèrera les stocker dans une pile : contrairement à la file (d'attente), c'est l'assiette rangée en dernier (en haut de la pile) qui sera réutilisée la première ;

— finalement, si les données sont des pièces du jeu d'échec, pour pouvoir jouer, on préfèrera les ranger sur un échiquier qui est une matrice (un tableau bidimensionnel) de 8 rangées sur 8 colonnes.

Dans ce cours, nous n'étudierons en détail que la structure de tableau, et nous utiliserons à l'occasion la structure de matrice.

3.1 Définition d'un tableau

Un tableau (parfois appelé *liste* en Python) est la manière la plus simple de ranger au même endroit un ensemble de données. Il est composé d'un ensemble de cases accolées les unes aux autres contenant les éléments dans un certain ordre. Par exemple, voici un tableau contenant 5 cases (on dit que le tableau est de longueur 5) :

17	64	5	1	38

Voici un tableau de caractères de longueur 10 :

'a'	'l'	'g'	'o'	'r'	'i'	't'	'h'	'm'	'e'

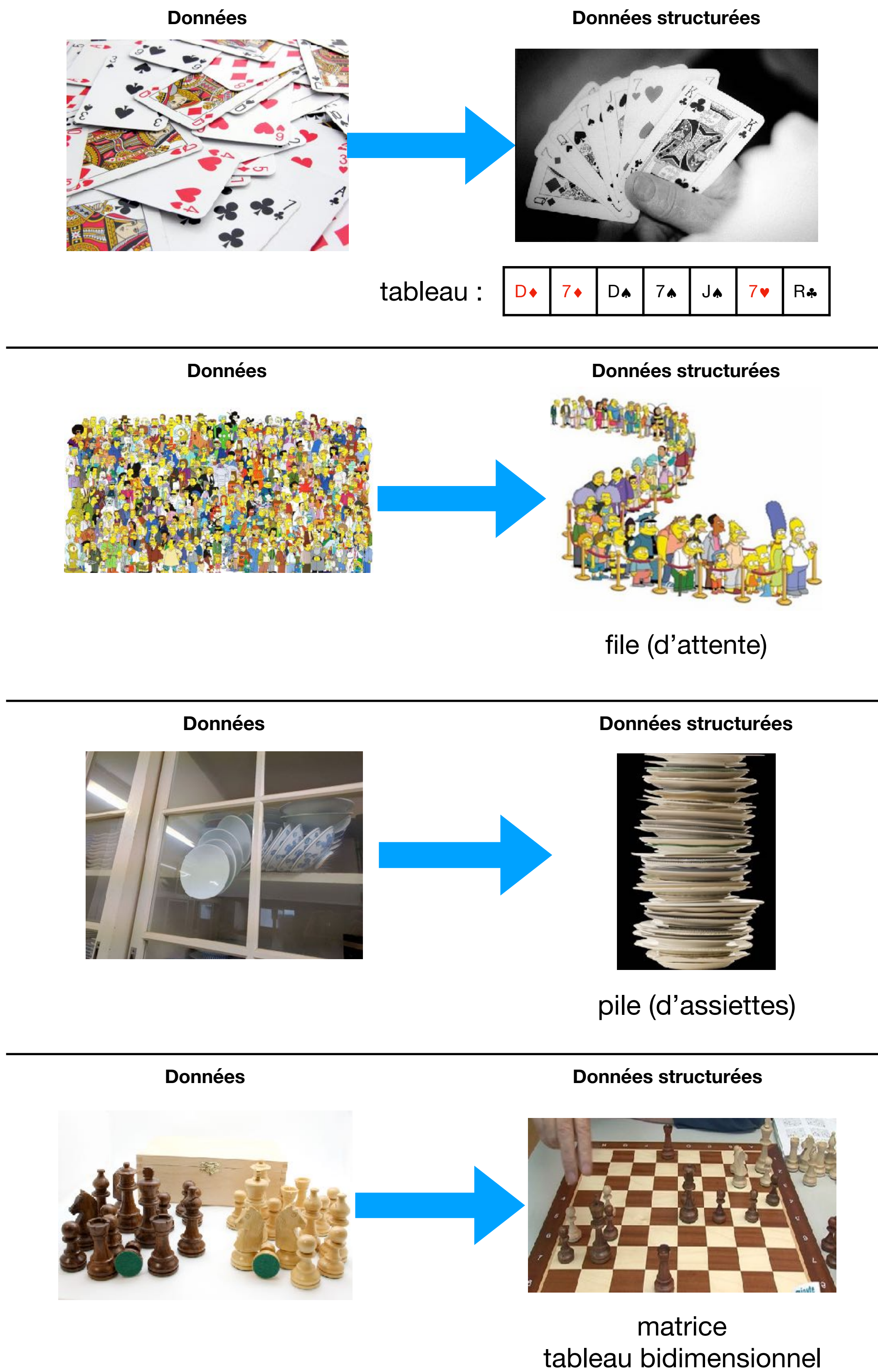


FIGURE 3.1 – Quatre structures de données : tableau, file, pile, matrice

Plus généralement, voici un tableau t de longueur $n \in \mathbf{N}$ quelconque :

$t[0]$	$t[1]$	$t[2]$	$t[3]$	$\cdots$	$t[n-2]$	$t[n-1]$

En Python, on peut déclarer un tableau en utilisant des crochets :

```
t = [5, 6, 8, 1, 0]
```

C'est pour cela qu'on notera les tableaux de la façon suivante : $\big[t[0], t[1], t[2], t[3], \ldots, t[n-2], t[n-1]\big]$. Classiquement, on commence à numéroter les cases d'un tableau à partir de 0, de sorte que la dernière case a le numéro $n-1$ (et pas n) : la i-ième case du tableau t (avec $0 \leq i \leq n-1$) contient une valeur qu'on note $t[i]$. On peut donc voir un tableau de longueur n contenant des éléments d'un ensemble E (des entiers, des caractères, etc.) comme une application $t\colon \{0, 1, \ldots, n-2, n-1\} \to E$ associant à chaque indice i une valeur notée $t[i]$ dans E.

Dans les algorithmes que nous allons écrire, on pourra :

— récupérer la longueur (*length* en anglais) d'un tableau `t` à l'aide de
```
n = len(t)
```
— initialiser un tableau de longueur `n` rempli d'une constante (par exemple 0) à l'aide de
```
t = [0] * n
```
— accéder au contenu de la `i`-ème case d'un tableau `t` à l'aide de `t[i]`

— modifier le contenu de la `i`-ème case d'un tableau `t` à l'aide de
```
t[i] = x
```

Exercice 16

1. On se donne trois variables entières a, b et c. Quel algorithme (on ne demande pas d'écrire une fonction, jusque quelques lignes de Python) permet d'effectuer une *permutation circulaire* vers la droite de ces trois variables, c'est-à-dire de faire en sorte qu'à la fin, a contienne la valeur originelle de c, b la valeur de a et c la valeur de b ?
2. Comment faire de même avec un tableau $t = \big[t[0], t[1], \ldots, t[n-1]\big]$ de n entiers, plutôt que trois variables ?
3. Combien d'opérations élémentaires (affectations, opérations arithmétiques) sont exécutées par votre algorithme en fonction de la longueur n du tableau ?

3.2 Tableaux et chaînes de caractères : application à la cryptologie

Un tableau peut permettre de stocker une chaîne de caractères. Par exemple, le tableau de caractères

'a'	'l'	'g'	'o'	'r'	'i'	't'	'h'	'm'	'e'

permet de représenter la chaîne de caractères « algorithme ». Grâce à cela, nous allons être en mesure d'écrire des algorithmes manipulant des chaînes de caractères. Un domaine d'application où c'est très utile est la cryptologie (étymologiquement, science du secret), un domaine

à la frontière des mathématiques et de l'informatique. Elle se sépare en deux pans de même importance. Le premier consiste à transformer une information afin de la rendre secrète, autrement dit à la "crypter" ou "chiffrer". Il s'agit de la *cryptographie* (étymologiquement, écriture secrète). Le second consiste à analyser les informations cryptées et trouver des méthodes et techniques afin d'en dévoiler le sens. Il s'agit de la *cryptanalyse*. Intéressons-nous à quelques procédés historiques simples de cryptologie, ce qui permettra d'écrire quelques algorithmes simples sur les tableaux.

Exercice 17

Historiquement, un procédé de cryptographie bien connu est le codage de César que Jules César utilisait dans ses correspondances. Le principe de chiffrement est simple. Étant donné un alphabet (ici, nous utiliserons l'alphabet latin) et un message, le message chiffré s'obtient en remplaçant chacune des lettres du message d'origine par une lettre à distance fixe toujours dans la même direction. Pour les dernières lettres, dans le cas d'une distance à droite, on reprend au début de l'alphabet. Il s'agit d'un chiffrement par décalage. À titre d'exemple, avec un décalage de 5, 'a' devient 'f', 'b' devient 'g', ..., 'y' devient 'd' et 'z' devient 'e'.

1. Soit le message "La vie est un long fleuve tranquille". Donner ses représentations chiffrées selon le codage de César avec les clés 3 et -7.
2. En utilisant des phrases (pas du code Python), donner une description précise de l'algorithme de chiffrement utilisé pour le codage de César.

Cela nous permet de revenir sur le codage de César (cf exercice 3.2) de manière un peu plus précise. On peut donc stocker le message en clair, ainsi que le message chiffré, à l'aide de tableaux de caractères. Comment peut-on écrire un algorithme procédant au chiffrement d'un message ?

Tout d'abord, comme on l'a vu précédemment, on code les caractères ('a', 'b', ..., 'z') avec des entiers. Supposons ici que la lettre 'a' est codée par l'entier 0, la lettre 'b' par l'entier 1, etc. On se donne alors une fonction `code(c)` permettant de connaître le code d'un caractère `c` et une fonction `caractère(p)` renvoyant le caractère dont le code est `p` (seulement s'il est entre 0 et 25). On a par exemple

```
code('g') = 6        caractère(12) = 'm'        caractère(code('x')) = 'x'
```

À l'aide de ces deux fonctions, on peut donc écrire un algorithme procédant au chiffrement d'un message :

```
def chiffrement_Cesar(message, clé):
      n = len(message)
      chiffré = ['a'] * n
      for i in range(n):
          p = code(message[i])
          p_décalé = (p + clé) % 26     # on décale puis on revient
                                        # dans l'intervalle [0,25]
          chiffré[i] = caractère(p_décalé)
      return chiffré
```

Cette fonction prend en entrée le message à chiffrer, ainsi que la clé de César qui est un entier correspondant au nombre de lettres pour le décalage. Elle commence par stocker dans la variable `n` la longueur du message. Elle crée ensuite un autre tableau de caractères, `chiffré`,

pour contenir le message chiffré qui est donc de longueur `n`. Elle l'initialise avec des caractères arbitraires. Ensuite, chiffrer un message consiste à parcourir les caractères du message les uns après les autres et, pour chacun d'entre eux, appliquer le décalage donné par la clé, avant de mettre le caractère chiffré dans la case correspondante du tableau `chiffré` : pour parcourir le message, on utilise une boucle `for` avec un `range` permettant à l'indice `i` de prendre les valeurs de 0 à $\texttt{n} - 1$. À la sortie de la boucle, on retourne le tableau `chiffré`.

Exercice 18

1. Exécuter l'algorithme précédent pour trouver ce que retourne l'appel `chiffrement_Cesar(['i','n','t','o','x'], 3)`. Pour exécuter la boucle `for`, on utilisera le tableau suivant :

i	message[i]	p	p_décalé	chiffré
avant la boucle				['a', 'a', 'a', 'a', 'a']
0	'i'	8	? ?	? ?
1	? ?	? ?	? ?	? ?
2	? ?	? ?	? ?	? ?
⋮				

2. L'algorithme précédent ne prend pas en charge les espaces auxquels on ne souhaite pas apporter de décalage. Ainsi, on aimerait que `chiffrement_Cesar(['u','n','␣','d','e','u','x'], 3)` renvoie le tableau `['x', 'q', '␣', 'g', 'h', 'x', 'a']`. Modifier l'algorithme précédent pour y parvenir.
3. Proposez un algorithme de déchiffrement, prenant en entrée le message chiffré et une clé et renvoyant le message décodé. À titre d'exemple, déchiffrez le message « kajex » sachant que la clé de chiffrement vaut 9.

Étudions désormais le nombre d'opérations élémentaires effectuées lors de l'algorithme de chiffrement de César.

— On affecte d'abord à la variable `n` la longueur du message, comptant pour 1 opération élémentaire.
— On crée ensuite un tableau de longueur n, ce qu'on compte également ici comme 1 opération élémentaire (on pourrait aussi compter ceci comme n opérations élémentaires, sans changer la complexité asymptotique).
— Ensuite, chaque itération consiste à calculer le code d'une lettre du message (1 opération), de décaler ce code à l'aide d'un calcul arithmétique (1 opération) et d'écrire le caractère chiffré dans le tableau (1 opération). Chaque itération coûte donc 3 opérations.
— Puisqu'on exécute n itérations, le coût total de la boucle `for` est de $3n$ opérations.
— Finalement, on retourne le message chiffré (1 opération).

Au total, on a donc utilisé $3n + 3$ opérations élémentaires.

Exercice 19

Admettons que quelqu'un vous envoie un message chiffré en vous spécifiant qu'il s'agit d'un codage de César mais sans vous donner la clé. Est-il possible de le déchiffrer ? Si oui, comment et est-ce efficace ?

On propose ensuite deux exercices permettant d'explorer un peu plus la cryptographie,

sans avoir recours formellement aux tableaux. *Ces exercices sont donc plutôt d'ordre culturel et peuvent donc être passés lors d'une lecture rapide du chapitre.*

Exercice 20

Dans l'armée de Spartes (autour du Vème siècle avant J.-C.), les militaires étaient parfois amenés à se transmettre des messages chiffrés. Pour ce faire, ils utilisaient un bâton, appelé bâton de Plutarque (ou scytale). L'émetteur du message prenait une fine lanière de tissu ne pouvant contenir qu'une seule lettre dans sa largeur, l'enroulait en spirale autour d'un bâton, et écrivait son message ligne par ligne dans la longueur du bâton de façon à avoir toutes les lignes remplies sauf éventuellement la dernière ligne :

Une fois déroulée, la lanière contenait le message chiffré. Pour déchiffrer ce message, le récepteur devait posséder un bâton de même diamètre (ayant le même nombre de circonvolutions) que celui de l'émetteur. Ce type de codage est un chiffrement par transposition.

À titre d'exemple, considérons le message suivant : « La vie c'est comme une boîte de chocolats, on ne sait jamais sur quoi on va tomber. » (*Forrest Gump*, de Robert Zemeckis) Considérons un bâton dont le périmètre permet 6 circonvolutions (*nous dirons pour simplifier que la clé du chiffrement est* 6). Le message chiffré est alors « Lo␣␣moamdoan␣meni␣ve␣␣svi␣cn␣aeuhes␣␣no␣utcecsro'␣oa␣mebliqbsoatuetî t␣or␣tsji.ce,a␣␣ »

1. Chiffrez la phrase « Les cons ça ose tout. » (*Les tontons flingueurs*, Michel Audiard) avec la clé 4 en vous aidant d'un ruban de papier que vous enroulerez autour d'un stylo par exemple. (C'est plus facile à faire à deux...)
2. Proposez une méthode décrivant ce procédé de chiffrement. En particulier, comment déterminer le nombre de colonnes à utiliser sur le bâton ? Illustrez votre méthode en chiffrant la phrase de la question précédente avec les clés 3 et 5.
3. Étant donné un message chiffré m de longueur n et une clé c, donnez un algorithme permettant de découvrir le message d'origine.
4. Déchiffrez les messages suivants :
    — "Ce'e'␣oceànos␣␣ntçln␣aeam␣siêq␣tmur." avec $c = 4$;
    — "Càq␣s'␣up␣emea ?so␣r␣titl␣␣␣ue␣" avec $c = 5$;
    — "L,␣snru␣s␣␣ekjutp.eeioè␣" avec $c = 6$.
5. À présent, vous recevez un message chiffré m sans la clé de chiffrement. Donnez un algorithme permettant de retrouver le message d'origine et évaluez-en l'efficacité en termes de temps.

	a	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**l**	**m**	**n**	**o**	**p**	**q**	**r**	**s**	**t**	**u**	**v**	**w**	**x**	**y**	**z**
a	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
b	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a
c	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b
d	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c
e	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d
f	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e
g	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f
h	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g
i	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h
j	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i
k	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j
l	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k
m	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l
n	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m
o	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n
p	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
q	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p
r	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q
s	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r
t	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
u	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t
v	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u
w	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v
x	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w
y	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x
z	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y

FIGURE 3.2 – Table de Vigenère.

Exercice 21

Revenons sur le codage de César. Le principal point faible de ce codage par décalage provient justement du type de décalage qui est identique quelle que soit la position de la lettre décalée dans le message d'origine.

1. Comment pourrait-on transformer le codage de César pour qu'il soit plus difficile à casser ?

L'objectif du codage de Vigenère est justement de remédier à ce défaut. Il a été décrit pour la première fois au XVIème siècle C'est également un chiffrement par décalage, mais la substitution est cette fois-ci poly-alphabétique. Cela signifie qu'une même lettre dans le message d'origine peut, selon sa position dans ce dernier, être remplacée par différentes lettres dans le message chiffré.
Plus précisément, ce nouveau chiffrement va utiliser une notion de clé différente des précédents. Ici, la clé va être une suite de caractères, qui prend généralement la forme d'un mot ou d'une phrase. Pour procéder au chiffrement, il faut parcourir le message d'origine lettre après lettre tout en parcourant circulairement les lettres de la clé pour effectuer la substitution. Bien sûr, plus la clé est longue et variée, plus le chiffrement est solide. La substitution à opérer est donnée par la table illustrée en Figure 3.2 page 53, appelée table de Vigenère, dans laquelle la première ligne correspond aux lettres du message d'origine et la première colonne à celles de la clé.
Ainsi, on remplace chaque lettre ℓ du message m d'origine par celle contenue dans la case (k,ℓ), avec k la lettre de la clé c correspondant à la position de ℓ dans m, modulo $|c|$.

À titre d'exemple, le message "Peu lui importe de quoi demain sera fait" associé à la clé "petit frère" sera chiffré de la manière suivante (en supprimant les accents) :

```
Message d'origine : Peu lui importe de quoi demain sera fait
Clé               : pet itf rerepet it frer epetit frer epet
                     ||ligne 't' colonne 'u' -> 'u' devient 'n'
                     |ligne 'e' colonne 'e' -> 'e' devient 'i'
                     ligne 'p' colonne 'p' -> 'P' devient 'E'
```

ce qui mène au message chiffré "Ein tnn zqgsgxx lx vlsz htqtqg xvvr jpmm".

2. Soit le message d'origine "Chacun voit sa voie de toi à moi". Chiffrez ce message avec les deux clés suivantes : "ntm" et "de personne je ne serai la cible". Qu'observez-vous ?
3. Étant donné un message d'origine m et une clé c, donnez un algorithme de chiffrement de m selon c.
4. Étant donné un message chiffré m et une clé c, donnez un algorithme de déchiffrement de m selon c. Continuez en déchiffrant le message "Diepe dorr n'owg naj k vrnnvr" avec la clé "keny arkana".

3.3 Rechercher dans un tableau

Un tableau peut aussi permettre de stocker simplement un ensemble de données telles que :
— un paquet de cartes à jouer ;
— des mots (avec leur définition) dans un dictionnaire ;
— ou des noms (avec leur adresse et numéro de téléphone) dans un annuaire.

Une opération cruciale lorsqu'on stocke un tel ensemble de données est de pouvoir tester si un élément appartient à l'ensemble ou non.

3.3.1 Recherche séquentielle

Imaginons pour commencer qu'on souhaite vérifier si notre paquet de cartes à jouer contient un joker ou pas. Pour cela, on a peu d'autres choix que de rechercher le joker en passant les cartes du paquet en vue l'une après l'autre, en commençant par un bout du paquet.

Dans le contexte des tableaux, cela revient à rechercher un élément dans le tableau en parcourant le tableau (de gauche à droite par exemple) et en s'arrêtant dès lors qu'on a trouvé l'élément : cet algorithme s'appelle la *recherche séquentielle* (ou recherche par balayage), puisqu'on visite les éléments du tableau séquentiellement, c'est-à-dire les uns à la suite des autres. On peut écrire cet algorithme de la façon suivante :

```
def rechercher_séquentiel(tableau, élément):
    n = len(tableau)
    for i in range(n):
        if tableau[i] == élément:
            return True
    return False      # élément pas trouvé !
```

L'algorithme renvoie `True` dès lors qu'il a trouvé une case du tableau contenant l'élément recherché : dès que l'instruction `return True` est exécutée, la fonction s'arrête et ne poursuit donc pas son exploration. Au contraire, si la boucle `for` s'est exécutée entièrement sans jamais trouver l'élément recherché, alors on peut renvoyer `False` puisqu'on est alors sûr que l'élément recherché ne se trouve pas dans le tableau.

Exercice 22

1. Modifier l'algorithme de recherche séquentielle pour qu'il renvoie l'indice où on a trouvé l'élément : ainsi, la recherche de l'élément 8 dans le tableau [3,5,8,2,8,1] devra renvoyer 2 (puisque les cases des tableaux sont numérotées à partir de 0). Si l'élément n'est pas trouvé, on renverra -1.
2. Modifier ensuite l'algorithme pour qu'il renvoie l'indice de la *dernière occurrence* de l'élément recherché : ainsi, la recherche de l'élément 8 dans le tableau [3,5,8,2,8,1] renverra désormais 4.

Imaginons désormais qu'on recherche un mot dans un dictionnaire, ou un nom dans un annuaire. A priori, vous n'utilisez pas l'algorithme de recherche séquentielle dans ce cas. Essayons de comprendre pourquoi en estimant la *complexité* de cet algorithme. Comment faire cela ? Une première possibilité consisterait à déclencher un chronomètre en même temps que le début de l'exécution de l'algorithme (par un humain ou un ordinateur), afin de voir le temps qu'il met avant de renvoyer le résultat. Cette première solution est malheureusement

peu précise car pas nécessairement reproductible : l'humain ou l'ordinateur qui exécute l'algorithme ne mettra pas toujours le même temps, selon qu'il est plus ou moins en forme, qu'il a plus ou moins d'autres choses à penser ou à faire en même temps. De plus, on arriverait alors pas à comparer les complexités d'algorithmes dont l'un serait exécuté par un humain et l'autre par un ordinateur, ou par deux ordinateurs différents.

Pour remédier à ce problème, on utilise une autre méthode pour estimer la complexité d'un algorithme : on compte plutôt le nombre d'*opérations élémentaires* que l'algorithme exécute *dans le pire des cas* pour des entrées de taille fixée.

- Une opération élémentaire, cela correspond *grosso modo* à une ligne de pseudo-code : de manière générale, il est important de se fixer un ensemble d'opérations élémentaires qu'on comptabilise ensuite une par une lors de l'exécution de l'algorithme.
- La définition précise *dans le pire des cas* puisqu'on voudrait pouvoir décrire la complexité de l'algorithme sur toutes les entrées possibles d'une taille fixée : mais il se peut que, pour certaines entrées, le résultat soit très rapide à calculer, et que, pour d'autres, ce soit beaucoup plus long. Pour régler ce problème, on considère donc le pire des cas possibles, qui borne donc la complexité de toutes les instances d'une taille fixée.

Dans l'algorithme de recherche séquentielle, l'affectation de la longueur du tableau dans la variable `n` peut être vue comme une opération élémentaire, de même que le test d'égalité entre le contenu de la case d'indice `i` et l'élément à chercher, ou le fait de retourner un résultat. On cherche donc à estimer le nombre de telles opérations élémentaires lorsque l'algorithme s'exécute sur un tableau de longueur n arbitraire (n étant supposé grand).

- Dans tous les cas, on affecte à la variable `n` la longueur du tableau, ce qui coûte une opération élémentaire.
- Une itération de la boucle `for` effectue une opération élémentaire (test d'égalité), et au plus une fois au total le retour de la valeur de sortie `True`.
- Puisque dans le pire des cas (c'est-à-dire lorsqu'on cherche un élément qui n'apparait pas dans le tableau), on exécute cette boucle n fois (une fois par case du tableau), au total, on exécute donc n opérations (sans compter le retour).
- Finalement, dans tous les cas, on exécute soit le retour (de `True`) au sein de l'itération, ou le retour (de `Faux`) en dehors de l'itération, qui coûte donc toujours une opération élémentaire.

Au total, on exécute donc $1+n+1$ opérations élémentaires dans le pire des cas, c'est-à-dire $n+2$ opérations élémentaires.

Considérons désormais le point de vue d'un tableau « très long », c'est-à-dire avec n très grand. Dans ce cas, il n'est pas très intéressant de distinguer $n+2$ de n : on préfère donc conserver uniquement l'*ordre de grandeur* de la complexité. À ce titre, introduisons la notation de Landau (du nom d'Edmund Landau qui l'a introduite) permettant de comparer deux telles *suites*.

Définition 4. Soient $u = (u_n)_{n\in\mathbf{N}}$ et $v = (v_n)_{n\in\mathbf{N}}$ deux suites à valeurs entières : les entiers u_n et v_n décrivent donc des nombres d'opérations élémentaires pour l'exécution d'un algorithme dans le pire des cas sur une entrée de taille n. On dit que u est en $O(v)$ (qui se lit « grand o de v », comme la lettre de l'alphabet...) s'il existe un entier N et une constante $c > 0$ tels que pour tout $n \geq N$, on a $u_n \leq cv_n$. Dans la suite, on s'autorise à écrire que u_n est en $O(v_n)$.

Par exemple, on obtient alors bien que $n+2$ est en $O(n)$: en effet, pour $N = 2$ et $c = 2$, on a bien que pour tout entier $n \geq 2$, $n+2 \leq n+n = 2n = cn$.

Dans ce cours, on considèrera donc que « $n+2$ ou n, c'est pareil ! ». On peut aller plus loin et montrer qu'en fait, « $2n$ ou n, c'est pareil ! » : en effet, pour $N = 0$ et $c = 2$, on obtient trivialement que pour tout $n \geq 0$, $2n \leq cn$. En terme d'informatique, cela veut dire qu'un algorithme A et un autre algorithme B qui va deux fois plus vite que A ne sont généralement pas distingué en terme de complexité dans le pire des cas : fondamentalement, c'est parce qu'il suffit d'avoir une machine deux fois plus puissante pour que B devienne aussi performant que A.

Au contraire, n^2 n'est pas en $O(n)$, c'est-à-dire que n^2 grossit beaucoup plus vite que n. Vous pouvez le vérifier en montrant que vous ne pouvez pas trouver N et c vérifiant la définition lorsque $u_n = n^2$ et $v_n = n$. Par contre, on a bien que n est en $O(n^2)$, mais ce n'est pas très intéressant : cela veut juste dire que n grossit moins vite que n^2, mais on y perd donc beaucoup en faisant cette approximation... L'ordre de grandeur d'une complexité de la forme $a_k n^k + a_{k-1} n^{k-1} + \cdots + a_2 n^2 + a_1 n + a_0$ (avec k fixé et $a_k \neq 0$) est donc en $O(n^k)$, à savoir le plus grand terme dans l'écriture polynomiale de la complexité.

Revenons alors à l'algorithme de recherche séquentielle dont on a vu qu'il exécutait dans le pire des cas $n+2$ opérations élémentaires. On dira donc qu'il a une complexité en $O(n)$, *linéaire* en la longueur du tableau en entrée. Si l'on recherche un mot dans un dictionnaire contenant 32 000 mots par exemple, cela demande donc un nombre d'opérations de l'ordre de 32 000 : si on le fait « à la main », même si on pouvait exécuter 10 opérations à la seconde, il nous faudrait alors 53 minutes pour rechercher un mot dans le dictionnaire...

On souhaite donc faire mieux en utilisant l'information qu'un dictionnaire, ou un annuaire, n'est pas un tableau quelconque. En effet, les mots du dictionnaire, ou les noms de l'annuaire, sont triés par ordre alphabétique croissant. On dit donc d'un tableau qu'il est trié si ses éléments sont classés par ordre croissant : le tableau [1,4,5,8] est donc trié, contrairement au tableau [8,3,4,5].

Exercice 23

1. Améliorer l'algorithme de recherche séquentielle dans le cas où le tableau donné en entrée est supposé trié, pour qu'il s'arrête dans son parcours du tableau dès lors qu'il a trouvé l'élément à chercher, ou bien qu'il est sûr que l'élément à chercher ne se trouve pas dans le tableau.
2. Quelle est l'ordre de grandeur de complexité dans le pire des cas de votre nouvel algorithme ?

3.3.2 Recherche dichotomique dans un tableau trié

Profitons donc mieux du fait qu'on recherche dans un tableau trié (dictionnaire ou annuaire) pour décrire un algorithme de recherche a priori plus performant. Lorsqu'on cherche dans un dictionnaire, on ne regarde pas les mots les uns à la suite des autres en commençant par la lettre A, d'autant plus si on recherche la définition du mot « pingouin »... On va plutôt essayer d'estimer la position du mot dans le dictionnaire, ouvrir le dictionnaire à la page correspondante, puis prendre une décision pour continuer notre recherche à gauche de la page courante, ou à droite. On continue ensuite ainsi de suite jusqu'à avoir trouvé le mot, ou être sûr que le mot n'existe pas dans ce dictionnaire.

Ce type de recherche dans un tableau trié s'appelle la *recherche dichotomique*, du grec $\delta\iota\chi o\tau o\mu \acute{\iota}\alpha$ qui signifie « division en deux parties égales ». Dans le cas général, on part donc

d'un tableau trié et on commence par comparer l'élément qu'on recherche avec l'élément qui se trouve au milieu (environ) du tableau. Partons du tableau trié d'entiers

1	4	5	7	8	9	10	14	17	25	26	27	38	47	56	64

et cherchons-y l'élément 26. On commence par regarder la case du milieu : le tableau étant de longueur 16, le milieu du tableau correspond à la case d'indice 7 (les cases sont indexées de 0 à 15), qui héberge l'élément 14. Il est différent de 26 : il faut donc continuer notre recherche. Par ailleurs, 14 est strictement inférieur à 26 : puisque le tableau est trié, cela implique que l'élément 26 ne peut pas se trouver dans la portion du tableau à gauche de l'élément 14. On peut donc se restreindre à rechercher 26 dans la partie non grisée du tableau :

1	4	5	7	8	9	10	14	17	25	26	27	38	47	56	64

Parmi les 8 éléments restants, on poursuit en comparant 26 avec l'élément du milieu du tableau restant, la case contenant 27. Ces deux éléments sont toujours différents, mais cette fois, la comparaison nous apprend que l'élément 26 ne peut pas se trouver dans la portion du tableau à droite de l'élément 27. On se restreint ainsi à la portion non grisée du tableau

1	4	5	7	8	9	10	14	17	25	26	27	38	47	56	64

Il reste une portion de longueur 3, dont 25 est l'élément du milieu. Une fois de plus, on supprime la partie de gauche de la portion restante pour ne laisser plus qu'une seule case qui abrite l'élément 26 que nous recherchions :

1	4	5	7	8	9	10	14	17	25	26	27	38	47	56	64

Sur le même tableau en entrée, si l'on recherche l'élément 6, voici les étapes par lesquelles on passe :

1	4	5	7	8	9	10	14	17	25	26	27	38	47	56	64
1	4	5	7	8	9	10	14	17	25	26	27	38	47	56	64
1	4	5	7	8	9	10	14	17	25	26	27	38	47	56	64
1	4	5	7	8	9	10	14	17	25	26	27	38	47	56	64
1	4	5	7	8	9	10	14	17	25	26	27	38	47	56	64

Cela nous permet de répondre avec certitude que le tableau de contient pas l'élément 6.

Voici une écriture sous forme de pseudo-code de l'algorithme que nous venons d'exécuter :

```
def rechercher_dichotomique(tableau, élément) :
    n = len(tableau)
    début = 0
    fin = n - 1
    while début <= fin:
        milieu := (début + fin) // 2
        if tableau[milieu] == élément:
            return True
        elif tableau[milieu] < élément:
            # l'élément est à droite
            début = milieu + 1
        else: # l'élément est à gauche
            fin = milieu - 1
    return False
```

On y maintient deux variables `début` et `fin` qui conservent en mémoire l'indice de la première et de la dernière case de la portion de tableau qu'il reste au cours de la recherche. On initialise donc ces deux variables avec les première et dernière cases du tableau respectivement, d'indice 0 et $\texttt{n} - 1$. La recherche se termine lorsqu'on est convaincu que l'élément recherché ne se trouve pas dans le tableau, c'est-à-dire lorsqu'il ne reste plus aucune portion de tableau à parcourir : c'est le cas lorsque l'indice `début` est strictement supérieur à `fin`. Il faut donc continuer la rechercher *tant que* cette condition n'est pas vérifiée, c'est-à-dire *tant que* `début` est inférieur ou égal à `fin`. Dans ce cas, il faut alors calculer l'indice de la case du milieu : le milieu de l'intervalle $[a,b]$ est le nombre $(a + b)/2$. Puisqu'on veut obtenir un indice de case qui est un entier, on choisit le quotient dans la division euclidienne de $a + b$ par 2 (à l'aide de l'opérateur `//` en Python). On stocke l'indice de la case du milieu dans une variable `milieu`. On peut alors comparer le contenu de cette case avec l'élément à rechercher : si on le trouve, on s'arrête en retournant `True` ; sinon, on effectue une comparaison supplémentaire pour savoir si l'élément a des chances d'apparaître à gauche ou à droite du milieu. Selon le cas, on met à jour soit le début de la portion, soit la fin de la portion. Notez l'utilisation du mot-clé `elif` qui permet d'enchaîner un `else` et un `if`.

Arrêtons-nous un moment sur cet exemple d'algorithme, qui est un peu plus complexe que tous ceux que nous avons vu jusqu'alors, en particulier du fait de l'utilisation d'une boucle `while`. Est-ce bien un algorithme selon la définition que nous nous étions donnée dans le chapitre précédent :

> ≪ Un algorithme est la description *non ambiguë* d'une séquence *finie* d'instructions permettant de résoudre un problème (informatique) ou d'obtenir un résultat. ≫

Terminaison

La définition insiste sur la finitude : un algorithme doit terminer après un nombre fini d'étapes. Est-on bien sûr que c'est le cas ici ? Dans cet algorithme, c'est l'utilisation d'une boucle `while` qui pourrait faire échouer cette condition de terminaison de l'algorithme : il se pourrait en effet que la condition d'arrêt de la boucle ne soit jamais vérifiée, l'algorithme bouclant alors à l'infini. Pourquoi est-on certain ici que cela n'arrivera pas ? Il s'agit de s'assurer que la boucle `while` termine, c'est-à-dire que le test `début <= fin` finit par devenir faux. Autrement dit, il faut s'assurer qu'à un moment de l'algorithme, on finit par avoir `début > fin`. C'est le cas, puisqu'à chaque étape de la boucle `while` :

— soit `début` augmente strictement ;
— soit `fin` diminue strictement.

Ainsi, à chaque itération `fin` − `début` diminue strictement : puisque c'est un entier, il finit par devenir négatif. Le test `début <= fin` finit donc par être faux.

En général, prouver la terminaison d'une boucle `while` se fait par la mise au jour d'un *variant de boucle*, c'est-à-dire une quantité entière positive ou nulle (dépendant des données de l'algorithme) qui décroit strictement à chaque tour de boucle. Comme il n'existe pas de suite infinie d'entiers strictement décroissante, on assure la terminaison de la boucle. Pour la recherche dichotomique, le variant de boucle est `fin` − `début`.

Correction

L'utilisation d'une boucle `while` complique également notre intuition sur la *correction* de l'algorithme : est-on sûr qu'il trouve bien un élément dès lors que celui-ci apparaît dans le

tableau et qu'il renvoie `False` uniquement lorsque l'élément ne s'y trouve pas ? S'assurer que l'algorithme est correct, c'est montrer qu'il fait bien ce qu'il est sensé faire :

— si l'élément qu'on cherche se trouve dans le tableau trié, alors l'algorithme doit renvoyer `True` ;

— si l'élément qu'on cherche ne se trouve pas dans le tableau trié, alors l'algorithme doit renvoyer `False` : ceci est facile à montrer puisque l'algorithme ne peut renvoyer `True` que s'il a trouvé un indice du tableau hébergeant l'élément recherché.

Concentrons-nous donc sur la première propriété. Pour s'assurer de cette propriété, on écrit un *invariant de boucle*, c'est-à-dire une propriété qui est vraie initialement, et qui reste vraie tout au long de l'algorithme. Ici, l'invariant de boucle est le suivant : si on suppose que le tableau en entrée est trié par ordre croissant et contient l'élément recherché, alors à tout moment de l'exécution de l'algorithme on est sûr qu'il existe un indice i entre `début` et `fin` tel que $\texttt{tableau}[i] = \texttt{élément}$.

C'est vrai en début de boucle puisque $\texttt{début} = 0$ et $\texttt{fin} = \texttt{n} - 1$ donc la portion est le tableau tout entier. On peut ensuite se convaincre que l'exécution d'une itération de la boucle `Tant que` préserve la propriété, puisqu'on a simplement retiré une portion du tableau où l'on est sûr que l'élément ne se trouve pas (c'est ici qu'on utilise le fait que le tableau est trié !).

Par le principe de récurrence (sur le nombre de tours de boucle), l'invariant de boucle est donc vrai pendant toute l'exécution : en particulier, il est vrai lorsque l'on sort de la boucle `while` (ce qui est sûr d'arriver puisque l'algorithme termine). Mais si l'on sort de la boucle sans avoir jamais renvoyé `True`, on a alors $\texttt{début} > \texttt{fin}$ et il n'existe plus aucun indice i entre `début` et `fin`, contredisant la propriété qu'on a démontré.

Complexité

Pour terminer l'étude de cet algorithme, il ne nous reste plus qu'à trouver sa complexité, pour la comparer avec celle de la recherche séquentielle qu'on a étudiée avant.

Supposons que le tableau a une longueur $n = 2^p$ pour simplifier le calcul. On cherche donc le nombre d'opérations élémentaires exécutées par l'algorithme dans le pire des cas sur un tableau de longueur 2^p. Le pire des cas intervient lorsqu'on ne trouve pas l'élément dans le tableau puisque la recherche ne s'interrompt alors pas prématurément. Ensuite, remarquons que chaque itération de la boucle `while` exécute 5 opérations élémentaires dans le pire des cas :

1. le test `début <= fin` ;
2. le calcul du nouveau milieu pour l'affectation de la variable `milieu` ;
3. le test `tableau[milieu] == élément` : le pire des cas se produit si ce test n'est pas satisfait ;
4. le test `tableau[milieu] < élément` ;
5. l'affectation de la nouvelle valeur de `début` ou `fin` selon le cas.

Il reste donc à connaître le nombre d'itérations de la boucle `while`. On peut la déduire en fonction de la longueur de la portion restante du tableau à explorer : cette longueur vaut toujours $\texttt{fin} - \texttt{début} + 1$. À chaque itération, la taille de la portion restante de tableau est au moins divisée par deux (d'où le nom de *dichotomie* !). Par conséquent,

— après 0 itération (au début), la longueur de la portion restante est 2^p ;

— après 1 itération, la longueur de la portion restante est au plus 2^{p-1} ;

— après 2 itérations, la longueur de la portion restante est au plus 2^{p-2} ;

— ...
— après k itérations, la longueur de la portion restante est au plus 2^{p-k} ;
— ...
— après $p+1$ itérations, la longueur de la portion restante est au plus $2^{p-(p+1)} = 1/2$, et puisqu'elle est entière, la longueur est nulle ; autrement dit le tableau est vide.

On est donc assuré qu'il y a au plus $p + 1$ itérations de la boucle `while`. Le nombre d'opérations élémentaires exécutées par la boucle dans le pire des cas est donc d'au plus $5 \times (p + 1)$.

Par ailleurs, en dehors de la boucle, l'algorithme exécute au plus 4 opérations élémentaires :

1. l'affectation de `n` ;
2. l'affectation de `début` ;
3. l'affectation de `fin` ;
4. le retour de la valeur `True` ou `False`.

Le nombre total d'opérations élémentaires est donc au plus $4+5\times(p+1)$. Puisque $n = 2^p$, on a $p = \log_2 n$, donc la complexité est en $4 + 5 \times (\log_2 n + 1)$. Pour n supérieur à 2 (de sorte que $1 \leq \log_2 n$), on a $4 + 5 \times (\log_2 n + 1) \leq 14 \times \log_2 n$. Ainsi, la complexité de la recherche dichotomique est de l'ordre de $O(\log_2 n)$, logarithmique en la longueur n du tableau.

Pour s'apercevoir de la différence cruciale entre une complexité linéaire (comme la recherche séquentielle) et une complexité logarithmique (comme la recherche dichotomique), observons que $\log_2(32\ 000) \approx 15$ ce qui veut dire que la méthode de la recherche dichotomique permet de trouver n'importe quel mot du dictionnaire contenant 32 000 mots en regardant au plus 15 pages du dictionnaire : à raison d'une page par seconde, cela donne le mot en 15 secondes au plus (à comparer aux 53 minutes qu'on avait calculé précédemment pour la recherche séquentielle). Lorsque n devient encore plus grand, l'écart est de plus en plus important, puisque la suite $(\log_2 n)_{n\in\mathbf{N}^*}$ croit exponentiellement moins vite que la suite $(n)_{n\in\mathbf{N}^*}$. Même pour un tableau (trié) comprenant autant de cases que le nombre d'atomes dans l'univers visible (environ 10^{79}), on parvient tout de même à rechercher un élément en un nombre d'opérations élémentaires de l'ordre de $\log_2(10^{79}) \approx 263$.

3.4 Tri d'un tableau

On a vu précédemment que pour rechercher un élément dans un tableau, la complexité était bien meilleure dès lors que le tableau était trié (c'est-à-dire que ses éléments sont rangés dans un ordre croissant). Il est donc naturel de se demander comment faire en sorte de trier un tableau et de la complexité nécessaire pour cela. Ce problème très naturel de tri d'éléments apparaît à de nombreuses reprises en informatique : par exemple, lorsqu'on a voulu visualiser la liste des restaurants à proximité par note moyenne décroissante, c'est un tri (par ordre décroissant, plutôt que croissant...) que l'on exécute.

3.4.1 Tri par insertion

Commençons par considérer la situation où l'on cherche à ranger dans sa main des cartes à jouer suivant un ordre précis (afin de séparer les couleurs, puis de ranger les cartes par ordre croissant dans chaque couleur, par exemple). Pour simplifier, considérons un cas particulier où nous n'avons que des cartes de carreau. Prenez une dizaine de cartes et rangez-les par ordre croissant : analysez alors votre façon de faire...

Je décris ici ma façon de faire lorsqu'il s'agit de trier un nombre conséquent de cartes. Je laisse le tas de cartes à trier face contre la table et prend les cartes les unes après les autres. Je prends la première carte dans ma main. Je retourne ensuite la seconde carte que je viens placer au bon endroit (avant ou après la première carte) dans ma main. Je retourne alors la troisième carte que je dois de même insérer au bon endroit dans ma main. Lorsque j'arrive à la treizième carte, il devient plus difficile de savoir où je dois l'insérer : si je décompose à nouveau ma routine, je vois que je *scanne* la suite des cartes de droite à gauche jusqu'à trouver l'endroit où je dois insérer la nouvelle carte. Essayez d'imiter ma méthode afin de classer un paquet de treize cartes de carreau, en décomposant bien chaque étape en opérations les plus simples possibles.

Cette procédure de tri s'appelle le *tri par insertion* du fait qu'on ne fait qu'insérer les éléments les uns après les autres au bon endroit dans la portion triée. On peut également l'appliquer pour trier un tableau. On suppose donc qu'on a en entrée de l'algorithme un tableau non trié, par exemple, le tableau [10,8,2,5,13]. On considère les éléments de gauche à droite, en cherchant à chaque fois à les placer au bon endroit dans la portion triée qui sera la partie gauche du tableau. Initialement, on considère donc le premier élément du tableau (10) qui est bien placé puisque c'est le seul élément considéré jusqu'alors. La portion triée est donc [10,... et il reste le tableau ...8,2,5,13] à considérer. On regarde ensuite le second élément (8) puis on le compare à l'élément à sa gauche, afin de l'insérer au bon endroit dans la portion triée : ici $8 < 10$ donc il faut échanger les deux éléments, ce qui termine l'insertion de 8 à sa place. On se retrouve alors avec la portion triée [8,10,... et le reste du tableau ...2,5,13]. Au coup suivant, on doit considérer l'élément 2 : il est inférieur à 10, donc on doit échanger sa place avec 10 (temporairement on obtient donc le tableau [8,2,10,5,13]), puis on le compare avec l'élément à sa gauche (8) pour arriver à la situation où la portion triée est [2,8,10,... et le reste ...5,13]. L'insertion de 5 se fait alors en deux étapes : on échange les éléments 10 et 5, puis les éléments 8 et 5, avant de voir que $2 < 5$ ce qui achève l'insertion de l'élément 5. Finalement, on considère l'élément 13 qui est directement supérieur à l'élément à sa gauche (10) stoppant dès le début son insertion. On termine donc avec le tableau trié [2,5,8,10,13].

