Design and Analysis of Algorithm

Pierre-Alain Fouque

Pierre-Alain.Fouque@univ-rennes1.fr

Université de Rennes 1

Version novembre 2020

Introduction

Bien que le vocable *informatique* ait été inventé dans les années soixante, on peut prétendre paradoxalement que l'informatique est à la fois une technique très récente et une science très ancienne.

C'est bien sûr une technique fort jeune : les premiers ordinateurs dignes de ce nom ont vu le jour à la fin de la deuxième guerre mondiale. Et c'est une technique qui consiste souvent en la mise en œuvre d'outils malheureusement tout aussi imparfaits qu'éphémères : langages de programmation, systèmes d'exploitation, méthodes de conduite de projets logiciels, pour ne citer que ceux-ci.

Par contre, en tant que science, l'informatique peut être considérée comme très ancienne, car elle ne saurait être dissociée du développement des mathématiques, qu'elle a accompagné depuis les premiers balbutiements jusqu'à l'époque contemporaine. En effet, à l'aube des temps historiques et des civilisations naissantes, les hommes ont appris à compter ; pour ce faire, ils ont inventé (ou découvert) les nombres et leur codage, puis les opérations arithmétiques élémentaires telles que l'addition ou la multiplication. Cet apport de génie a seul permis le réel décollage de l'activité économique (et donc culturelle), par le biais de l'élevage et de l'emploi de la monnaie — il fallait bien compter les troupeaux et vendre ses productions. Or, codage des nombres et opérations arithmétiques de base forment justement le cœur de nos ordinateurs.

Ce cours constituent précisément une introduction à l'aspect scientifique de l'informatique. Bien que cet aspect recouvre plusieurs théories distinctes (quoique interconnectées), telles que logique et calculabilité, conception et sémantique des langages de programmation, compilation, validation des programmes, etc., sa partie la plus essentielle est l'*algorithmique*, dont nous donnons tout d'abord la définition extraite du petit Larousse :

> **ALGORITHME** n.m. (ar. *al-khārezmi*, surnom d'un mathématicien arabe). MATH. et INFORM. Suite finie d'opérations élémentaires constituant un schéma de calcul ou de résolution d'un problème.
>
> **ALGORITHMIQUE** adj. De la nature de l'algorithme. • n.f. Science des algorithmes, utilisés notamment en informatique.

On étudiera ici principalement les *algorithmes* et *structures de données*, i.e. la structuration de l'information en machine, les plus communément employés en informatique, mais dont certains sont utiles dans d'autres domaines des sciences de l'ingénieur.

Le but d'un tel cours est triple :

— d'abord, faire comprendre que l'informatique ne se limite pas à la simple écriture de programmes, mais qu'elle nécessite une réflexion mathématique

approfondie afin de déterminer le bon algorithme et la bonne structure de données permettant de résoudre le problème auquel on est confronté ; en particulier, on songera à toujours se faire une idée de la *complexité* spatiale et temporelle d'un algorithme : informellement, il s'agit de l'espace mémoire et du temps nécessaire à son exécution ;
- ensuite, fournir à l'ingénieur une connaissance suffisante des principaux types d'algorithmes et de structures de données pour qu'il évite de "réinventer la roue", et puisse se référer aisément à ce qui existe déjà, c'est-à-dire aux différentes "Bibles" de l'algorithmique : citons en particulier la fameuse trilogie *The Art of Computer Programming* [5, 6, 7] de Donald E. Knuth, la première étude mathématique moderne sur le sujet, et qui reste une référence des plus utiles ; dans la même veine, on peut également mentionner *The Design and Analysis of Computer Algorithms* [1] de Aho, Hopcroft et Ullman ; *Introduction à l'algorithmique* [3] de Cormen, Leiserson et Rivest est peut-être le nec plus ultra actuel ; on pourra encore consulter les livres de Gonnet et Baeza-Yates ou de Sedgewick [4, 9]. En ce qui concerne l'algorithmique numérique, *Numerical Recipes* (en C, FORTRAN ou PASCAL) [8] est la référence fondamentale ;
- enfin, indiquer les limitations de l'usage de l'ordinateur : tous les problèmes ne sont pas solubles par ordinateur, ou bien à cause d'une complexité spatiale ou temporelle rédhibitoire des algorithmes qui leur correspondent, ou bien même parce qu'ils sont *indécidables*, c'est-à-dire réellement trop compliqués pour être résolus par un procédé automatique.