Voici une écriture sous forme de pseudo-code de cet algorithme :

```
def trier_par_insertion(tableau) :
    n = len(tableau)
    for i in range(1, n):
        x = tableau[i]
        # insérer x parmi les i premiers éléments
        j = i
        while (j > 0) and (x < tableau[j-1]):
            # décaler d'un élément
            tableau[j] = tableau[j-1]
            j = j-1
        # ici, x ≥ tableau[j−1] ou bien j=0
        tableau[j] = x
    # le tableau est trié !
```

Exécutez l'algorithme sur le tableau [8,2,10,5,13] pour bien comprendre comment il fonctionne : vous verrez qu'il ne fait pas exactement ce qui est dit au-dessus en s'épargnant des échanges de cases inutiles...

Exercice 24

1. Que se passe-t-il si on remplace la troisième ligne de l'algorithme par `for i in range(n):` ?
2. Montrons que cet algorithme termine. La seule raison pour laquelle il pourrait ne pas terminer est la boucle `while`, comme dans l'algorithme de recherche dichotomique. Montrer que cette boucle termine bien dans tous les cas.

Notons également que cet algorithme ne retourne aucun résultat : on a effectivement choisi de faire un tri *en place*, c'est-à-dire qu'on ne renvoie pas un nouveau tableau, mais qu'on a modifié le tableau donné en entrée directement... Il est donc inutile de le retourner, puisqu'on a directement modifié le tableau dans la mémoire. Notons qu'en Python, le passage d'un tableau en argument d'une fonction se fait bien « par référence » et donc la modification au sein de la fonction se répercute bien sur la mémoire globale : attention, ce n'est pas le cas dans tous les langages de programmation !

Pour mieux comprendre cet algorithme (puis le comparer avec d'autres algorithmes, dans la suite), estimons sa complexité dans le pire des cas. Comme pour les algorithmes de recherche étudiés auparavant, il faut donc comptabiliser les opérations élémentaires.

Vu l'imbrication de boucles de cet algorithme, il convient de commencer par considérer les boucles les plus internes, c'est-à-dire le code qui est le plus « décalé à droite ». Considérons donc d'abord la boucle `while` qui exécute deux opérations élémentaires à chaque itération (une affectation dans `tableau[j]` et une décrémentation de `j`), auxquelles il faut ajouter les opérations élémentaires nécessaires pour tester si l'on doit continuer la boucle ou pas : ce test `(j > 0) and (x < tableau[j-1])` requiert deux comparaisons qu'on comptabilise donc comme deux opérations élémentaires. Une itération réclame donc 4 opérations élémentaires. Il faut aussi considérer le dernier test qui fait sortir de la boucle `while`, qui nécessite aussi 2 tests dans le pire des cas. Pour totaliser la complexité de cette boucle interne, il nous suffit donc de savoir le nombre de fois que cette boucle s'exécute. Dans le pire des cas, l'élément `x` est inférieur à tous les éléments de la portion triée, ce qui pousse alors à le décaler jusqu'au tout début du tableau : la variable `j` prend alors toutes les valeurs de `i` à 1, avant d'être égale à 0, auquel cas on sort de la boucle. Dans le pire des cas, on exécute donc `i` fois la boucle, générant donc au total $4\mathtt{i} + 2$ opérations élémentaires.

On peut ensuite passer à la boucle `for` :

- on y affecte la variable `x`
- ainsi que la variable `j` ;
- on exécute la boucle `while`, coûtant donc $4\mathtt{i} + 2$ opérations élémentaires dans le pire des cas ;
- finalement, on modifie la valeur de `tableau[j]`.

Lors de l'itération correspondant à une valeur particulière de `i`, on exécute donc $1+1+4\mathtt{i}+2+1 = 4\mathtt{i}+5$ opérations élémentaires. Ce nombre dépend de l'itération `i` : on ne peut donc pas simplement multiplier le nombre d'opérations par le nombre d'itérations. À la place, on fait la somme de toutes les contributions des itérations : le nombre total d'opérations élémentaires exécutées au sein de la boucle `for` vaut donc

$$(4 \times 1 + 5) + (4 \times 2 + 5) + \cdots + (4 \times (n-1) + 5)$$

qu'on peut écrire sous la forme d'une somme qu'on décompose en deux sommes indépendantes :

$$\sum_{i=1}^{n-1}(4i+5) = \sum_{i=1}^{n-1} 4i + \sum_{i=1}^{n-1} 5 = 4\sum_{i=1}^{n-1} i + 5 \times (n-1)$$

Il ne reste plus qu'à calculer la somme de gauche, qui n'est rien d'autre que la somme des $n-1$ premiers entiers dont on sait par ailleurs qu'elle vaut $(n-1)n/2$. On obtient donc un nombre d'opérations élémentaires égal à $2(n-1)n + 5(n-1) = 2n^2 + 3n - 5$.

Finalement, il faut aussi comptabiliser les opérations élémentaires en dehors de la boucle `for` : il n'y en a qu'une, l'affectation de la variable `n`. In fine, on exécute donc, dans le pire des cas, $2n^2 + 3n - 4$ opérations élémentaires. L'ordre de grandeur est donc en $O(n^2)$ (pour s'en convaincre ici, il suffit de remarquer que pour tout n supérieur à 3, $2n^2 + 3n - 4 \leq 2n^2 + 3n \times n = 5n^2$). L'algorithme de tri par insertion est donc un algorithme de complexité *quadratique* en la longueur du tableau à trier.

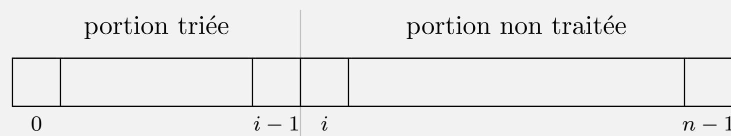
Exercice 25

On vient de voir que, dans le pire des cas, la complexité du tri par insertion est en $O(n^2)$. On peut raisonnablement se poser la question de savoir si on a surestimé le nombre d'opérations élémentaires. En fait, il n'en est rien. Trouver donc une suite de tableaux $(t_n)_{n\in\mathbf{N}}$ avec t_n un tableau de longueur n telle que le nombre C_n d'opérations élémentaires effectuées lors du tri du tableau t_n est de la forme $an^2 + bn + c$ avec a, b, c des constantes et $a > 0$.

Avant d'étudier un autre algorithme de tri, avec une meilleure complexité, l'exercice suivant propose l'étude d'un autre tri très célèbre en informatique, car particulièrement visuel.

Exercice 26

Intéressons-nous à un autre algorithme de tri, fondé sur la méthode dite "des bulles". Il s'agit d'une méthode qui opère par permutations successives sur le tableau à trier. On trie ainsi le tableau de gauche à droite, comme pour le tri par insertion. Voilà la situation, une fois qu'on a trié les i premiers éléments, avec $1 \leq i < n-1$:



Pour augmenter la longueur de la portion triée, on crée une bulle qui enserre la case $n-1$ du tableau : il faut désormais imaginer que le tableau est dessiné de manière verticale, la case $n-1$ étant le fond d'un aquarium, alors que la séparation entre les cases $i-1$ et i représente le niveau de l'eau à l'heure actuelle. La bulle va donc remonter du fond de l'aquarium jusqu'au niveau de l'eau : mais attention, lorsqu'elle passe d'une case à l'autre, si les deux éléments consécutifs ne sont pas ordonnés, la bulle fait remonter la case qu'elle enserre et les éléments sont donc permutés. Au contraire, si les deux éléments consécutifs sont dans le bon ordre, la bulle remonte sans entraîner avec elle de permutation de cases. À titre d'exemple, si l'on prend le tableau [B,A,T,E,A,U,X] (qu'on écrit BATEAUX dans la suite de cet exercice pour raccourcir les notations), la méthode des bulles le trie selon l'ordre alphabétique de la manière suivante :

i	trié / non traité	min. partie non traitée	après "remontée des bulles"
0	/ BATEAUX	A	A / BATEUX
1	A / BATEUX	A	AA / BETUX
2	AA / BETUX	B	AAB / ETUX
3	AAB / ETUX	E	AABE / TUX
4	AABE / TUX	T	AABET / UX
5	AABET / UX	U	AABETU / X
6	AABETU / X		

Attention, notez bien ce qu'il s'est passé lors de la remontée de la première bulle :

- le niveau de l'eau est pour l'instant tout à gauche du tableau et la bulle commence autour de la lettre X ;
- la bulle remonte et puisque U est une lettre inférieure (dans l'ordre alphabétique) à X, la bulle passe de X à U sans permuter les éléments ;
- il en va de même lorsqu'elle passe de U à A ;
- par contre, la bulle entraîne la lettre A avec elle puisque E est supérieure à A : les lettres A et E sont donc permutées ;
- la lettre suivante est un T, qui est supérieure au contenu de la bulle (A) donc les deux cases sont aussi permutées ;
- la bulle rencontre une autre lettre A sur son passage, et continue donc à monter sans besoin de permuter les cases ;
- finalement, B étant inférieur à A, la bulle entraîne la lettre A avec elle et la fait passer au-dessus du niveau de l'eau.

On obtient donc le tableau ABATEUX : on a fait sortir la lettre A, tout en faisant remonter la seconde lettre A au niveau de la première... Lors de la deuxième étape, cela explique pourquoi on obtient le tableau AABETUX (et pas AABTEUX !).

1. Exécuter le tri à bulles sur le tableau [I,N,F,O,R,M,A,T,I,Q,U,E] (vous représenterez le résultat sous forme d'un tableau tel que celui donné ci-dessus).
2. Écrire (sous forme de pseudo-code) l'algorithme décrivant le tri à bulles.
3. Évaluer la complexité de cet algorithme, en donnant un ordre de grandeur du nombre d'opérations élémentaires réalisées dans le pire des cas sur un tableau de longueur n.
4. En regardant à nouveau l'exécution de l'algorithme sur BATEAUX et INFORMATIQUE, qu'observez-vous ? Pourrait-on l'améliorer afin d'éviter des comparaisons inutiles ?
5. Proposer un nouvel algorithme de tri à bulles qui améliore significativement le précédent en tenant compte de vos observations précédentes. *Il est possible de faire s'arrêter une fonction en plaçant l'instruction* `return`*, qui ne renvoie rien, mais stoppe l'exécution où elle en est...*
6. Bien que cet algorithme améliore en moyenne le temps d'exécution, il n'en est rien dans le pire cas : vous pouvez ainsi vérifier qu'une estimation rapide de la complexité fournit toujours une borne en $O(n^2)$ sur le nombre d'opérations. Pour s'en convaincre, pouvez-vous trouver une suite de tableaux $(t_n)_{n\in\mathbf{N}}$ avec t_n de longueur n telle que le nombre d'opérations élémentaires effectuées par

votre algorithme optimisé sur t_n est de la forme $an^2 + bn + c$ avec a, b, c des constantes et $a > 0$?

3.4.2 Tri par fusion

Proposons maintenant une autre façon de trier un tableau, utilisant une méthode tout à fait différente, consistant à diviser-pour-régner. Illustrons-la sur l'exemple du tableau

$$[3, 16, 14, 1, 12, 7, 10, 4, 5, 11, 15]$$

On va diviser le problème en deux sous-problèmes de la même forme : ici, il suffit de couper le tableau en deux, pour obtenir deux sous-tableaux $[3, 16, 14, 1, 12, 7]$ et $[10, 4, 5, 11, 15]$ de taille (presque) identique. Imaginons qu'on sache trier ces deux sous-tableaux : on obtient alors les tableaux $[1, 3, 7, 12, 14, 16]$ et $[4, 5, 10, 11, 15]$. Pour obtenir le grand tableau trié, il suffit donc de fusionner ces deux tableaux triés. Mais ceci est très facile à faire (imaginez que vous avez deux jeux de cartes triés et que vous voulez les fusionner en conservant le tri...). Il suffit de remarquer que le plus petit élément du grand tableau trié doit forcément être 1 ou 4 (les premiers éléments des deux petits tableaux triés), c'est donc 1. Pour obtenir la suite du grand tableau trié, on continue ce même raisonnement avec les deux petits tableaux $[3, 7, 12, 14, 16]$ et $[4, 5, 10, 11, 15]$: c'est 3 le plus petit élément des deux qui doit donc être à la suite de 1 dans le tableau. Puis 4 vient ensuite : on se retrouve alors avec deux petits tableaux de la forme $[7, 12, 14, 16]$ et $[5, 10, 11, 15]$. On continue ainsi jusqu'à avoir complètement fusionné les deux petits tableaux triés et obtenu le tableau

$$[1, 3, 4, 5, 7, 10, 11, 12, 14, 15, 16]$$

La question reste cependant entière : comment fait-on pour trier les deux petits tableaux $[3, 16, 14, 1, 12, 7]$ et $[10, 4, 5, 11, 15]$? La réponse est simple : « on recommence ! ». En effet, la tactique décrite ci-dessus pour trier le grand tableau peut être aussi utilisée pour traiter ces deux petits tableaux, de manière indépendante. On peut ainsi visualiser dans la Figure 3.3 le cheminement complet pour trier le grand tableau : la division en deux sous-tableaux est marquée par l'éclair rouge et les deux flèches bleues, puis la fusion des deux tableaux triés est représentée par les flèches orange.

Comment écrire dans du pseudo-code la formule magique « on recommence ! » ?!? La manière la plus simple de faire est la suivante :

```
def tri_fusion(tableau) :
    n = len(tableau)
    if n <= 1:
        return tableau
    else:
        gauche = tableau[0 : n//2]    # portion gauche
        droite = tableau[n//2 : n]    # portion droite
        gauche_trié := tri_fusion(gauche)
        droite_trié := tri_fusion(droite)
        retourner( fusionner(gauche_trié, droite_trié) )
    FinSi
```

On a utilisé la notation Python `t[a : b]` qui permet de renvoyer un nouveau tableau contenant les éléments $t[a], t[a+1], \ldots, t[b-1]$ d'indice a (inclus) à b (exclus). Lorsque $a = 0$

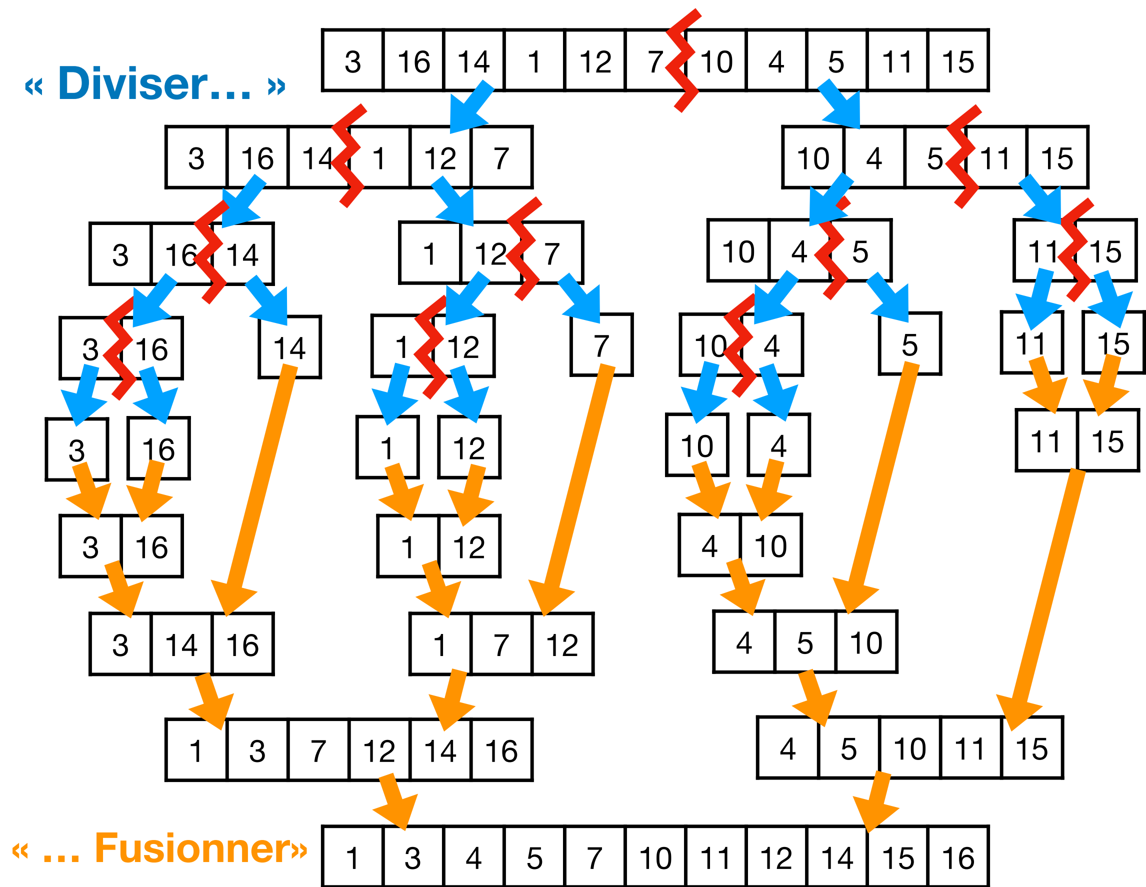


FIGURE 3.3 – Représentation graphique du tri par fusion

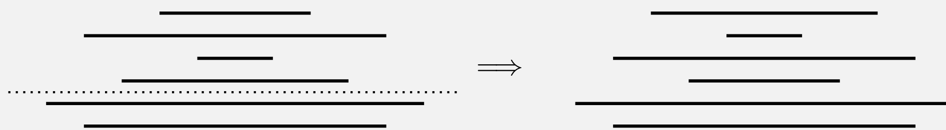
(comme c'est le cas pour la portion gauche), on peut simplifier en `t[:b]`. Lorsque b est égal à la longueur du tableau (comme c'est le cas pour la portion droite), on peut simplifier en `t[a:]`. Contrairement au tri par insertion, cette fonction retourne un tableau : elle n'est pas *en place*. On a également utilisé une fonction `fusionner` dont on ne fournit pas le code : c'est la fonction qui prend deux petits tableaux triés en entrée et doit les fusionner pour produire un grand tableau trié, comme expliqué ci-dessus. Finalement, le « on recommence ! » est écrit par l'appel de la fonction `tri_fusion` elle-même au sein de sa propre définition : on appelle cela des *appels récursifs* ; la fonction elle-même s'appelle une *fonction récursive*. Nous reverrons plus loin ce mécanisme.

En terme de complexité, il n'est pas très difficile de se convaincre que la fusion de deux tableaux triés peut s'exécuter en temps linéaire ($O(n)$) en la taille du grand tableau. Par ailleurs, comme on peut le voir en Figure 3.3, on fait assez peu d'appels récursifs finalement : comme pour la recherche dichotomique, l'intérêt de la méthode est qu'à chaque appel récursif, on a divisé par deux la longueur du tableau à trier. On ne peut donc faire qu'au plus $O(\log_2 n)$ appels récursifs imbriqués. Ceci explique pourquoi la complexité du tri par fusion est en $O(n \log_2 n)$, ce qu'on admet dans ce cours.

On est donc passé d'un tri de complexité $O(n^2)$ à un tri de complexité $O(n \log_2 n)$: c'est évidemment bien mieux de la même façon que la recherche dichotomique de complexité $O(\log_2 n)$ est bien meilleure que la recherche séquentielle de complexité $O(n)$. Mais peut-on encore mieux faire ? En fait, on peut montrer qu'il n'est pas possible de mieux faire, dès lors qu'on ne considère que des tris qui procèdent par comparaison des éléments deux par deux, comme c'est le cas pour les tris qu'on a étudiés jusque-là. Cependant, si on lève cette restriction, il est possible d'obtenir une meilleure complexité, comme on l'étudie dans l'exercice suivant.

Exercice 27

Dans une crêperie du Vieux Port, deux crêpiers se relaient en cuisine. Toutes les crêpes n'ont pas exactement le même diamètre. Au moment du changement de crêpier, s'il reste des crêpes déjà précuites, le crêpier sur le départ, un peu psychorigide sur les bords, veut laisser à son collègue une pile de crêpes triée de la plus grande en bas, jusqu'à la plus petite en haut. Oui, mais voilà : la cuisine est minuscule ! La seule possibilité pour le crêpier psychorigide est d'utiliser sa grande spatule, de la planter entre deux crêpes de la pile de crêpes et de retourner la totalité de la pile de crêpes au-dessus, le tout sur la même pile de crêpe. Par exemple, dans la pile de crêpes à gauche ci-dessous, si le crêpier plante sa spatule à l'endroit des pointillés et retourne la pile du dessus, il obtient la pile de crêpes de droite.



1. Comment le crêpier doit-il s'y prendre pour trier sa pile de crêpes ? Décrire une méthode que le crêpier peut utiliser facilement : en particulier, notez qu'il est facile pour le crêpier de trouver la plus grande crêpe dans une pile de crêpes...
2. Décrire votre algorithme en Python en supposant que la pile de crêpe est représentée par le tableau `t` des diamètres des crêpes en commençant par la crêpe la plus basse. Vous pourrez utiliser

- la fonction `retourner_spatule(t, i)` pour dire au crêpier de planter sa spatule au-dessus de la `i`-ième crêpe de la pile `t` (supposée non vide) et de retourner la pile au-dessus (l'exemple du début correspond donc à `retourner_spatule(t, 2)`) ;
- la fonction `plus_grande_crêpe(t, i)` qui retourne la plus grande crêpe dans la pile `t` (supposée non vide) au-dessus de la `i`-ème crêpe, c'est-à-dire dans le sous-tableau `t[i:]` : ainsi `plus_grande_crêpe(t, 0)` renvoie la plus grande crêpe, alors que `plus_grande_crêpe(t, 1)` fait de même en ignorant la crêpe du bas.

3. Combien d'opérations élémentaires (recherche de la plus grande crêpe dans une sous-pile et retournement d'une sous-pile) effectue le crêpier s'il utilise votre algorithme dans le pire des cas, en fonction du nombre n de crêpes dans la pile ?
4. Proposer des algorithmes en pseudo-code pour réaliser les fonctions `plus_grande_crêpe` et `retourner_spatule`, en ne s'autorisant comme opérations élémentaires sur les tableaux que la lecture d'une case et l'écriture dans une case.
5. En supposant désormais qu'on compte comme opérations élémentaires les comparaisons d'éléments du tableau et les lectures et écritures dans le tableau, quelle est la complexité du tri du crêpier psychorigide ?

Chapitre 4

Algorithmes sur les entiers et les flottants

On a vu dans le chapitre précédent comment on pouvait stocker des ensembles ordonnés de données dans un tableau. C'est particulièrement pratique, par exemple pour stocker les informations des différents restaurants à proximité. Si l'on souhaite ensuite calculer la distance entre l'utilisateur et l'un de ces restaurants, il nous faut calculer avec des paires de réels (plus précisément de nombre flottants) représentant des coordonnées cartésiennes.

Cela nous amène à réfléchir à la façon dont une machine compte, c'est-à-dire calcule avec des nombres. On peut par exemple se demander ce que fait une machine lorsqu'on lui demande d'ajouter 1 à une variable n entière, comment une machine ajoute deux nombres entiers, comment elle teste que n est congru à 4 modulo 27 (c'est-à-dire que le reste dans la division euclidienne de n par 27 vaut 4), comment elle vérifie si deux entiers a et b sont premiers entre eux (c'est-à-dire n'ont que 1 comme diviseur commun positif), comment elle calcule l'exponentielle de 23, ou le logarithme de 14. Ce chapitre a pour objectif d'éclaircir ces différentes questions.

4.1 Addition d'entiers

Nous avons déjà vu un algorithme dans le chapitre d'introduction *incrémentant* une variable, c'est-à-dire ajoutant un à cette variable : `n = n`+1. On l'a vu aussi, les entiers sont représentés en binaire dans une machine tel que notre téléphone ou notre ordinateur. Dès lors, passer une variable de 13 à 14, signifie passer son codage en binaire de 1101 à 1110. Nous avions vu alors l'algorithme pour incrémenter la représentation binaire d'une variable :

(i) commencer par le bit de poids faible (celui qui est le plus à droite) ;

(ii) inverser le bit ;

(iii) tant que ce bit est à zéro, recommencer l'étape (ii) avec le bit situé à sa gauche ;

(iv) si on arrive au bout de la représentation binaire, ajouter un bit 1.

À l'époque, nous n'avions pas pu décrire plus précisément cet algorithme, faute de syntaxe pour les algorithmes et de structure de données adaptée pour stocker un code binaire. Désormais, nous pouvons stocker le codage binaire de tout entier dans un tableau. Ainsi, on représente l'entier $n = 207$ par le tableau

1	1	0	0	1	1	1	1

L'incrément peut alors s'écrire de la manière suivante en pseudo-code qui prend en entrée un tableau de bits et le modifie (la fonction ne renvoie donc rien, comme lors du tri par insertion dans le chapitre précédent) :

```
def incrémenter(n_en_binaire):
    i = len(n_en_binaire)-1
    while (i > 0) and (n_en_binaire[i] == 1):
        n_en_binaire[i] = 0
        i = i - 1
    n_en_binaire[i] = 1
```

Par exemple, sur le tableau précédent, représentant 207, on initialise la variable `i` à 7 (la longueur du tableau valant 8, c'est-à-dire qu'on a codé l'entier sur un octet). On commence par s'apercevoir que la case d'indice 7 héberge un bit à 1, vérifie la condition de la boucle `while` qui permet de passer ce bit à 0 et de passer la valeur de `i` à 6. Le tableau est donc devenu

1	1	0	0	1	1	1	0

On fait de même pour les trois bits à 1 suivants, arrivant à la situation où `i` vaut 3 et le tableau est

1	1	0	0	0	0	0	0

Désormais, la case d'indice `i`= 3 héberge un bit 0 et ne vérifie donc plus la condition de la boucle `while`. On continue avec la suite du pseudo-code, après la boucle. On passe donc la valeur de cette case à 1 et on s'arrête. À la fin de l'algorithme, le tableau `n_en_binaire` est donc

1	1	0	1	0	0	0	0

qui est bien la représentation en binaire de 208.

Vous avez peut-être remarqué qu'on est en train d'utiliser un algorithme pour calculer $n+1$ à partir de n, et que cet algorithme utilise l'affectation `i = i - 1` : on pourrait donc naturellement se dire qu'on est en train de tricher. En fait, il n'en est rien car il faut simplement se rendre compte que ce sont des opérations bien différentes :

— l'incrémentation qu'on cherche à effectuer prend en entrée un entier qui pourrait être codé sur la totalité de la mémoire de notre ordinateur, et donc un nombre gigantesque et inconnu ;

— au contraire, l'affectation `i = i - 1` est simplement un subterfuge pour dire à l'ordinateur de *se déplacer d'une case à gauche dans la mémoire* : c'est une opération de base que l'ordinateur est capable de faire sans rien calculer du tout...

Réitérons désormais l'expérience d'incrémentation avec le tableau suivant en entrée :

1	1	1	1	1	1	1	1

Cette fois-ci, le test `n_en_binaire[i]`= 1 n'échoue jamais, mais le test `i`> 0 finit par échouer, lorsque la variable `i` devient égale à 0. On sort alors du tableau (rempli de 0) et on passe la première case à 1 de sorte qu'on termine avec le tableau suivant :

1	0	0	0	0	0	0	0

Clairement, le résultat n'est pas correct puisque 11111111 est le codage binaire de l'entier $2^8 - 1 = 255$, alors que 10000000 est le codage binaire de l'entier $2^7 = 128$. La raison de l'échec est ce qu'on appelle un *dépassement de capacité* : 11111111 est le plus grand entier naturel qu'on peut stocker sur 8 bits, de sorte qu'ajouter 1 est impossible sans dépasser les 8 bits autorisés.

Exercice 28

Il n'est pas franchement satisfaisant, même dans le cas d'un dépassement de capacité que l'entier 128 suive l'entier 255, lorsqu'on incrémente un entier stocké sur 8 bits (pour simplifier, puisqu'en réalité, on stocke les entiers sur 32 ou 64 bits dans les ordinateurs modernes). Essayons donc de faire mieux.

1. Supposons désormais qu'on stocke sur 8 bits des entiers relatifs, avec un bit de signe (1 pour les nombres négatifs et 0 pour les nombres positifs). Exécuter l'algorithme d'incrémentation précédent sur les deux codes de l'entier 0 (à savoir 00000000 et 10000000), le code d'un entier strictement positif différent du plus grand entier qu'on peut coder sur 8 bits et le code d'un entier strictement négatif différent du plus petit entier qu'on peut coder sur 8 bits. Est-ce correct ?
2. La situation semble encore moins favorable qu'avec les entiers naturels. Pour arranger cela, on utilise en fait un autre codage des entiers relatifs, basé sur la technique du *complément à deux*. On ne modifie pas le codage d'un entier positif : on commence par le bit de signe 0 puis le codage de l'entier sur les 7 bits restants. En revanche, voici la méthode pour coder un entier négatif $-n$ avec $n \in \mathbb{N}$:
   - on commence par coder sur 8 bits l'entier naturel n ;
   - on inverse tous ses bits (les 1 deviennent des 0 et vice versa) ;
   - puis on ajoute 1 (en utilisant l'algorithme d'incrémentation vu plus haut).

   Par exemple, la représentation de l'entier -4 est obtenu en partant de la représentation de 4 sur 8 bits, à savoir 00000100, en inversant les bits, pour obtenir 11111011, puis en ajoutant 1 : c'est donc 11111100.

   Donner la représentation en complément à deux sur 8 bits des entiers $+10$, -17, -76.
3. Notez que le nombre 0 a désormais un unique codage en complément à deux (00000000 sur 8 bits). Reprendre les exemples trouvés à la question 1 et utiliser l'algorithme vu plus haut pour les incrémenter : est-ce que le résultat est désormais correct ?
4. Quel est le plus grand entier positif qu'on peut coder en complément à deux sur 8 bits ? Et le plus petit entier négatif ? Essayer désormais d'ajouter 1 au plus grand entier positif. Vous devriez alors comprendre pourquoi l'on peut dire que le codage par complément à deux est dit *cyclique*.

L'addition de deux entiers codés en binaire s'effectue de manière similaire. L'exercice 2 de ce cours proposait d'ailleurs de réaliser de telles additions binaires à la main :

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

Exercice 29

1. Compléter la fonction suivante pour qu'elle renvoie le tableau contenant la somme des entiers m et n représentés par les tableaux en entrée, sans faire attention à d'éventuels dépassements de capacité ?

```
def additionner(m_en_binaire, n_en_binaire):
    N = len(n_en_binaire)
    résultat = [0] * N
    retenue = 0
    for i in range(........):
        ....
    return résultat
```

2. Tester cet algorithme en additionnant 63 et 42. Tester ensuite cet algorithme en additionnant les représentations en complément à deux sur 8 bits (cf exercice précédent) des entiers 63 et -42. Qu'obtient-on ?

On suppose donc à partir de maintenant qu'on sait effectuer des additions et des soustractions d'entiers. On s'abstrait dans la suite du chapitre de la représentation en binaire des entiers.

4.2 Divisibilité

4.2.1 Division euclidienne

Une autre opération très courante qu'on exécute souvent consiste à calculer le quotient q et le reste r dans la division euclidienne d'un entier naturel n par un entier naturel m, c'est-à-dire l'unique paire $(q,r) \in \mathbf{N} \times \{0,1,\dots,m-1\}$ tel que

$$n = m \times q + r$$

Cette fois-ci, par souci de simplicité, on donne un algorithme qui ne suit pas la méthode *à la main* qu'on exécute habituellement. Par exemple, si on cherche à diviser 382 par 27, généralement, voilà comment on procède :

— d'abord on se demande combien de fois on peut mettre 27 dans 38 : c'est une fois donc on *pose* 1, et il reste 11 ;

— on fait descendre le chiffre 2 des unités dans 382 et on se demande combien de fois on peut mettre 27 dans 112 : c'est quatre fois donc on *pose* 4, et il reste 4.

On a donc le diagramme suivant :

$$\begin{array}{rccc|cc} & 3 & 8 & 2 & 2 & 7 \\ \cline{5-6} - & 2 & 7 & & 1 & 4 \\ \cline{2-3} & 1 & 1 & 2 & & \\ - & 1 & 0 & 8 & & \\ \cline{2-4} & & & 4 & & \end{array}$$

C'est une façon compacte et efficace de mener à bien une division euclidienne : elle est cependant plus délicate à écrire sous forme de code Python. On utilise à la place une méthode plus élémentaire, mais moins efficace. Elle consiste à soustraire de 382 le nombre 27 autant de

fois que nécessaire jusqu'à trouver un entier strictement inférieur à 27 : le nombre de fois qu'il a fallu soustraire 27 est le quotient et l'entier restant finalement est le reste. Par exemple,

$$
\begin{aligned}
382 - 27 &= 355 & 193 - 27 &= 166 \\
355 - 27 &= 328 & 166 - 27 &= 139 \\
328 - 27 &= 301 & 139 - 27 &= 112 \\
301 - 27 &= 274 & 112 - 27 &= 85 \\
274 - 27 &= 247 & 85 - 27 &= 58 \\
247 - 27 &= 220 & 58 - 27 &= 31 \\
220 - 27 &= 193 & 31 - 27 &= 4
\end{aligned}
$$

On retrouve donc bien la division euclidienne $382 = 27 \times 14 + 4$.

Cette méthode par soustraction successive est aisée à écrire, dans une fonction qui prend en entrée deux entiers (le dividende n et le diviseur m), et qui renvoie une paire de deux entiers, le quotient et le reste dans la division euclidienne de n par m.

```
def division_euclidienne(dividende, diviseur) :
    quotient = 0
    while dividende >= diviseur:
        quotient = quotient + 1
        dividende = dividende - diviseur
    reste = dividende
    return (quotient, reste)
```

Dans la suite, on se permet d'utiliser

— le quotient dans la division euclidienne de n par m, qui se note `n//m` en Python ;
— le reste dans la division euclidienne de n par m, qui se note `n%m` en Python.

4.2.2 Calcul du plus grand diviseur commun : algorithme d'Euclide

La division euclidienne a de multiples applications : l'une d'entre elles est de tester si deux entiers positifs a et b sont premiers entre eux, c'est-à-dire si leur seul diviseur positif commun est 1. Par exemple, 14 et 15 sont premiers entre eux, mais 14 et 21 ne le sont pas puisqu'ils sont tous les deux divisibles par 7. Si on note $\text{pgcd}(a,b)$ le plus grand diviseur commun de a et b, on a

$$a \text{ et } b \text{ sont premiers entre eux } \iff \text{pgcd}(a,b) = 1$$

Pour tester si deux entiers sont premiers entre eux, il suffit donc de calculer leur pgcd. Pour cela, le fameux *algorithme d'Euclide* s'applique. Il consiste à faire des divisions euclidiennes successives en remplaçant le dividende par le diviseur et le diviseur par le reste, jusqu'à ce que ce ne soit plus possible car le reste devient nul. Par exemple, pour trouver le pgcd de 21 et 14, on effectue la division euclidienne de 21 et 14 :

$$21 = {\color{blue}14} \times 1 + {\color{red}7}$$

puis on remplace le dividende par 14 et le diviseur par 7 pour obtenir la deuxième division euclidienne :

$${\color{blue}14} = {\color{red}7} \times 2 + 0$$

Le reste devient nul et on en déduit alors que le pgcd de 21 et 14 est le dernier reste non nul, à savoir 7 : $\text{pgcd}(21,14) = 7$.

Exercice 30

Calculer le pgcd de 799 et 345 à la main, par divisions euclidiennes successives.

On peut représenter visuellement l'algorithme d'Euclide avec le diagramme en Figure 4.1.

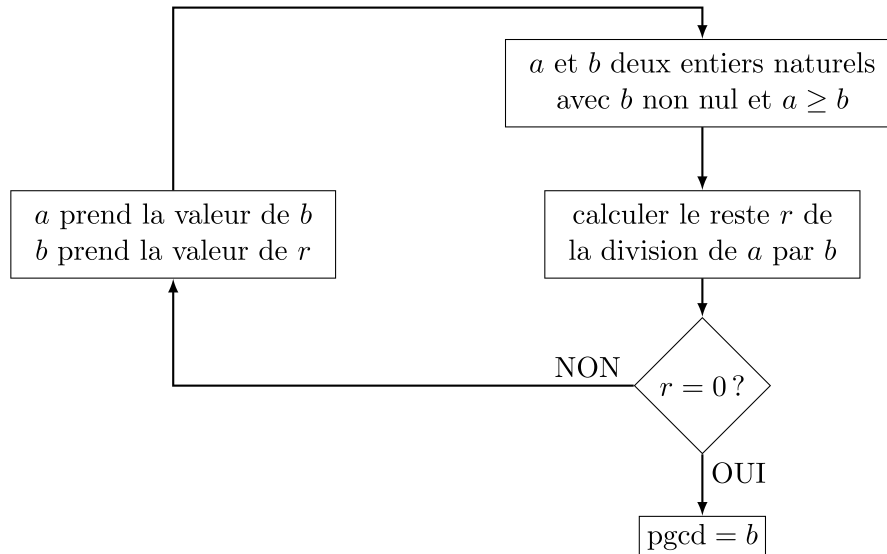


FIGURE 4.1 – Algorithme d'Euclide

Cette représentation graphique illustre qu'on doit faire une action (donner à a la valeur de b et à b la valeur de r, puis calculer le reste r de la division euclidienne de a par b) *tant que* r est non nul. On peut donc écrire tout naturellement le code Python suivant :

```
def pgcd(a, b) :
    # a > b deux entiers non nuls
    r = a % b
    while r > 0:
        a = b
        b = r
        r = a % b
    return b
```

La correction de cet algorithme provient du fait qu'à tout moment dans l'algorithme le pgcd des variables a et b reste inchangé, ce qui tient du fait que si r est le reste dans la division euclidienne de a par b on a

$$\text{pgcd}(a,b) = \text{pgcd}(b,r)$$

La preuve de ce théorème est simple : il suffit de démontrer que les diviseurs communs de a et b sont les mêmes que ceux de b et r. Notons q le quotient dans la division euclidienne de a par b de sorte que $a = b \times q + r$.

- Soit d un diviseur commun de a et b. Alors d divise a et d divise b, donc d divise $a - b \times q = r$. Donc d est un diviseur commun de b et r.
- Soit d un diviseur commun de b et r. Alors d divise $b \times q + r = a$ donc d est un diviseur commun de a et b.

La terminaison de l'algorithme est également simple à démontrer puisque le long d'une itération de la boucle `while` la valeur de la variable r passe à `b % r` c'est-à-dire le reste dans

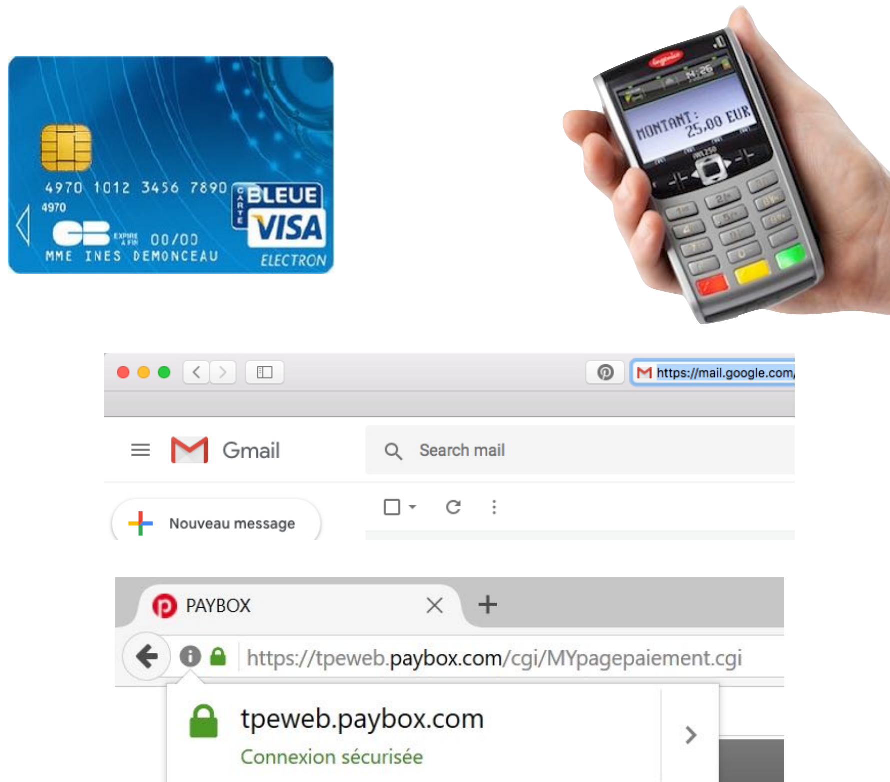


FIGURE 4.2 – Applications du test de primalité réciproque

la division euclidienne de b par r, qui par définition est strictement inférieur à r. La valeur r est donc un variant de boucle. Puisque la boucle `while` s'arrête dès lors que r devient négatif ou nul, on est sûr qu'on ne peut boucler qu'un nombre fini de fois.

4.2.3 Applications de la divisibilité

Mais finalement, est-ce vraiment utile de savoir si deux entiers n et m sont premiers entre eux ? En fait, non seulement c'est utile mais c'est même indispensable dans notre vie de tous les jours. C'est une des opérations de base que l'on exécute très souvent comme illustré en Figure 4.2.

— Lorsqu'on paie avec une carte de crédit dans un commerce ou qu'on retire de l'argent à un distributeur, des informations personnelles critiques transitent sur Internet et la banque envoie l'autorisation de transaction : il faut donc que le terminal de paiement ou le distributeur de billets soit sûr qu'il est bien en train de communiquer avec la banque de manière sécurisée.

— Lorsqu'on se connecte à sa boîte mail, sur un réseau social ou qu'on paie une transaction en ligne, on envoie son mot de passe ou son numéro de carte bancaire sur Internet. On veut donc garantir que personne n'est capable de récupérer facilement ces informations privées. On utilise alors le protocole de communication *https*, la lettre s signifiant *sécurisé*.

Plus généralement, la tâche où s'applique le test de primalité réciproque concerne la cryp-

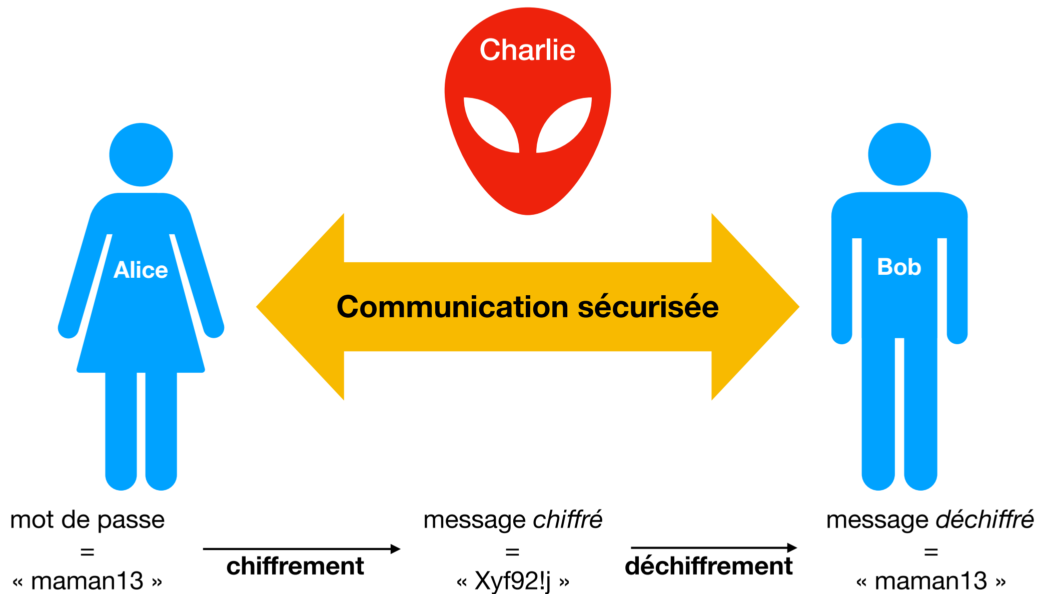


FIGURE 4.3 – Communication sécurisée entre deux entités

tographie. Alice et Bob veulent ainsi communiquer des informations de manière sécurisée, c'est-à-dire sans qu'un intrus, Charlie, puisse récupérer les informations. Par exemple, Alice peut vouloir se connecter à son compte Twitter en envoyant son mot de passe à Bob (qui est alors ici le serveur de Twitter...), comme en Figure 4.3.

Nous avons déjà vu dans les exercices des chapitres précédents des méthodes de chiffrement, que ce soit le chiffrement de César ou de Vigenère. À chaque fois, on s'est aperçu que ces techniques de chiffrement souffraient d'un lourd défaut : on peut rapidement casser le chiffrement en réalisant une cryptoanalyse (en essayant toutes les clés possibles ou bien en guidant sa recherche via l'utilisation de statistiques sur la langue du message à chiffrer). On n'utilise donc pas ces méthodes en réalité. À la place, on utilise des systèmes basés sur l'arithmétique, tels que le cryptosystème RSA (du nom de ses créateurs Ronald Rivest, Adi Shamir et Leonard Adleman). Il s'agit d'un système de chiffrement asymétrique, c'est-à-dire qu'Alice et Bob n'ont pas la même clé de chiffrement/déchiffrement (contrairement aux codes de César ou Vigenère où les différents participants partagent la même clé de décalage). C'est un système basé sur l'utilisation de grands entiers premiers qui implique que les messages eux-mêmes soient d'abord transformés en entiers : c'est facile en utilisant par exemple la table ASCII transformant chaque caractère en entier. La clé d'Alice est une paire d'entiers (e,n), celle de Bob une paire d'entiers (d,n). La procédure de chiffrement, illustrée en Figure 4.4, consiste, pour Alice, à partir d'un message M et de le chiffrer grâce à l'opération $M^e \mod n$. Symétriquement, la procédure de déchiffrement d'un chiffré C consiste, pour Bob, à calculer $C^d \mod n$.

Mais comment sont choisis les clés d'Alice et Bob pour rendre robuste et correct ce protocole de chiffrement ?

1. Tout d'abord, on choisit deux grands entiers premiers distincts p et q.
2. On calcule ensuite leur produit $n = p \times q$.
3. On calcule également $\varphi(n) = (p-1)(q-1)$.

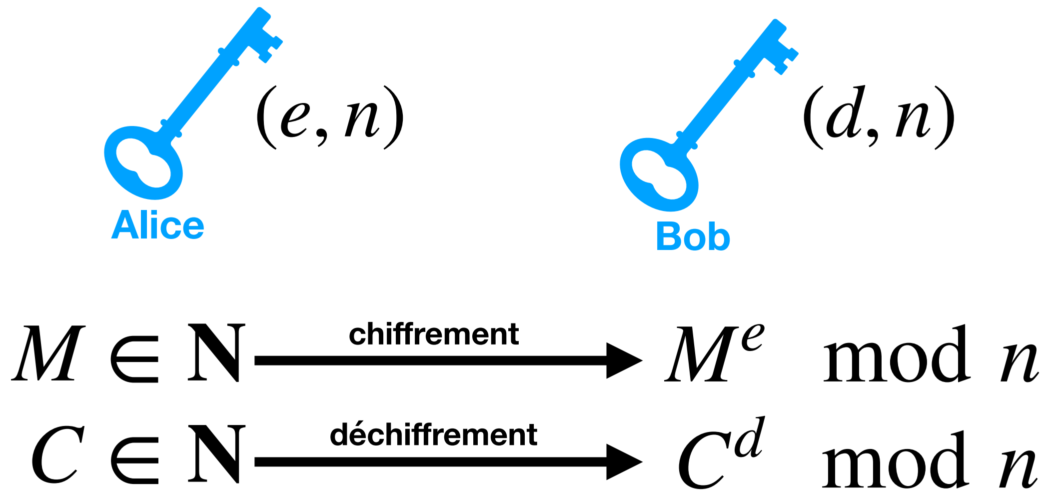


FIGURE 4.4 – Cryptosystème RSA

4. On choisit alors un entier e premier avec $\varphi(n)$.
5. On calcule l'entier d inverse de e modulo $\varphi(n)$, c'est-à-dire un entier d tel que

$$d \times e \equiv 1 \; [\varphi(n)]$$

Il est très facile de calculer l'entier d à l'aide de l'algorithme d'Euclide (étendu). En effet, le fait que e et $\varphi(n)$ soient premiers entre eux (c'est-à-dire que leur pgcd soit égal à 1) garantit l'existence d'une paire de Bézout (d,k) tels que $de + k\varphi(n) = 1$. L'algorithme d'Euclide, légèrement enrichi, permet de calculer une telle paire (d,k).

Une fois crées les clés (e,n) et (d,n), on peut oublier p et q. La propriété clé qui garantit que le cryptosystème RSA est correct est le théorème suivant, qu'on admet dans ce cours (mais dont vous pourrez facilement trouver une preuve sur Internet si cela vous intéresse) :

Théorème 1. *Pour tout message $M \in \mathbf{N}$, $(M^e)^d \equiv M \; [n]$, c'est-à-dire que si on déchiffre le message chiffré obtenu à partir du clair M, on retombe sur le message M initial.*

On le voit, l'arithmétique des entiers premiers (et premiers entre eux) est donc indispensable pour chiffrer efficacement et sûrement des messages qu'on s'échange à longueur de journée.

4.3 Exponentiation

Une question se pose cependant au vu de la procédure de chiffrement et de déchiffrement RSA qu'on vient de découvrir : comment la machine fait-elle pour calculer `M**e % n` ? Plus généralement, comment la machine fait-elle pour calculer l'élévation à une puissance entière positive, par exemple pour calculer 21^{32} ou e^{23} ?

Étant donné un nombre x (cela peut-être un entier ou un réel, qu'on représente alors en machine à l'aide d'un flottant) et un entier naturel n, on souhaite donc calculer le nombre

$$x^n = \underbrace{x \times x \times \cdots \times x}_{n \text{ fois}}$$

Si on suppose que la machine sait réaliser une multiplication (ce qui est du même genre que le calcul d'une addition qu'on a étudié plus tôt), on a donc un algorithme très simple qui consiste à multiplier $n-1$ fois le nombre x avec lui-même. La fonction suivante réalise ces multiplications.

```
def puissance(x, n):
    résultat = x
    for i in range(n - 1):
        résultat = résultat * x
    return résultat
```

L'opération coûteuse dans ce calcul est clairement la multiplication. Combien en effectue-t-on au cours de cet algorithme ? Autant qu'il y a de tours de la boucle `for`, c'est-à-dire $n-1$. Peut-on faire mieux ?

Par exemple, pour calculer x^8, l'algorithme précédent revient aux 7 multiplications

$$x^8 = x \times x \times x \times x \times x \times x \times x \times x$$

Mais on peut clairement faire mieux en remarquant que $x^8 = x^4 \times x^4$, puis que $x^4 = x^2 \times x^2$ puis que $x^2 = x \times x$. En seulement 3 multiplications (qu'on représente ci-dessous par des élévations au carré), on obtient donc

$$x^8 = ((x^2)^2)^2$$

Si on veut calculer x^{13}, on peut de même utiliser l'algorithme précédent réalisant 12 multiplications :

$$x^{13} = x \times x \times x \times x \times x \times x \times x \times x \times x \times x \times x \times x \times x$$

ou bien on peut remarquer que $x^{13} = x^6 \times x^6 \times x$. Si on sait calculer x^6, il faut alors ajouter deux multiplications supplémentaires pour obtenir x^{13}. De même, $x^6 = x^3 \times x^3$ et $x^3 = x^2 \times x$, avec $x^2 = x \times x$. On obtient donc x^{13} à l'aide de 5 multiplications seulement

$$x^{13} = ((x^2 \times x)^2)^2 \times x$$

On voit qu'on ne réalise donc pas le même découpage selon que la puissance n est paire ou impaire, ce qui nous amène à considérer l'algorithme suivant, qu'on appelle d'*exponentiation rapide.*

```
def puissance_rapide(x, n):
    a = 1
    b = x
    m = n
    while m > 0:
        if m % 2 == 1:
            a = a * b
        m = m // 2
        b = b * b
    return a
```

Exercice 31

1. Exécuter l'algorithme d'exponentiation rapide lors du calcul de 2^{10}, en précisant la valeur des variables lors de chaque étape de la boucle `while`.
2. Donner une preuve du fait que l'algorithme termine toujours.
3. Pour prouver que l'algorithme est correct, c'est-à-dire qu'il renvoie bien x^n, une méthode consiste à vérifier qu'à tout moment de l'algorithme on a $\mathtt{x}^{\mathtt{n}} = \mathtt{a} \times \mathtt{b}^{\mathtt{m}}$. Montrer que c'est vrai avant de rentrer dans la boucle `while`, puis que si c'est vrai au début d'une itération de la boucle `while`, alors c'est vrai à la fin de cette itération. Conclure alors à l'aide d'un raisonnement par récurrence.
4. Combien de multiplications sont effectuées par l'algorithme lors du calcul de 2^{10}. Plus généralement, pouvez-vous donner un ordre de grandeur du nombre de multiplications effectuées pour calculer x^n par l'algorithme d'exponentiation rapide, en fonction de n ? Comparer avec l'algorithme `puissance` étudié plus haut.
5. Comme on l'a vu, le cryptosystème RSA demande de savoir calculer des *puissances modulaires*, c'est-à-dire le reste de x^n dans la division euclidienne par k. Sachant que

$$\text{si } a \equiv b \ [k] \qquad \text{alors} \qquad a^n \equiv b^n \ [k]$$

on a intérêt à calculer les puissances en prenant les restes dans la division euclidienne par k à chaque étape. Modifier alors la fonction `puissance_rapide`, en ajoutant un troisième argument k, afin qu'elle calcule le reste de x^n dans la division euclidienne par k. : on s'autorise à utiliser comme opération élémentaire supplémentaire le calcul de `y % k`, le reste dans la division euclidienne de y par k.

4.4 Recherche d'un zéro d'une fonction

Terminons ce chapitre en nous intéressant au calcul de logarithmes, particulièrement pertinent par exemple pour calculer une valeur en décibel d'atténuation de signal (en acoustique par exemple). Développé par John Napier au début du XVIIème siècle, le logarithme *népérien* avait initialement pour objectif de simplifier des calculs, en particulier en remplaçant des multiplications par des sommes grâce à la propriété

$$\forall a \quad \forall b \qquad \ln(a \times b) = \ln(a) + \ln(b)$$

Figure 4.5 – Tables de logarithmes

En l'absence de calculatrices, en effet, les sommes sont bien plus simples à réaliser à la main que des multiplications. En revanche, cela demandait à savoir passer aisément d'une valeur à son logarithme et vice versa. Pendant trois siècles, cela s'est fait à l'aide de tables de logarithmes qui étaient regroupées dans des petits livres, comme l'illustre la Figure 4.5.

Évidemment, il n'est plus question de telles tables de logarithmes aujourd'hui, les calculatrices et ordinateurs faisant le travail pour nous. Mais comment font-ils ? En fait, ils utilisent des techniques mathématiques un peu plus avancées que celles que vous connaissez sans doute à l'heure actuelle (développements limités, séries numériques, résolutions d'équations différentielles...), mais on peut déjà étudier une méthode simple pour faire de tels calculs, en supposant simplement qu'on sait calculer l'exponentielle de n'importe quel nombre flottant (et pas seulement d'un nombre entier positif comme on l'a fait dans la section précédente).

En effet, une fois qu'on sait calculer l'exponentielle, on peut remarquer que

$$x = \ln(14) \iff e^x = 14$$

Si on sait trouver un antécédent par la fonction exponentielle d'un nombre, 14 par exemple, alors on a trouvé son logarithme népérien ln(14). De manière similaire, pour le logarithme en base 2 :

$$x = \log_2(152) \iff 2^x = 152$$

Remarquons désormais que résoudre $e^x = 14$ revient à résoudre $e^x - 14 = 0$. On se ramène donc à rechercher un réel x tel que $e^x - 14 = 0$ ou $2^x - 152 = 0$, c'est-à-dire le zéro d'une fonction mathématique f, comme illustré en Figure 4.6.

Ce problème ressemble beaucoup au problème de la recherche d'un élément dans un tableau, qu'on a étudié dans le chapitre précédent. La différence est qu'on est face à un *tableau infini* ne permettant pas de réaliser la recherche séquentielle consistant à visiter tous les

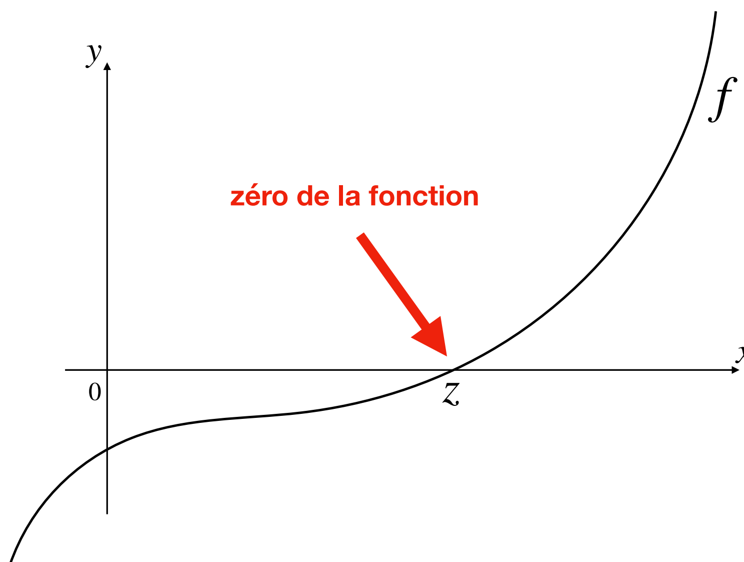


FIGURE 4.6 – Zéro d'une fonction f : abscisse z telle que $f(z) = 0$

éléments du tableau de gauche à droite. En revanche, rien n'empêche d'utiliser une technique de recherche dichotomique (cf Figure 4.7). On commence par encadrer un des zéros de la fonction qu'on suppose continue : on suppose donc connu un intervalle $[a,b]$ tel que f est continue sur cet intervalle et change de signe, c'est-à-dire $f(a) \times f(b) < 0$. Par le théorème des valeurs intermédiaires, on est assuré de l'existence d'un zéro z dans l'intervalle $[a,b]$. Considérons le milieu de l'intervalle $\frac{a+b}{2}$. Deux cas se présentent :

— si $f((a+b)/2) < 0$ (comme dans la figure), alors on sait, toujours grâce au théorème des valeurs intermédiaires, que la fonction f possède un zéro sur l'intervalle $[\frac{a+b}{2},b]$;

— si $f((a+b)/2) > 0$, alors on sait que la fonction f possède un zéro sur l'intervalle $[a,\frac{a+b}{2}]$.

On continue donc ainsi à resserrer l'intervalle encadrant un zéro de la fonction f : sur la figure, on considère ensuite m le milieu de l'intervalle $[\frac{a+b}{2},b]$, puis m' le milieu de l'intervalle $[\frac{a+b}{2},m]$. Quand s'arrête-t-on ? On s'arrête si on finit par tomber exactement sur un zéro, ou alors après avoir obtenu un intervalle $[x,x']$ suffisamment petit pour avoir obtenu une approximation suffisante du zéro recherché.

Outre cet algorithme de recherche dichotomique, d'autres solutions simples existent. L'une d'entre elles est l'*algorithme de Newton* qui consiste à « descendre le long de la courbe ». Cette fois-ci, on suppose de plus que la fonction f est dérivable, de sorte qu'on connait une équation de la tangente à la courbe de f en le point d'abscisse x_0 : $y = f(x_0) + f'(x_0) \times (x - x_0)$. La méthode de Newton se base sur l'idée qu'on peut approcher la courbe d'une fonction par sa tangente : si x est suffisamment proche de x_0 alors $f(x)$ est proche de $f(x_0) + f'(x_0)(x - x_0)$.

En particulier, si on calcule l'abscisse du point d'intersection de la tangente à la courbe en x_0 avec l'axe des abscisses, on espère obtenir une nouvelle abscisse x_1 plus proche du zéro comme le montre la Figure 4.8.

On résout donc l'équation $0 = f(x_0) + f'(x_0)(x - x_0)$ afin de trouver (si la dérivée $f'(x_0)$ est non nulle) l'abscisse $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$. Plus généralement, on construit donc la suite $(x_n)_{n\in\mathbb{N}}$

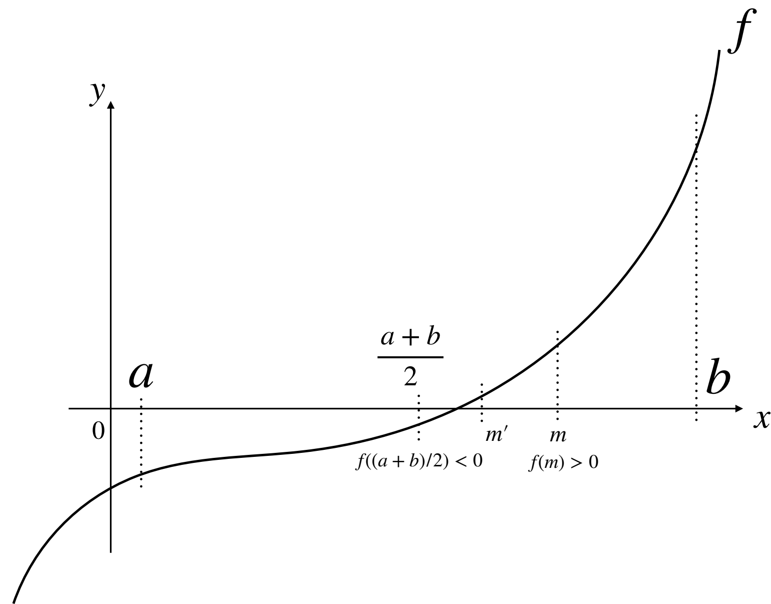


FIGURE 4.7 – Recherche dichotomique du zéro d'une fonction f

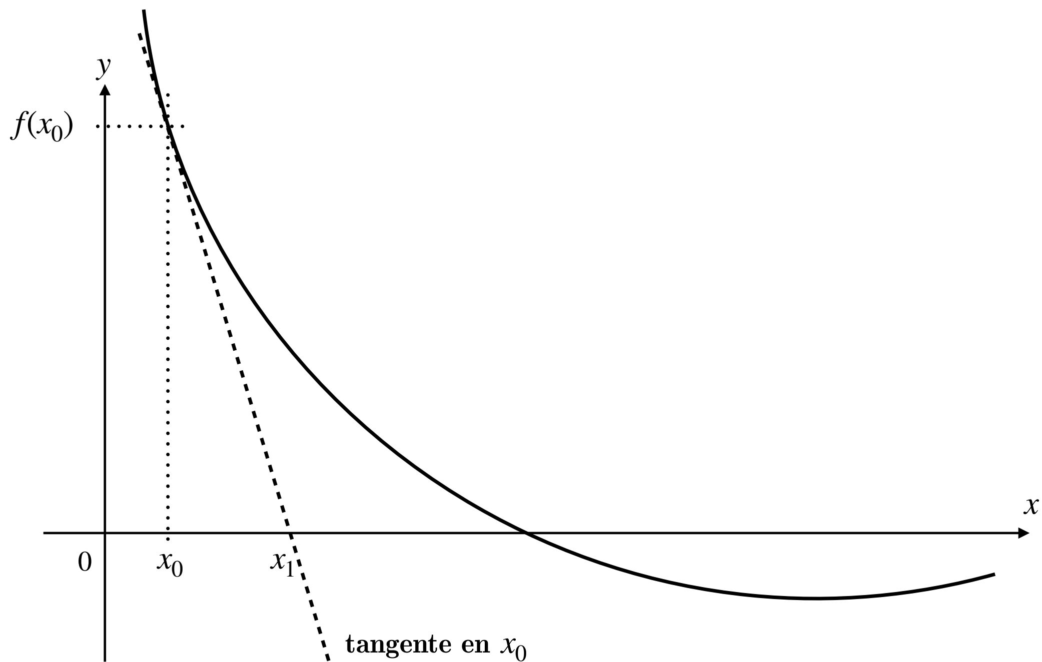


FIGURE 4.8 – Première étape de la méthode de Newton

par récurrence, à partir du point x_0, en posant pour tout n

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

tant que la dérivée $f'(x_n)$ est non nulle (sinon la méthode échoue).

Exercice 32

Exécuter les deux itérations suivantes de l'algorithme de Newton sur le schéma de la Figure 4.8, afin de trouver x_2 et x_3.

On peut donc écrire un algorithme qui calcule les valeurs successives de la suite $(x_n)_{n\in\mathbb{N}}$ qu'on stocke dans une variable `a` évoluant au cours du temps, nécessitant d'avoir en argument non seulement la fonction f, mais aussi sa dérivée f'.

```
def approximation_zero_Newton(f, f', x0):
    a = x0
    while f(a) != 0:
        a = a - f(a)/f'(a)
    return a
```

Comme précédemment dans la recherche dichotomique, cet algorithme n'a aucune raison de terminer a priori, puisqu'on n'est pas garanti de finir par trouver un zéro exactement. On modifie donc cet algorithme pour qu'il ait plus de chances de terminer (même si ce n'est pas encore garanti !), en lui ajoutant une précision $\varepsilon > 0$ (on utilise traditionnellement la lettre grecque « epsilon » pour décrire ce paramètre d'erreur) en dessous de laquelle on se satisfait de l'approximation trouvée, c'est-à-dire qu'on cherche une abscisse z telle que $|f(z)| \leq \varepsilon$. On continue donc la boucle `while` si on n'a pas cette condition vérifiée, c'est-à-dire tant que $|f(a)| > \varepsilon$:

```
def approximation_zero_Newton(f, f', x0, epsilon):
    a = x0
    while abs(f(a)) > epsilon:
        a = a - f(a)/f'(a)
    return a
```

où l'on a utilisé la fonction `abs` de Python pour calculer la valeur absolue.

C'est un algorithme très utilisé qui fonctionne plutôt bien en pratique, mais il faut bien voir qu'il n'est pas correct en général : il se peut que l'algorithme termine et renvoie un résultat qui ne soit pas « proche » d'un zéro de la fonction f. C'est le cas dans l'exemple de la Figure 4.9 où l'algorithme s'arrête à la fin de la première itération, loin du zéro de f.

Exercice 33

On propose un autre critère d'arrêt pour l'algorithme de Newton en remarquant que lorsque l'on s'approche du zéro de f, les abscisses x_n deviennent de plus en plus proches : précisément, la distance de x_n à x_{n+1} tend alors vers 0 lorsque n tend vers l'infini.

1. Comment modifier l'algorithme pour que la condition d'arrêt porte ainsi sur cette distance entre deux abscisses successives ?
2. Pouvez-vous trouver un exemple montrant que ce nouvel algorithme est également incorrect ? On demande donc de dessiner le graphe d'une fonction f et de choisir x_0 et un paramètre d'erreur tels que le nouvel algorithme termine (sans échouer du fait d'une dérivée nulle) en renvoyant un résultat « loin » du zéro de la fonction f.

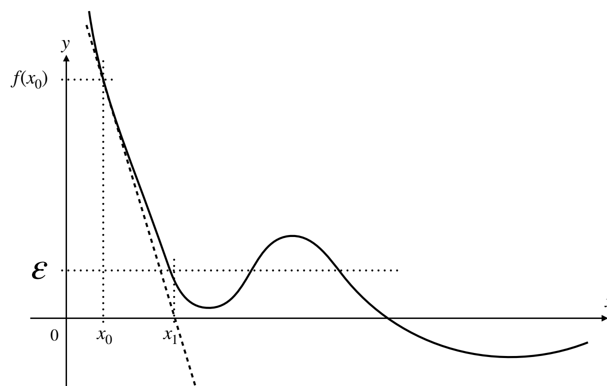


FIGURE 4.9 – Exécution erronée de l'algorithme de Newton : l'algorithme s'arrête mais renvoie une abscisse loin du zéro de la fonction

En partant de la fonction f associant à x la valeur $f(x) = e^x - 14$, on peut appliquer la méthode de Newton pour trouver une valeur approchée de $\ln(14)$. Il suffit de trouver la dérivée de f qui vaut $f'(x) = e^x$ et de partir de n'importe quelle abscisse, par exemple $x_0 = 0$. De même, si on choisit la fonction f associant à x la valeur $f(x) = 2^x - 152$, alors on a $f'(x) = \ln(2) \times 2^x$. Puisqu'on sait estimer précisément $\ln(2)$ (comme précédemment), on sait calculer la fonction dérivée. À nouveau en partant de $x_0 = 0$, cela permet d'estimer précisément la valeur de $\log_2(152)$.

Exercice 34

On cherche désormais à appliquer l'algorithme de Newton pour calculer la racine carrée d'un nombre r positif : cette estimation de la racine carrée $\sqrt{r}$ d'un nombre positif est appelée *algorithme de Héron*. Pour cela, on utilise la fonction f qui à tout réel x associe $x^2 - r$.

1. Décrire explicitement la suite récurrente $(x_n)_{n\in\mathbf{N}}$ de la méthode de Newton appliquée à cette fonction.
2. En déduire un algorithme (n'utilisant pas la fonction `approximation_zero_Newton`) qui donne une approximation de la racine carrée d'un nombre r donné en argument, avec une précision ε donnée en argument également : on souhaite renvoyer un résultat a vérifiant $|a - \sqrt{r}| \leq \varepsilon$, mais attention on ne peut pas utiliser la valeur de $\sqrt{r}$...

Chapitre 5

Graphes : modélisation et parcours

Nous savons désormais mieux comment on (ou une machine) peut calculer avec des algorithmes, sur des ensembles d'objects stockés dans des tableaux ou des données numériques. En particulier, on a vu comment trier des tableaux, ce qui permet de répondre à l'un des problèmes posé dans le chapitre d'introduction (cf Figure 1.17). Un autre problème que nous posions alors, dans la Figure 1.16, est le calcul du plus court chemin dans Google Maps : on avait dit alors qu'on pouvait abstraire le problème à l'aide d'un graphe avec des nœuds représentant les intersections de rue et des arcs reliant ces nœuds avec la durée en minutes du trajet correspondant. Ce chapitre et le suivant ont pour objectif de mieux comprendre cette structure de données supplémentaire que sont les graphes. Dans ce premier chapitre, nous allons voir que les graphes sont partout et nous allons répondre à la question que pose Google Maps. Le chapitre suivant permettra d'étudier d'autres problèmes en lien avec les graphes.

5.1 Les graphes sont partout

Commençons par observer différents graphes qu'on croise, consciemment ou pas, dans notre vie. Les réseaux de métro (cf Figure 5.1) sont sans doute l'un des exemples les plus visibles de graphes où des stations sont connectées par des rails de métro (de différentes lignes).

Nous sommes sans doute moins conscients qu'Internet est également un gigantesque graphe dans lequel des utilisateurs (représentés par une adresse, qu'on appelle *adresse IP*) sont reliés entre eux via des serveurs, comme illustré en Figure 5.2.

Toujours sur Internet, il existe des sous-réseaux spécifiques, par exemple ceux de réseaux sociaux tels que Twitter ou Instagram.

Mais nous n'avons pas attendu la création d'Internet pour créer des réseaux connectant des personnes : il existe ainsi des graphes représentant les collaborations entre chercheurs, les liens dans le graphe représentant un article écrit en commun. La Figure 5.4 montre un exemple de tel graphe, centré sur le chercheur mathématicien Paul Erdös.

5.2 Graphes : application au diamètre des réseaux sociaux

Définissons donc plus formellement ce qu'est un graphe, pour pouvoir mieux raisonner dessus.

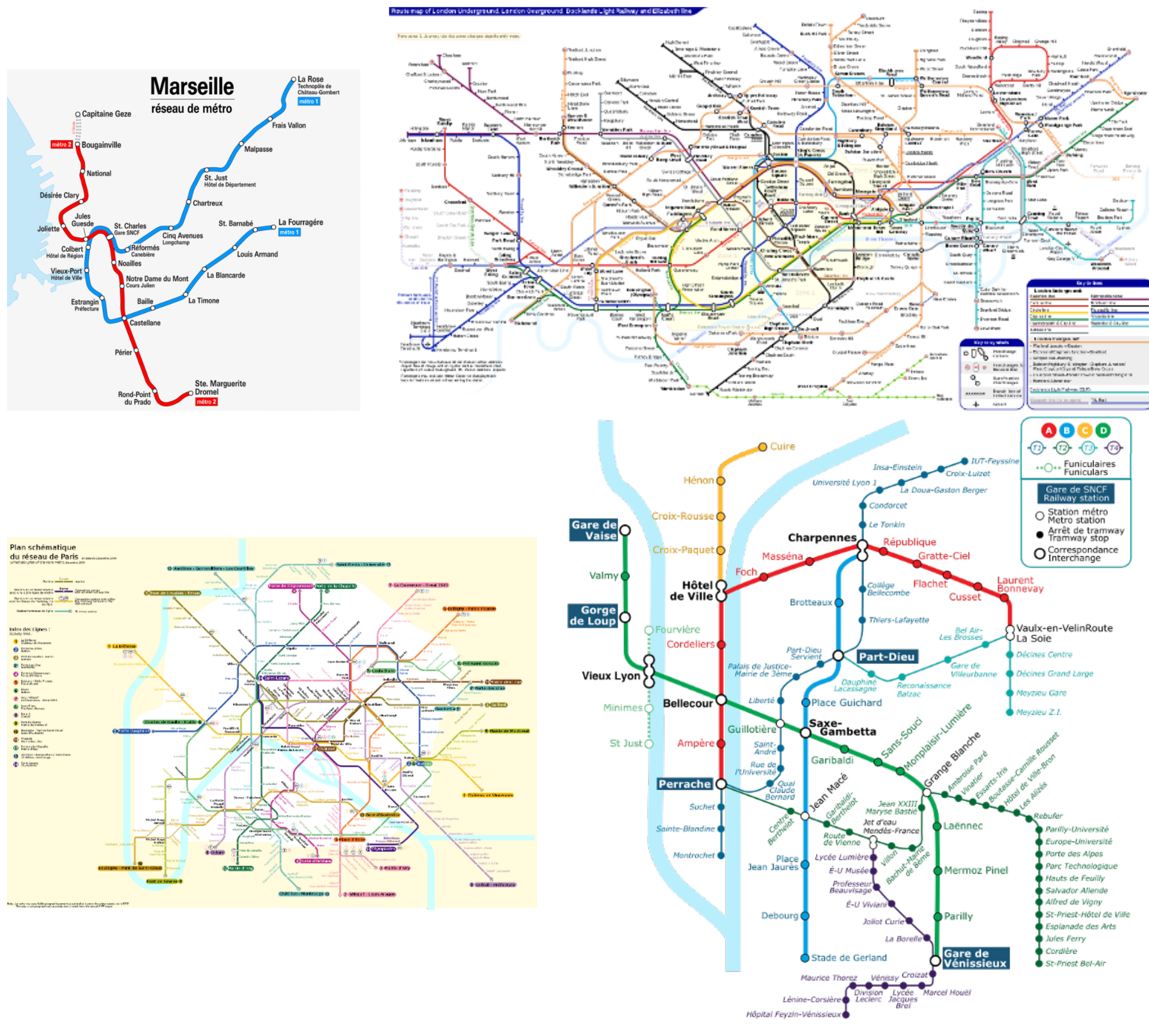


Figure 5.1 – Réseaux de métro

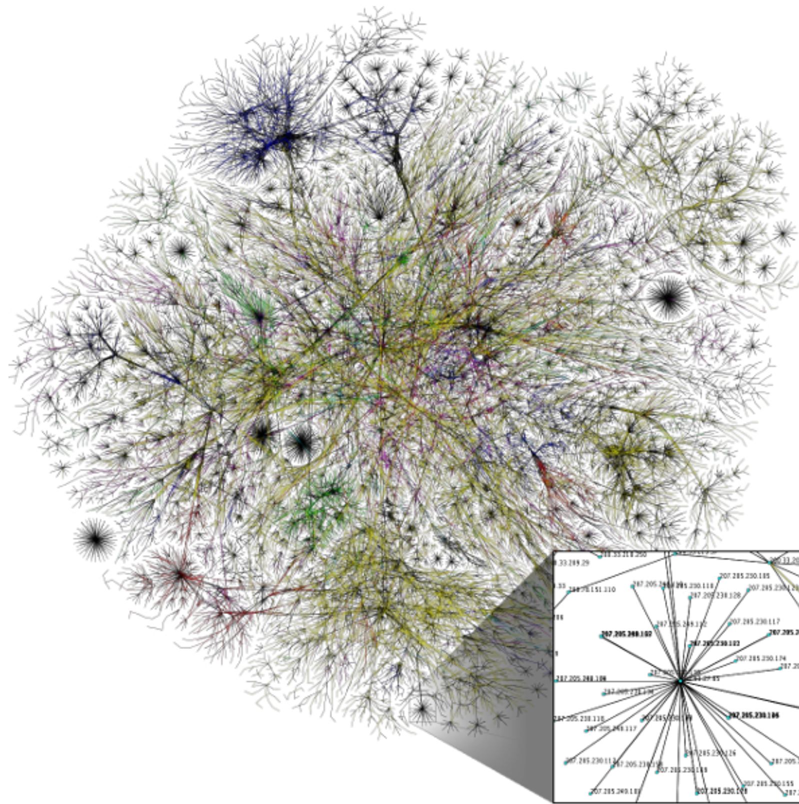


FIGURE 5.2 – Illustration d'une partie du graphe d'Internet

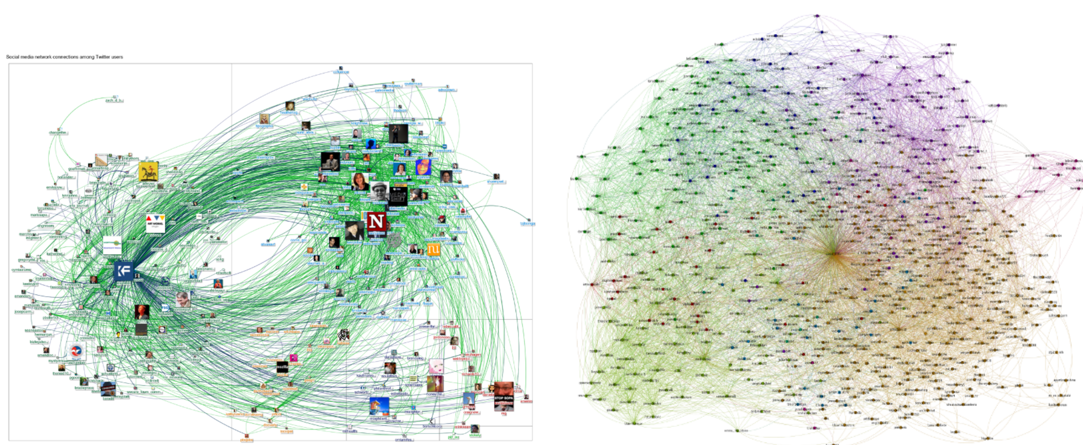


FIGURE 5.3 – Graphes tirés des connexions entre utilisateurs de réseaux sociaux tels que Twitter (à gauche) ou Instagram (à droite)

FIGURE 5.4 – Graphe de collaborations scientifiques centré sur Paul Erdös

Définition 5. Un graphe est la donnée d'un ensemble fini S de *sommets* (ou nœuds) et d'un ensemble fini A de paires de sommets qu'on appelle *arêtes* : si u et v sont deux sommets, l'arête $\{u,v\}$ représente le fait que les sommets u et v sont reliés dans le graphe. On note parfois $G = (S,A)$ le graphe.

Voici un exemple de graphe pour lequel l'ensemble de sommets est $S = \{0,1,2,3,4\}$ et l'ensemble d'arêtes est $A = \{\{0,1\}, \{0,3\}, \{1,2\}, \{1,4\}, \{2,4\}, \{2,3\}\}$:

Un *chemin* (on dit parfois aussi *chaîne*) dans un tel graphe $G = (S,A)$ est une suite de sommets $s_1,s_2,s_3,\ldots,s_{n-1},s_n$ qui sont reliés par une arête, c'est-à-dire tel que $\{s_1,s_2\} \in A$, $\{s_2,s_3\} \in A$, ..., et $\{s_{n-1},s_n\} \in A$. La longueur d'un chemin est le nombre d'arêtes qu'il emprunte. Par exemple 0,1,2,4,2,3 est un chemin de longueur 5 dans le graphe précédent.

Un graphe non orienté permet, par exemple, de représenter les relations de connaissance ou d'amitié dans un réseau social (de telles relations sont généralement symétriques : Marc est ami avec Sarah si et seulement si Sarah est amie avec Marc...). La Figure 5.5 donne un

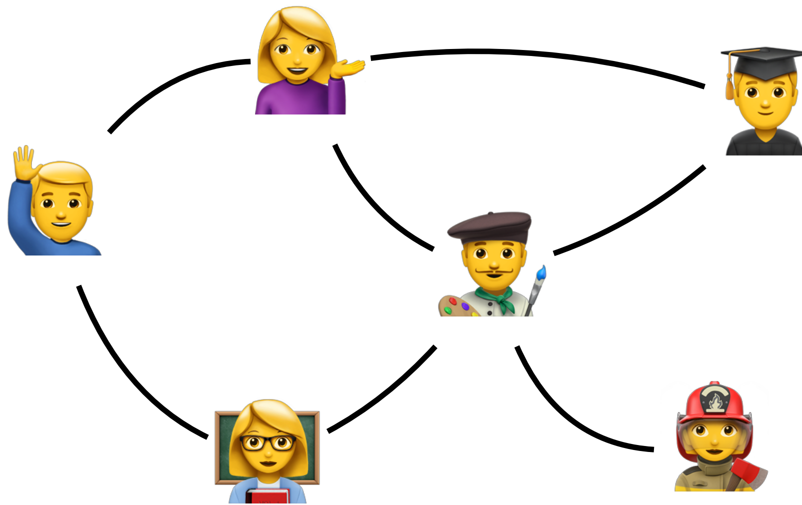


FIGURE 5.5 – Aperçu du graphe d'un réseau social

petit aperçu d'un tel réseau social : les sommets sont les utilisateurs du réseau social et les arêtes représentent les liens de connaissance ou d'amitié.

Modéliser un tel réseau social sous la forme d'un graphe permet de visualiser le réseau (comme dans les figures du début du chapitre...), mais aussi de raisonner sur le réseau. Par exemple, des recherches ont montré qu'« Un utilisateur de Facebook (parmi les 1,59 milliards d'utilisateurs) est connecté à n'importe quelle autre personne par le biais de 3,5 personnes en moyenne » (voir le post en anglais de Facebook Research `https://research.fb.com/three-and-a-half-degrees-of-separation/`) : cela veut dire que sur Facebook par exemple, il y a de fortes chances que Madonna soit amie avec un des amis des amis de vos amis. Cela fait référence à l'expérience que le psychologue social Stanley Milgram avait conduite dans les années 60, montrant qu'en moyenne, deux individus américains ne sont séparés que par 6 degrés de connaissance en moyenne.

Formellement, la distance entre deux sommets est la longueur minimale d'un chemin qui les relie. La distance moyenne entre deux sommets du graphe de Facebook est donc estimée à 3,5. Cela permet d'en déduire des informations sur le diamètre du graphe de Facebook : le *diamètre* d'un graphe est la distance maximale séparant deux sommets quelconques du graphe.

Exercice 35

Du temps de Pagnol, une bergère veut traverser la rivière Durance avec un chou, un mouton et un loup. Malheureusement, pas de pont à l'horizon et elle ne possède qu'une minuscule barque dans laquelle elle peut naviguer avec un seul de ses « compagnons » d'aventure. En sa présence, le mouton n'ose pas manger le chou, pas plus que le loup n'ose manger le mouton, mais ils n'hésiteraient pas à satisfaire leurs appétits si la bergère tournait le dos. Comment doit-elle s'y prendre pour amener tout le monde de l'autre côté de la rivière ? Utilisons un graphe pour l'aider !

1. Quelles sont les configurations possibles de cette aventure ? On pourra les décrire en observant les personnages qui peuvent se trouver sur la rive de départ et la rive d'arrivée.

FIGURE 5.6 – Graphe sur-imprimé sur un morceau du plan de Marseille

2. Modéliser alors le problème sous la forme d'un graphe dont les sommets sont les configurations possibles et les arêtes représentent l'évolution possible de l'aventure.
3. Résoudre le problème à l'aide du graphe précédent. Décrire à la bergère les allers-retours qu'elle doit faire. Combien de traversées la bergère doit-elle faire ? Existe-t-il plusieurs solutions avec ce nombre minimal de traversées ? Les donner toutes.

5.3 Graphes orientés : application aux graphes de configuration

Considérons désormais le plan d'une ville : il y a des routes qui se croisent à des intersections. On peut donc sur-imprimer au-dessus du plan un graphe dont les sommets sont les intersections et les arêtes sont les morceaux de route sans intersection. On a représenté une partie d'un tel graphe en Figure 5.6.

Trouver un chemin pour aller d'un point A à un point B dans la ville, revient donc à trouver un chemin du sommet A au sommet B dans le graphe. Si cela fonctionne parfaitement pour trouver un chemin pour un piéton, c'est nettement moins intéressant pour une voiture, puisqu'on n'a pas l'information des sens interdits ! Il nous faut donc ajouter cette information dans les graphes, qu'on appelle ainsi *orientés*. Un exemple de graphe orienté pour le plan de Marseille est donné en Figure 5.7.

Définition 6. Un graphe orienté est la donnée d'un ensemble fini S de *sommets* (ou nœuds)

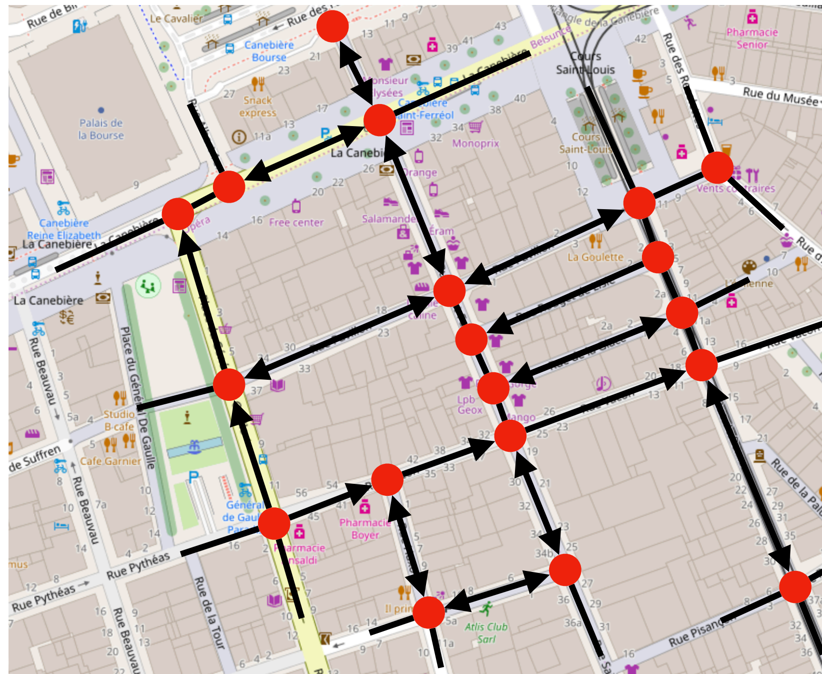
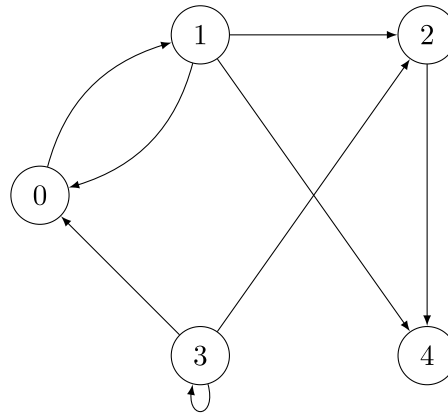


FIGURE 5.7 – Graphe orienté sur-imprimé sur un morceau du plan de Marseille

et d'un ensemble fini A de couples de sommets qu'on appelle *arcs* : si u et v sont deux sommets, l'arc (u, v) représente le fait qu'il existe un arc partant du sommet u et allant au sommet v. On note parfois $G = (S, A)$ le graphe.

Notez qu'on utilise la notation $\{u, v\}$ pour une paire de sommets (différents), une arête dans un graphe non orienté, alors qu'on utilise la notation (u, v) pour un couple de sommets (potentiellement le même sommet deux fois), un arc dans un graphe orienté. En particulier, si 1 et 2 sont deux sommets du graphe, les arêtes $\{1, 2\}$ et $\{2, 1\}$ sont les mêmes, alors que les arcs $(1, 2)$ et $(2, 1)$ sont différents. Pour les rues dans un plan, une rue à sens unique sera représentée par un arc dans un seul sens, alors qu'une rue à double-sens est représentée par deux arcs, un pour chaque sens.

Voici un exemple de graphe orienté pour lequel l'ensemble de sommets est $S = \{0, 1, 2, 3, 4\}$ et l'ensemble d'arcs (qu'on représente par des liens avec des flèches) est $A = \{(0, 1), (1, 0), (1, 2),$ $(1, 4), (2, 4), (3, 0), (3, 2), (3, 3)\}$:



Un *chemin* dans un tel graphe orienté $G = (S,A)$ est une suite de sommets $s_1,s_2,s_3,\ldots,$ s_{n-1},s_n qui sont reliés par un arc, c'est-à-dire tel que $(s_1,s_2) \in A$, $(s_2,s_3) \in A$, ..., et $(s_{n-1},s_n) \in A$. La longueur d'un chemin est le nombre d'arcs qu'il emprunte. Par exemple 3,3,0,1,4 est un chemin de longueur 4 dans le graphe orienté précédent.

Les graphes orientés permettent de représenter des *graphes des configurations* qu'on utilise dans diverses applications. Si on cherche à modéliser un système qui évolue dans le temps, on appelle configuration d'un tel système son état à un instant donné : l'évolution du système, sa dynamique, peut alors parfois être représentée comme un graphe dans lequel les sommets abritent les configurations et un arc relie une configuration c_1 à une configuration c_2 s'il est possible de passer de manière élémentaire de c_1 à c_2 lors de l'évolution du système.