Les algorithmes décrits plus loin seront présentés de manière informelle en français, ou dans un pseudo-code proche de la syntaxe du C et facilement compréhensible, ou encore sous la forme d'un programme C. Outre les calculs de complexité, on insistera au travers de nombreux exercices sur la nécessité de programmer les algorithmes, ce qui est une bonne méthode pour s'assurer les avoir bien compris.

Table des matières

Introduction **3**

1 Complexité **7**
1.1 Définitions . . . 7
1.1.1 Comparaison asymptotique de fonctions réelles . . . 7
1.1.2 Complexité d'un algorithme . . . 8
1.2 Exemples d'algorithmes de complexité différente . . . 9
1.2.1 Suite de Fibonacci . . . 10
1.2.2 Problème du tri . . . 13
1.3 Tri par insertion . . . 14
1.3.1 Description de l'algorithme . . . 14
1.3.2 Implémentation en C . . . 15
1.3.3 Complexité . . . 16
1.4 Qu'est ce qu'un algorithme efficace ? . . . 17
1.5 Exercices . . . 19

2 Récursivité **21**
2.1 Fonctions numériques récursives . . . 21
2.1.1 Suite de Fibonacci . . . 23
2.1.2 Fonction d'Ackermann . . . 24
2.2 Procédures récursives . . . 24
2.2.1 Tri fusion . . . 25
2.2.2 Tri rapide . . . 26
2.2.3 Complexité des algorithmes de tri . . . 29
2.2.4 Tours de Hanoï . . . 29
2.3 Complément : définitions inductives . . . 30
2.4 Exercices . . . 34

3 Structures de données élémentaires **35**
3.1 Tableaux . . . 35
3.2 Listes chaînées . . . 37
3.2.1 Définition . . . 37
3.2.2 implémentation en C . . . 38
3.2.3 Opérations courantes sur les listes . . . 40
3.3 Piles et files . . . 43
3.4 Application : évaluation des expressions arithmétiques préfixées . . . 44
3.5 Exercices . . . 48

4 Recherche en table **49**
4.1 Adressage direct 49
4.2 Recherche séquentielle 50
4.3 Recherche dichotomique 51
4.4 Tables de hachage 53
4.5 Récapitulatif 55
4.6 Exercices 55

5 Arbres **57**
5.1 Terminologie 57
5.2 Parcours d'arbre 61
5.3 Arbres binaires de recherche 63
5.3.1 Définition 63
5.3.2 Opérations sur les arbres binaires de recherche 64
5.4 Files de priorité et tas 66
5.4.1 Définition 67
5.4.2 Opérations sur les tas 67
5.4.3 Implémentation 70
5.5 Arbres équilibrés 70
5.6 Exercices 71

6 Graphes **73**
6.1 Définitions 73
6.2 Représentation des graphes 74
6.2.1 Matrice d'adjacence 74
6.2.2 Listes de successeurs 75
6.3 Parcours de graphe 76
6.3.1 Parcours en largeur d'abord 76
6.3.2 Parcours en profondeur d'abord 77
6.4 Applications des parcours de graphe 78
6.4.1 Tri topologique 78
6.4.2 Calcul des composantes fortement connexes 78
6.5 Recherche de chemins les plus courts dans un graphe 81
6.5.1 Calcul des chemins à partir de la matrice d'adjacence . . . 81
6.5.2 Algorithme de Aho-Hopcroft-Ullman 83
6.6 Exercices 85