Exercice 36

Dans le film *Die Hard 3* (*Une journée en enfer*), les deux héros, John McClane et Zeus Carver, doivent résoudre l'énigme de Simon Gruber pour arrêter le compte à rebours d'une bombe. Voici l'énigme : « Sur la fontaine, il y a deux bidons : l'un a une contenance de 5 gallons, l'autre de 3 gallons. Remplissez l'un des bidons de 4 gallons d'eau exactement et placez-le sur la balance. La minuterie s'arrêtera. Soyez extrêmement précis : un gramme de plus ou de moins et c'est l'explosion ! ». Les nerfs de John McClane sont alors mis à rude épreuve pour trouver une solution. Il commence par remarquer très justement qu'on ne peut pas remplir le bidon de 3 gallons avec 4 gallons d'eau. Il faut donc trouver le moyen de mettre exactement 4 gallons d'eau dans le bidon de 5 gallons. Dans la scène du film (que vous pouvez consulter en français sur `https://www.youtube.com/watch?v=pmk2mNf9iqE`), John commence par donner une première idée peu convaincante puisqu'elle termine par la nécessité de remplir le bidon de 3 gallons au tiers, ce qu'on ne sait faire précisément... Le film propose ensuite une solution très partielle, coupée au montage. Appliquons donc des méthodes de graphes orientés pour retrouver la meilleure solution possible que les héros appliquent pour s'en sortir.

1. Une configuration du système correspond au volume d'eau contenu dans chacun des deux bidons. On peut donc représenter une telle configuration par une paire (a,b) où a est le volume d'eau contenu dans le bidon de 5 gallons et b le volume d'eau contenu dans le bidon de 3 gallons, avec $0 \leq a \leq 5$ et $0 \leq b \leq 3$. Les actions élémentaires possibles du système sont de remplir un des deux bidons (qu'il soit initialement vide ou pas), vider un des deux bidons (qu'il soit ini-

tialement plein ou pas) et transférer le contenu d'un des bidons dans l'autre jusqu'à ce que ce dernier soit plein. En partant de la configuration initiale (0,0), construire le graphe orienté décrivant entièrement l'ensemble des configurations et la dynamique du système. *Attention, certains mouvements ne sont possibles que dans un seule sens, raison pour laquelle nous utilisons un graphe orienté dans ce cas.*

2. En déduire une solution la plus courte possible permettant d'aller de la configuration (0,0) à une configuration de la forme $(4,b)$ où exactement 4 gallons d'eau se trouvent dans le gros bidon. Décrire à John McClane la suite d'opérations qu'il doit effectuer pour arrêter la bombe au plus vite.
3. Imaginons la suite de l'énigme désormais. Dans une autre fontaine, le terroriste a placé deux bidons, l'un de 6 gallons et l'autre de 15 gallons. S'il demande à John McClane de remplir l'un des deux bidons avec exactement 5 gallons d'eau, pourra-t-il éviter l'explosion de la bombe ?

5.4 Codage d'un graphe

Pour pouvoir calculer sur un graphe, il faut savoir comment on va coder un graphe. La façon la plus simple (il existe d'autres façons qu'on n'étudiera pas dans ce cours) consiste à stocker le graphe à l'aide d'une *matrice d'adjacence*. Il s'agit d'un tableau bidimensionnel M dont les lignes et les colonnes sont indexées par les sommets du graphe et tel que la case $M_{u,v}$ en ligne u et en colonne v vaut 1 dès lors qu'il existe un arc (u, v) ou une arête $\{u, v\}$ dans le graphe, et vaut 0 sinon. En Python, on accèdera à cette case à l'aide de `M[u][v]` et on utilise donc cette notation dans tout ce chapitre. Voici les matrices d'adjacence du graphe non orienté (à gauche) et orienté (à droite) présentés plus tôt :

$$M_1 = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix} \qquad M_2 = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Par exemple, on a $M_1[2][3] = 1$ signifiant qu'il existe une arête reliant les sommets 2 et 3 dans le graphe non orienté, et $M_2[2][1] = 0$ signifiant qu'il n'y a pas d'arc allant de 2 à 1 dans le graphe orienté.

Notons que la matrice d'adjacence d'un graphe non orienté est *symétrique* (vis-à-vis de la diagonale Nord-Ouest - Sud-Est), c'est-à-dire que pour tous sommets u et v, $M[u][v] = M[v][u]$: il existe une arête $\{u, v\}$ dans le graphe si et seulement s'il existe l'arête $\{v, u\}$.

Exercice 37

1. Quelle propriété d'un graphe orienté est représentée par le fait que dans sa matrice d'adjacence M, on a pour tout sommet u, $M[u][u] = 1$?
2. Dans un graphe non orienté, que vaut $M[u][u]$ pour tout sommet u ?
3. Quelle condition sur la matrice d'adjacence M d'un graphe orienté permet de représenter le fait que « pour tout chemin de longueur 2 allant d'un sommet u

à un sommet v, il existe un raccourci, c'est-à-dire un arc de u à v » ?

De la même manière qu'on a parcouru un tableau (unidimensionnel) à l'aide d'une boucle `for`, on peut parcourir toutes les cases d'une matrice (bidimensionnelle) à l'aide de deux boucles `for` imbriquées. Par exemple, si on souhaite vérifier qu'une matrice d'adjacence est symétrique, on peut utiliser le pseudo-code suivant :

```
def est_symétrique(M):
    n = len(M)
    for u in range(n):
        for v in range(n):
            if M[u][v] != M[v][u]:
                return False
    return True
```

On y utilise la fonction `len` qui renvoie le nombre de lignes de la matrice `M`. Ensuite, on recherche une preuve de non symétrie de la matrice (auquel cas l'algorithme s'arrête en plein milieu en retournant `False` dès que possible) : s'il n'a pas trouvé de preuve de non symétrie, c'est que la matrice est symétrique et on renvoie donc `True`. Noter qu'on fait deux fois trop de travail dans ce code, puisqu'on teste chaque couple de sommets (u,v) deux fois... On peut évidemment faire mieux, en ne testant qu'une seule fois chaque couple, en supposant toujours que $u < v$ (en particulier, il n'y a pas besoin de vérifier la propriété pour $u = v$) :

```
def est_symétrique(M):
    n = len(M)
    for u in range(n-1):
        for v in range(u+1, n):
            if M[u][v] != M[v][u]:
                return False
    return True
```

Exercice 38

1. Écrire un algorithme en pseudo-code, qui prend en entrée la matrice d'adjacence d'un graphe orienté, et renvoie le nombre d'arcs dans le graphe.
2. Comment modifier votre algorithme pour qu'il compte le nombre d'arêtes d'un graphe non orienté ?

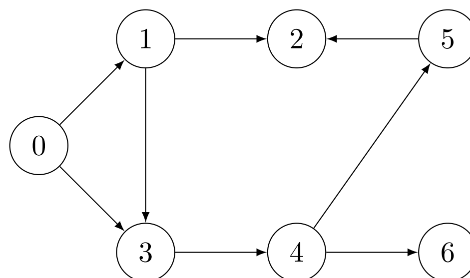
5.5 Parcours de graphe

Il est temps d'automatiser les calculs que l'on souhaite faire dans des graphes, que ce soit la recherche d'un itinéraire pour aller d'un point A à un point B, ou la résolution d'énigmes telles que celles de Pagnol ou de Die Hard 3. Le point commun de ces différents problèmes est la recherche d'un chemin allant d'un sommet source à un sommet cible. On résout ce problème d'*accessibilité* dans un graphe à l'aide d'un algorithme qui *parcourt* le graphe, c'est-à-dire visite les sommets du graphe petit à petit en suivant des arêtes/arcs du graphe. Pour simplifier, on donne les explications qui suivent uniquement dans le cas des graphes orientés, mais tout fonctionne de la même façon avec des graphes non orientés.

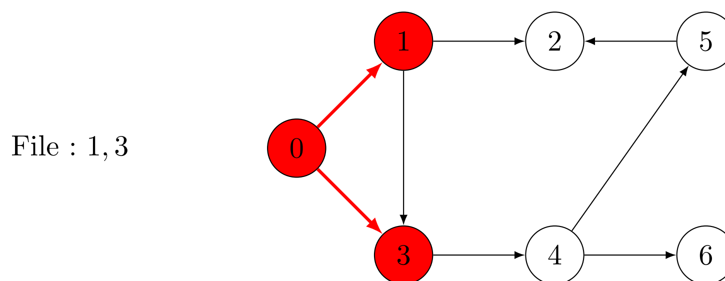
L'algorithme de parcours qu'on va considérer est le *parcours en largeur*. On part d'un sommet source s du graphe. L'idée est alors d'imiter le gonflement d'un ballon de baudruche

depuis s : au fur et à mesure où le ballon se gonfle, il voit de plus en plus de sommets du graphe, ceux qui sont de plus en plus éloignés du sommet s. Sur le cours Ametice en ligne, une activité vous est proposé pour découvrir le parcours en largeur, à l'aide d'une vidéo et d'exercices associés. Si vous pouvez y accéder, faites-le avant ou en même temps que vous lisez la suite... Sinon, toutes les informations sont redonnées dans la suite et fin de ce chapitre.

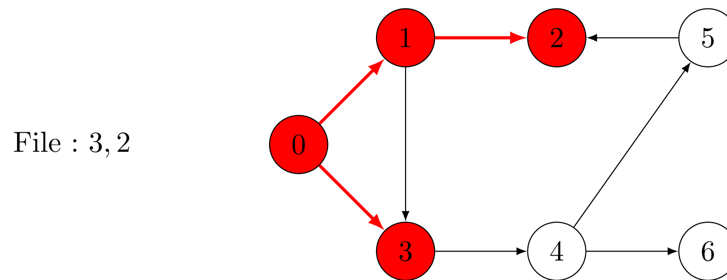
Pour exécuter le parcours en largeur, nous allons utiliser une des structures de données dont nous avons parlé au début du chapitre 3, les files qui permettent de stocker un ensemble de clients en se souvenant de l'ordre dans lequel ils sont arrivés afin de servir les clients dans leur ordre d'arrivée. Dans le cas du parcours en largeur, les clients seront les sommets du graphe. Considérons ainsi le graphe ci-dessous qu'on souhaite parcourir à partir de la source $s = 0$:



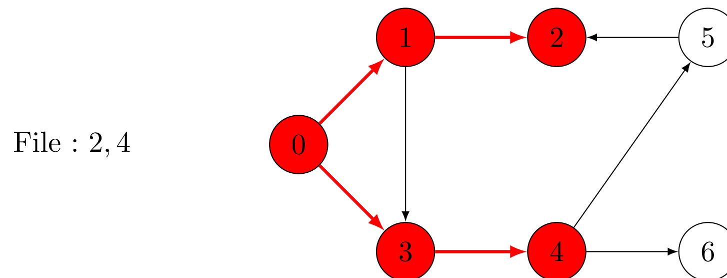
On va colorier les sommets avec la couleur rouge (en plus du blanc initial) pour enregistrer de l'information. La première chose qu'on fait est d'ailleurs de colorer en rouge la source $s = 0$ et de l'insérer dans la file d'attente. Ensuite, on traite l'unique sommet dans la file d'attente. Traiter un sommet u signifie regarder tous ses voisins, c'est-à-dire considérer tous les sommets v tels que (u, v) est un arc du graphe. En l'occurrence, puisqu'on traite le sommet 0, on va donc considérer ses voisins 1 et 3. Pour chacun de ses voisins, s'ils sont blancs, on les colorie en rouge et on les insère dans la file d'attente. Après avoir traité entièrement le sommet 0, on arrive donc dans la situation suivante, dans laquelle on colorie également en rouge les arcs qui nous ont permis d'aller de la source 0 aux différents sommets déjà découverts :



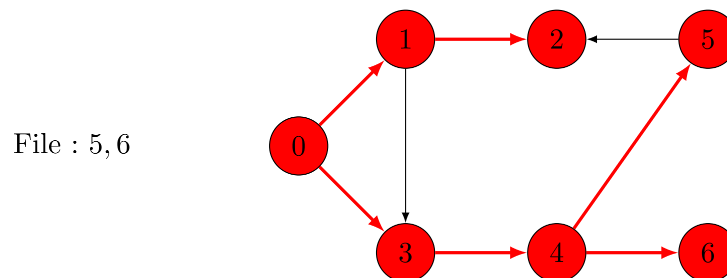
La file d'attente contient désormais les sommets 1 et 3, et on va supposer qu'on y a inséré d'abord 1, puis 3. Ainsi, on extrait d'abord de la file d'attente le sommet 1. Il a deux voisins, 2 et 3. Pour le sommet 2, comme avant, on le colore en rouge et on l'insère dans la file. Mais le sommet 3, lui, est déjà rouge et on ne le considère donc pas : en particulier, on ne colorie pas en rouge l'arc $(1, 3)$. Une fois le sommet 1 entièrement traité, on est donc dans la situation suivante :



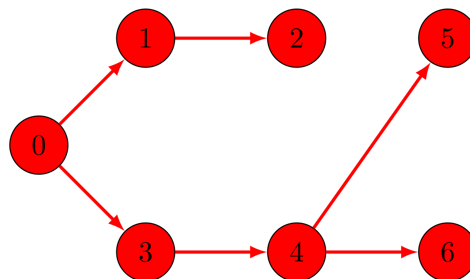
On traite ensuite le sommet inséré il y a le plus longtemps dans la file d'attente, c'est-à-dire le sommet 3. Il n'a qu'un seul voisin, le sommet 4, qu'on colorie donc en rouge :



On traite ensuite le sommet 2, qui n'a aucun voisin. Il ne reste ensuite que le sommet 4 dans la file d'attente. Il a deux voisins blancs, les sommets 5 et 6, qu'on traite donc pour obtenir la situation



Il reste ensuite les sommets 5 et 6 à traiter, dans cet ordre, mais tous les sommets étant déjà rouges, on ne fait rien de plus. Le graphe précédent est donc la situation finale sur laquelle on s'arrête. Observons qu'on a bien découvert tous les sommets du graphe qu'on pouvait atteindre via un chemin depuis le sommet 0. De plus, si on ne regarde que les arêtes colorées en rouge :



on obtient une représentation d'un chemin possible pour aller de la source $s = 0$ à n'importe quel sommet rouge : pour aller de 0 à 5 par exemple, il faut suivre le chemin $0, 3, 4, 5$. Notons

qu'il s'agit d'un plus court chemin (en terme du nombre d'arcs visités) pour aller de 0 à 5. C'est une propriété toujours satisfaite par le parcours en largeur : il permet de construire les plus courts chemins issus de la source s.

Voici une façon de décrire, à l'aide d'un pseudo-code, l'algorithme de parcours en largeur, dans lequel on suppose que les n sommets du graphe sont numérotés $\{0, 1, 2, \ldots, n-1\}$ comme dans les exemples précédents :

```
def parcours_en_largeur(M, s):
    n = len(M)
    H = graphe_vide(n)          #  graphe sans arcs avec n sommets
    F = file_vide()
    couleur = ["blanc"] * n
    couleur[s] = "rouge"
    insérer(F, s)
    while not est_vide(F):
        u = extraire(F)
        for v in range(n):
            if (M[u][v] == 1) and (couleur[v] == "blanc"):
                couleur[v] = "rouge"
                H[u][v] = 1
                insérer(F, v)
    return H                    #  graphe des chemins minimaux
```

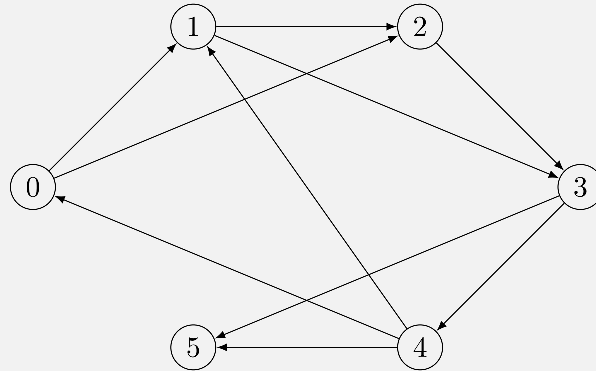
Il prend en argument la matrice d'adjacence M du graphe, et le sommet source s. Cet algorithme retourne la matrice d'adjacence d'un graphe H qui ne contient que les arcs traversés en parcourant un chemin de s à chacun des autres sommets accessibles, c'est-à-dire les arcs rouges dans l'explication précédente. On suppose connue une fonction `graphe_vide` qui retourne une matrice carrée remplie de 0, c'est-à-dire la matrice d'adjacence d'un graphe à n sommets et sans arc. On suppose aussi connue les fonctions permettant de travailler avec une file d'attente :

- `file_vide()` qui initialise une file vide ;
- `est_vide(F)` qui teste si la file d'attente `F` est vide ;
- `insérer(F,u)` qui insère le sommet `u` dans la file d'attente `F` ;
- `extraire(F)` qui renvoie le sommet en tête de la file d'attente `F` et le supprime de la file.

Les couleurs des sommets sont stockées dans un tableau `couleur`.

Exercice 39

1. Exécuter l'algorithme de parcours en largeur sur le graphe G représenté ci-dessous à partir du sommet $s = 0$, en montrant toutes les étapes de l'algorithme (avec la coloration des sommets et l'état de la file dans les configurations intermédiaires). Représenter aussi le graphe H retourné par la fonction.



2. Écrire en Python une fonction `calculer_chemin`(H, s, t) qui prend en argument le graphe des chemins minimaux H construit par la fonction `parcours_en_largeur`(G, s) et affiche le chemin de H qui commence par le sommet source s et se termine avec le sommet t. Pour simplifier, on pourra afficher les sommets du chemin en ordre inverse : par exemple, si le chemin entre s et t est $s \to u \to v \to t$, on pourra afficher ≪ t v u s ≫.

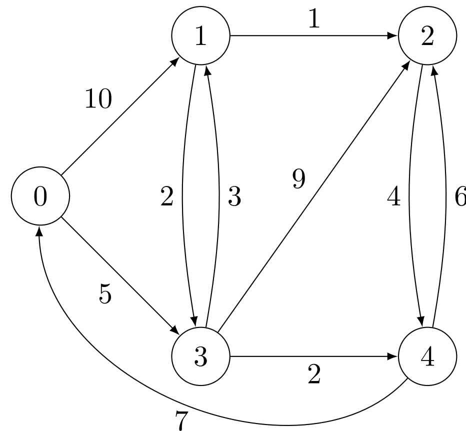
5.6 Plus courts chemins dans des graphes pondérés : algorithme de Dijkstra

L'algorithme de parcours en largeur permet donc de trouver des plus courts chemins dans des graphes. Ici, plus court veut dire ≪ qui utilise le moins d'arcs/arêtes possible ≫. C'est suffisant pour résoudre le problème de Pagnol et de Die Hard 3, mais pas pour trouver des plus courts chemins dans un plan, puisqu'alors ce n'est pas tant le nombre d'arcs qui compte que le temps ou la distance parcourue au total le long de l'itinéraire choisi.

Pour résoudre ce problème, il faut donc incorporer dans la modélisation en graphe une information supplémentaire, que ce soit le temps pour parcourir l'arc (c'est-à-dire le temps qu'il faut pour aller en voiture d'une intersection à une autre), ou plus simplement la longueur de l'arc (c'est-à-dire la distance qui sépare les deux intersections). On utilise pour cela la notion de graphe pondéré.

Définition 7. Un graphe pondéré est la donnée d'un graphe orienté (S, A) et d'une fonction $p\colon A \to \mathbf{N}$ associant un poids (qu'on suppose entier naturel ici) à chaque arc du graphe. On le représente à l'aide d'une matrice d'adjacence M à coefficients dans $\mathbf{N} \cup \{+\infty\}$ dans laquelle le coefficient $M[u][v]$ vaut $p(u, v)$ si $(u, v) \in A$, et vaut $+\infty$ sinon.

Voici un exemple de graphe pondéré dans lequel on fait apparaître le poids de l'arc à côté de celui-ci :

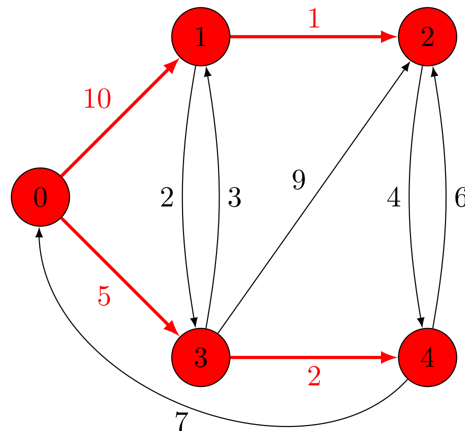


On a ainsi, par exemple, $p(0,1) = 10$ et $p(4,0) = 7$. On peut coder ce graphe pondéré avec la matrice d'adjacence

$$M = \begin{pmatrix} +\infty & 10 & +\infty & 5 & +\infty \\ +\infty & +\infty & 1 & 2 & +\infty \\ +\infty & +\infty & +\infty & +\infty & 4 \\ +\infty & 3 & 9 & +\infty & 2 \\ 7 & +\infty & 6 & +\infty & +\infty \end{pmatrix}$$

Pour un chemin $s_1, s_2, s_3, \ldots, s_{n-1}, s_n$ dans le graphe (S, A), le *poids* du chemin est la somme des poids des arcs empruntés, c'est-à-dire $p(s_1, s_2) + p(s_2, s_3) + \cdots + p(s_{n-1}, s_n)$. Un *plus court chemin* d'un sommet u à un sommet v est un chemin de poids minimal parmi tous les chemins allant de u à v (s'il existe un tel chemin). Par exemple, le chemin $0, 1, 2$ est un chemin de 0 à 2, mais ce n'est pas un plus court chemin car le chemin $0, 3, 1, 2$ est plus *court* en terme de poids : c'est d'ailleurs un plus court chemin pour aller de 0 à 2. On appelle *distance* de u à v le poids d'un plus court chemin de u à v. Par exemple, la distance de 0 à 2 est donc 9, alors que la distance de 0 à 4 vaut 7.

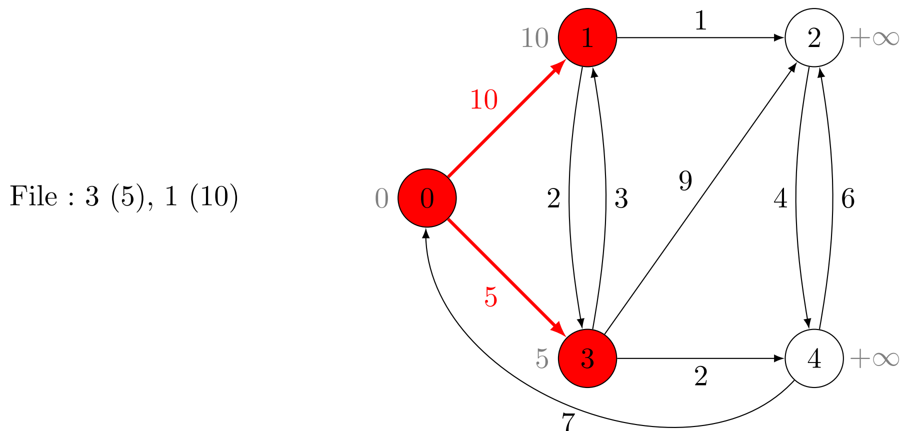
On cherche donc un algorithme qui permet de calculer la distance d'une source s à tout autre sommet, de la même manière que le parcours en largeur permettait de trouver tous les sommets qu'on peut atteindre à partir de s. En plus des distances, on aimerait aussi pouvoir trouver des plus courts chemins. Clairement l'algorithme de parcours en largeur n'est plus correct, puisqu'il renvoie le résultat suivant pour le graphe pondéré précédent, à partir de la source $s = 0$:



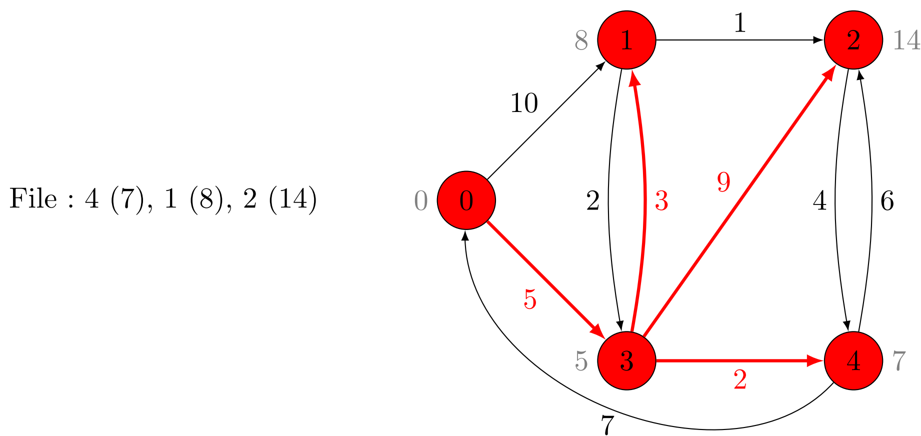
ce qui est incorrect puisqu'alors on en déduirait que la distance de 0 à 1 vaut 10, alors même qu'elle vaut 8.

On doit donc corriger l'algorithme. C'est Edsger Wybe Dijkstra (né en 1939 et mort en 2002) qui a proposé une façon élégante de corriger l'algorithme de parcours en largeur pour qu'il fonctionne dans des graphes pondérés. Son idée : plutôt que de stocker les sommets à traiter dans une file d'attente, utilisons donc plutôt une file de priorité. Il a donc utilisé le concept de file prioritaire à la caisse d'un magasin, dans le contexte des graphes pondérés... Ici, chaque sommet u aura donc une priorité qui correspond à l'estimation courante qu'on a de la distance de la source s à u. Au début, on part avec une estimation très pessimiste associant une distance $+\infty$ à tout sommet u, sauf la source à qui on associe la distance 0. À chaque étape, on traite le sommet qui a *la plus faible priorité* dans la file d'attente, c'est-à-dire celui dont on pense qu'il est le plus proche de la source possible.

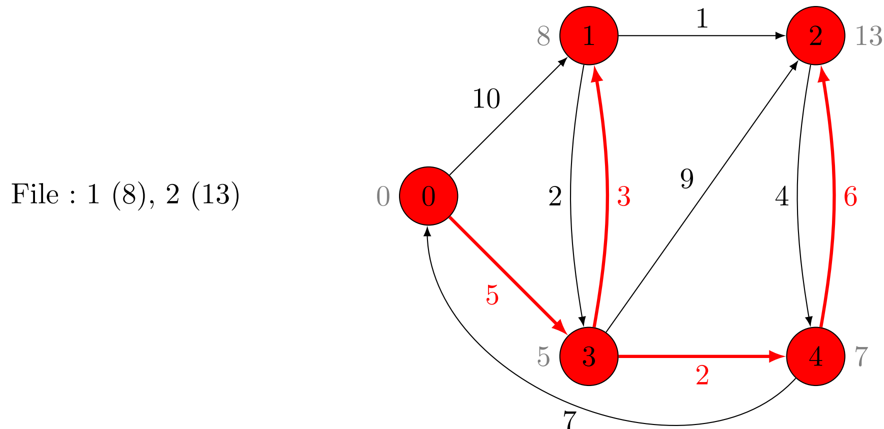
Exécutons l'algorithme de Dijkstra sur l'exemple précédent. Comme pour le parcours en largeur, on commence par colorier la source $s = 0$ en rouge et lui associer la distance 0. On traite alors ce sommet en considérant ses deux voisins, 1 et 3. On peut donc mettre à jour les distances connues de la source à ces deux sommets. On insère alors dans la file de priorité les sommets avec ces distances. On représente ci-dessous la file de priorité en rappelant pour chaque sommet de la file sa priorité. On imprime aussi à côté de chaque sommet du graphe la distance estimée courante.



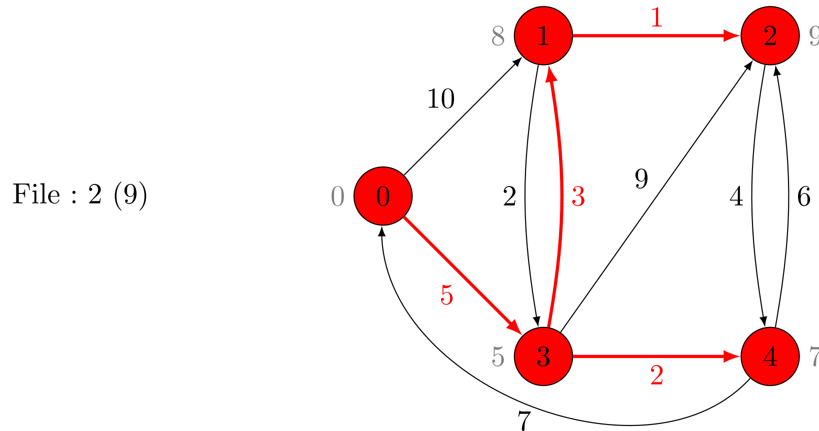
Bien qu'on ait sans doute traiter le voisin 1 avant le voisin 3, la priorité de 3 est plus faible que celle de 1 : c'est donc avec ce sommet qu'on continue. On considère donc les voisins de 3, à savoir 1, 2 et 4. C'est là que l'algorithme se distingue du parcours en largeur. Ici, même si le sommet 1 est rouge, il faut mettre à jour l'information de distance, puisqu'on a trouvé un chemin $0, 3, 1$ de poids 8, plus petit que l'estimation courante de la distance. On doit donc mettre à jour les distances et les priorités dans la file (cela revient à dire qu'on a une file à la caisse d'un magasin dans laquelle les priorités des clients peuvent changer au cours du temps...). On met également à jour la coloration des arcs, puisqu'on souhaite que les arcs colorés en rouge représentent les plus courts chemins. On traite ensuite les voisins 2 et 4 en les insérant dans la file d'attente. On arrive donc à la situation suivante :



Le sommet prioritaire dans la file est désormais le sommet 4. Rien à faire avec son voisin 0, puisque l'estimée courante de la distance, 0, est imbattable. Par contre, on met à jour l'estimation de la distance pour le voisin 2 :



On traite de même le sommet 1 qui remet à jour la distance estimée au sommet 2 :



Il reste le sommet 2 à traiter qui n'induit aucun changement : le graphe précédent est donc la situation finale. Ce graphe contient les informations des distances de 0 à chacun des sommets et on peut également reconstituer des plus courts chemins, en ne considérant que les arcs rouges.

Afin d'écrire cet algorithme [1], il nous faut disposer d'une file de priorité. On peut l'initialiser à l'aide d'une fonction `file_priorité_vide()` qui renvoie une file de priorité vide F. Par rapport à une file d'attente, les éléments de F seront associés à une priorité (un entier naturel). On suppose connue les opérations suivantes :

— on peut insérer un nouvel élément u dans la file, en précisant sa priorité $p \in \mathbf{N}$ à l'aide de la fonction `insérer_priorité(F, u, p)` ;

— `est_vide(F)` qui teste si la file de priorité `F` est vide ;

— l'élément prioritaire est celui de **plus petite priorité** : on peut récupérer cet élément à l'aide de la fonction `extraire_prioritaire(F)` qui supprime au passage l'élément renvoyé ;

1. Rassurez-vous, ce sera l'algorithme le plus difficile de ce cours, il n'est donc pas impossible qu'il vous donne quelques sueurs froides...

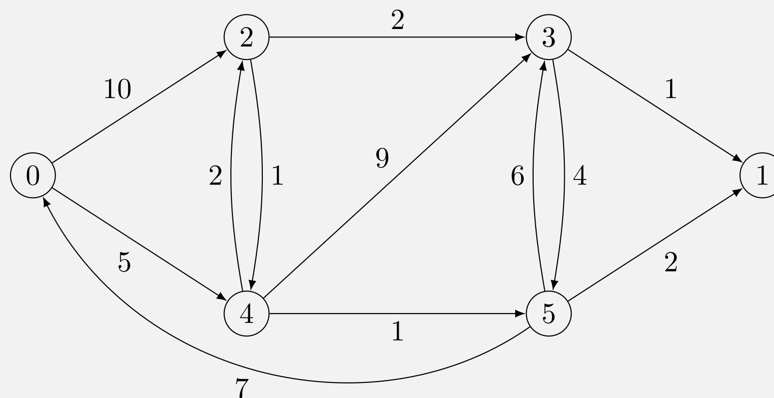
— on peut mettre à jour la priorité d'un élément u de la file pour lui attribuer la nouvelle priorité p dans F, à l'aide de la fonction `mettre_à_jour_priorité(F, u, p)`.

Une autre différence notable entre le parcours en largeur et l'algorithme de Dijkstra réside dans la nécessité de maintenir les estimations des distances (on utilise un tableau `distance` pour cela) et de mettre à jour les arcs rouges : pour cela, plutôt que de maintenir un graphe `H` comme précédemment, on utilise plutôt un tableau `prédécesseur` qui associe à chaque sommet v rouge son prédécesseur dans le graphe `H`, c'est-à-dire l'unique sommet u tel que (u, v) est un arc rouge de `H`. Si le sommet est blanc (ou pour la source qui n'a pas de tel prédécesseur), on utilise le symbole $\perp$ pour dénoter l'absence de prédécesseur. Ces transformations permettent d'obtenir l'algorithme suivant :

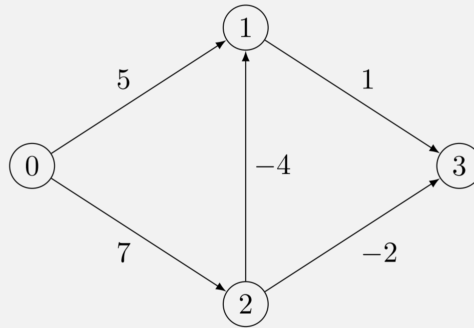
```
def dijkstra(M, s):
    n = len(M)
    distance = [+∞] * n
    prédécesseur = [⊥] * n
    F = file_priorité_vide()
    couleur = ["blanc"] * n
    couleur[s] = "rouge"
    distance[s] = 0
    insérer_priorité(F, s, 0)
    while not est_vide(F) faire
        u = extraire_prioritaire(F)
        d = distance[u]
        for v in range(n):
            if (couleur[v] == "blanc") and (M[u][v] != +∞):
                couleur[v] = "rouge"
                prédécesseur[v] = u
                distance[v] = d + G[u][v]
                insérer_priorité(F, v, distance[v])
            elif d + G[u][v] < distance[v]:
                prédécesseur[v] = u
                distance[v] = d + G[u,v]
                mettre_à_jour_priorité(F, v, distance[v])
    return prédécesseur
```

Exercice 40

1. Exécuter l'algorithme de Dijkstra sur l'exemple ci-dessous en partant de la source $s = 0$:



2. En déduire un plus court chemin du sommet 0 au sommet 1.
3. Jusque-là, nous avons étudié uniquement des graphes pondérés avec des poids entiers positifs ou nuls : pourtant, on pourrait imaginer des graphes avec des poids négatifs, par exemple si le poids de l'arc représente des échanges d'argent (vente ou achat de produits). Exécuter l'algorithme de Dijkstra sur l'exemple ci-dessous où plusieurs arcs ont des poids négatifs :



4. Qu'en déduisez-vous sur l'algorithme de Dijkstra ?

Chapitre 6

Théorie des graphes : chemins eulériens et coloration

On l'a vu dans le chapitre précédent, les graphes sont très utiles pour représenter des situations dans lesquelles on se pose une question d'accessibilité d'un sommet à un autre : un algorithme de parcours (ou l'algorithme de Dijkstra dans le cas des graphes pondérés) permet alors de résoudre cette question.

Il existe d'autres problèmes pertinents qu'on peut se poser sur les graphes, nous amenant à raisonner sur les graphes, avant de produire à nouveau des algorithmes pour résoudre automatiquement ces problèmes. Dans ce chapitre, nous allons en étudier deux bien différents : les graphes eulériens et la coloration de graphe.

6.1 Graphes eulériens : le problème des sept ponts de Königsberg

Considérons dans un premier temps un problème concret que Leonhard Euler, un mathématicien du XVIIIème siècle, alors résident à Saint-Petersbourg, s'est posé en visitant la ville de Königsberg (qui s'appelle Kaliningrad de nos jours). Cette ville est séparée en plusieurs parties par la rivière Pregel au-dessus de laquelle sept ponts passent, comme le montre le plan en Figure 6.1.

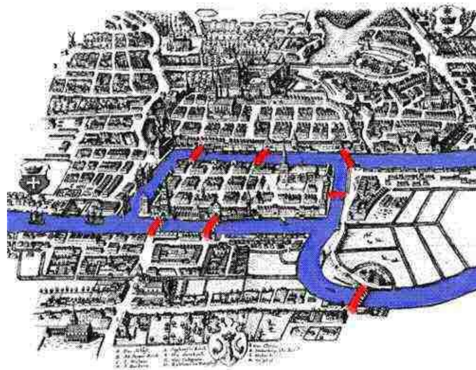


Figure 6.1 – Les sept ponts de Königsberg

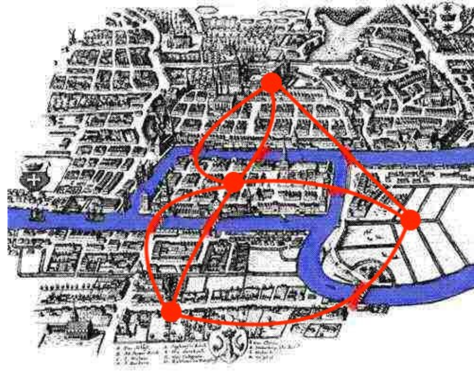
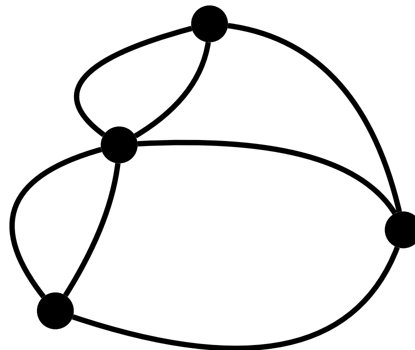


Figure 6.2 – Le graphe sous-jacent au problème des sept ponts de Königsberg

Un guide touristique se pose alors la question, pour attirer une foule de touristes toujours plus grande dans ses visites, de savoir s'il est possible de faire visiter la ville en faisant un circuit passant une fois par chaque pont, mais sans passer deux fois par le même pont. On veut donc partir d'un endroit dans la ville, faire un tour qui passe par chaque pont une et une seule fois, puis être revenu au point de départ.

On voit bien que la carte précédente recèle bien trop de détails qu'on peut ignorer pour répondre à la question. Il suffit de ne garder que les ponts et la façon dont ils sont reliés entre eux, par les rives et les îles de Königsberg, comme représenté en Figure 6.2.

Si on ne garde que le graphe représenté en rouge, on arrive à la modélisation suivante :



Notez que ce graphe est un peu particulier. C'est un graphe non orienté ayant quatre sommets, mais il y a plusieurs arêtes reliant la même paire de sommets : on appelle souvent ce genre de graphe des multi-graphes, signifiant qu'on s'autorise à mettre plusieurs arêtes plutôt qu'une seule entre certaines paires de sommets. Dans un tel graphe non orienté, on appelle *cycle eulérien* tout cycle (un chemin qui revient à son point de départ) qui traverse chaque arête du graphe une et une seule fois. Partez donc de n'importe quel sommet, puis essayez de créer un cycle eulérien dans le graphe précédent.

Vraisemblablement, vous n'y arriverez pas : il vous restera toujours une arête au moins non visitée alors que vous serez coincé dans un sommet d'où vous avez déjà visité toutes les arêtes y arrivant... Rassurez-vous, ce n'est en rien de votre faute. Leonhard Euler lui-même s'est fait la même réflexion et a fini par se convaincre que, dans ce graphe, il n'y a pas de cycle eulérien. Il en a même déduit une recette miraculeuse permettant de décider

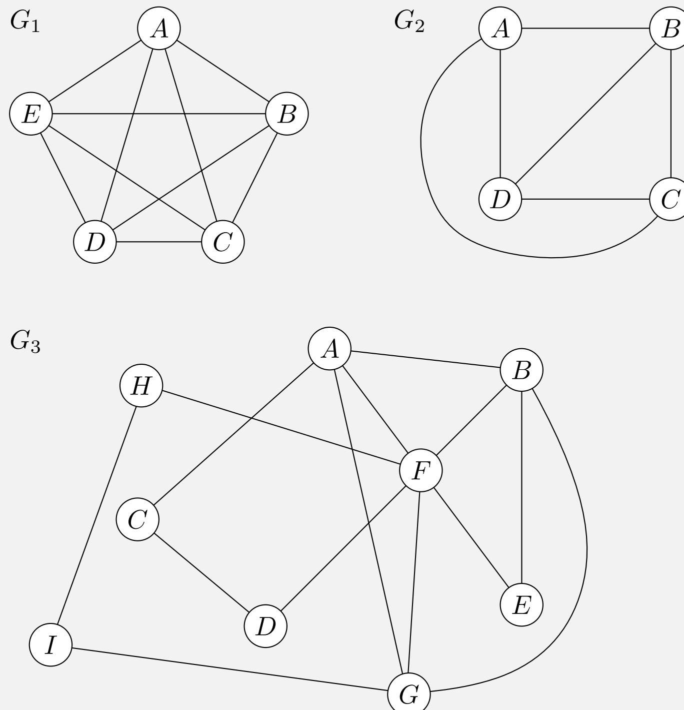
en un instant s'il y a un cycle eulérien ou pas dans un graphe. Pour cela, notons dans un premier temps que la présence d'un cycle visitant toutes les arêtes du graphe n'est possible que si le graphe est en un seul morceau, c'est-à-dire qu'il est possible de relier toute paire de sommets par un chemin (une suite d'arêtes consécutives) : on parle alors de graphes *connexes*. Le graphe de Königsberg est connexe. Sa recette miraculeuse consiste à compter, pour un sommet quelconque, le nombre d'arêtes arrivant dans ce sommet, qu'on appelle le *degré* du sommet. Dans le graphe de Königsberg, trois sommets ont degré 3 et le sommet représentant l'île centrale a degré 5. Euler remarqua que s'il existe un cycle eulérien, alors celui-ci visite toutes les arêtes du graphe (par définition) : pour chaque sommet, on doit donc rentrer autant de fois dedans qu'on en sort. Autrement dit, le degré de chaque sommet doit être pair. Ce qui est intéressant, c'est que cette condition est aussi suffisante :

Théorème 2. *Un graphe non orienté connexe admet un cycle eulérien si et seulement si chaque sommet est de degré pair.*

Cette condition permet tout de suite d'affirmer qu'il n'y a donc pas de cycle eulérien dans le graphe de Königsberg, puisque les quatre sommets ont un degré impair.

Exercice 41

En utilisant la caractérisation précédente, dites pour chacun des graphes suivants s'ils admettent un cycle eulérien. *Attention, on ne demande pas de trouver un tel cycle eulérien s'il existe, mais vous pouvez bien sûr essayer si vous le voulez !*



Comment prouver la condition suffisante du théorème d'Euler, à savoir que si chaque sommet d'un graphe connexe a un degré pair, alors il existe un cycle eulérien ? Utilisons une preuve par algorithme, c'est-à-dire qu'on va donner un algorithme qui construit un cycle eulérien dès lors que les sommets sont tous de degré pair : la preuve de terminaison et de correction de l'algorithme donne alors une preuve du théorème d'Euler.

C'est l'algorithme de Hierholzer qui nous permet de construire un cycle eulérien (s'il en existe un). On peut le décrire de manière informelle ainsi :

— Choisir n'importe quel sommet initial v

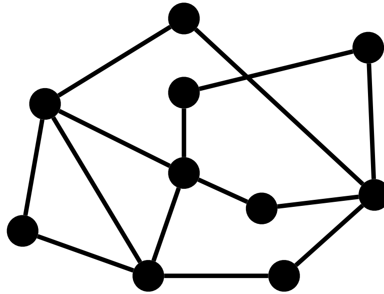
— Suivre un chemin arbitraire d'arêtes jusqu'à retourner à v, obtenant ainsi un cycle c

— **Tant qu**'il y a des sommets u dans le cycle c avec des arêtes qu'on n'a pas encore choisies :

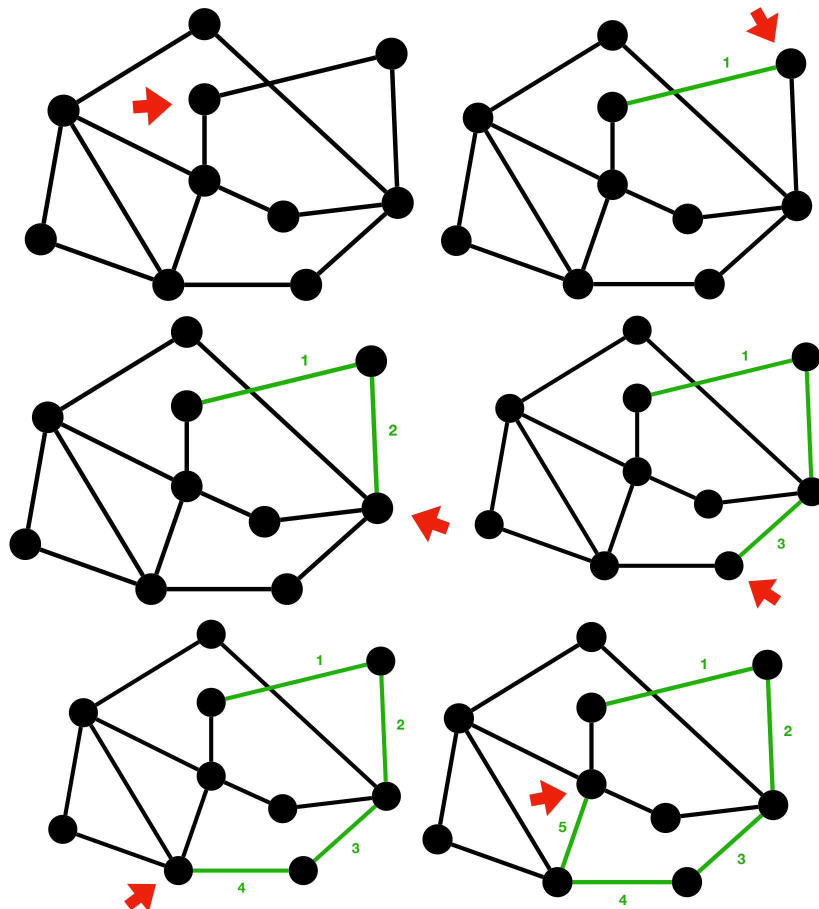
  — Suivre un chemin à partir de u, n'utilisant que des arêtes pas encore choisies, jusqu'à retourner à u, obtenant un cycle c'

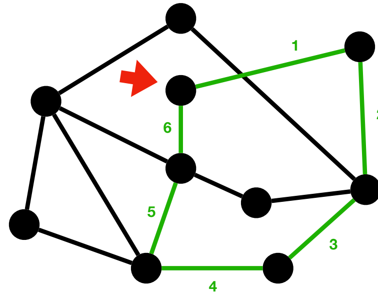
  — Prolonger le cycle c par c'

Voici une exécution de l'algorithme sur le graphe suivant :

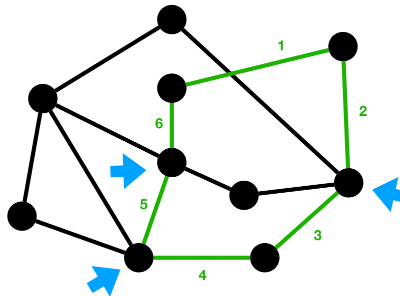


On choisit arbitrairement un sommet initial (marqué par la flèche rouge), puis on suit un chemin arbitraire d'arêtes jusqu'à être retourné au sommet initial :

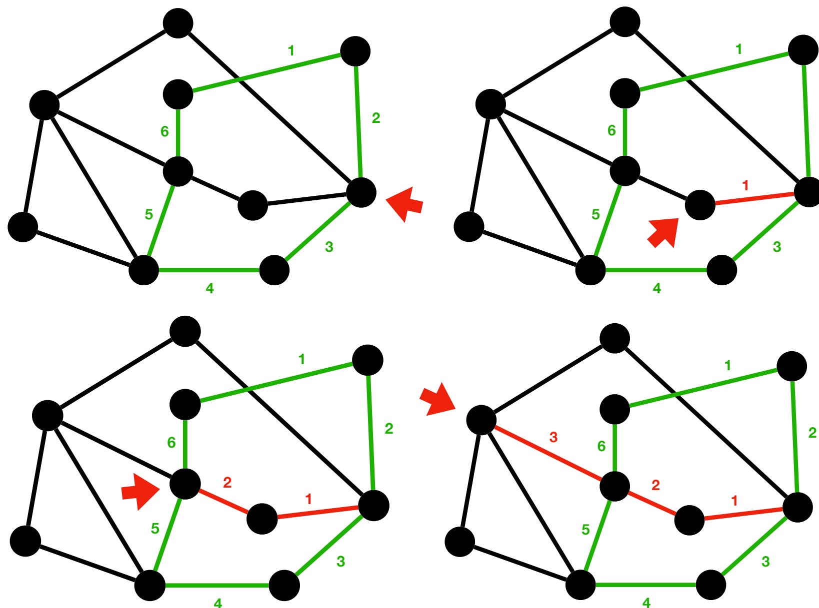


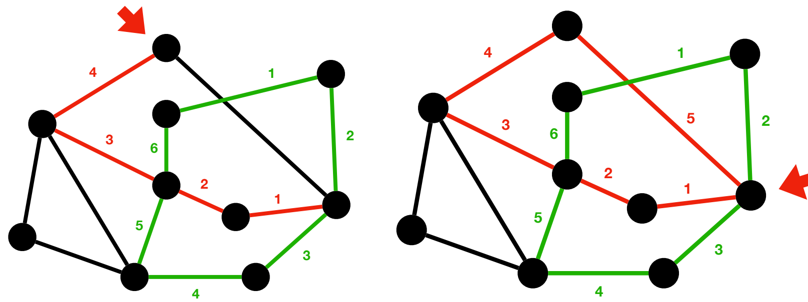


On a obtenu un cycle c, mais ce n'est pas encore un cycle eulérien puisque certaines arêtes ne sont pas visitées. Dans ce cycle, trois sommets (marqués par les flèches bleues ci-dessous) ont des arêtes qu'on n'a pas encore choisies :

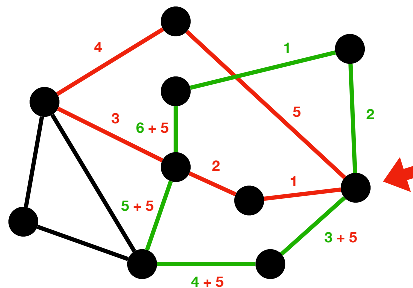


On choisit l'un de ces sommets, puis on recherche à nouveau un cycle c' n'empruntant aucune des arêtes préalablement choisies :

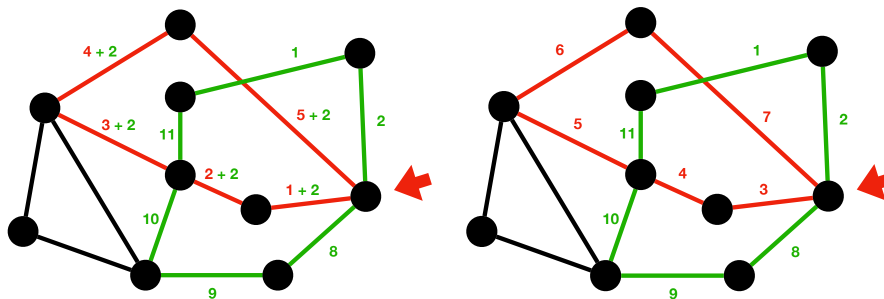




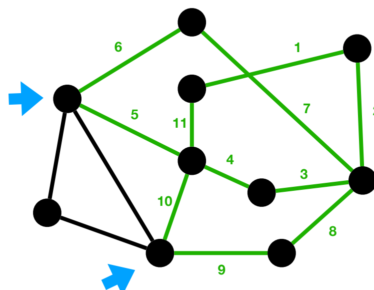
Il reste à insérer le cycle c' au sein du cycle c, c'est-à-dire qu'après avoir visité les deux premières arêtes de c, on se met en pause pour visiter le cycle c' entièrement, avant de terminer avec les quatre dernières arêtes de c. Pour faire cela, il suffit d'ajouter au numéro des quatre dernières arêtes de c la longueur de c' :



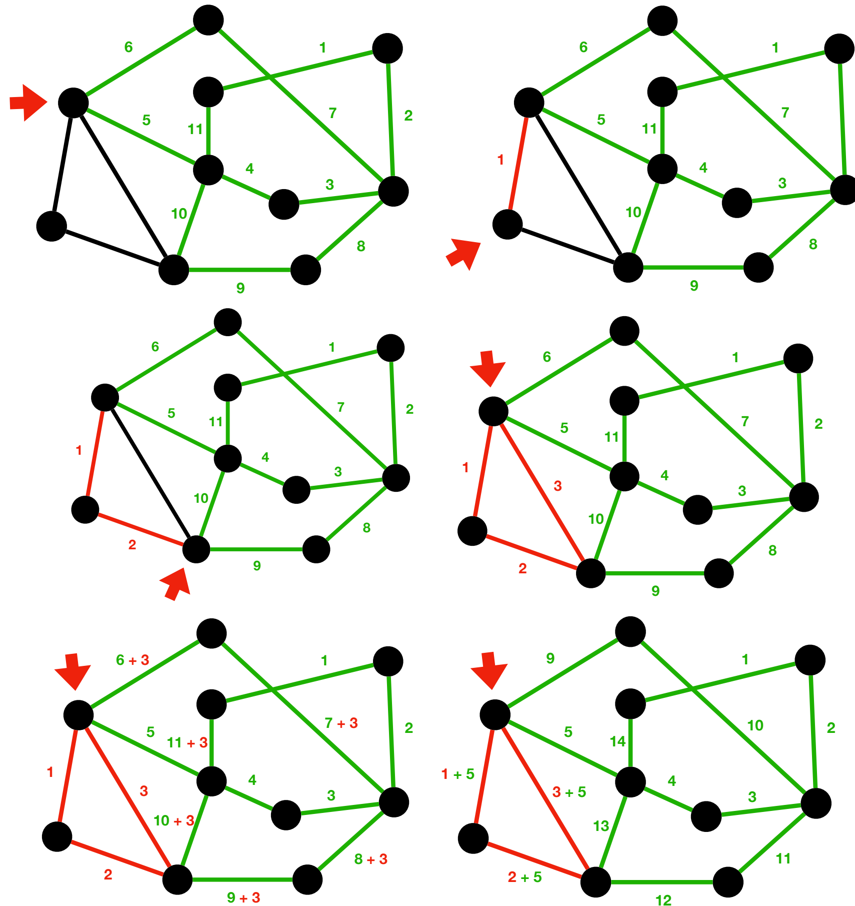
puis de décaler les numéros des arêtes du cycle c de deux pour l'insérer au bon endroit :



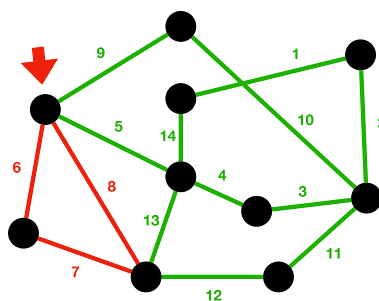
On obtient donc un cycle un peu plus long qui ne visite pas deux fois la même arête. Ce n'est toujours pas un cycle eulérien : deux sommets dans le cycle ont encore des arêtes non visitées



On en choisit un puis on recommence la même recherche de cycle, puis l'insertion du petit cycle dans le grand :



On obtient finalement le cycle eulérien suivant :

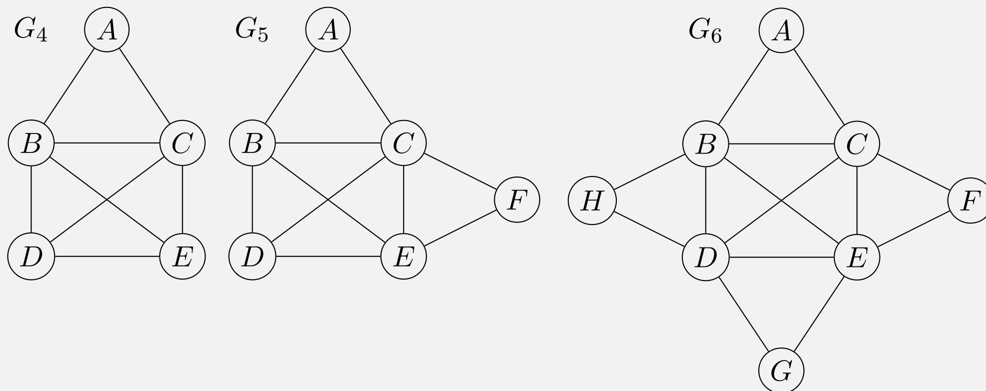


Pourquoi cet algorithme termine-t-il et est-il correct si le graphe vérifie la condition de connexité et sur les degrés de ses sommets ? Tout d'abord, remarquons qu'on ne peut pas rester bloqué à un moment dans la recherche d'un nouveau cycle : en effet, du fait de la parité du degré des sommets, si on entre dans un sommet, on peut toujours en sortir. La recherche du premier cycle finit tôt ou tard par terminer puisqu'on finit nécessairement par retourner dans le sommet initial v : l'ensemble des arêtes est fini et le sommet v a un nombre impair d'arêtes inexplorées une fois qu'on en est parti (donc il reste au moins une arête pour

y entrer). Ces arguments sont encore vrai pour la recherche du cycle autour de u, ce qui fait que l'algorithme de Hierholzer termine. Il produit bien un cycle qui visite toutes les arêtes du graphe, sinon, par connexité, au moins un des sommets devraient avoir encore une arête non visitée. Finalement, il ne peut pas visiter deux fois la même arête, par construction : il produit donc bien un cycle eulérien. Cela prouve donc le théorème d'Euler.

Exercice 42

1. Pour les graphes de l'exercice 41 qui admettent un cycle eulérien, trouver un tel cycle en appliquant l'algorithme de Hierholzer.
2. La notion de cycle eulérien peut être étendue : un *chemin eulérien* est un chemin (pas nécessairement un cycle) qui traverse chaque arête du graphe une et une seule fois. Parmi les graphes suivants, quels sont ceux pour lesquels vous pouvez trouver un chemin eulérien ?



3. À partir de vos observations, conjecturer une caractérisation des graphes possédant des chemins eulériens en termes de degrés des sommets du graphe (ressemblant à la caractérisation des graphes possédant des cycles eulériens).

6.2 Coloration de graphes et des cartes

Considérons un autre problème lié aux graphes. Imaginons la tâche d'un géographe qui souhaite colorier les pays d'une carte du monde avec le moins de couleurs possibles, tout en garantissant toujours que deux pays partageant une frontière terrestre n'ont pas la même couleur. On peut par exemple produire la carte de la Figure 6.3, utilisant quatre couleurs. Est-ce un hasard ? En fait, non, c'est toujours possible comme le dit le théorème des quatre couleurs :

Théorème 3. *On peut colorer n'importe quelle carte avec un maximum de quatre couleurs de sorte que les zones adjacentes reçoivent toujours deux couleurs distinctes.*

Ce théorème très simple a posé de fortes résistances à la communauté scientifique. Conjecturé dès le milieu du XIXème siècle, plusieurs preuves erronées ont été proposées, jusqu'à ce que, dans les années 1960 et 1970, des chercheurs utilisent des ordinateurs pour les aider à produire une preuve. En effet, on peut prouver le théorème avec des arguments relativement simples dès lors que la carte à colorier contient suffisamment de pays : mais cela laisse de nombreux « petits » graphes à étudier qu'il est très pénible, sinon impossible à tous faire à

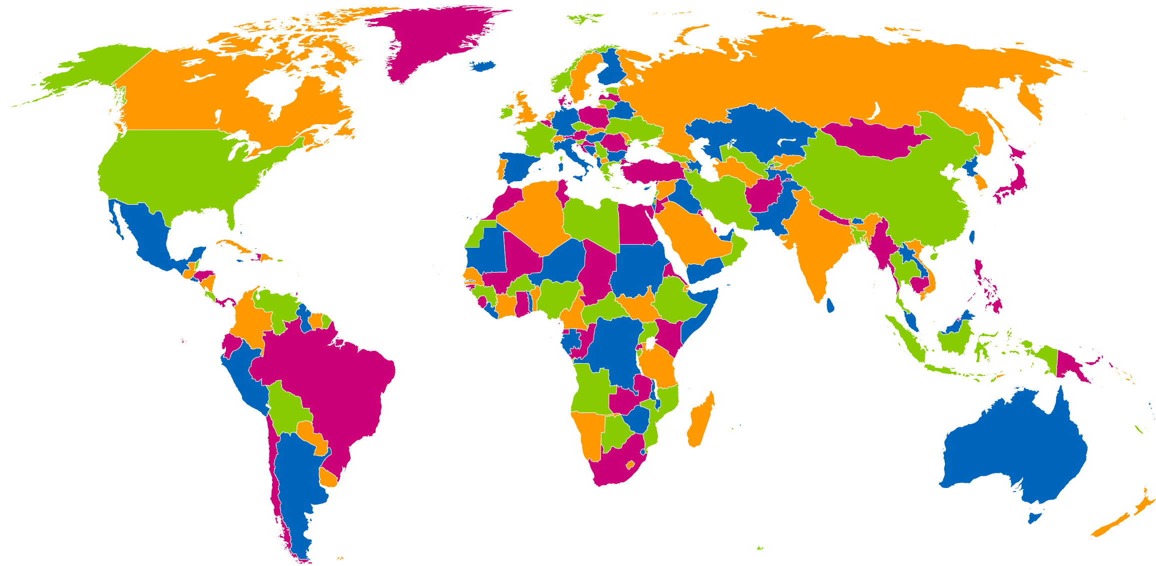
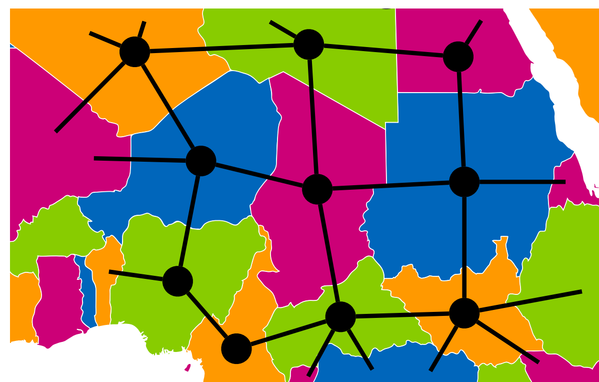


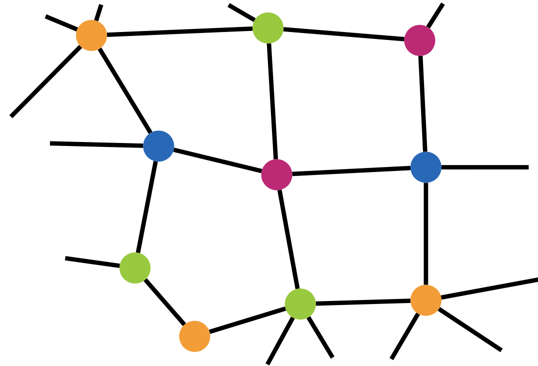
FIGURE 6.3 – Mappemonde colorée avec quatre couleurs

la main. L'utilisation de l'ordinateur s'avère donc indispensable mais a alors partagé la communauté scientifique pour savoir si cela pouvait effectivement encore s'appeler une preuve. Ce qui est sûr, c'est que le problème de prouver le théorème se déplace (comme dans le cas de l'algorithme de Hierholzer pour le théorème d'Euler) sur la preuve de correction de l'algorithme qui sert à vérifier les « petits » graphes. À l'heure actuelle, on ne connait toujours pas de preuve entièrement à la main de ce théorème pourtant simple à énoncer.

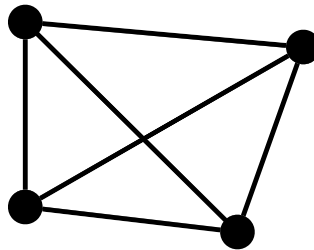
Mais quel rapport avec les graphes ? Si on zoome un peu sur la carte, on peut modéliser le problème avec un graphe non orienté dont les sommets seront les pays à colorier et les arêtes sont les frontières reliant ces pays :



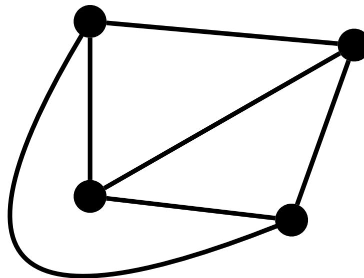
Colorier la carte revient donc à colorier les sommets du graphe, de sorte que deux sommets reliés par une arête ne sont pas coloriés avec la même couleur :



On appelle graphe *planaire* tout graphe qu'on peut obtenir par cette méthode à partir d'une carte quelconque : c'est donc un graphe qu'on peut dessiner sur un plan (une feuille de papier) sans qu'aucune arête n'en croise une autre. Attention, certains graphes ne semblent pas planaires, comme celui-ci...



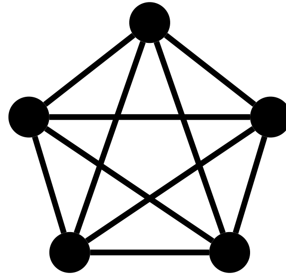
mais le sont en fait dès lors qu'on tord un peu certaines de leurs arêtes :



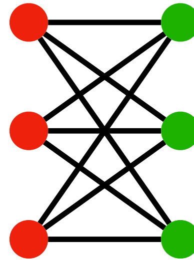
Le théorème des quatre couleurs peut donc s'énoncer de la façon suivante :

Théorème 4. *On peut colorer les sommets de tout graphe non orienté planaire avec un maximum de quatre couleurs de sorte que les sommets adjacents (c'est-à-dire reliés par une arête) reçoivent toujours deux couleurs distinctes.*

Ce théorème permet de se convaincre que certains graphes ne sont pas planaires. Il suffit pour cela de montrer qu'ils ne sont pas coloriables avec quatre couleurs. C'est par exemple le cas du graphe complet (c'est-à-dire tel que toute paire de sommets est une arête) à cinq sommets (ou plus) :



On ne peut pas le colorier avec quatre couleurs puisque chaque sommet a quatre sommets adjacents qui sont eux-même reliés entre eux. Par contre, noter que le théorème des quatre couleurs ne donne pas une condition nécessaire et suffisante pour être un graphe planaire. Le graphe suivant n'est pas planaire, mais il peut être colorié avec deux couleurs :



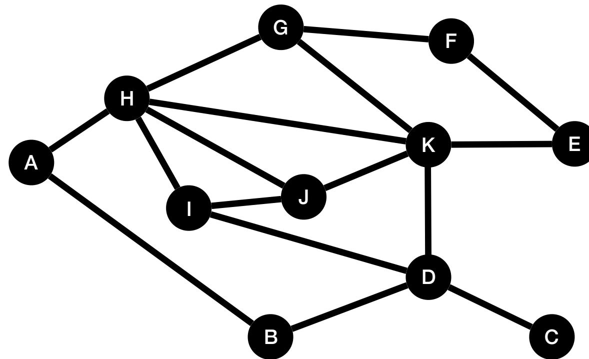
Le théorème des quatre couleurs a une grosse lacune : il dit qu'il est possible de colorer un graphe planaire avec quatre couleurs, mais ne donne pas de méthode pour faire cela. Comment donc colorer effectivement un graphe planaire, ou une carte de géographie, à partir de là ?

Une première méthode naïve pourrait consister à essayer toutes les possibilités de coloriage. Dans le pire des cas, cela demande à essayer toutes les possibilités de coloriage des sommets du graphe avec quatre couleurs. Un coloriage est une fonction de l'ensemble S des sommets dans un ensemble de couleurs $\{0,1,2,3\}$ à quatre éléments : ces fonctions sont au nombre de $4^{|S|}$, qui est exponentiel en fonction du nombre de sommets. Il n'est donc clairement pas envisageable d'essayer toutes les possibilités, dès lors que le graphe est un peu gros.

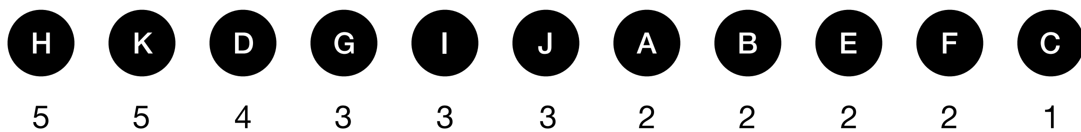
À la place, on étudie l'algorithme de Welsh-Powell, qu'on peut écrire de la manière suivante :

- Trier les sommets du graphe par ordre de degré décroissant
- couleur $\leftarrow$ 0 (*couleur initiale*)
- **Tant qu**'il y a encore des sommets non colorés
  - Parcourir la liste triée des sommets et colorer en *couleur* les sommets non colorés qui ne sont pas connectés à d'autres sommets de la même couleur
  - couleur $\leftarrow$ couleur + 1 (*choisir une nouvelle couleur*)

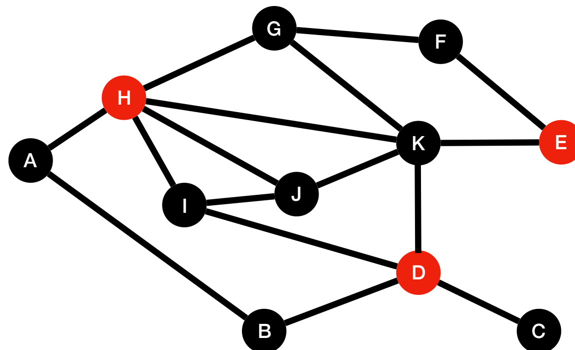
L'idée est donc très simple : on colorie les uns après les autres les sommets du graphe, en le traitant dans l'ordre décroissant de leur degré, et on essaie d'attribuer à un maximum de sommets la couleur 0, puis la couleur 1, puis la couleur 2, etc. Considérons par exemple le graphe suivant dont les sommets sont les lettres de A à K :



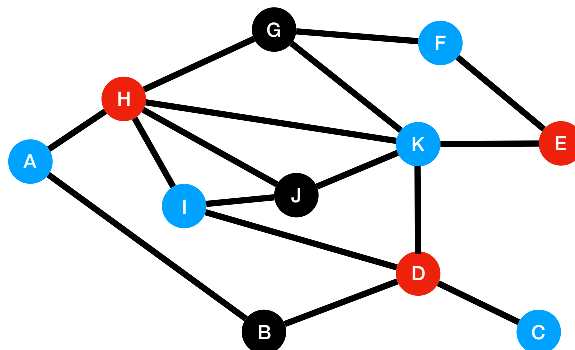
On commence par trier les sommets par ordre décroissant de degrés :



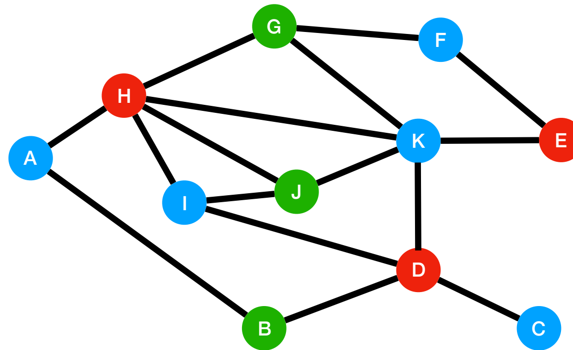
On commence par vouloir attribuer la couleur 0, disons rouge. On parcourt ainsi la liste précédente des sommets et on colorie le sommet en rouge, dès lors qu'aucun de ses voisins n'a déjà cette couleur. On peut donc colorer les sommets H, D et E :



On passe alors à la couleur suivante, disons bleu. On parcourt la liste des sommets non encore colorés, et on colorie le sommet en bleu dès lors qu'aucun de ses voisins n'a déjà cette couleur. On peut alors colorer les sommets K, I, A, F et C :

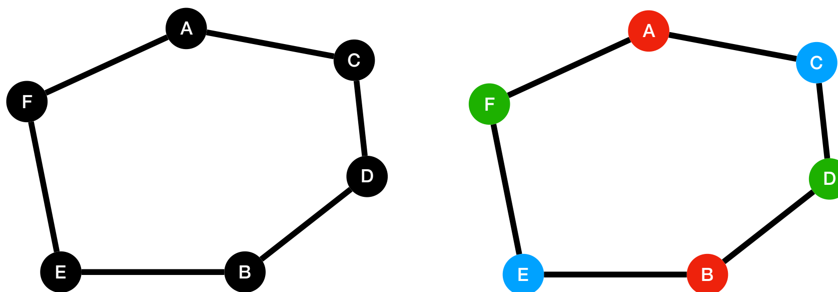


On fait de même avec la couleur suivante, le vert par exemple, pour colorer les sommets restants G, J et B :



On a donc réussi à colorer ce graphe planaire avec trois couleurs seulement. Notons qu'il n'est pas possible de faire mieux, c'est-à-dire de n'utiliser que deux couleurs seulement. En effet, ce graphe possède un *triangle*, c'est-à-dire trois sommets reliés les uns aux autres, par exemple les sommets H, I et J : un tel triangle nécessite trois couleurs puisque chaque sommet est relié à deux autres sommets reliés entre eux. Sur ce graphe, l'algorithme de Welsh-Powell renvoie donc une coloration *optimale*, c'est-à-dire avec le nombre minimal de couleurs possible.

Ce n'est malheureusement pas toujours le cas. Par exemple, sur le graphe suivant, où tous les sommets ont degré 2, si on suit l'ordre alphabétique de traitement des sommets, on arrive à la coloration à droite avec trois couleurs :

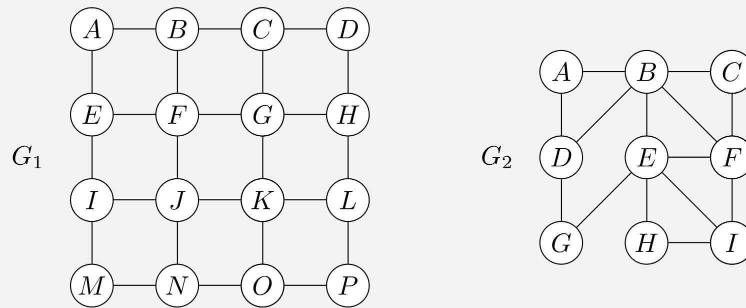


En fait, ce graphe ne nécessite que deux couleurs, puisqu'on peut colorer les sommets A, D et E d'une même couleur, puis B, C et F d'une autre même couleur.

Exercice 43

À chaque fois que deux sommets ont le même degré, *on les suppose triés par ordre alphabétique*. Par exemple, si les sommets A, D, B sont tous de degré 3, on suppose qu'ils seront triés dans l'ordre A, B, D.

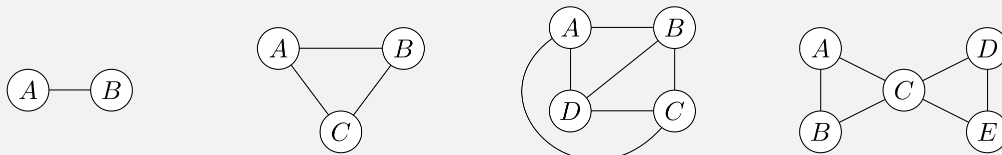
1. Colorier les graphes suivants à l'aide de l'algorithme de Welsh-Powell.



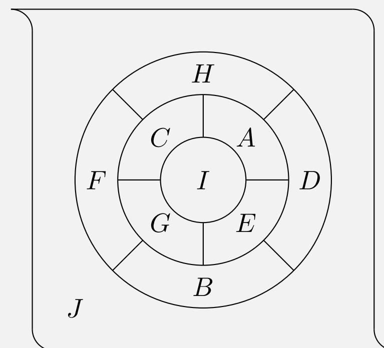
2. Les colorations obtenues sont-elle optimales en ce qui concerne le nombre de couleurs utilisées ? Si oui, pourquoi ? Si non, trouver une meilleure coloration.

Exercice 44

1. Dessiner une carte pour chacun des graphes suivants.



2. À l'aide de l'algorithme de Welsh-Powell, colorier la carte ci-dessous. *Noter que la région externe J correspond, elle aussi, à l'un des sommets du graphe associé. (Comme dans l'exercice précédent, si deux sommets ont le même degré, choisir d'abord le plus petit selon l'ordre alphabétique.)*



3. La coloration obtenue est-elle optimale en ce qui concerne le nombre de couleurs ? Si oui, pourquoi ? Si non, trouver une meilleure coloration.

Chapitre 7

Arbres

Passons à l'étude de graphes très particuliers, les arbres. Eux aussi, nous les retrouvons un peu partout dans notre vie quotidienne, et nous allons voir que leur étude permet de résoudre des problèmes informatiques intéressants.

7.1 Exemples d'arbres déjà rencontrés

Un arbre généalogique (cf Figure 7.1) est, par exemple, un graphe particulier où les sommets sont les personnes et les arêtes sont les liens de parenté directe entre ces personnes.

Un autre exemple, plus proche de l'informatique, concerne l'arborescence de fichiers stockés sur un ordinateur. Par exemple, en Figure 7.2, le répertoire principal de la machine contient une image `souris.jpg` ainsi que trois répertoires. Chacun de ses répertoires contient à son tour un certain nombre de fichiers et/ou d'autres répertoires. Un répertoire peut aussi être vide, comme le répertoire `System`. Notez que la représentation de l'arbre est inversé, le tronc tout en haut et les branches partant vers le bas... C'est souvent la tradition en informatique.

Mais vous avez aussi vu des arbres dans votre cursus scolaire. Par exemple, si on vous décrit une situation où une urne contient deux boules rouges et trois boules bleues, et qu'on vous demande de calculer la probabilité de tirer deux boules de couleurs différentes, si on ne fait pas de remise, alors vous savez qu'une méthode de résolution consiste à utiliser un arbre de probabilités, comme en Figure 7.3. La probabilité d'une branche est alors le produit des probabilités le long de la branche. La probabilité d'un évènement, c'est-à-dire un ensemble de branches, est la somme des probabilités des branches composant l'évènement. Ainsi, la probabilité recherchée est

$$\frac{2}{5} \times \frac{3}{4} + \frac{3}{5} \times \frac{1}{2} = \frac{3}{5}$$

Finalement, même dans ce cours, nous avons déjà utilisé des arbres, sans vraiment le savoir. Il s'agissait de l'exercice 8 où nous avions considéré deux codages (cf Figure 7.4) pour les 27 symboles d'un texte (les 26 lettres de l'alphabet plus l'espace) en associant à chaque symbole un mot binaire. Par exemple, le mot « encore » se code en codage fixe à l'aide de

001000110100010011101000100100

soit 30 bits, et en codage variable à l'aide de

11010010100100000101110

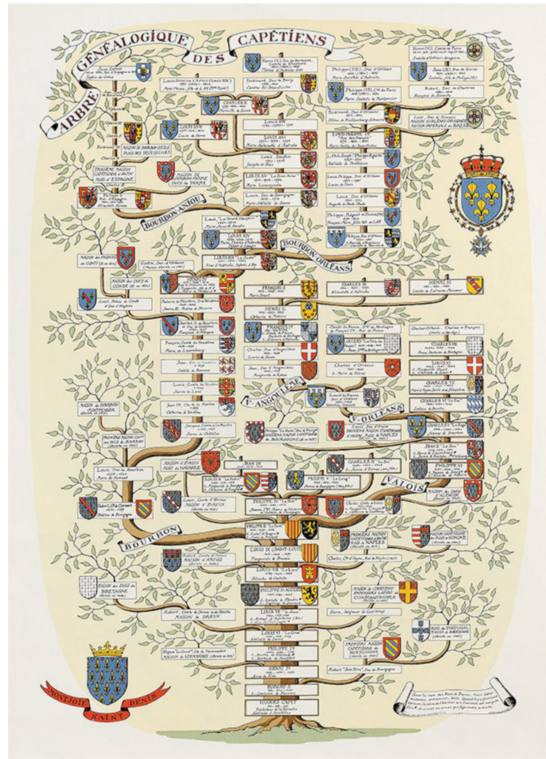


FIGURE 7.1 – Un arbre généalogique

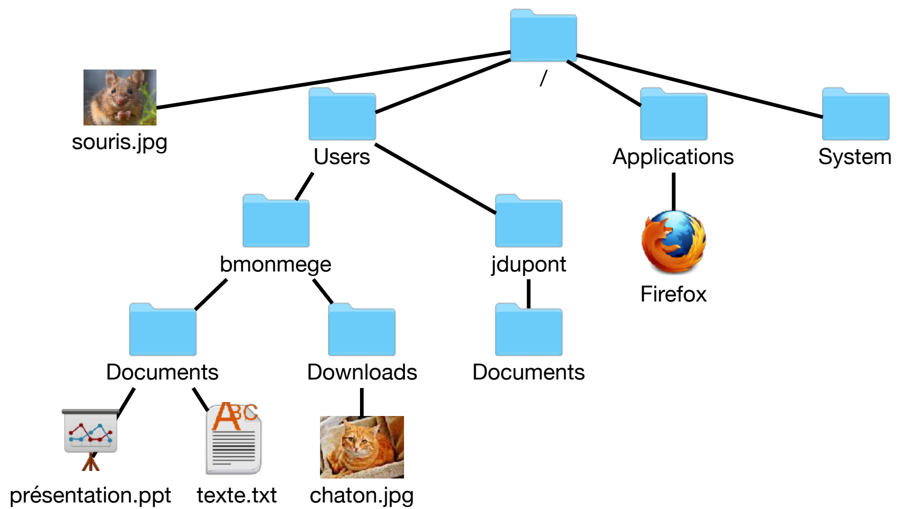


FIGURE 7.2 – Arborescence de fichiers

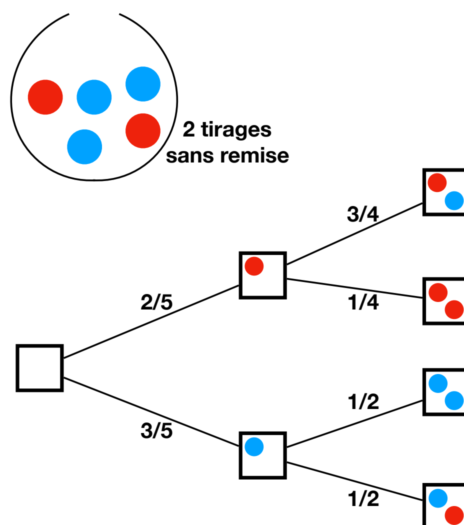


FIGURE 7.3 – Un arbre de probabilités

soit 23 bits. Nous avions vu alors que souvent le codage variable permet d'obtenir un code plus court pour les mots de la langue française. En revanche, un des points négatifs était que le décodage semble plus difficile. Si on donne simplement la suite de bits, par exemple,

1000100101111000000010100101101001101000000001001

il faut un peu de courage pour réussir à décoder le mot ≪ information ≫. Cependant, en pratique, nous avions vu que nous n'avons jamais le choix du décodage. La raison est qu'aucun code de symbole n'est le *préfixe* d'un autre code, c'est-à-dire ne commence par le code d'un autre symbole. Ceci permet de placer les symboles codés dans un arbre, comme en Figure 7.5. Depuis le sommet tout en haut, on va à gauche si le code commence par un 0 et à droite s'il commence par un 1. On continue ainsi à chaque sommet de l'arbre jusqu'à ce que, en concaténant les bits déjà vus, on obtienne le code d'un symbole, auquel cas on place le symbole à cet endroit de l'arbre. Le symbole ≪ t ≫ est ainsi placé au bout du chemin qui va à gauche, puis deux fois à droite, puis une fois à gauche, puisque son code est 0110. Dans ces arbres, les sommets représentent donc des séquences de bits et les arêtes représentent l'ajout d'un bit 0 ou 1 à la fin de la séquence. Une fois représenté l'arbre en entier, on observe aisément une propriété qui fait la force de cet arbre : les lettres les plus utilisées dans la langue française se retrouvent haut dans l'arbre, alors que les lettres les moins utilisées sont placées plus profondément dans l'arbre. Il s'agit du codage de Huffman qui, tout en conservant la propriété qu'aucun code n'est préfixe d'un autre pour permettre le décodage, attribue les codes les plus courts aux symboles très usités pour diminuer la taille du codage de mots de la langue française. L'arbre de Huffman est donc fortement dépendant de la langue du texte à coder.

7.2 Définitions

Qu'ont en commun les arbres qu'on a vus jusque-là ? Ce sont des graphes non orientés particuliers. De plus, ils sont tous d'un seul tenant : nous avons vu dans le chapitre précédent le

lettre	codage variable
a	1010
b	0010011
c	01001
d	01110
e	110
f	0111100
g	0111110
h	0010010
i	1000
j	011111110
k	011111111001
l	0001
m	00101
n	1001
o	0000
p	01000
q	0111101
r	0101
s	1011
t	0110
u	0011
v	001000
w	011111111000
x	01111110
y	0111111111
z	01111111101
espace	111

lettre	codage fixe
a	00000
b	00001
c	00010
d	00011
e	00100
f	00101
g	00110
h	00111
i	01000
j	01001
k	01010
l	01011
m	01100
n	01101
o	01110
p	01111
q	10000
r	10001
s	10010
t	10011
u	10100
v	10101
w	10110
x	10111
y	11000
z	11001
espace	11010

FIGURE 7.4 – Deux codages possibles des lettres de l'alphabet

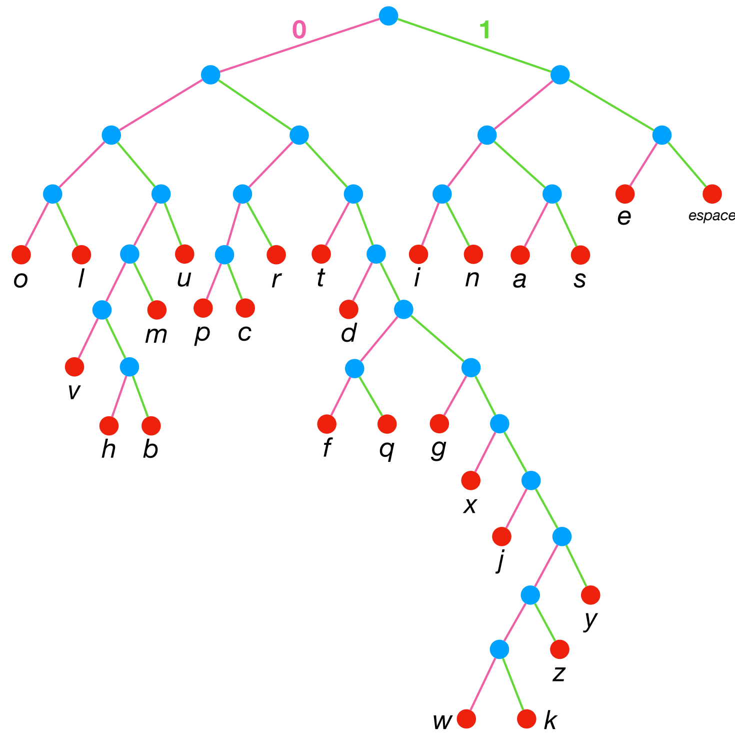
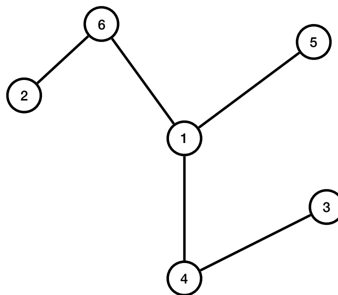


FIGURE 7.5 – Arbre de Huffman

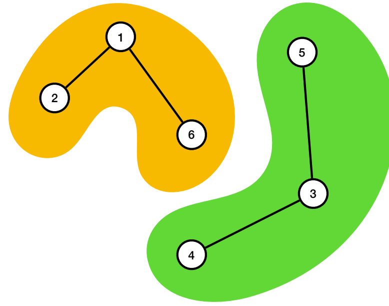
terme pour qualifier cette propriété, il s'agit de *graphes connexes*. Ils ont aussi la particularité de ne contenir aucun chemin cyclique qui n'ait besoin de « rebrousser chemin » : on dit effectivement que le graphe est *sans cycle* ou *acyclique*. Il se trouve que ce sont les seules propriétés.

Définition 8. Un arbre est un graphe non orienté connexe et sans cycle.