7 Analyse syntaxique **87**
7.1 Définitions 87
7.1.1 Algorithme naïf de pattern-matching 88
7.1.2 Algorithme de Rabin-Karp 88
7.2 Pattern-matching à base d'automates finis 89
7.2.1 Définition des automates finis déterministes 89
7.2.2 Automate de recherche de motif 91
7.2.3 Calcul de l'automate de recherche 91
7.3 Exercices 92

8 Conclusion : les limites de l'algorithmique **93**

Chapitre 1

Complexité

La principale motivation de l'algorithmique est de résoudre des problème informatiques, i.e. de traitement automatique de l'information, de la manière la plus efficace possible. Or, même si intuitivement on croit comprendre ce qu'*efficace* semble signifier, la réalité est loin d'être aussi simple. La théorie de la complexité s'attache à la formalisation du concept d'efficacité d'un algorithme. Dans ce premier chapitre, nous nous efforcerons de saisir ce que l'on entend par complexité d'un algorithme ainsi que les conséquences pratiques qui découlent de conclusions d'ordre théorique.

1.1 Définitions

1.1.1 Comparaison asymptotique de fonctions réelles

Commençons par un rappel de quelques notations : pour deux fonctions réelles $f(n)$ et $g(n)$, on écrira :

$$f(n) = O(g(n)) \tag{1.1}$$

si et seulement s'il existe deux constantes strictement positives n_0 et c telles que :

$$0 \leq f(n) \leq c \times g(n)$$

pour tout n supérieur à n_0. On note encore :

$$f(n) = \Omega(g(n)) \tag{1.2}$$

quand $g(n) = O(f(n))$, et :

$$f(n) = \Theta(g(n)) \tag{1.3}$$

quand on a à la fois les propriétés (1.1) et (1.2), i.e.

$$\exists (c, c') \in \left(\mathbb{R}_+^*\right)^2 \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 \quad c \times g(n) \leq f(n) \leq c' \times g(n)$$

Notons que la formule (1.3) ne signifie pas que $f(n)$ est équivalente à $g(n)$ (noté $f(n) \sim g(n)$), qui se définit comme :

$$\lim_{n \to \infty} \frac{f(n) - g(n)}{g(n)} = 0$$

Cependant, si $f(n) \sim g(n)$ on a $f(n) = \Theta(g(n))$. Enfin, il est clair que $f(n) = \Theta(g(n))$ si et seulement si $g(n) = \Theta(f(n))$. Intuitivement, la notation Θ revient à "oublier" le coefficient multiplicatif constant de $g(n)$.

Voici quelques exemples de comparaisons de fonctions :
— $n^2 + 3n + 1 = \Theta(n^2) = \Theta(50n^2)$
— $n/\ln(n) = O(n)$
— $50n^{10} = O\left(n^{10{,}01}\right)$
— $2^n = O(\exp(n))$
— $\exp(n) = O(n!)$
— $n/\ln(n) = \Omega(\sqrt{n})$

On peut ainsi établir une hiérarchie (non exhaustive) entre les fonctions :

$$\boxed{\log(n) \ll \sqrt{n} \ll n \ll n\log(n) \ll n^2 \ll n^3 \ll 2^n \ll \exp(n) \ll n!}$$

Voici une comparaison numérique de ces fonctions. On notera en particulier la croissance très rapide des fonctions exponentielles (2^n, $\exp(n)$ et $n!$). A titre de comparaison, on estime le nombre de particules dans l'univers à 10^{80} !