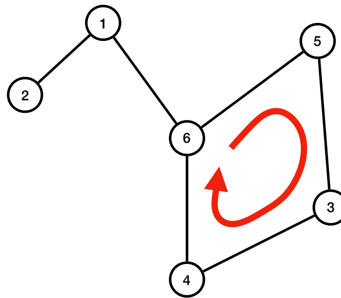
Voilà donc un exemple d'arbre :



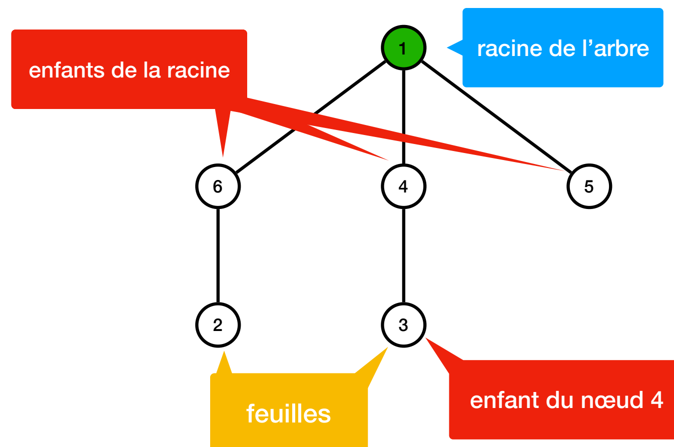
En revanche, l'exemple ci-dessous n'est pas un arbre puisque ce graphe n'est pas connexe : il est en deux morceaux détachés



Finalement, ce dernier exemple ci-dessous est bien connexe, mais possède un cycle : ce n'est donc pas non plus un arbre



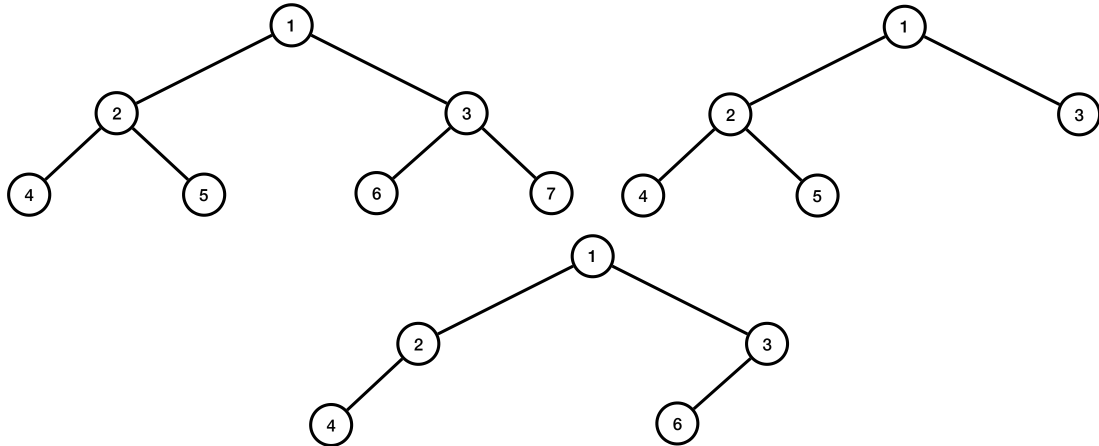
Une autre particularité des graphes vus précédemment est qu'ils ont un sommet ayant un statut particulier : c'est la *racine* de l'arbre. Il s'agit simplement d'un sommet désigné que l'on dessine généralement tout en haut de l'arbre (tout en bas dans le cas de l'arbre généalogique, tout à gauche dans le cas de l'arbre de probabilités). Un arbre possédant une racine s'appelle souvent un *arbre enraciné*. On peut donc reprendre l'arbre précédent, choisir 1 comme sommet racine, puis comme si on remontait le sommet 1 tout en haut en le tenant entre nos doigts, on obtient la représentation du même arbre :



Dans ce cas, les sommets 4, 5 et 6 sont les *enfants* du sommet 1. De même, le sommet 3 est l'unique enfant du sommet 4. Les sommets 2, 3 et 5 n'ont pas d'enfants : on dit qu'ils sont des *feuilles*, pour conserver l'analogie botanique. On dit d'ailleurs qu'un chemin allant de la racine à une feuille est une *branche* de l'arbre.

7.3 Arbres binaires

Parmi tous les arbres possibles, on distingue une classe importante en pratique : les *arbres binaires*. Ce sont des arbres enracinés dont chaque sommet a au plus 2 enfants : on distingue alors d'ailleurs son *enfant gauche* de son *enfant droit*. Les arbres ci-dessous sont donc des arbres binaires :

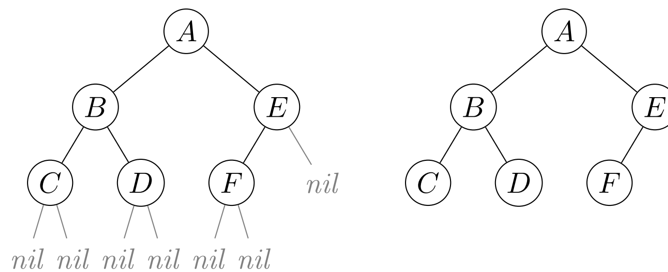


On peut alors donner une définition différente de ces arbres, plus proche de leur représentation dans un ordinateur. Il s'agit d'une définition *récursive* : cela veut dire qu'on s'autorise à utiliser la notion d'arbre binaire pour définir ce qu'est un arbre binaire. On fait de même lorsqu'on décide qu'un entier (naturel) est soit 0 soit le successeur $n+1$ d'un entier naturel n. Pour les arbres, on fait ainsi :

Définition 9. Un arbre binaire est :
— soit l'arbre vide, qu'on note souvent *nil* (et qu'on ne représente généralement pas dans les dessins) ;
— soit une racine ayant un enfant gauche et un enfant droit, qui sont tous deux des arbres binaires.

Dans le cas d'arbres enracinés, on appelle parfois *nœuds* les sommets de l'arbres.

À gauche ci-dessous le graphe est un arbre binaire, qu'on représente dans la suite comme dessiné à droite, sans les arbres *nil* :



La racine de cet arbre est le nœud A qui a deux enfants : l'enfant gauche a B pour racine, et l'enfant droit a E pour racine. Les enfants du nœud C sont deux arbres vides *nil*. Une feuille est un nœud qui possède l'arbre vide comme enfants gauche et droit : les feuilles de l'arbre du dessus sont C, D et F.

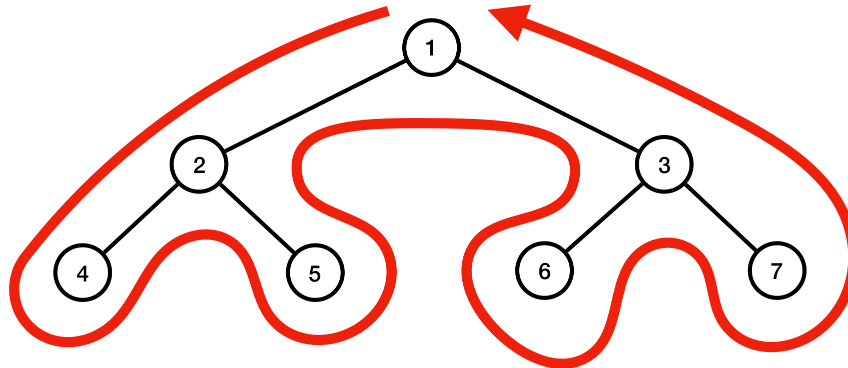
7.4 Affichage d'une arborescence de fichiers : parcours préfixe

Considérons désormais des problèmes de la vie réelle que l'on va pouvoir traiter à l'aide d'arbres. Reprenons l'exemple de l'arborescence de fichiers de la Figure 7.2. Parfois, on a besoin d'afficher cette arborescence sous forme d'une liste de fichiers. Un outil permet de faire cela sous un système d'opérations Unix, tel que Linux ou Mac : l'utilitaire `ls` qui lorsqu'on l'exécute dans un terminal avec l'option `-R` pour signifier qu'on souhaite rentrer récursivement dans les sous-répertoires, on obtient le résultat suivant

```
$ ls -R
souris.jpg
Users
Applications
System

./Users:
bmonmege
jdupont

./Users/bmonmege:
Documents
Downloads

./Users/bmonmege/Documents:
présentation.ppt
texte.txt

./Users/bmonmege/Downloads:
chaton.jpg

./Users/jdupont:
Documents

./Users/jdupont/Documents:

./Applications:
Firefox

./System:
```

Le programme `ls` commence par imprimer les noms des fichiers et répertoires contenus directement dans le répertoire racine. Ensuite, il imprime le contenu du premier répertoire entièrement, avant de passer au second répertoire, puis le troisième répertoire. L'impression du premier répertoire `/Users` est longue puisque, de même, il faut commencer par imprimer le nom des deux répertoires, puis le contenu de chacun, et ainsi de suite.

Remis dans le contexte d'un arbre binaire (plutôt qu'un arbre quelconque comme pour l'arborescence de fichiers), cela revient à considérer le parcours suivant

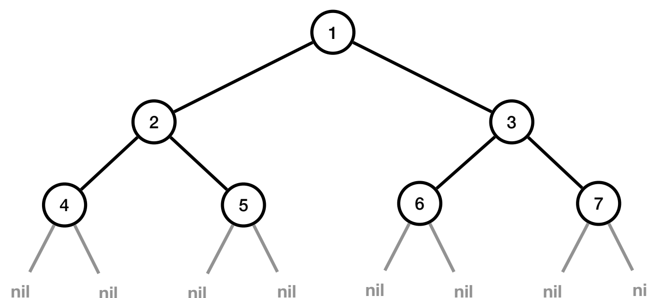


De plus, on imprime le contenu du sommet la première fois qu'on le visite, ce qui fait que, dans ce cas, on imprimerait donc les sommets dans l'ordre

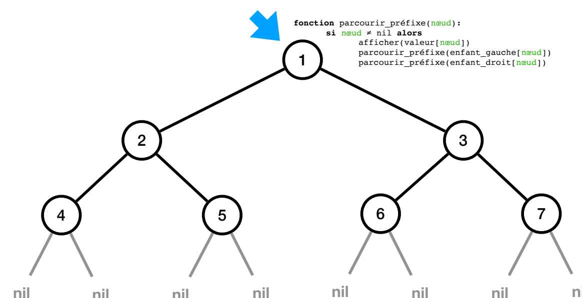
Un tel parcours, qu'on appelle *parcours préfixe* puisqu'il traite chaque nœud *avant* de traiter ses enfants, peut être implémenté par un algorithme récursif (nous avons déjà vu un algorithme récursif lorsque nous avons étudié le tri par fusion) :

```
def parcours_préfixe(nœud):
    if nœud != nil:
        afficher(valeur[nœud])
        parcours_préfixe(enfant_gauche[nœud])
        parcours_préfixe(enfant_droit[nœud])
```

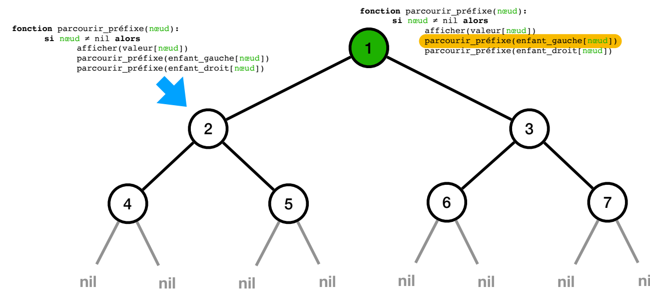
Testons-le sur l'arbre précédent pour mieux comprendre comment s'exécute un algorithme récursif. On commence par faire réapparaître les arbres vides :



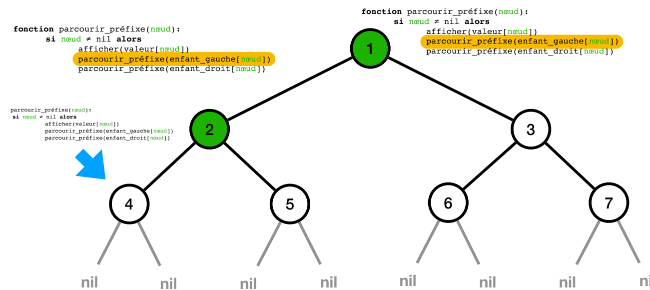
et on lance l'algorithme sur le nœud racine :



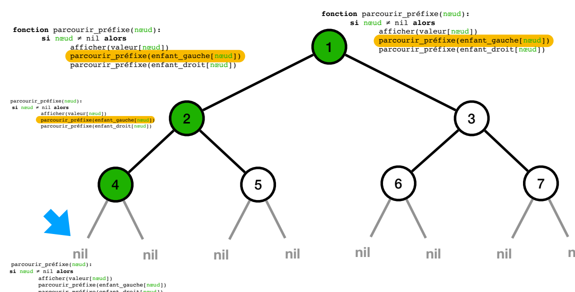
On commence par le test **nœud != nil** qui est vrai puisque le nœud 1 n'est pas l'arbre vide. On exécute donc les trois lignes internes, en commençant par afficher la valeur du nœud. On appelle ensuite récursivement le même algorithme sur l'enfant gauche. Cela veut dire qu'on met en pause l'exécution courante et qu'on démarre une exécution indépendante depuis le début sur le nœud 2 :



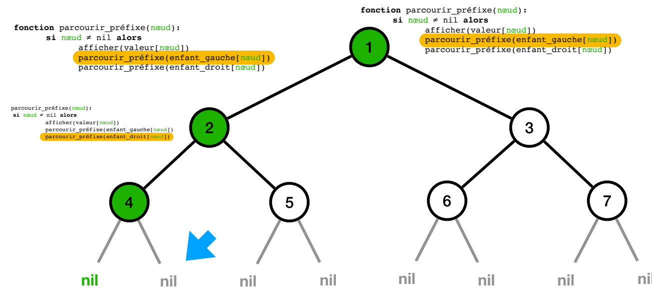
De même, on commence par imprimer le contenu du nœud 2, puis on fait l'appel récursif dans l'enfant gauche :



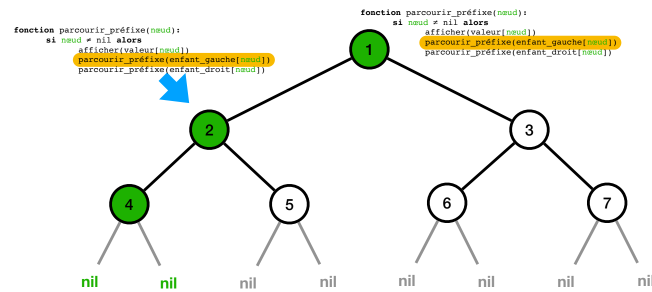
De même on affiche le contenu du nœud 4 et on s'appelle récursivement sur l'enfant gauche :



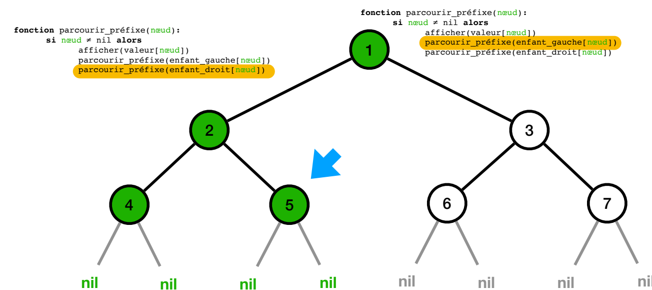
Cette fois, ce nœud est l'arbre vide. Le test **nœud** $\neq$ **nil** est donc faux ce qui implique que la fonction s'arrête tout de suite, sans rien faire. On revient donc dans l'algorithme mis en pause sur le nœud 4 : c'est comme si vous aviez le contrôle du direct sur plusieurs télévisions en parallèle, dès qu'une émission se termine sur l'une, le programme se remet en route sur une autre... La prochaine ligne de l'algorithme demande à exécuter l'algorithme sur le sous-arbre droit, vide également :



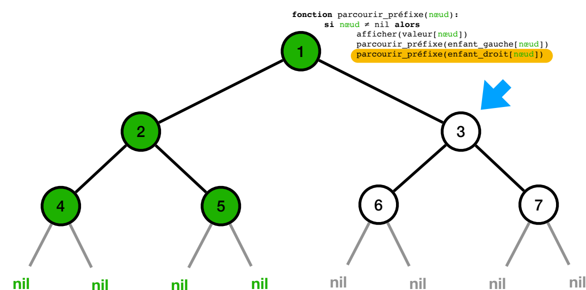
L'exécution s'arrête donc également immédiatement et on revient dans l'exécution sur le nœud 4 : mais celle-ci est arrivée à son terme et s'arrête donc naturellement. On retourne donc finalement dans l'exécution sur le nœud 2, là où on s'en était arrêté :



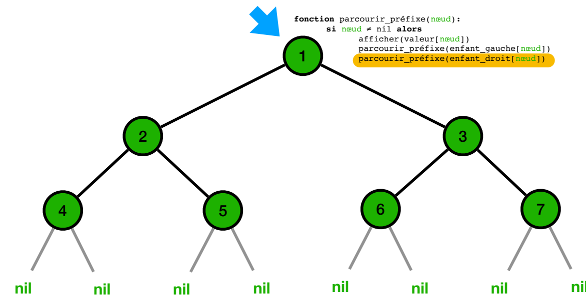
La prochaine ligne consiste à s'appeler récursivement sur l'enfant droit :



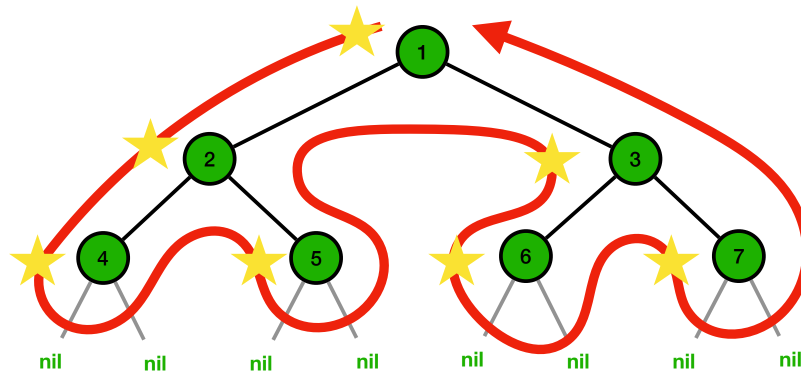
Comme pour le nœud 4, on va donc imprimer le contenu du nœud 5, puis s'appeler récursivement sur les deux arbres vides avant de terminer. L'exécution du nœud 2 termine ainsi, et on retourne donc dans l'exécution du nœud 1 :



Le parcours de son enfant droit permet d'imprimer les contenus des nœuds 3, puis 6, puis 7, avant de revenir à la fin de l'exécution du nœud 1 et de terminer :

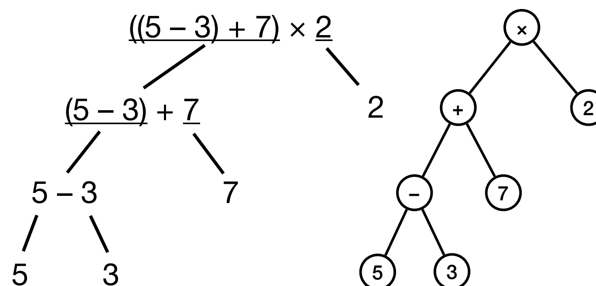


Au final, on a donc imprimé les contenus des nœuds dans l'ordre 1, 2, 4, 5, 3, 6, 7. Le long du parcours d'arbre, on traite donc les nœuds la première fois qu'on les a visités, c'est-à-dire sur les étoiles dans le diagramme suivant :



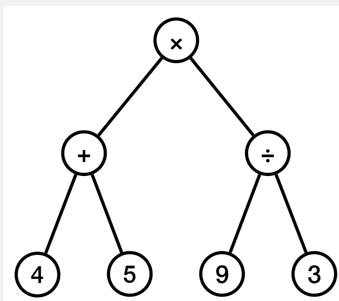
7.5 Expressions arithmétiques et parcours postfixe

Les arbres sont partout... même dans vos calculatrices. Lorsque vous tapez un calcul, par exemple $\big((5-3)+7\big)\times 2$, la calculatrice, elle, stocke un arbre binaire en mémoire. Pour cela, elle essaie de décomposer le calcul en deux tant que c'est possible. Au début, elle trouve donc l'opération la plus prioritaire, le produit, pour décomposer le calcul en le produit de $(5-3)+7$ et de 2. À nouveau, elle décompose le sous-calcul $(5-3)+7$ comme la somme de $5-3$ et de 7, et ainsi de suite jusqu'à ce que les sous-calculs soient tous des constantes. On obtient donc la représentation par l'arbre ci-dessous, qu'on résume à droite en conservant dans chaque nœud uniquement l'opération qui permet la décomposition :



Exercice 45

Quelle est l'expression arithmétique dont l'arbre est le suivant ?



Qu'est-ce qu'un calcul pour une calculatrice ? Considérons, pour simplifier la présentation, une calculatrice très simple avec uniquement les opérations de somme, de soustraction, de produit et de division. Ainsi, ce que reçoit la calculatrice de l'utilisateur est une expression arithmétique qui est :

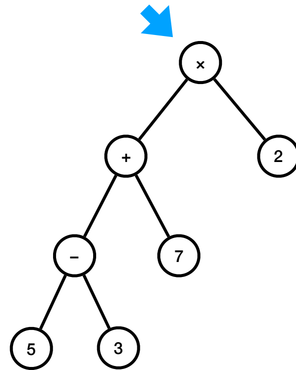
— soit un nombre n ;
— soit une somme $(e_1 + e_2)$;
— soit une soustraction $(e_1 - e_2)$;
— soit un produit $(e_1 \times e_2)$;
— soit une division $(e_1 \div e_2)$;

avec, à chaque fois, e_1 et e_2 deux expressions arithmétiques. On retrouve ici une définition par récurrence, où on utilise la notion d'expression arithmétique pour définir une expression arithmétique. C'est parce que cette définition est récursive qu'elle se prête bien à la représentation par un arbre :

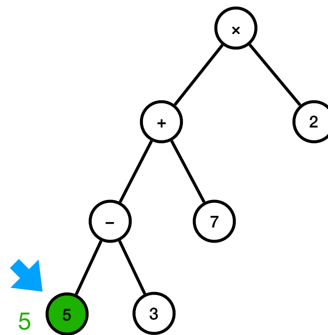
— un nombre n est représenté par une feuille contenant n ;
— une somme $(e_1 + e_2)$ est représentée par un nœud contenant $+$ avec l'arbre binaire de e_1 en enfant gauche et l'arbre binaire de e_2 en enfant droit ;
— et de même pour les trois autres opérations.

Si la calculatrice stocke l'arbre de l'expression dans sa mémoire, comment fait-elle pour l'évaluer, c'est-à-dire pour répondre à l'utilisateur que le résultat du calcul $\big((5-3)+7\big) \times 2$ est 18 ? Elle fait comme nous si nous voulions le faire de tête : elle commence par les calculs « les plus à l'intérieur ». En effet, dans le calcul précédent, la seule chose qu'on peut calculer, sans utiliser de propriétés sur les opérateurs arithmétiques, est le sous-calcul $5-3$, qui vaut 2. On peut donc remplacer dans le calcul originel et obtenir $(2+7) \times 2$. À nouveau, on exécute le sous-calcul le plus à l'intérieur, soit $2+7$, dont on sait que cela vaut 9. On obtient alors 9×2 qui s'évalue en 18.

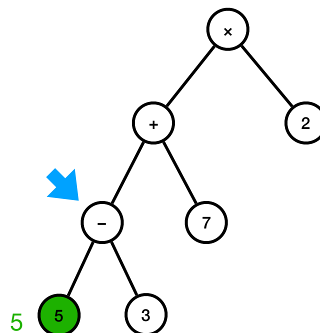
Il se trouve que ce calcul consistant à d'abord évaluer les sous-calculs les plus à l'intérieur est à nouveau un parcours de l'arbre. On commence donc à la racine de l'arbre.



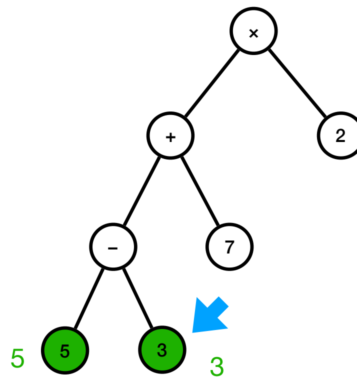
Mais contrairement au parcours préfixe, on affiche rien et on part directement dans l'enfant gauche. Petit à petit, appel récursif après appel récursif, on atterrit donc la feuille 5 qui s'évalue directement en 5 sans avoir besoin d'appels récursifs :



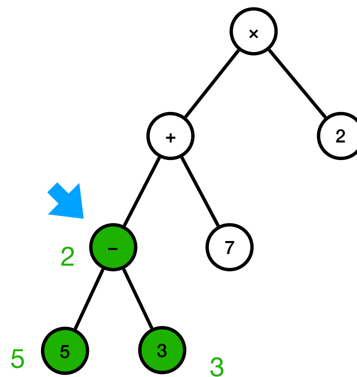
On peut donc remonter dans le nœud parent :



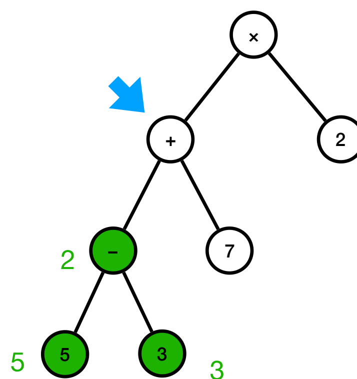
Comme pour le parcours préfixe, on passe ensuite à l'appel récursif sur l'enfant droit :



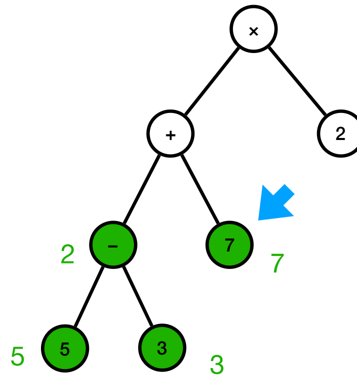
puis on remonte une dernière fois vers le nœud $-$ où on a désormais toutes les informations pour pouvoir évaluer l'opération $5 - 3$:



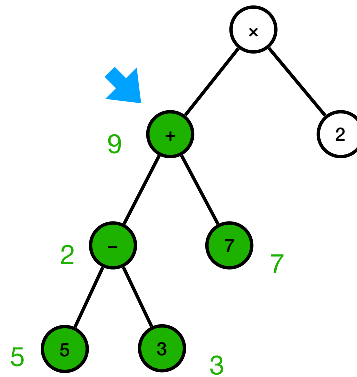
On continue donc à remonter dans l'arbre, sur le nœud $+$ où, à nouveau, on n'a pas encore tous les éléments pour effectuer le calcul :



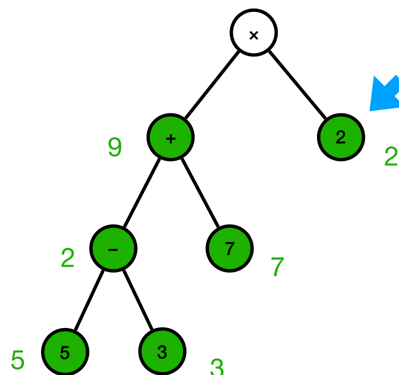
Il faut d'abord parcourir l'enfant droit :



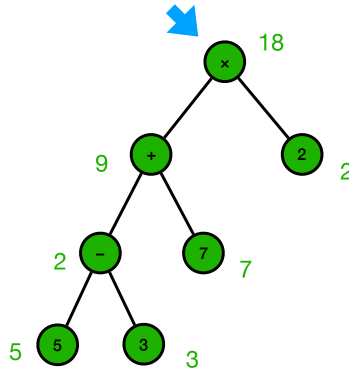
puis revenir pour évaluer le calcul $2 + 7$:



On repart à la racine de l'arbre, pour ensuite aller évaluer son enfant droit :



puis finalement évaluer le calcul 9×2 à la racine :

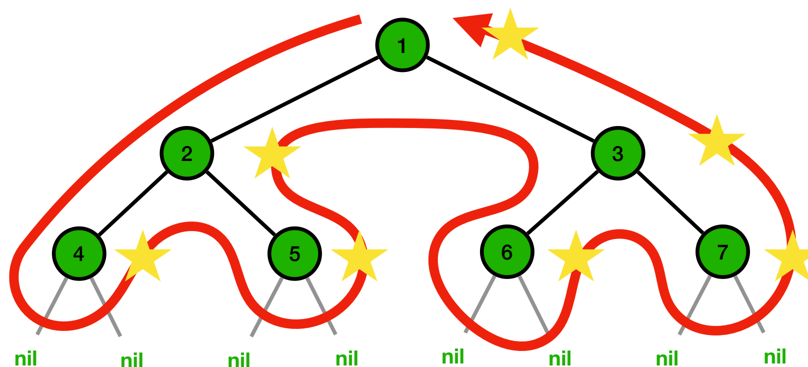


Voici l'opération qu'on a effectué, en admettant écrite une fonction testant si un nœud est une feuille (c'est-à-dire si son enfant gauche et son enfant droit sont égaux à *nil*) :

```
def évaluer_expression(nœud):
    if est_feuille(nœud):
        return valeur[nœud]
    else:
        m = évaluer_expression(enfant_gauche[nœud])
        n = évaluer_expression(enfant_droit[nœud])
        if valeur[nœud] == "+":
            return m + n
        elif valeur[nœud] == "-":
            return m - n
        elif valeur[nœud] == "*":
            return m * n
        else:    # dans ce cas valeur[nœud] = "/"
            return m / n
```

Notons que les appels récursifs sont désormais exécutés avant de traiter plus longuement le nœud (traitement qui consiste en l'occurrence à prendre le résultat de l'enfant gauche et de l'enfant droit et d'exécuter le calcul élémentaire demandé).

En matière d'affichage d'un arbre binaire, l'évaluation d'une expression arithmétique consiste en le parcours suivant :



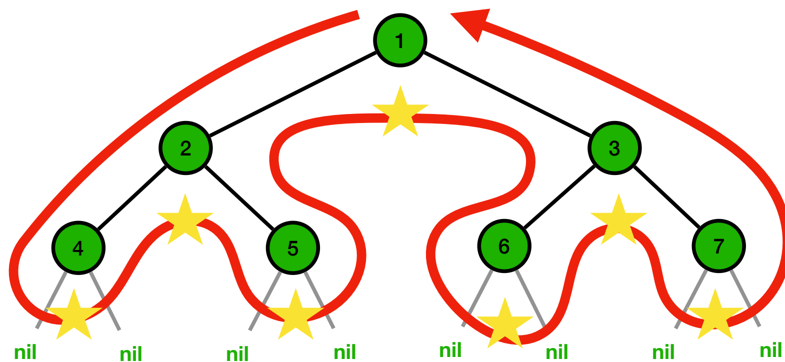
dans lequel on affiche chaque nœud lors du dernier passage par celui-ci, après le traitement de ses enfants gauche et droit. En opposition au parcours préfixe précédent, on appelle ce parcours le *parcours postfixe*. On peut l'écrire de la manière suivante à l'aide d'un algorithme récursif :

```
def parcours_postfixe(nœud):
    if nœud != nil:
        parcours_postfixe(enfant_gauche[nœud])
        parcours_postfixe(enfant_droit[nœud])
        afficher(valeur[nœud])
```

Pour l'arbre précédent, le parcours postfixe affiche donc les nœuds dans l'ordre 4, 5, 2, 6, 7, 3 et 1.

7.6 Parcours infixe : affichage d'une expression

Nous avons vu deux parcours d'arbres différents : le parcours préfixe dans lequel on traite d'abord le nœud avant de traiter son enfant gauche puis son enfant droit, et le parcours postfixe où on commence par traiter les deux enfants avant de traiter le nœud lui-même. On peut aisément imaginer un troisième parcours dans lequel on traite le nœud *entre* le traitement de l'enfant gauche et celui de l'enfant droit : on appelle cela le *parcours infixe*. Il peut se visualiser ainsi



et affiche donc les nœuds dans l'ordre 4, 2, 5, 1, 6, 3 et 7. C'est comme si on avait aplati l'arbre en le passant sous une enclume. L'algorithme récursif de ce parcours est naturellement le suivant :

```
def parcours_infixe(nœud):
    if nœud != nil:
        parcours_infixe(enfant_gauche[nœud])
        afficher(valeur[nœud])
        parcours_infixe(enfant_droit[nœud])
```

Les trois parcours ne diffèrent donc que par l'ordre dans lequel on visite la racine, l'enfant gauche et l'enfant droit.

Le parcours infixe est très naturel, en particulier à nouveau dans les calculatrices. Il permet d'afficher à l'écran le calcul (stocké sous forme d'arbre par la calculatrice) sous un format compréhensible par nous. En fait, afficher une expression,

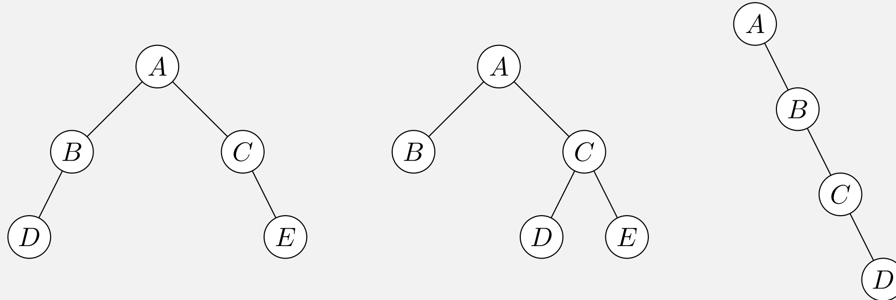
— c'est afficher le nombre n si on est à une feuille ;
— sinon c'est afficher une parenthèse ouvrante, afficher l'enfant gauche, afficher l'opération contenue dans le nœud courant, puis afficher l'enfant droit et enfin afficher une parenthèse fermante.

Exercice 46

1. Exécuter l'algorithme affichage d'une expression sur l'arbre correspondant à l'expression arithmétique $\big((5+3)\times(7+2)\big)$ en vous assurant que vous retrouvez bien l'expression attendue.
2. Écrire l'algorithme récursif permettant d'afficher une expression arithmétique stockée sous la forme d'un arbre binaire.

Exercice 47

1. Appliquer les trois algorithmes de parcours depuis la racine des arbres binaires suivants, en montrant les valeurs affichées.



2. Trouver un arbre binaire dont le parcours préfixe est A, B, C, D, E, F et le parcours infixe est C, B, E, D, F, A. (*Attention, on cherche bien un seul arbre qui admet ces deux parcours à la fois !*)
3. Trouver deux arbres binaires différents dont le parcours préfixe est A, B, D, C, E, G, F et le parcours postfixe est D, B, G, E, F, C, A.

7.7 Arbres binaires de recherche

Pour finir ce chapitre, considérons une dernière application des arbres en informatique. Pour cela, revenons à une des motivations du chapitre 3 : le stockage d'un annuaire téléphonique dans lequel on veut pouvoir chercher facilement un nom (pour connaître le numéro de téléphone qui lui est associé dans l'annuaire). Nous avons alors proposé une solution sous forme de tableau. En comparant deux algorithmes de recherche dans un tableau (et donc de recherche d'un nom dans un annuaire), l'algorithme de recherche séquentielle et l'algorithme de recherche dichotomique dans un tableau trié, nous pouvons déduire que la meilleure implémentation d'un annuaire jusque-là est un tableau trié (qui représente bien l'annuaire physique où les noms sont triés dans l'ordre alphabétique) : l'algorithme de recherche dichotomique permet alors de rechercher un nom dans l'annuaire avec une complexité $O(\log n)$ si l'annuaire contient n noms. Même si on place dans l'annuaire 60 millions d'habitants, le temps de recherche reste minuscule, puisque $\log_2(60\ 000\ 000) \approx 26$.

Que se passe-t-il si un nouvel abonné arrive en cours d'année, ou si un abonné ne souhaite plus avoir son nom dans l'annuaire ? Avec les bottins imprimés, pas d'autre solution que d'attendre la prochaine édition où la totalité de l'annuaire sera réimprimée. Que peut-on faire si on utilise un annuaire numérique ? Imaginons un annuaire stocké dans un tableau trié par ordre alphabétique des noms :

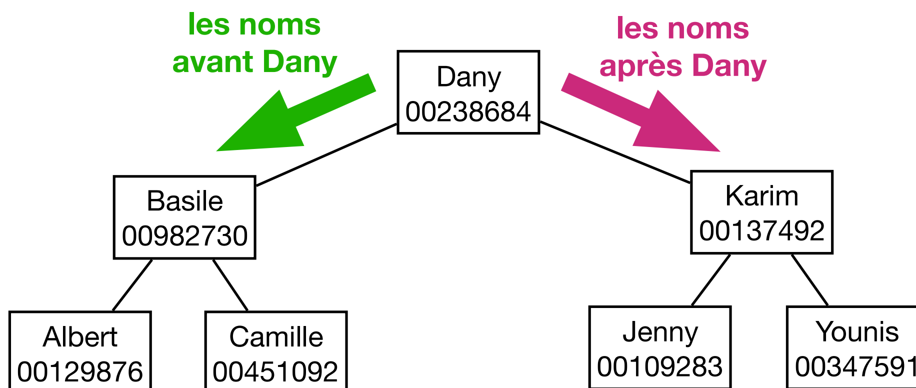
Albert 00129876	Camille 00451092	Dany 00238684	Jenny 00109283	Karim 00137492	Younis 00347591

Basile est un nouvel abonné qu'il nous faut ajouter dans l'annuaire. On peut utiliser la recherche dichotomique pour trouver très rapidement la position dans le tableau où il faut l'insérer (entre Albert et Camille).

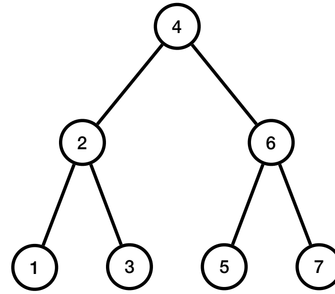
Albert 00129876	Basile 00982730	Camille 00451092	Dany 00238684	Jenny 00109283	Karim 00137492	Younis 00347591

Mais l'insérer dans le tableau nécessite de « déplacer » tous les noms qui le suivent dans l'annuaire : c'est donc très coûteux ! Pour un bottin, cela veut dire qu'il faut refaire toute la mise en page des pages qui suivent celle où on l'a ajouté. Même pour un tableau stocké dans la mémoire d'un ordinateur, cela demande de déplacer d'une case vers la droite le contenu de toutes les cases de la dernière jusqu'à la case qu'on doit libérer pour insérer Basile : cet algorithme a donc complexité $O(n)$ dans le pire des cas (c'est-à-dire quand on doit insérer un nouvel abonné en tête du tableau).

À la place, nous allons proposer une nouvelle structure de données pour stocker un annuaire (ou un dictionnaire) dont l'insertion d'un nouvel élément sera, comme la recherche, de complexité $O(\log n)$. Il s'agit d'un arbre binaire :



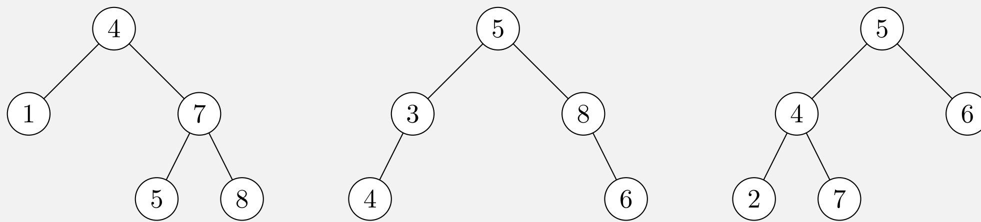
Plus précisément, c'est un *arbre binaire de recherche* (ABR). Il s'agit d'un arbre tel que chaque nœud x de l'arbre vérifie la propriété suivante : toutes les valeurs des nœuds dans l'enfant gauche de x sont inférieures à la valeur de x, et toutes les valeurs des nœuds dans l'enfant droit de x sont supérieures à la valeur de x. Sur l'exemple précédent, tous les noms avant Dany dans l'annuaire sont à gauche de la racine, et tous les noms après Dany dans l'annuaire sont à sa droite. Mais aussi, parmi tous les noms à gauche de Dany, ceux avant Basile sont à sa gauche et ceux après Basile sont à sa droite. Voici un autre ABR où les nœuds sont des entiers :



À gauche de la racine se trouve des entiers inférieurs à 4. À droite du nœud 6, ne se trouve que des entiers supérieurs à 6.

Exercice 48

Lesquels des arbres suivants sont des ABR et pourquoi ?



Comment rechercher une valeur dans un ABR, comme par exemple un nom dans l'annuaire ? On peut utiliser un algorithme récursif qui *descend* dans l'arbre en partant de la racine et en continuant à gauche ou à droite à l'aide d'une comparaison entre la valeur à chercher et le nœud courant. Notez la proximité entre cet algorithme et la recherche dichotomique qui décidait aussi de continuer dans la moitié gauche ou droite du tableau restant, à l'aide d'une comparaison entre la valeur cherchée et le milieu du tableau : c'est la racine de l'ABR qui joue le rôle du milieu du tableau.

```
def est_présent_abr(nœud, x):
    if nœud == nil:
        return False
    elif x == valeur[nœud]:
        return True
    elif x < valeur[nœud]:
        return est_présent_abr(enfant_gauche[nœud], x)
    else:
        retourner est_présent_abr(enfant_droit[nœud], x)
```

Dans l'ABR précédemment donné en exemple, la recherche de la valeur 3 se déroule comme suit :

- à la racine, on compare 3 et 4 : puisque $3 < 4$, on continue la recherche dans l'enfant gauche, le nœud 2 ;
- puisque $2 < 3$, on continue la recherche dans l'enfant droit ;
- c'est le nœud 3 et la recherche s'arrête donc avec succès.

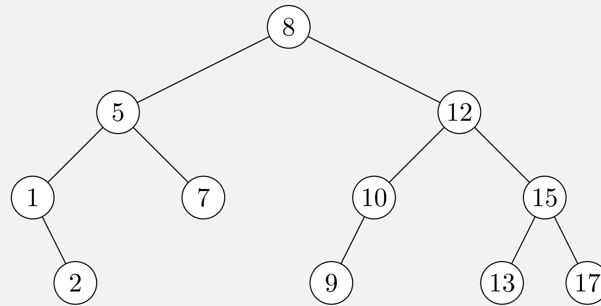
Si on recherche la valeur 8, voici l'exécution :

- à la racine, on a $4 < 8$, donc on continue la recherche dans l'enfant droit, le nœud 6 ;
- puisque $6 < 8$, on continue à nouveau la recherche à droite, dans le nœud 7 ;
- à nouveau $7 < 8$, on continue donc la recherche à droite ;

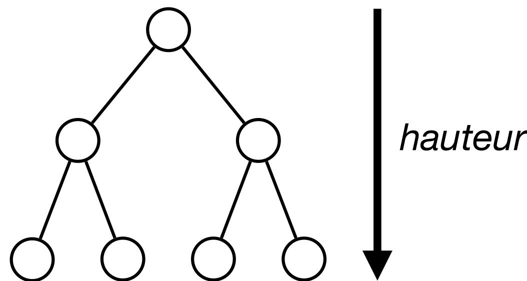
— c'est désormais l'arbre vide `nil` : nous n'avons pas trouvé la valeur 8 dans l'arbre et on peut s'arrêter sur cet échec.

Exercice 49

Appliquer l'algorithme de recherche pour tester la présence des valeurs 15, 7, 2, 11, 3 dans l'ABR suivant, en donnant la liste des nœuds où l'algorithme s'appelle récursivement.



Quelle est la complexité de cet algorithme ? À chaque appel récursif, on plonge dans l'enfant gauche ou l'enfant droit, donc on descend d'un niveau à chaque appel. Ainsi, les appels récursifs suivent une branche (un chemin de la racine à l'une des feuilles de l'arbre) de l'arbre et la complexité est donc proportionnelle à la longueur maximale d'une de ses branches. Par analogie avec nos dessins, on appelle *hauteur* cette longueur maximale d'une branche de l'arbre. Par exemple, l'ABR suivant a hauteur 2, de sorte qu'un arbre réduit à sa racine aura hauteur 0 par convention :



Comment la hauteur d'un ABR se compare-t-elle au nombre n de nœuds que l'arbre contient ? Malheureusement pas toujours très bien : la hauteur peut être de l'ordre de n dans le pire des cas. Mais si l'ABR est *équilibré* (comme c'est le cas au-dessus), c'est-à-dire qu'on essaie de conserver autant de nœuds dans l'enfant gauche et l'enfant droit de n'importe quel nœud de l'arbre, alors la hauteur devient proportionnelle à $\log_2 n$. Dans ce cas, la complexité de l'algorithme de recherche dans un ABR a complexité $O(\log n)$, comme pour la recherche dichotomique.

Attelons-nous finalement au problème initial : l'ajout de nouveaux abonnés dans l'annuaire ! Cela correspond à l'insertion d'une nouvelle valeur dans un ABR. En fait, nous n'avons pas beaucoup à faire, le principal étant de savoir chercher la position où devrait se trouver l'élément à insérer. De deux choses l'une :

— soit on trouve l'élément à insérer : dans ce cas, on renvoie une erreur puisqu'on ne souhaite pas avoir deux occurrences de la même valeur dans l'ABR ;

— soit on ne le trouve pas, ce qui veut dire que l'algorithme de recherche s'est retrouvé en-dessous d'une feuille, dans un arbre vide *nil* (comme lorsqu'on a recherché 8 dans l'ABR précédemment) : il ne reste plus alors qu'à remplacer cet arbre vide par une nouvelle feuille avec la valeur à insérer. On obtient bien un nouvel ABR avec tous les nœuds de l'ABR initial auquel s'ajoute la valeur à insérer.

L'algorithme se résumant finalement à une recherche dans l'ABR, sa complexité est du même ordre que la complexité de la recherche : $O(n)$ dans le pire des cas, et $O(\log n)$ dans le cas d'un ABR équilibré.

Exercice 50

1. Construire un ABR en insérant une par une (à partir de l'arbre vide) les valeurs $9, 5, 12, 2, 15, 7, 11$ dans l'ordre. Ensuite, construire un autre ABR en insérant les mêmes valeurs, mais dans l'ordre $2, 5, 7, 9, 11, 12, 15$. Quelle est la hauteur des deux arbres et quelle différence y aura-t-il du point de vue de la complexité lors d'une recherche dans le pire des cas ?
2. Appliquer l'algorithme de parcours infixe aux arbres binaires construits dans la question précédente. Dans quel ordre l'algorithme affiche-t-il les valeurs ?
3. En déduire un algorithme de tri de tableau (qu'on pourra décrire avec des phrases) qui utilise un ABR comme structure de données auxiliaire.

Chapitre 8

Calculabilité

Nous avons vu dans le dernier chapitre de nombreuses applications des arbres, ces graphes particuliers qui ne possèdent pas de cycles. Dans ce chapitre, nous allons étudier des arbres (puis des graphes) particuliers servant de modèle de calcul. Jusqu'à maintenant, nous avons surtout étudié des algorithmes et des structures de données permettant de résoudre des problèmes concrets. Mais, dans l'introduction, nous avons évoqué le problème de savoir si un problème possède une solution : nous avons vu le problème géométrique de la quadrature du cercle qui s'est avéré ne pas avoir de solution. Existe-t-il de telles impossibilités en informatique également, c'est-à-dire des problèmes tels qu'aucun calcul, aucun algorithme n'est capable de le résoudre ? Pour pouvoir répondre à cette question, il nous est nécessaire de préciser ce que nous entendons par *impossible*, et donc ce qui est *possible* également, à savoir *calculable*. Ce chapitre évoque dans un premier temps une notion assez faible de calculabilité, les arbres de décision, que nous enrichirons ensuite avec les automates, puis finalement avec les machines de Turing.

8.1 Arbres de décision

Commençons donc par utiliser des arbres pour prendre des décisions. C'est très naturel et c'est ce que nous faisons sans même le savoir tous les jours. Par exemple, imaginons

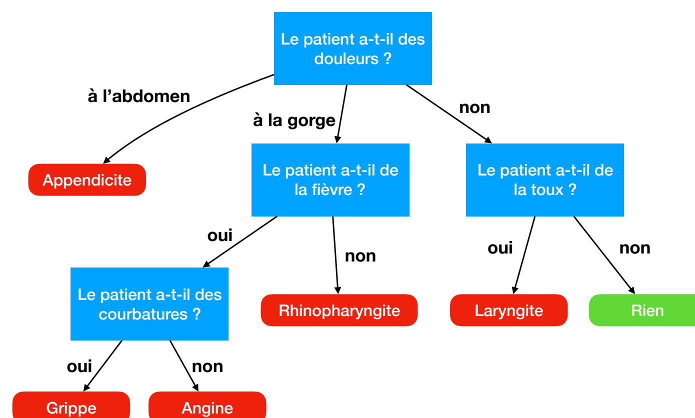


FIGURE 8.1 – Un arbre de décision pour décider la maladie du patient

FIGURE 8.2 – Le jeu du *Qui est-ce ?*

un médecin qui doit diagnostiquer un patient entrant dans son cabinet. Comment procède-t-il ? Une façon de faire est de commencer par poser une première question, par exemple « Avez-vous des douleurs ? » : suivant la réponse, le médecin apprend de l'information et il peut continuer à poser des questions, jusqu'à avoir appris suffisamment d'informations pour prendre une décision, c'est-à-dire déclarer si le patient est malade et si oui, quelle maladie il a probablement. Les différents états de connaissance du médecin au cours du diagnostic sont reliés par des arcs selon les réponses aux questions qu'il pose : cela forme un arbre (en effet, un cycle dans ce diagramme de diagnostic n'aurait pas beaucoup de sens puisque cela voudrait dire que le médecin pose deux fois la même question), qu'on appelle *arbre de décision*. Un exemple très simplifié de tel arbre de décision est représenté en Figure 8.1.

Lorsque nous jouons au jeu du *Qui est-ce ?* (cf Figure 8.2), nous procédons par élimination successive de personnages en posant à l'autre joueur des questions auxquelles il répond par oui ou non. On peut à nouveau représenter une stratégie de l'un des joueurs à l'aide d'un arbre de décision : chaque nœud de l'arbre est à nouveau une question qu'il pose à son adversaire et selon la réponse apportée (oui ou non), on continue dans le sous-arbre gauche ou le sous-arbre droit. Le début d'un arbre de décision possible pour ce jeu est représenté en Figure 8.3 : les feuilles de cet arbre sont les situations où un seul visage reste possible auquel cas le joueur a deviné (sauf erreur ou tricherie de l'adversaire...) le personnage choisi par l'adversaire.

Un dernier exemple d'arbre de décision que nous utilisons tous les jours sans le savoir consiste en la détection de spams dans les applications d'emails. Par exemple, considérons les deux emails suivants :

Bravo,
Vous venez de gagner à notre grand tirage internet. Cliquez ici pour recevoir 1.000.000 de dollars ! ! ! Et pour gagner d'autres cadeaux, cliquez ici.

Salut Benoît,
Demain je joue contre Bruno. Si je gagne la partie, je me qualifie directement. Si je ne gagne pas demain mais que je gagne la suivante, je monte en pool 3 l'année prochaine !
Bises, Sandra

Comment détecter si ce sont des spams ou des messages réguliers qu'il ne faut pas filtrer ? La méthode la plus simple, très efficace, consiste à utiliser des arbres de décision. Les questions sont des critères qu'on peut vérifier sur chaque email, par exemple la comparaison du nombre d'occurrences d'un mot (ou sa fréquence d'apparition dans l'email) avec une constante. Un tel arbre de décision simplifié est donné en Figure 8.4.

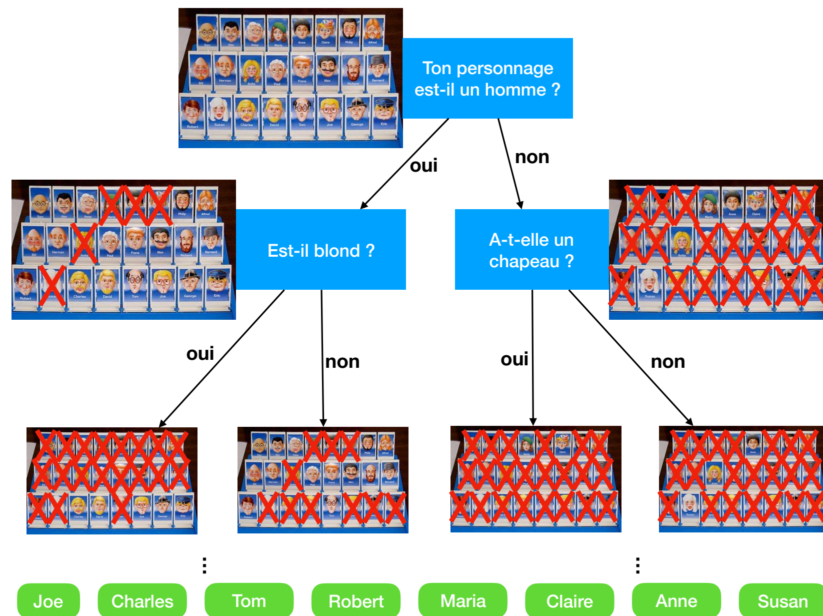


FIGURE 8.3 – Un extrait de l'arbre de décision pour représenter une stratégie au *Qui est-ce ?*

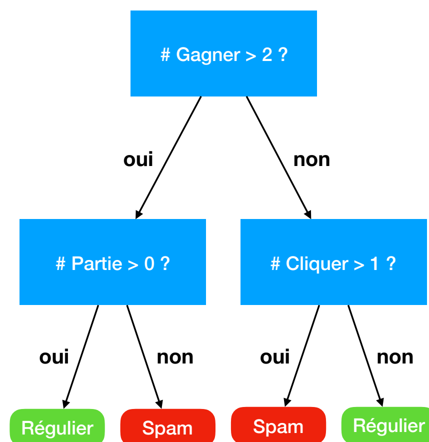
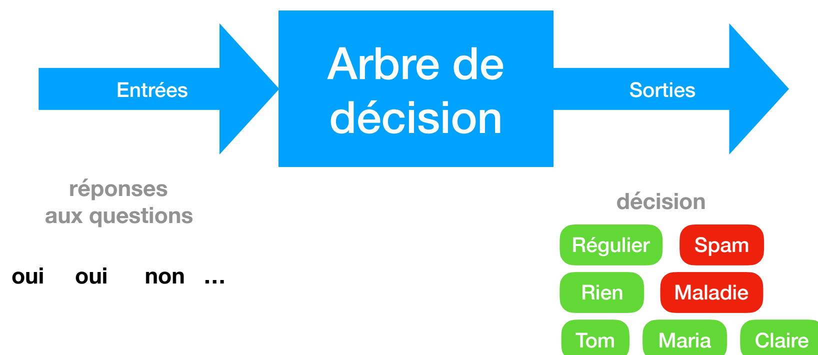


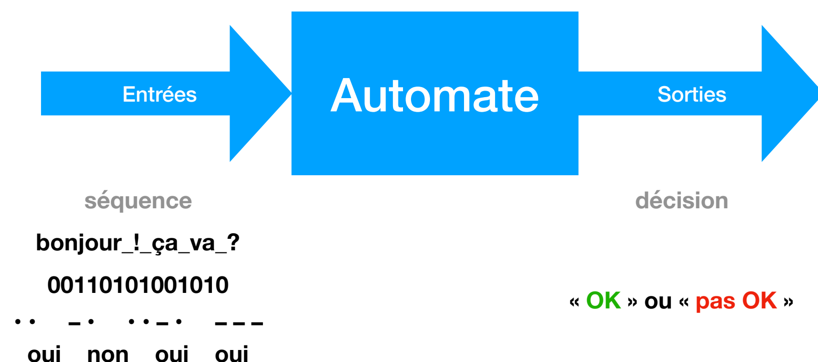
FIGURE 8.4 – Un arbre de décision simplifié pour détecter les spams

Pour le message de gauche, le nombre de fois que le mot « gagner » apparaît est inférieur ou égal à 2, mais le mot « cliquer » (sous une forme conjuguée) apparaît strictement plus d'une fois : l'arbre de décision conclut donc que c'est un spam (la troisième feuille en partant de la gauche). Pour le second message en revanche, le mot « gagner » apparaît strictement plus de deux fois, mais le mot « partie » apparaît aussi, donc l'arbre de décision permet de détecter qu'il s'agit probablement d'un email régulier. En pratique, les arbres de décision utilisées pour détecter les spams sont bien plus grands et ils sont mis au point à l'aide de méthodes d'apprentissage automatique, à partir d'une grande quantité d'emails qui sont déjà classifiés comme étant réguliers ou spams : on parle d'*apprentissage supervisé*.

Essayons désormais de préciser ce qu'on entend par « calculer avec un arbre de décision ». Dans les exemples que nous avons vus jusqu'alors, un arbre de décision est un processus de calcul qui prend en entrée des réponses (oui ou non) aux questions qu'il pose, et les utilise afin de produire une sortie, la décision attendue (régulier/spam, malade/pas malade, le nom du personnage du *Qui est-ce ?*) :



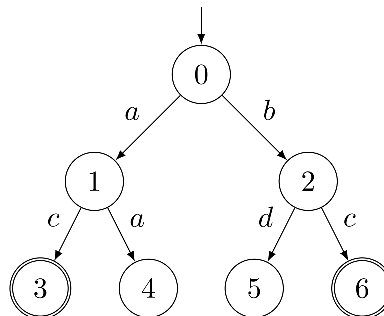
Si on s'abstrait un peu des exemples, on peut considérer qu'on prend en entrée une séquence de réponses prises dans un ensemble fini de réponses possibles, et qu'on prend une décision « OK » ou « pas OK », suite à la séquence de réponses observée. Cette étape de généralisation de ce que peut prendre un arbre de décision en entrée nous invite à changer de nom : désormais, nous appellerons un tel processus prenant des séquences en entrée et renvoyant « OK »/« pas OK » en sortie un *automate* :



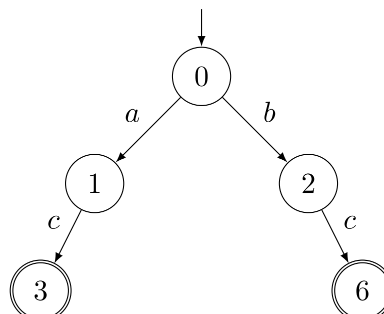
En entrée, on peut donc imaginer que l'automate prend des caractères pour lire des séquences de caractères (du texte, donc), ou bien des séquences de 0 et de 1 (pour représenter des entiers en binaire par exemple), ou bien encore des points et des traits, éléments de base

du code Morse. Les arbres de décision sont donc des automates particuliers, qui lisent des séquences de « oui »/« non ». Dans le cas général, les sommets de l'arbre de décision portent le nom d'*états* dans les automates : l'état d'où l'on part au début du processus (la racine de l'arbre de décision) est appelé *état initial*. Les éléments qu'on lit (des caractères, des 0/1, des oui/non...) sont appelés des *lettres* et les arcs d'un état à l'autre, étiquetés par une lettre, sont appelés des *transitions*. Certains états sont les états où l'automate prend une décision : certains sont *acceptants* (ceux où on déclare la séquence de lettres lues comme étant « OK ») et les autres sont non acceptants.

Voici un premier exemple d'automate sur l'alphabet $\{a, b, c, d\}$:

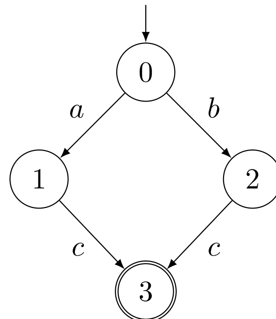


Cet automate possède sept états numérotés de 0 à 6. L'état 0 est l'état initial. Seuls les états 3 et 6 sont acceptants (parmi les feuilles $\{3, 4, 5, 6\}$) : on les représente par un double cercle. Un automate décrit un ensemble de séquences « valides », c'est-à-dire celles pour laquelle il décide « OK ». Pour l'exemple ci-dessus, seules deux séquences de lettres permettent d'aller de l'état initial à un état acceptant : la séquence *ac* et la séquence *bc*. Les séquences *aa* et *bd* parviennent à des feuilles mais ne sont tout de même pas acceptées par l'automate. On dit que le *langage reconnu par l'automate* est $\{ac,bc\}$. Notez que la séquence a n'est pas reconnue par l'automate, mais on peut l'étendre de façon à atteindre un état acceptant. En revanche, non seulement *aa* n'est pas reconnue, mais aucune suite possible ne mène à un état acceptant. C'est la même chose que pour la séquence c ou d : elles ne sont pas acceptées et aucune suite possible ne permet d'atteindre un état acceptant. C'est la raison pour laquelle on ne les a même pas représentées sur le dessin. De même, on peut donc ignorer les transitions menant de 1 à 4 et de 2 à 5. L'automate ci-dessous est donc équivalent au précédent, au sens où il accepte le même langage :

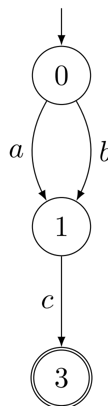


Maintenant qu'on a supprimé des états inutiles, il apparait qu'on peut encore simplifier cet automate. En effet, on peut essayer de *partager* des états, quitte à ne plus insister sur le fait que l'arbre de décision soit un *arbre*. Par exemple, il ne rime à rien de distinguer les

états 3 et 6 qui sont tous deux acceptants mais ne permettent pas de continuer à lire des lettres. On peut donc les fusionner sans crainte :



Mais alors, on s'aperçoit qu'il est aussi inutile de distinguer les états 1 et 2 qui sont tous deux non acceptants, mais permettent de lire uniquement la lettre c en se rendant dans le même état 3 : ils réagissent de la même manière dans le futur, donc on peut sans crainte les fusionner également. On obtient donc l'automate suivant :



Cela n'a plus rien d'un arbre, mais on appelle toujours cela un automate. Voyons donc une définition formelle de ces automates finis (car on se concentre sur les automates qui possèdent un ensemble fini d'états).

8.2 Automates finis

Un automate permet de décrire des ensembles de séquences sur un alphabet A donné :

— si l'alphabet est $A = \{0, 1\}$, l'automate décrit un ensemble de séquences de 0 et 1 : un automate peut ainsi accepter des codes binaires d'entiers ;

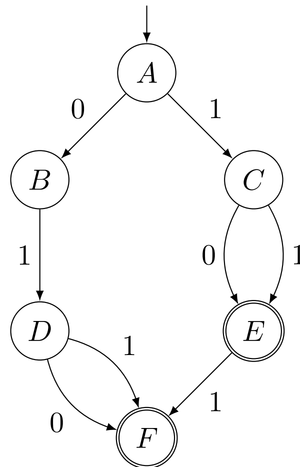
— si l'alphabet est $A = \{a, b, c, \ldots, y, z\}$, l'automate décrit un ensemble de mots sur l'alphabet latin : un automate peut ainsi accepter l'ensemble des mots du dictionnaire français ;

— si l'alphabet est $A = \{oui, non\}$, l'automate accepte des séquences de décisions à certaines questions : un automate permet alors de représenter une stratégie au jeu du *Qui est-ce ?* dans un arbre de décision.

Un automate est composé d'*états* et de *transitions* qui sont des arcs menant d'un état (potentiellement acceptant, contrairement aux exemples vus jusqu'à maintenant) à un autre

(potentiellement le même), étiquetés par des lettres de l'alphabet A. Un automate possède également un état *initial* et des états *acceptants*.

On représente ci-dessous un automate $\mathcal{A}$, sur l'alphabet $\{0, 1\}$:



Ses états sont A, B, C, D, E et F. L'état initial est A, distingué par une flèche entrante. Les états acceptants sont E et F distingués par un double cercle. Cet automate possède 8 transitions : par exemple, il y en a une de l'état A vers l'état B étiquetée par la lettre 0, et une autre de l'état E à l'état F étiquetée par la lettre 1.

Notons que tous les automates vus jusqu'alors vérifient la propriété suivante :

> « Pour tout état e de l'automate et pour toute lettre ℓ de l'alphabet, il existe au plus une transition sortant de l'état e étiquetée par la lettre ℓ. »

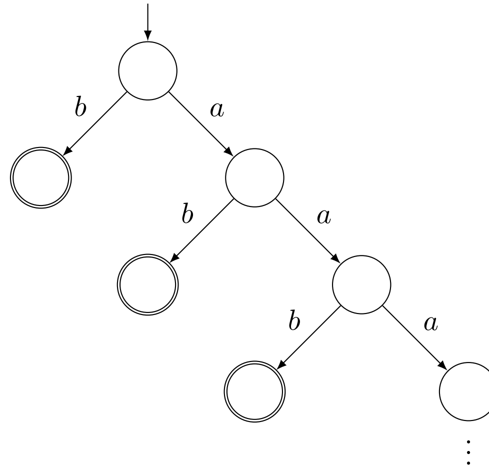
On dit que l'automate est *déterministe* en cela qu'à tout moment il n'y a aucun choix pour continuer la lecture d'une séquence de lettres fixée à l'avance : soit on bloque car il n'y a pas de transition étiquetée par la lettre voulue (par exemple, si on veut lire deux lettres 0 dans l'automate du dessus, on bloque en B qui ne peut pas lire une lettre 0 en suivant une transition), soit on suit l'unique transition étiquetée par la lettre voulue. Dans ce cours, on considèrera uniquement des automates déterministes.

Un automate accepte un *langage*, qui est un ensemble de séquences. Une séquence s est *acceptée* par l'automate s'il existe un chemin (et s'il existe, il est unique) de l'état initial à un état acceptant dont la séquence des étiquettes des transitions visitées (dans l'ordre) est s. Ainsi, la séquence 011 est acceptée par l'automate $\mathcal{A}$ ci-dessus, mais pas la séquence 01 qui ne termine pas dans un état acceptant, ni la séquence 110 pour laquelle il n'y a pas de chemin avec cette étiquette (on bloque en E lorsqu'il s'agit de lire la lettre 0). L'ensemble des séquences acceptées par l'automate $\mathcal{A}$ est $\{010, 011, 10, 11, 101, 111\}$: ce sont l'ensemble des codages binaires sur moins de 3 bits représentant des entiers premiers (2, 3, 5, 7).

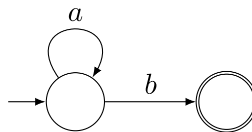
Jusque-là, nous n'avons vu que des automates qui acceptent un langage fini de séquences. Comment peut-on accepter un langage infini ?[1] Prenons un exemple : peut-on trouver un automate acceptant toutes les séquences sur l'alphabet $\{a, b\}$ qui ne contiennent que des a sauf la dernière lettre qui doit être un b, c'est-à-dire le langage $\{b, ab, aab, aaab, aaaab, \ldots\}$. Clairement, ce langage contient un ensemble infini de séquences. Notre première tentative

1. Attention, c'est bien le langage dont on veut qu'il contienne un ensemble infini de séquences, pas les séquences elles-même qui sont toujours finies dans ce cours.

pourrait être de dessiner un arbre de décision : depuis l'état initial, soit on lit un b et on accepte directement, soit on lit un a et on va dans un nouvel état qui, à nouveau soit lit un b et accepte directement, soit lit un a et poursuit une étape de plus, etc.

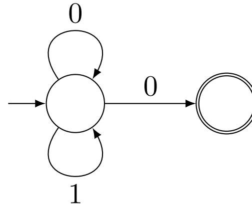


Malheureusement, cet automate n'est pas fini : il possède un nombre infini d'états. Plus précisément, il a un état acceptant par séquence à accepter... Comme précédemment cependant, on se rend compte que tous les états acceptants peuvent être aisément fusionnés puisqu'ils sont tous acceptants et ne permettent de lire aucune lettre supplémentaire. Une fois qu'on a fait cela, on peut se rendre compte que tous les états non acceptants sont alors ≪ similaires ≫ : il permettent tous de partir vers l'unique état acceptant en lisant un b, ou continuer vers eux en lisant un a. On peut donc tous les fusionner et on obtient finalement l'automate fini (car il possède un nombre fini d'états) suivant :

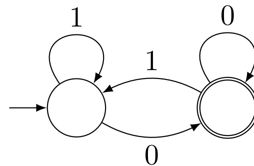


Dans les dessins, il n'est pas toujours nécessaire de donner des noms aux états, comme dans l'exemple de l'automate $\mathcal{B}$. Mais on peut évidemment donner des noms si on le souhaite, pour aider à comprendre la signification de l'état : ici, l'état de gauche de $\mathcal{B}$ pourrait s'appeler ≪ que des a ≫ et l'état de droite ≪ b à la fin ≫, ou simplement ≪ A ≫ et ≪ B ≫ si on veut des noms plus courts...

Revenons sur l'alphabet {0,1} pour un nouvel exemple de langage infini. Les séquences sur cet alphabet représentent des entiers codés en binaire. Peut-on trouver un automate acceptant l'ensemble des codes des entiers pairs ? Rappelons-nous qu'un entier est pair si et seulement si son codage en binaire termine par un 0. On souhaite donc reconnaître l'ensemble des séquences de 0 et de 1 qui terminent par un 0 : cela ressemble au langage précédent (à renommage près du a en 1 et du b en 0), mais cette fois-ci, on doit pouvoir accepter n'importe quelle séquence avant de lire le *dernier* 0. On pourrait donc se dire qu'un automate permettant d'accepter ce langage est :



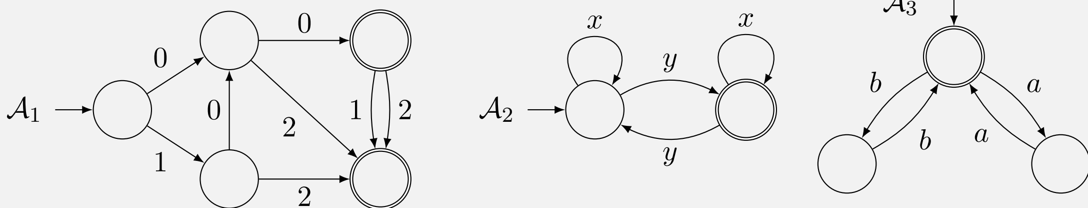
On lit n'importe quelle séquence de 0 et de 1 en restant dans l'état de gauche, avant de passer dans l'état de droite en lisant un 0. Malheureusement, cet automate n'est pas *déterministe* : dans l'état de gauche, en lisant un 0, l'automate a le choix entre rester dans l'état de gauche, ou passer dans l'état de droite. On ne s'autorise pas de tels automates à choix dans ce cours. Il nous faut donc modifier notre premier essai. En fait, notons qu'une fois qu'on vient de lire un 0, on doit nécessairement se trouver dans un état acceptant puisqu'il se pourrait que ce 0 soit le dernier bit du codage binaire. Par contre, lorsqu'on vient de lire un 1, on doit se trouver dans un état non acceptant. Cela nous amène directement à cette seconde tentative :



Cet automate est déterministe et il accepte bien toutes les séquences de bits qui terminent par un 0, c'est-à-dire les codages binaires des entiers pairs.

Exercice 51

Pour chacun des trois automates $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{A}_3$ ci-dessous, décrire l'alphabet ainsi que le langage de toutes les séquences acceptées : lorsque le langage est infini, on peut le décrire avec une phrase à défaut de pouvoir énumérer toutes les séquences.



Exercice 52

Dessiner un automate qui accepte l'ensemble des séquences sur l'alphabet $\{0, 1\}$ possédant un nombre de 1 multiple de 3 : à titre d'exemple, la séquence 0011010 doit être acceptée, mais pas la séquence 1101011.

Exercice 53

On se place, dans cet exercice, sur l'alphabet $\{0, 1\}$.

1. Dessiner un arbre de décision qui accepte l'ensemble des séquences de longueur au plus 4 qui possède autant de 0 que de 1 (dans n'importe quel ordre).
2. Simplifier l'arbre de décision précédent pour obtenir un automate avec un

nombre minimal d'états : on ne demande pas de montrer que l'automate a un nombre minimal d'états. (*Indication : commencer par regrouper les états acceptants feuilles de l'arbre de décision, puis regrouper les états qui ont le même comportement, comme on l'a fait avant dans le cours.*)

3. À partir de l'automate obtenu à la question précédente, en déduire un automate qui accepte l'ensemble des séquences de longueur au plus 6 qui possède autant de 0 que de 1.

Exercice 54

L'alphabet Morse donne un code pour les 26 lettres de l'alphabet composé d'impulsions courtes (•) et longues (▬) :

A •▬	B ▬•••	C ▬•▬•	D ▬••	E •	F ••▬•
G ▬▬•	H ••••	I ••	J •▬▬▬	K ▬•▬	L •▬••
M ▬▬	N ▬•	O ▬▬▬	P •▬▬•	Q ▬▬•▬	R •▬•
S •••	T ▬	U ••▬	V •••▬	W •▬▬	X ▬••▬
Y ▬•▬▬	Z ▬▬••				

Trouver un automate, ayant le plus petit nombre d'états possibles, qui accepte exactement l'ensemble des codes Morse.

8.3 Applications des automates finis

Les automates finis sont très utiles pour modéliser des situations de la vie courante. Ils sont d'ailleurs intensivement utilisés à ces fins dans l'industrie. Prenons un exemple : modéliser le comportement normal d'un distributeur de café. Voici les spécifications données par l'industriel qui les construit :

— le café est à 30 centimes d'euro ;
— le thé est à 50 centimes d'euro ;
— la machine n'accepte que les pièces de 10 et 20 centimes d'euro ;
— on peut insérer 50 centimes d'euro au maximum dans la machine ;
— on peut annuler à tout moment (et récupérer la monnaie).

Décrivons alors l'ensemble des fonctionnements normaux du distributeur à l'aide d'un automate. Il admet comme alphabet les cinq lettres suivantes : « insertion d'une pièce de 10 centimes », « insertion d'une pièce de 20 centimes », « appui sur le bouton *Café* et distribution du café », « appui sur le bouton *Thé* et distribution du thé », « appui sur le bouton *Cancel* et récupération de la monnaie ». On ne considère donc pas comme un fonctionnement normal l'appui sur le bouton *Café* ou *Thé* sans distribution de café ou de thé : dis autrement, cela veut dire que le distributeur ne doit rien faire dans le cas où l'utilisateur appuierait sur l'un de ces deux boutons dans un autre cas que ceux considérés dans le fonctionnement normal. Afin de pouvoir savoir quand le distributeur doit délivrer du café ou du thé, il nous faut retenir le montant déjà introduit dans la machine : ce sont le rôle des états dans un automate, qui sont l'unique mémoire disponible. On créé donc 6 états : 0 centimes (début), 10 centimes, 20 centimes, 30 centimes, 40 centimes et 50 centimes. Il ne reste plus qu'à ajouter les transitions. Une représentation de l'automate est donné en Figure 8.5. On y a représenté l'état

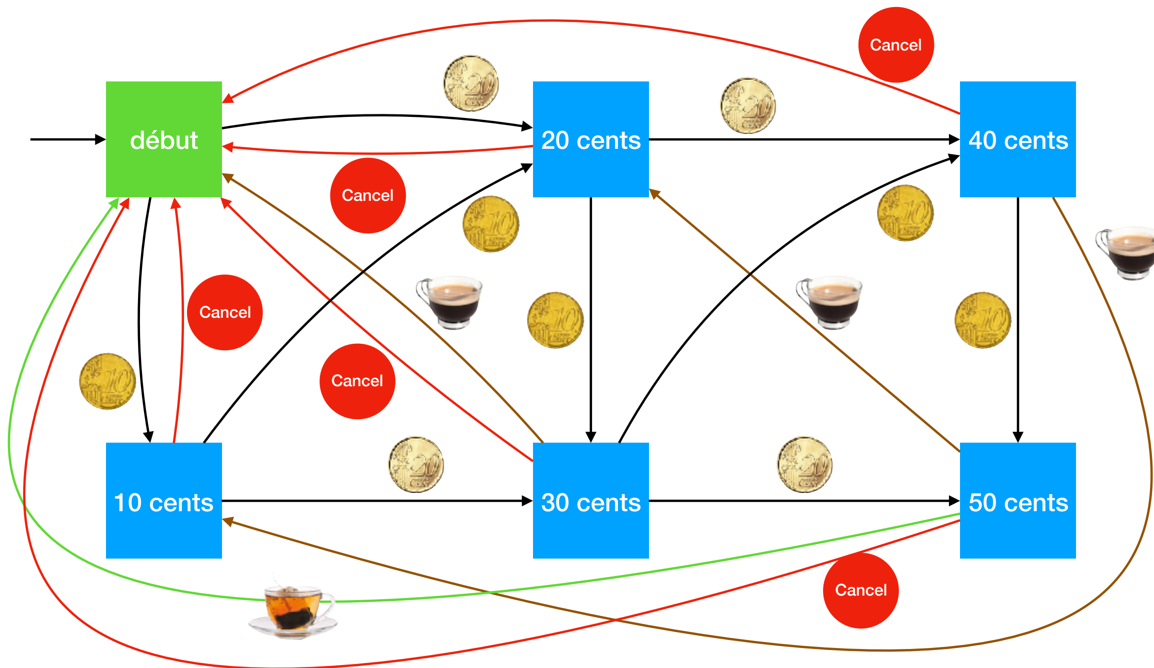


FIGURE 8.5 – Un automate pour le distributeur de café

début en vert pour signifier qu'il pourrait être considéré comme état acceptant de l'automate, c'est-à-dire un état où l'on peut sans souci éteindre le distributeur sans léser un consommateur qui a déjà inséré de l'argent dedans.

Exercice 55

L'objectif de l'exercice est de modéliser le comportement des portes automatiques de bus par le biais d'un automate. On peut demander à ouvrir la porte grâce à un bouton. La porte est aussi équipée d'un capteur de proximité qui permet d'empêcher la fermeture de la porte si une personne est proche. Le principe est que la porte ne doit pas s'ouvrir si un individu passe simplement devant mais s'il l'indique explicitement en appuyant sur le bouton. Le bouton est soit dans l'état *relâché* si personne ne le touche, soit dans l'état *appuyé* si quelqu'un est en train d'appuyer sur le bouton. Le système possède 4 états possibles :

— Etat 0 : personne n'est à proximité de la porte. La porte est fermée.
— Etat 1 : un individu est à proximité de la porte (le capteur de proximité le détecte) et la porte est fermée.
— Etat 2 : un individu a appuyé sur le bouton. La porte est ouverte et le capteur de proximité détecte la personne.
— Etat 3 : un individu est à proximité de la porte (le capteur de proximité le détecte), il n'a pas la main sur le bouton mais la porte est ouverte. Ce cas survient si l'individu appuyait auparavant sur le bouton.

Les événements qui permettent la transition entre états sont :

— **présence** : le capteur de proximité détecte un individu à proximité mais celui-ci ne touche encore pas le bouton.

— **a**bsence : le capteur de proximité n'a détecté aucun individu à proximité depuis au moins 2 secondes. On suppose dans ce cas que personne ne peut appuyer sur le bouton.
— **t**ouche : l'individu vient d'appuyer sur le bouton alors qu'il était déjà dans le champ du capteur de proximité mais qu'il n'avait pas encore touché le bouton.
— **r**elâche : le bouton vient d'être relâché mais l'individu est toujours dans le champ du capteur.

Définir un automate permettant de modéliser le comportement de la porte automatique tel qu'il est décrit ci-dessus : l'automate doit accepter toutes les séquences d'évènements valides, si bien que tous les états sont acceptants. Quelles sont les transitions de votre automate où le système doit donner l'ordre au vérin de la porte d'ouvrir ou de fermer la porte ?

Une autre application possible des automates concerne la recherche de motif dans un texte. Par exemple, dans un navigateur web ou dans un traitement de texte, vous avez toujours la possibilité de rechercher un mot dans le document. Cette tâche est exécutée très efficacement par un automate (à tel point que c'est souvent à l'aide d'automates que les algorithmes de recherche sont effectivement décrits en pratique). Trouver un mot dans un texte, cela revient à décrire un automate qui accepte la partie initiale du texte qui termine par le mot recherché. Typiquement, considérons le texte :

> Du journal « Le petit bachelier » : comme les professeurs
> se plaisent à le rabâcher, le baccalaureat est important... !

Si on recherche le mot « bac » dans ce texte, en ignorant les majuscules et les accents, on trouve trois occurrences :

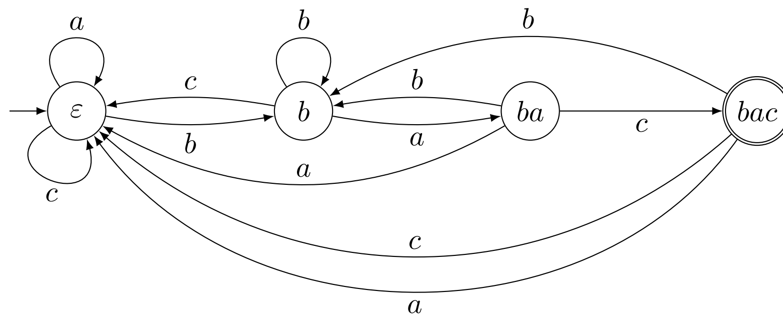
> Du journal « Le petit bachelier » : comme les professeurs
> se plaisent à le rabâcher, le baccalaureat est important... !

Un automate à qui l'on donnerait ce texte devrait donc accepter les trois *préfixes* suivant du texte, tous ceux qui finissent par le motif « bac » :
— Du journal « Le petit bac
— Du journal « Le petit bachelier » : comme les professeurs se plaisent à le rabâc
— Du journal « Le petit bachelier » : comme les professeurs se plaisent à le rabâcher, le bac

Repérer une occurrence du motif revient donc à accepter dans l'automate le préfixe du texte qui termine sur cette occurrence.

Considérons toujours la recherche du motif « bac » mais dans des textes sur l'alphabet simplifié $\{a,b,c\}$, pour éviter d'obtenir un automate difficile à représenter visuellement. Dans ce cas, les mots *aabbbac* et *aabacbbac* doivent être acceptés, mais pas les mots *aabacbba* (même s'il contient *bac* puisqu'il ne termine pas par *bac*), *acababbaa* ou *babc*. L'automate doit accepter le mot *bac* lui-même donc il est raisonnable de commencer par créer quatre états permettant de lire successivement les lettres b, a puis c. Le premier état doit être initial, et le dernier acceptant. Il faut ensuite *compléter* les autres transitions de l'automate en maintenant sa correction. Par exemple, dans l'état initial, il faut pouvoir également lire les lettres a et c : celles-ci ne font pas avancer dans la reconnaissance d'une occurrence du motif, donc, en les lisant, on boucle sur l'état initial. De même, si on lit un c dans le second état, on remet à

zéro notre avancée dans la recherche du motif *bac*, donc on retourne dans l'état initial. En revanche, si on lit un *b* dans le second état, on doit rester dans cet état, puisqu'on continue à voir la première lettre du motif *bac*. En continuant de compléter les transitions jusqu'au dernier état, on obtient l'automate suivant :



Exercice 56

On se place, dans cet exercice, sur l'alphabet $\{a,b,c\}$.

1. Dessiner un automate qui accepte l'ensemble des séquences qui se terminent par *cca* : ainsi, le mot *bacbcca* doit être accepté, mais pas le mot *bccba*, ni le mot *bccab*.
2. Dessiner un automate qui accepte l'ensemble des séquences qui se terminent par *abba* : attention, le mot *abbabba* doit être accepté !

8.4 Langage non accepté par un automate fini

Il est possible de trouver des langages qui ne sont acceptés par aucun automate fini. Reprenons le résultat de l'exercice 53 par exemple, où vous avez cherché des automates pour accepter les mots de longueur 4 puis 6 qui ont autant de lettres 0 que de lettres 1. Vous avez remarqué que la taille de l'automate grossit lorsqu'on passe de la longueur 4 à la longueur 6, et vous devriez avoir acquis l'intuition que l'automate continuerait à grossir si on le demandait pour une longueur $2n$ quelconque. En particulier, cela donne l'impression qu'aucun automate *fini* ne peut accepter le langage des séquences de 0 et de 1 qui possèdent autant de 0 que de 1. C'est en effet le cas :

Théorème 5. *Le langage L des mots sur l'alphabet $\{0,1\}$ qui possèdent autant de 0 que de 1 ne peut pas être accepté par un automate fini. C'est aussi le cas du langage $\{0^n1^n \mid n \in \mathbb{N}\backslash\{0\}\}$.*

La preuve de ce résultat dépasse largement le programme de ce cours, mais l'intuition est la suivante : pour accepter les mots de la forme 0^n1^n, c'est-à-dire les mots de L avec les 0 tous avant le premier 1, un automate devrait avoir une mémoire infinie puisqu'il devrait être capable, une fois qu'il voit le premier 1, de s'être rappelé combien de 0 il a vu jusqu'alors pour accepter exactement le même nombre de 1. Un tel automate devrait donc avoir un nombre infini d'états, ce qui est interdit dans les automates finis.

Cependant, il est facile d'écrire un algorithme qui prend en entrée un tableau contenant des lettres 0 et 1, et renvoie `True` si ce tableau contient autant de 0 que de 1, et `False` sinon.

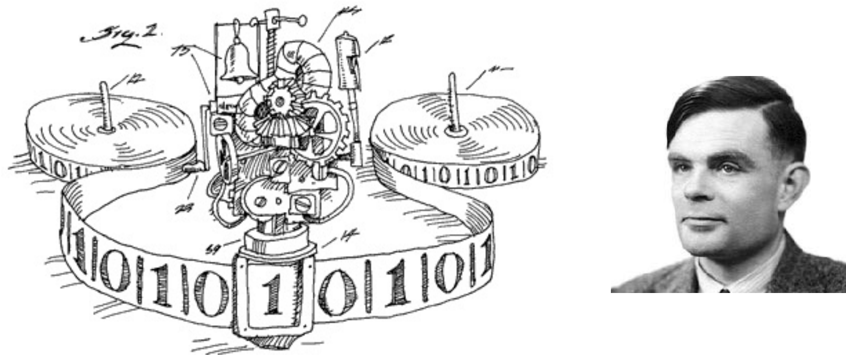


FIGURE 8.6 – À gauche : image illustrant le concept de machine de Turing. À droite : Alan Turing (1912-1954).

Exercice 57

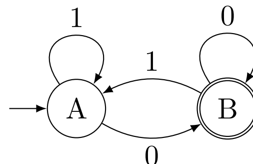
Écrire un algorithme en Python, qui prend en entrée un tableau contenant des lettres 0 et 1, et renvoie `True` si ce tableau contient autant de 0 que de 1, et `False` sinon.

Le modèle d'automates finis, aussi intéressant et puissant soit-il, ne peut donc pas représenter *ce qui est calculable*, puisqu'un algorithme très simple reconnaît un langage qu'un automate ne peut pas accepter. Enrichissons donc ce modèle d'automates finis pour parer à ce problème.

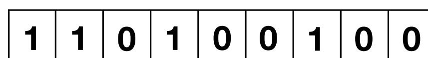
8.5 Des automates vers les machines

Que manque-t-il donc aux automates finis pour pouvoir représenter *tout ce qui est calculable* ? Afin de pouvoir simplement ajouter des caractéristiques supplémentaires, commençons par présenter les automates finis sous un format un peu généralisé, nous rapprochant du comportement d'un ordinateur, d'une machine. Imaginons donc que la séquence à traiter par l'automate est écrite sur un ruban découpé en cases toutes semblables. Le ruban passe au travers d'une tête de lecture fixe, tel que représenté à gauche de la Figure 8.6. La tête de lecture possède un état (parmi un ensemble fini d'états) qui lui permet de retenir un ensemble fini d'informations, de sorte qu'elle réagit différemment ensuite aux lettres qu'elle lit sur le ruban.

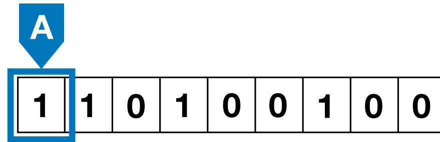
Revenons par exemple sur l'automate suivant ayant deux états :



Exécutons la machine sous-jacente sur l'entrée 110100100. On commence donc avec ce mot écrit sur un ruban :



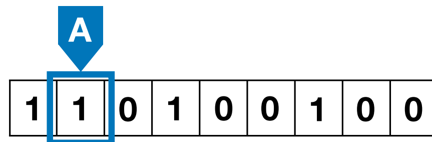
L'état A est l'état initial donc on place le ruban sur la tête de lecture dans l'état A :



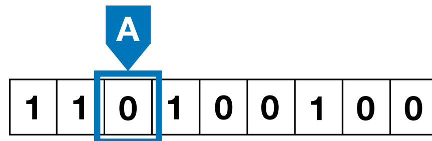
À partir de là, la machine doit savoir ce qu'elle doit faire lorsqu'elle est dans l'état A et qu'elle lit la lettre 1. L'automate nous dit qu'elle doit rester dans l'état A, et elle continue en lisant la lettre suivante du ruban. On peut représenter les quatre transitions possibles de la manière suivante :

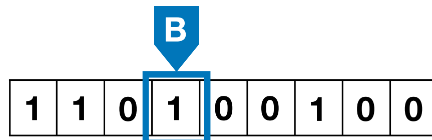
Etat	Symbole	Nouvel état
A	1	A
A	0	B
B	1	A
B	0	B

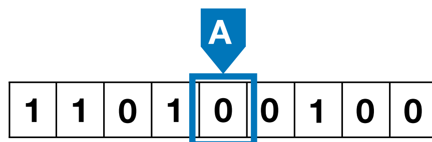
Ainsi, on arrive dans la situation suivante après une étape :

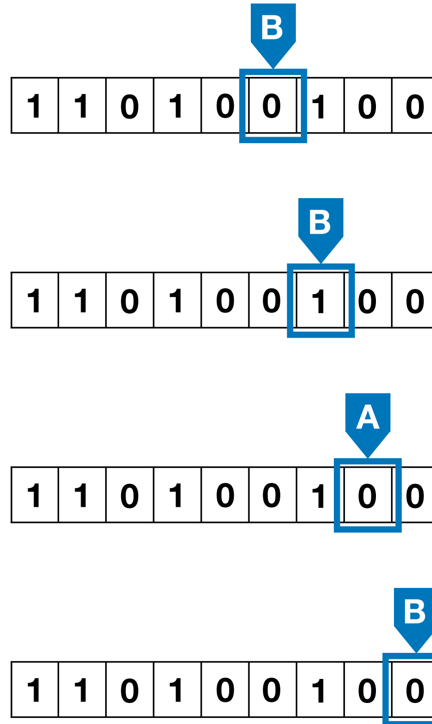


On continue ainsi, via les étapes suivantes :

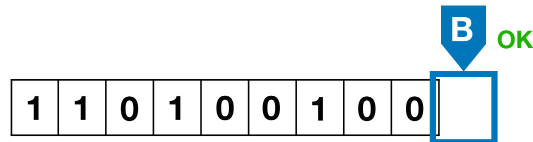






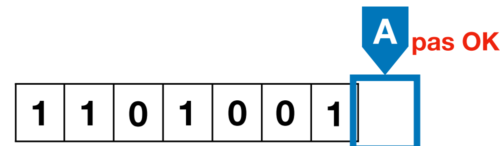


La prochaine étape sort du ruban de lecture :



Il n'y a plus rien à lire et on est dans l'état B qui est acceptant : la machine accepte et on déclare la séquence 110100100 comme acceptée par la machine.

Si on reprend l'exécution de la machine au début, avec la séquence 1101001 sur le ruban, on arrive à la situation suivante :



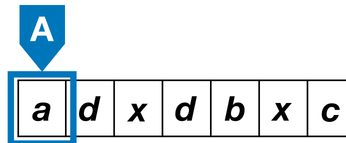
Il n'y a plus rien à lire et on est dans l'état A non acceptant : la machine rejette donc et on déclare la séquence 1101001 comme rejetée par la machine.

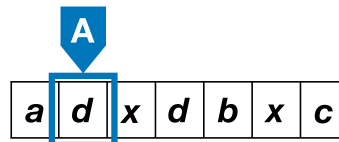
Ajoutons donc désormais des caractéristiques supplémentaires à ces machines très simples, en commençant par la possibilité de changer de sens de lecture. Désormais, lorsque la machine lit une lettre dans un certain état, elle peut soit continuer en allant lire la lettre à droite sur le ruban, soit aller lire la lettre à gauche sur le ruban. On représentera ce sens par une flèche dans les tables de transition.

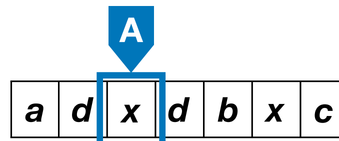
Exécutons par exemple la machine dont un extrait de la table est

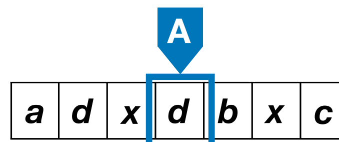
Etat	Symbole	Sens	Nouvel état
A	*a,b,c,…,z*	→	A
A		←	B
B	*a*	←	B*a*
B	*b*	←	B*b*
B	*c*	←	B*c*
…	…	…	…
B	*z*	←	B*z*
B*c*	*a,b,d,…,z*	←	B*c*
B*c*		→	**OK**

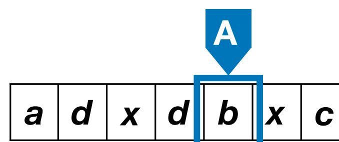
Elle utilise comme alphabet l'ensemble des caractères latins minuscules $\{a,b,c,\ldots,z\}$ ainsi qu'un caractère blanc. Elle a pour états $\{\mathrm{A}, \mathrm{B}, \mathrm{B}a, \mathrm{B}b, \mathrm{B}c, \ldots, \mathrm{B}z, \mathrm{OK}\}$. La première ligne signifie que lorsque la machine est dans l'état A, si elle lit une lettre différente du blanc, elle continue sur la case de droite sur le ruban, toujours dans l'état A. Si on lance la machine avec la séquence *adxdbxc* sur le ruban, on passe donc par les configurations suivantes :

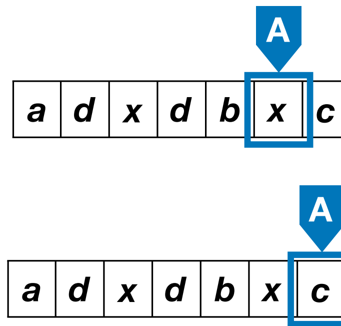




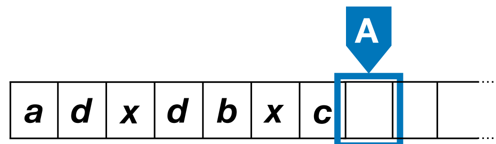




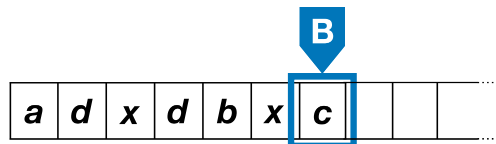




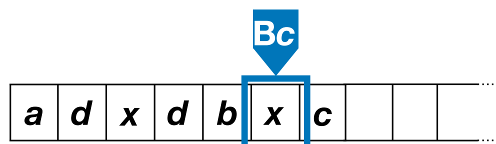
À ce point, la tête de lecture va vouloir aller à droite, mais le ruban semble s'arrêter. Pour éviter qu'une telle situation ne se produise, on étend le ruban à droite en le remplissant de cases blanches. La prochaine configuration est donc



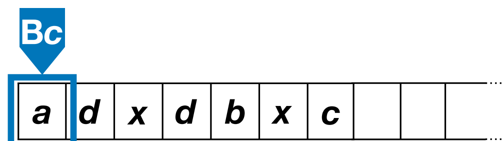
En suivant la deuxième ligne de la table de transition, on arrive donc dans la configuration



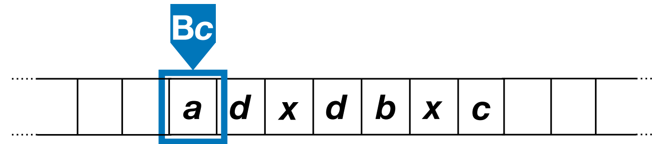
Les lignes suivantes de la table diffèrent suivant la lettre lue. Ici, le ruban contient la lettre c, on passe donc dans l'état Bc et on continue à gauche :



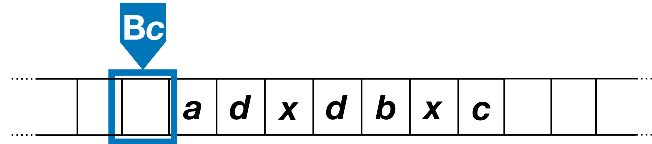
Désormais, on continue à aller à gauche tant qu'on lit une des lettres a,b,d, . . . ,z, c'est-à-dire toutes les lettres sauf c ou un blanc. Une autre transition permet cependant de traiter le cas d'une case blanche, mais la table de transitions ne permet pas de lire un c : si on se trouvait dans ce cas, comme pour un automate, la lecture serait bloquée et on rejetterait la séquence. En l'occurrence, nul autre c sur le ruban à gauche, donc la tête de lecture continue à gauche jusqu'à



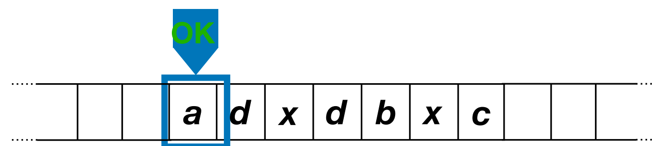
Comme avant, on est arrivé au bout de la partie écrite du ruban, mais on suppose qu'on est en fait sur un ruban infini (à gauche comme à droite), en ajoutant des cases blanches à gauche :



Après avoir lu la lettre a, on arrive donc dans la configuration



La dernière ligne de la table de transition permet alors d'arriver dans l'état OK, qui est l'état acceptant de cette machine :



Cette machine vérifie donc que la dernière lettre de la séquence est différente de toutes les autres lettres de la séquence : la table de transitions donnée plus haut n'est pas complète, puisqu'elle ne traite que le cas où la dernière lettre serait un c mais il est aisé de la compléter... Il est bien plus agréable de pouvoir faire un *aller-retour* sur le ruban, pour aller chercher le dernier symbole puis vérifier qu'il est différent de tous les autres. Cependant, il est possible d'accepter ce langage avec un automate fini, même si le nombre d'états nécessaires est bien plus important. En fait, c'est un résultat plus général :

Théorème 6. *Tout langage accepté par une machine lisant le ruban et pouvant changer de sens peut aussi être reconnu par une machine qui lit le ruban de gauche à droite uniquement (c'est-à-dire par un automate fini).*

Cette caractéristique supplémentaire ne permet donc pas d'ajouter de nouveaux langages. Il faut aller plus loin.

8.6 Machines de Turing

C'est Alan Turing (cf Figure 8.6) qui permit d'aller plus loin, en proposant un modèle de machine qui peut lire son ruban, se déplacer dans les deux sens, mais aussi *écrire* sur le ruban. [2]

L'objectif d'Alan Turing était justement de caractériser ce qui est *calculable*. À l'époque, les calculs, au sens mathématique du terme, était entièrement réalisés par des êtres humains, avec un papier et un crayon. À titre d'exemple, une photographie prise dans les années 1950 aux États-Unis, et reproduite en Figure 8.7, montre les calculateurs et calculatrices humain(e)s

2. Historiquement parlant, Alan Turing a décrit ses machines éponymes dans un article en 1937, bien avant que la notion d'automates finis n'apparaisse : ces derniers ont été introduits bien après, en 1956, par Stephen Cole Kleene (1909-1994), qui étudiait l'impact d'une diminution de la puissance de calcul des machines de Turing.

FIGURE 8.7 – Photographie du Comité consultatif national pour l'aéronautique avec les calculateurs et calculatrices humaines

du Comité consultatif national pour l'aéronautique : les calculs extrêmement compliqués de trajectoire des fusées et autres missiles étaient donc bien réalisés par des hommes et des femmes. Le film *Hidden Figures* (*Les figures de l'ombre* en français) reporte cette époque, en prenant comme héroïne Katherine Johnson, physicienne et mathématicienne, ayant calculé les trajectoires du programme Mercury et de la mission Apollo 11 vers la Lune, à la NASA.

Dans son article fondateur *On computable numbers*, Alan Turing écrit :

> « Normalement, on calcule en écrivant certains symboles sur le papier. [...] Je considère qu'on effectue le calcul sur un papier unidimensionnel, c'est-à-dire, sur un ruban divisé en carrés. »

L'objectif d'Alan Turing est donc de rendre systématique le calcul effectué par un homme ou une femme, sur un papier. Pour simplifier l'écriture, mais sans perte de généralité, il suppose que le papier est unidimensionnel. Il est évidemment possible d'encoder dans un ruban unidimensionnel les informations écrites sur un papier bidimensionnel. Alan Turing poursuit :

> « Je suppose aussi que le nombre de symboles qu'on peut écrire est fini. Si on permettait une infinité de symboles, il y aurait des symboles qui diffèreraient dans une mesure arbitrairement faible. [...] On peut toujours utiliser une séquence de symboles au lieu d'un symbole simple. »

Il justifie ainsi que l'alphabet utilisé pour remplir les cases du ruban n'a pas besoin d'être infini. Typiquement, on peut décomposer un nombre en une suite de chiffres, ou un mot en une suite de lettres.

> « La différence, de notre point de vue, entre les symboles simples et composites est qu'on ne peut pas observer les symboles composites en un coup d'œil, s'ils sont trop longs. Cela est conforme à l'expérience. On ne peut pas établir en un coup d'œil si 9999999999999999 et 999999999999999 sont égaux. »

Il observe que le champ visuel d'un calculateur humain est limité. Pour pouvoir lire une longue séquence de symboles, il doit déplacer son regard. Quitte à se déplacer un peu plus,

Alan Turing va considérer que le champ visuel du calculateur est limité à un unique symbole. À propos des états, il énonce :

> « Le comportement du calculateur à chaque moment est déterminé par le symbole qu'il observe et son *état d'esprit* à ce moment. »

Typiquement, lorsqu'un calculateur doit additionner 12932 et 19, il commence par mémoriser le symbole 2 (le chiffre des unités du premier nombre), puis le symbole 9 (le chiffre des unités du second nombre), il sait que leur somme fait 11 et qu'il doit donc écrire comme résultat un 1 et qu'il doit retenir une retenue égale à 1. Il mémorise ensuite les chiffres des dizaines, 3 et 1, qu'il additionne en ajoute la retenue : il peut donc écrire 5 comme résultat, et poursuivre son calcul.

> « On suppose également que le nombre d'états d'esprit qu'on doit prendre en compte est fini. Les raisons pour cela sont de la même nature que celles qui restreignent le nombre de symboles. »

De la même manière que des symboles trop proches visuellement sont indiscernables – justifiant qu'un nombre fini de symboles suffit – Alan Turing nous convainc que des états (d'esprit) trop proches ne peuvent être distingués – justifiant donc qu'un nombre fini d'états suffit. Il finit de nous convaincre en précisant qu'on peut utiliser le ruban comme brouillon, pour retenir plus de choses :

> « On peut éviter l'utilisation d'états d'esprit plus compliqués en écrivant plus de symboles sur le ruban. »

Et c'est donc là que la machine de Turing se distingue d'un automate fini, et même des machines vues précédemment qui peuvent se déplacer sur le ruban dans les deux sens : une machine de Turing doit avoir la possibilité de prendre des notes sur le ruban et donc doit pouvoir écrire dans une case du ruban, qu'elle soit déjà remplie ou qu'elle soit blanche pour l'instant.

Voici donc à quoi ressemble la table de transitions d'une machine de Turing :

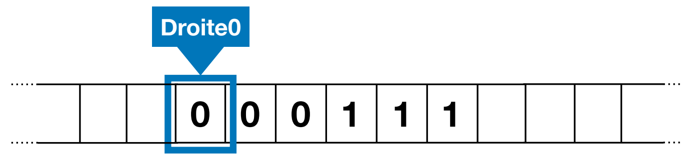
Etat	Symbole	Sens	Nouveau symbole	Nouvel état
Droite0	0	→	~~0~~	Droite1
Droite1	0	→	0	Droite1
Droite1	1	→	1	Droite1
Droite1		←		Gauche1
Gauche1	1	←	~~1~~	Gauche0
Gauche0	1	←	1	Gauche0
Gauche0	0	←	0	Gauche0
Gauche0	~~0~~	→	~~0~~	Droite0
Droite1	~~1~~	←	~~1~~	Gauche1
Droite0	~~1~~	→	~~1~~	OK

Cette nouvelle machine a cinq états : Droite0, Droite1, Gauche0, Gauche1 et un état « OK » pour signifier que la machine accepte. On supposera qu'elle démarre dans l'état Droite0 avec un

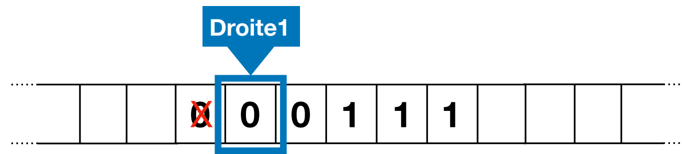
ruban contenant une séquence de 0 et de 1, ainsi que des cases blanches à droite et à gauche, de sorte que le ruban est en fait infini (même si à tout moment nous n'aurons besoin que d'une portion finie de celui-ci). La machine utilisera des symboles supplémentaires, en s'autorisant de barrer les lettres présentes sur le ruban. Les premières lignes de la table de transitions doivent donc se lire comme le pseudo-code suivant, utilisant une variable `état` se rappelant de l'état courant de la machine de Turing :

```
Si état = Droite0 et on lit 0 alors
    état := Droite1
    écrire 0̸ sur le ruban
    se déplacer à droite
Sinon Si état = Droite1 et on lit 0 ou 1 alors
    se déplacer à droite
Sinon Si état = Droite1 et on lit une case blanche alors
    état := Gauche1
    se déplacer à gauche
Sinon ...
```

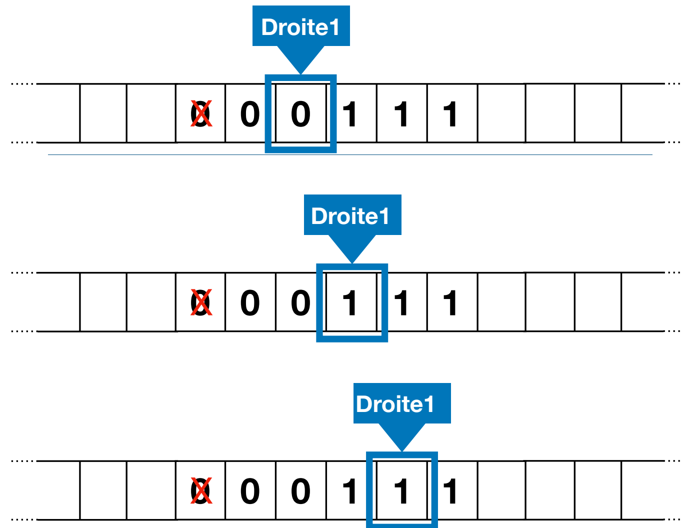
Exécutons la machine sur la séquence 000111 pour comprendre son fonctionnement. On commence donc à gauche de la séquence écrite sur le ruban, dans l'état initial Droite0.

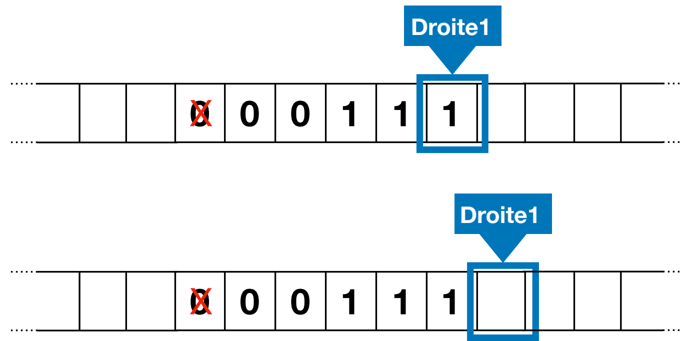


La première transition barre le 0, passe dans l'état Droite1 et se déplace vers la droite :

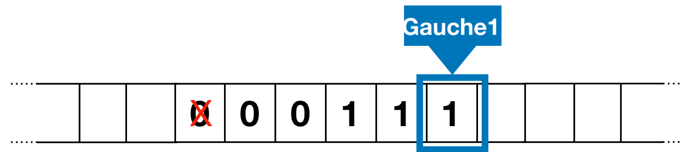


Les lignes 2 et 3 de la table de transition ne font que se déplacer vers la droite tant qu'on lit des 0 et des 1 :

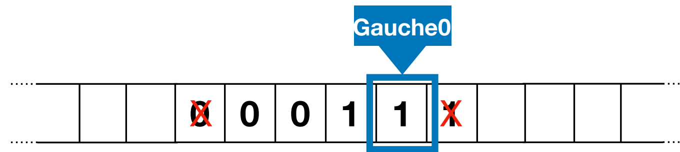




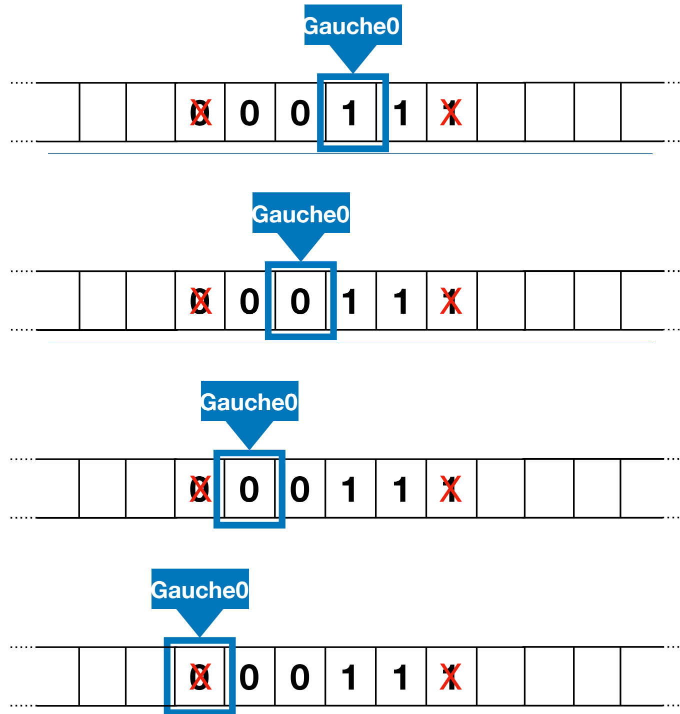
Lorsqu'on voit une case blanche, la quatrième ligne de la table de transitions demande à passer dans l'état Gauche1 en se déplaçant vers la gauche :



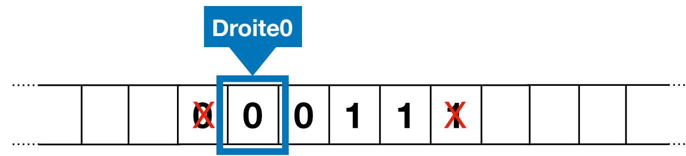
On applique la transition de la cinquième ligne de la table :



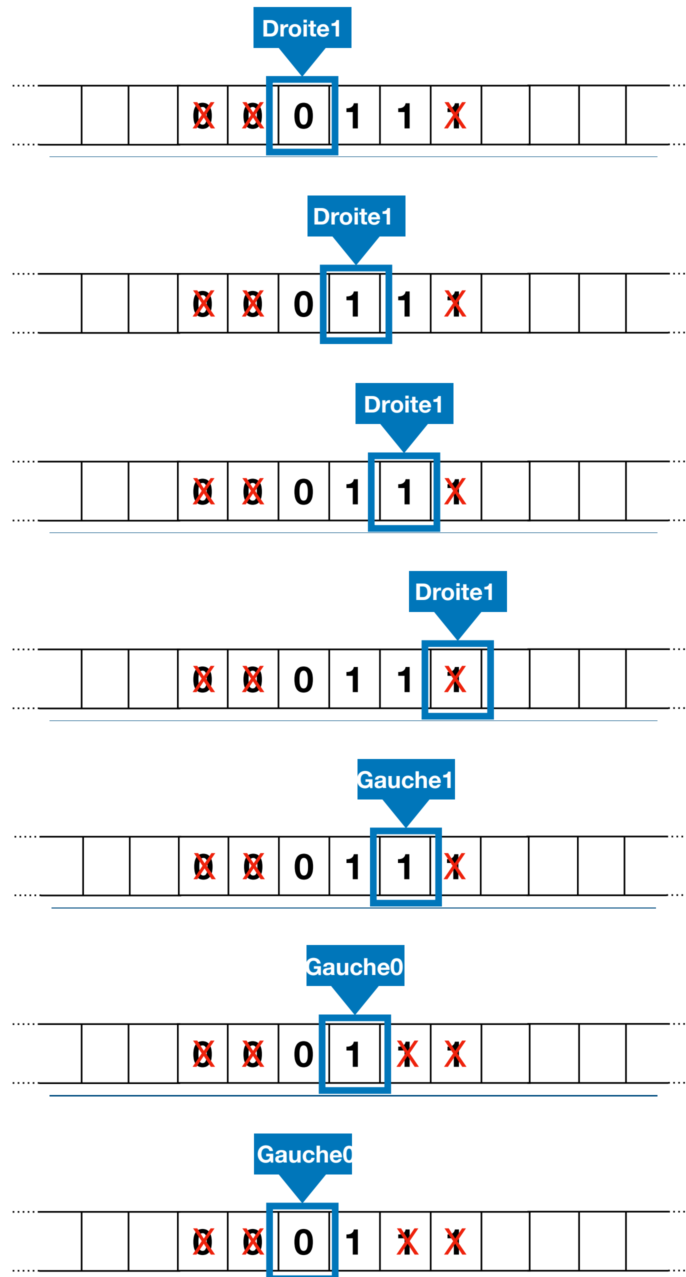
puis les transitions des lignes 6 et 7 restant dans l'état Gauche0 :

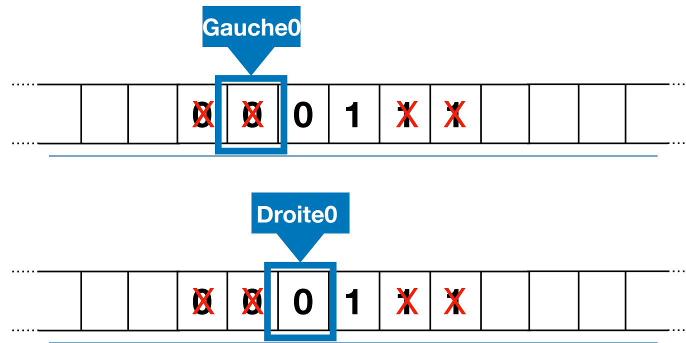


jusqu'à rencontrer un 0 barré, auquel cas la huitième ligne de la table prescrit de passer à nouveau dans l'état initial Droite0 en se déplaçant d'une case vers la droite :

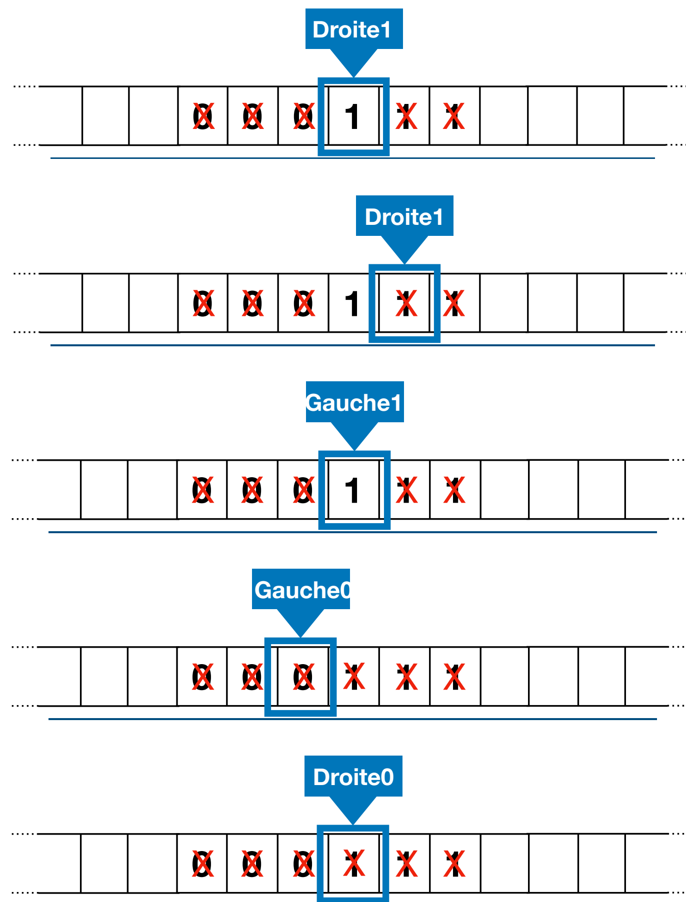


On se retrouve ainsi presque dans la configuration initiale et on effectue à nouveau un aller-retour consistant à barrer le 0 tout à gauche et aller barrer le 1 pas encore barré tout à droite du ruban :

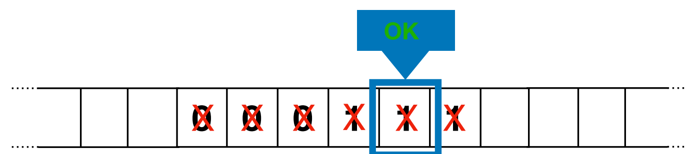




À nouveau, on se retrouve dans la même situation initiale, et on effectue un dernier aller-retour :



À ce moment, la situation est un peu différente, puisqu'on est dans l'état initial Droite0 et qu'on lit un 1 barré. Dans ce cas, la dernière ligne de la table de transition passe dans l'état « OK » :



La machine accepte donc la séquence 000111. On peut en fait montrer que cette machine accepte exactement toutes les séquences de la forme 0^n1^n (avec $n > 0$), c'est-à-dire toutes les séquences de 0 suivies de 1 avec autant de 0 que de 1 : nous avons vu au Théorème 5 que ce langage ne peut pas être accepté par un automate fini, ni même par une machine qui peut se déplacer dans les deux sens sans pouvoir écrire sur le ruban (par le Théorème 6). Voici donc un langage que seule une machine de Turing qui peut écrire sur son ruban peut accepter. Notons cependant qu'elle accepte ce langage d'une façon bien différente de la manière dont vous avez dû vous y prendre dans l'exercice 57...

Exercice 58

Exécuter cette machine de Turing, depuis l'état initial Droite0, sur les séquences 00011, 00111, 10 puis 0011 pour se convaincre sur quelques exemples qu'elle reconnaît bien le langage $\{0^n1^n \mid n \in \mathbb{N} \setminus \{0\}\}$.

8.7 Lien entre machines de Turing et pseudo-code

La thèse d'Alan Turing consiste à convaincre que la puissance d'une machine de Turing équivaut à celle d'un calculateur humain. Qu'en est-il des ordinateurs modernes, ou disons, pour simplifier, du pseudo-code que nous utilisons depuis le début de ce cours, ou du langage Python que vous utilisez dans le cours de Mise en œuvre informatique ?

On l'a vu, le pseudo-code qu'on utilise se base sur un certain nombre de structures de contrôle simples qu'on peut enchaîner, ainsi que de structures de données pour stocker des informations. Se convaincre qu'une machine de Turing a exactement autant de puissance que le pseudo-code que nous utilisons demande à exécuter deux raisonnements :

1. il faut se convaincre qu'on peut exécuter dans une machine de Turing n'importe quel algorithme écrit en pseudo-code ;
2. et il faut aussi se convaincre qu'on peut écrire un pseudo-code qui exécute n'importe quelle machine de Turing.

Tout d'abord, voyons pourquoi on peut exécuter n'importe quel pseudo-code dans une machine de Turing. Pour simplifier, il s'agit de savoir faire un peu d'arithmétique pour stocker des entiers, de savoir écrire des structures conditionnelles (`if ..: .. else: ..`), des itérations (`while...` par exemple), enchaîner de telles instructions et représenter des structures de données telles que des tableaux :

- L'enchaînement d'instructions est simple : on peut utiliser le mécanisme d'états d'une machine de Turing pour ordonner à la machine de passer dans un autre état une fois qu'on a terminé d'exécuter une certaine instruction.
- Stocker des entiers ou des tableaux sur le ruban d'une machine de Turing est facile en utilisant le codage binaire des entiers, puis en écrivant le contenu des cases du tableau sur le ruban, en les séparant par un caractère spécial (par exemple une virgule).
- Faire un test `if ..: .. else: ..` est aussi très simple dans une machine de Turing, puisque c'est exactement ce genre de test qu'elle effectue pour choisir la transition qu'elle doit exécuter à chaque instant.
- Itérer à l'aide d'une boucle`while...` est aussi aisé, puisqu'il suffit de revenir dans un état particulier pour recommencer une portion du code : c'est ce que nous avons fait dans l'exemple précédent où nous revenions dans l'état Droite0 pour recommencer un nouvel aller-retour sur le ruban.

— Finalement, il nous reste à savoir faire un peu d'arithmétique pour pouvoir, par exemple, incrémenter un compteur i, c'est-à-dire transformer une portion du ruban contenant le codage binaire d'un entier i, en cette même portion de ruban qui abritera le codage binaire de $i+1$. Vous devriez facilement être convaincu qu'une machine de Turing va pouvoir faire ça, de la même manière que nous l'avons fait en début de Chapitre 4. L'exercice suivant vous permet de le vérifier.

Exercice 59

Voici la table de transitions d'une machine de Turing ayant trois états (D, G et OK), dont l'état D est l'état initial, et OK l'état acceptant :

Etat	Symbole	Sens	Nouveau symbole	Nouvel état
D	0	→	0	D
D	1	→	1	D
D		←		G
G	1	←	0	G
G	0	←	1	OK
G		←	1	OK

Exécuter cette machine sur plusieurs séquences d'entrée composées de 0 et de 1 pour vous convaincre qu'elle code l'opération d'incrémentation.

Cela donne donc l'intuition (il existe évidemment des preuves plus formelles de ce résultat, bien au-delà de l'ambition de ce cours) que tout pseudo-code peut être exécuté par une machine de Turing. C'est finalement exactement la façon dont le code Python qu'on écrit s'exécute : il est transformé en une séquence d'opérations très simples qui peuvent être comprises par le processeur de l'ordinateur.

Réciproquement, il est aussi facile de se convaincre qu'on peut écrire un pseudo-code qui exécute n'importe quelle machine de Turing. Nous avons déjà vu précédemment que la table de transition d'une machine de Turing pouvait être encodée dans du pseudo-code utilisant une variable `état` pour maintenir l'état courant et un tableau pour stocker le contenu du ruban (avec une variable supplémentaire pour se souvenir de la position courante de la tête de lecture sur le ruban). Il ne reste plus qu'à observer que la répétition des étapes d'exécution d'une machine de Turing peut être obtenue par une boucle `while...` comme illustré en Figure 8.8.

Machines de Turing et pseudo-code ont donc le même pouvoir d'expression : c'est la raison pour laquelle on dit que les machines de Turing sont une version idéalisée des ordinateurs programmables qui seront inventés par la suite.

8.8 Peut-on tout calculer ?

On est donc en mesure de dire qu'un problème est *calculable* s'il l'est par une machine de Turing. Plus précisément, nous avons vu depuis le début des machines de Turing qui acceptent des langages, c'est-à-dire qui répondent « OK »/« pas OK » sur des séquences écrites au début sur leur ruban. On dit donc qu'un langage L (c'est-à-dire un ensemble de

Etat	Symbole	Sens	Nouveau symbole	Nouvel état
A	0	→	1	B
A	1	←	A	A
A		←	OK	B

```
while état != OK:
   if état == A and lettre == 0:
      état = B
      nouvelle_lettre = 1
      déplacer_droite()
   elif état == A and lettre == 1:
      état = A
      nouvelle_lettre = A
      déplacer_gauche()
   elif ...
```

FIGURE 8.8 – Exécuter une machine de Turing à l'aide d'un pseudo-code

séquences) sur un alphabet fini est *calculable* s'il existe une machine de Turing qui acceptent exactement l'ensemble des séquences de L. Une machine de Turing accepte une séquence dès lors qu'elle termine dans un état acceptant. Mais, elle a plusieurs manières de *ne pas accepter* une séquence : elle peut soit s'arrêter sur un blocage car elle n'a pas de transition permettant de continuer son exécution, ou bien elle peut ne jamais s'arrêter sans visiter un état d'acceptation. C'est le cas de la machine donnée dans l'exercice suivant.

Exercice 60

Voici la table de transitions d'une machine de Turing ayant deux états (D et G), dont l'état G est l'état initial :

Etat	Symbole	Sens	Nouveau symbole	Nouvel état
G		→	0	D
G	0	←	0	G
G	1	←	1	G
D		←	1	G
D	0	→	0	D
D	1	→	1	D

Exécuter cette machine à partir du ruban ne contenant que des cases blanches. Observer que cette machine ne s'arrête pas.

Maintenant qu'on a une définition claire de ce qui est calculable, nous pouvons enfin revenir à la question initiale : « peut-on tout calculer ? ». La réponse est négative et nous

allons même pouvoir exhiber un problème qu'on ne peut pas résoudre avec une machine de Turing (et donc un ordinateur). Il s'agit du *problème de l'arrêt* :

Il prend en entrée la table de transitions d'une machine de Turing (ou le pseudo-code qui peut permettre de l'exécuter) ainsi qu'une séquence finie écrite initialement sur son ruban de départ, et pose la question de savoir si la machine s'arrête (qu'elle accepte ou qu'elle n'accepte pas, par ailleurs) avec ce ruban de départ. On se pose donc la question de savoir si une machine de Turing peut accepter exactement les machines de Turing qui s'arrêtent sur un ruban donné : *on donne donc à manger* à une machine de Turing une autre machine de Turing (ou plus précisément sa table de transition), de la même manière qu'on peut donner à manger à une fonction en pseudo-code (ou en Python) une autre fonction...

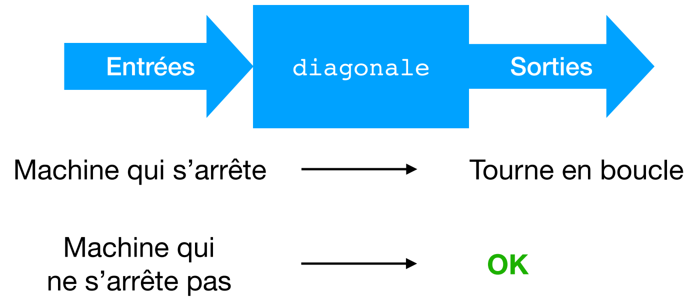
Théorème 7. *Il n'existe pas de machine de Turing permettant de résoudre le problème de l'arrêt.*

Avec les maigres outils dont nous disposons, nous sommes déjà en mesure de prouver ce résultat (contrairement à la preuve de l'impossibilité de la quadrature du cercle qui fait appel à des structures mathématiques assez complexes). Raisonnons par l'absurde en supposant qu'il existe une machine de Turing qui résolve le problème de l'arrêt : cette machine prend donc en entrée la table de transitions d'une machine M et un ruban d'entrée, et décide si, oui ou non, M s'arrête.

Construisons alors une nouvelle machine de Turing à partir de celle-ci. On la définit plutôt à l'aide d'une fonction Python (mais on sait que cela est équivalent d'après la section précédente) :

```
def diagonale(M):
    if M s'arrête sur un ruban de départ contenant la table de M:
        x = 0
        while x >= 0 faire
            x = x + 1
    else:
        return OK
```

Le test sur l'arrêt de M (en rouge) est réalisé par la machine de Turing dont on a supposé l'existence préalablement. Cette nouvelle machine peut être schématisée ainsi :



Notons en effet que si la machine M donnée en entrée s'arrête sur un ruban de départ qui contient sa propre table de transitions, alors la machine `diagonale` ne s'arrête pas puisqu'on entre dans une boucle infinie. Au contraire, si on déduit que la machine M ne s'arrête pas, alors la machine `diagonale` s'arrête et accepte.

Parvenons désormais à une contradiction en *donnant à manger* à la machine `diagonale` la table de transitions de la machine `diagonale` elle-même. Que donne alors l'exécution de `diagonale`(`diagonale`) ? Deux cas sont possibles :

- si `diagonale` s'arrête lorsqu'on lui donne en entrée la table de transitions de la machine `diagonale`, alors on entre dans la boucle infinie donc on ne s'arrête pas ;
- au contraire, si `diagonale` ne s'arrête pas lorsqu'on lui donne en entrée la table de transitions de la machine `diagonale`, alors on retourne OK donc on s'arrête en acceptant.

Ainsi, `diagonale`(`diagonale`) s'arrête si et seulement si `diagonale`(`diagonale`) ne s'arrête pas ! Ce qui est évidemment une contradiction à l'existence d'une machine de Turing résolvant le problème de l'arrêt. On a ainsi prouvé par l'absurde qu'une telle machine ne pouvait pas exister.

Un tel problème qu'aucune machine de Turing ne peut résoudre est souvent dit *indécidable*. Le fait que le problème de l'arrêt soit indécidable est une très mauvaise nouvelle pour la science informatique : cela veut dire qu'on ne peut pas écrire d'algorithme qui conclut si un autre algorithme termine ou non. Or, nous avons vu qu'il est crucial de savoir si un algorithme se termine (et même ensuite de déterminer sa complexité) : aucun espoir qu'un ordinateur puisse jamais répondre à coup sûr à cette question cependant !

D'autres problèmes très importants sont aussi indécidables :

- Est-ce qu'une machine de Turing donnée atteint un certain état (par exemple, un état acceptant) ? Ce problème, et en fait tout problème non trivial sur les machines de Turing (c'est le théorème de Rice, énoncé par Henry Gordon Rice en 1951), est indécidable, ce qui se montre par réduction à partir du problème de l'arrêt.
- Est-ce qu'une proposition mathématique donnée est un théorème (c'est-à-dire, peut-on en trouver une preuve) ? Une proposition mathématique est une formule avec des quantificateurs et des opérateurs logiques : par exemple, la formule

$$\forall x \in \mathbf{R} \quad \forall y \in \mathbf{R} \quad \big(x \neq y \Longrightarrow \exists z \in \mathbf{R} \quad x < z < y\big)$$

  est vraie, puisqu'entre deux réels différents, on peut toujours trouver un autre réel (on dit que l'ensemble des réels est *dense*). La recherche d'une preuve pour une proposition mathématique est *le problème de la décision* énoncé par David Hilbert (*Entsheidungsproblem* en allemand) qui motiva initialement Alan Turing lorsqu'il mit au point ses machines éponymes dans son article de 1936 dont le titre complet est *On Computable Numbers, with an Application to the Entscheidungsproblem.*

— David Hilbert (1862-1943, né à Königsberg, ville dont nous avons déjà évoqué l'existence avec le problème des graphes eulériens) a listé 23 problèmes mathématiques en 1900 dont certains sont encore non résolus à ce jour. Le 10ème problème était de trouver un algorithme déterminant si une équation diophantienne a des solutions. Une équation diophantienne est une équation polynomiale à coefficients entiers, à une ou plusieurs inconnues, dont on cherche des solutions entières. Par exemple, voici un système d'équations diophantiennes :

$$\begin{cases} x^3y - 3y^2z = 20 \\ -7y^4 + 4yz^3 = 0 \end{cases}$$

C'est en 1970 que Iouri Matiassevitch (né en 1947) démontra l'indécidabilité du 10ème problème de Hilbert.

Chapitre 9

Conclusion

Ce cours vous a proposé de découvrir la science informatique, trouvant sa source bien avant même la mise au point d'ordinateurs tels que nous les connaissons aujourd'hui. La science informatique, c'est la science du calcul, c'est-à-dire la science s'interrogeant sur ce qu'est le calcul, ce qu'on peut ou ne peut pas calculer, et, lorsqu'on peut le calculer, comment et à quel coût.

Cela nous a amené à découvrir les algorithmes et la manière de les décrire avec un pseudo-code (ou un langage de programmation tel que Python). Nous avons comparé ces algorithmes en calculant leur complexité, préférant ainsi la recherche dichotomique à la recherche séquentielle lorsque le tableau est trié, puis le tri par fusion au tri par insertion. Nous avons étudié des algorithmes numériques classiques, avec des applications à la cryptographie.

Pour résoudre davantage de problèmes, nous avons introduit des structures de données plus complexes que les tableaux, à savoir les graphes et les arbres, omniprésents en informatique et dans d'autres disciplines.

Finalement, dans notre recherche de ce qui est calculable (et donc de ce qui ne l'est pas), nous avons découvert les automates finis (dont un cas particulier, les arbres de décision, sont très utiles en pratique), puis les machines de Turing, dont les automates ne sont qu'un cas particulier. Cela nous a permis de mettre au jour un problème indécidable, c'est-à-dire dont on sait qu'aucun algorithme ne viendra jamais à bout.

Ce n'est évidemment pas la fin de la science informatique, une science qui évolue constamment. En parallèle des travaux d'Alan Turing pour caractériser la notion de calcul, d'autres informaticiens se sont intéressés à la capacité d'auto-reproduction des machines. Ainsi, John von Neumann (1903-1957) a mis au point les automates cellulaires, dont vous connaissez peut-être l'exemple du *jeu de la vie*. Ces automates sont des machines très simples qu'on dispose en ligne ou sur une grille bidimensionnelle, et qui interagissent alors entre elles pour imiter les interactions naturelles. Von Neumann a ainsi montré que de telles machines très simples avaient des capacités d'auto-réplication. Plus tard, Stephen Wolfram (né en 1959) a étudié en détails ces automates cellulaires pour montrer qu'il en existe de très complexes, chaotiques, à tel point que certains automates cellulaires montrent les mêmes capacités qu'une machine de Turing : on retrouve ainsi sous une forme très différente la même notion de calculabilité. C'est la force de la thèse d'Alan Turing !

Alan Turing s'est aussi distingué pour ses travaux en intelligence artificielle, par exemple en mettant au point le test de Turing pour distinguer une intelligence artificielle d'un humain. Ce domaine est désormais extrêmement actif puisque, combiné avec la puissance de calcul

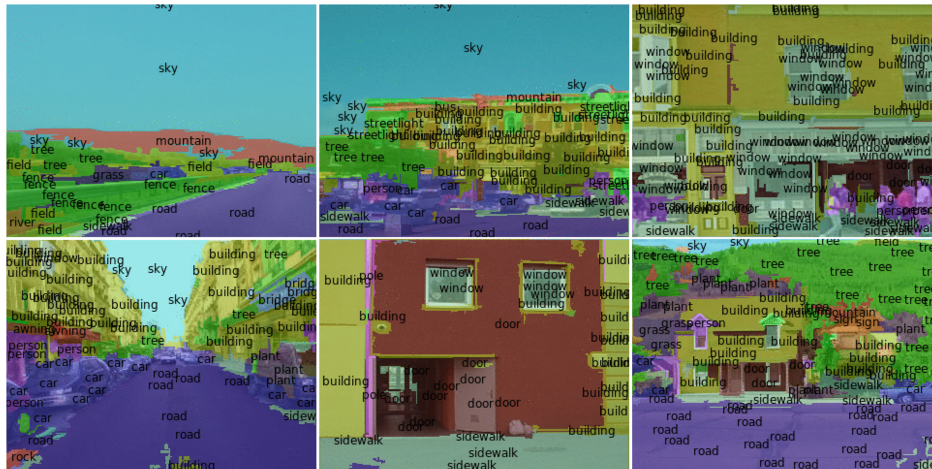


FIGURE 9.1 – Segmentation et étiquetage d'images

des ordinateurs actuels, les méthodes d'apprentissage statistique ont permis de résoudre des problèmes jusqu'alors hors de portée des ordinateurs. De tels algorithmes, basés sur l'étude statistique de masses de données, permettent par exemple de segmenter et étiqueter des images ou des vidéos avec les objets qu'elles contiennent (cf Figure 9.1), de reconnaitre des écritures manuscrites ou la voix dans les assistants vocaux, de prédire l'évolution de la bourse, d'explorer Mars, de jouer aux échecs ou au Go, ou de conduire des voitures autonomes. Différentes méthodes d'intelligence artificielle et d'apprentissage statistique permettent de résoudre ces problèmes : la dernière en date, ayant montré le plus de résultats époustouflants récemment, consiste en l'utilisation de réseaux de neurones artificiels, imitant la structure cérébrale d'un cerveau humain pour décomposer un problème complexe en sous-problèmes de plus en plus spécialisés. L'application de telles méthodes statistiques pour des applications telles que la conduite de voitures autonomes pose des questions éthiques puisque des vies humaines sont en jeu : se satisfera-t-on d'intelligences artificielles mises au point par des méthodes d'apprentissage statistique, sans que quiconque ne comprenne vraiment comment ces intelligences prennent leurs décisions ? En cas d'accident, qui sera déclaré responsable ? Et accepterons-nous que nos données personnelles soient utilisées par des compagnies privées afin d'alimenter les algorithmes d'apprentissage statistique ? Voici autant de questions qui sont les challenges d'aujourd'hui et de demain dans ce domaine.