$\log(n)$	3.3	6.6	10
$\sqrt{n}$	3.1	10	32
n	10	100	1000
$n\log(n)$	33	664	10^4
n^2	100	10^4	10^6
n^3	10^3	10^6	10^9
2^n	10^3	10^{30}	10^{300}
$\exp(n)$	2×10^4	10^{43}	10^{434}
$n!$	3.6×10^6	10^{158}	10^{2568}

1.1.2 Complexité d'un algorithme

Afin de fournir une mesure intrinsèque de l'efficacité d'un algorithme, indépendante de l'environnement d'exécution, nous allons en définir la complexité. Pour cela, on considère qu'un algorithme reçoit des données initiales qu'il manipule afin de fournir un résultat. Parmi les opérations effectuées, on en considère les plus importantes, celles qui sont représentatives de l'effort de calcul à produire. Informellement, la complexité d'un algorithme est une fonction exprimant le nombre d'opérations nécessaires en fonction de la taille des données à traiter.

Une telle définition pose plus de problèmes qu'elle n'en résout. Il convient en effet tout d'abord de **définir ce que l'on entend par taille des données**. De manière générale, la taille d'un objet doit refléter l'espace nécessaire à son codage dans la mémoire d'un ordinateur. Ainsi, un entier n se code en binaire avec $\lfloor \log_2(n) \rfloor + 1$ bits ; une bonne mesure de la taille d'un entier est par conséquent son logarithme en base 2. Bien entendu, on pourrait considérer d'autres codages, par exemple en unaire, pour lesquels la quantité de mémoire nécessaire est bien supérieure mais de telles approches ne sont pas naturelles. Comme autre exemple, si l'on a un algorithme recevant un tableau d'entiers comme donnée initiale, le

nombre d'éléments de ce tableau en mesure correctement la taille. Bien entendu, ceci sous-entend que les entiers occupent un espace de taille fixe en mémoire, par exemple 4 octets.

Le second point important à résoudre est le **choix des opérations élémentaires que l'on veut comptabiliser**. Si l'on souhaite obtenir une estimation de complexité représentative des temps de calcul effectifs, il faut choisir les opérations qui seront le plus souvent réalisées et qui nécessiteront en pratique le plus de temps de calcul. Par exemple, pour les algorithmes de tri, on mesure traditionnellement le nombre de comparaison entre éléments du tableau à trier. Tout comme pour la taille des données initiales, on va donner une mesure approximative, simplifiée, à l'aide des notations O, Ω et Θ. En effet, seul le comportement **asymptotique** des algorithmes nous intéresse. Par exemple, si nous comptabilisons $3n^2 - 8n + 4$ opérations pour traiter des données de taille n, nous dirons simplement que l'algorithme est en $\Theta(n^2)$; le terme en n, et a fortiori le terme constant, deviennent en effet négligeables devant le terme quadratique lorsque n devient grand. De plus, le coefficient du terme dominant (3 dans notre exemple) est de peu d'intérêt en pratique. Considérons par exemple un algorithme effectuant des additions et des multiplications ; on admet en général qu'une multiplication est plus complexe et prend plus de temps qu'une addition mais comment chiffrer cette différence ? Doit on considérer que l'une est 50 fois plus lente que l'autre comme c'était le cas sur les premiers microprocesseurs ou au contraire dire que les temps de calcul sont comparables (comme sur certains processeurs modernes) ? Cet exemple montre que les coefficients numériques apportent peu d'information en pratique. Par contre, une complexité en $\Theta(n^2)$ nous apprend par exemple que quelque soit l'ordinateur employé, quelque soit le langage utilisé, le temps de calcul sera en gros multiplié par 4 si l'on double la taille des données initiales.

Enfin, si l'on considère l'ensembles des données initiales de taille fixée, il est clair que le nombre d'opérations peut fortement varier d'une donnée à l'autre. On imagine sans peine qu'avec certains algorithmes un tableau déjà trié sera bien plus facile à ordonner qu'un tableau aléatoire. Suivant le cas auquel on s'intéresse, on parle de *complexité dans le cas le pire* lorsque l'on mesure la complexité maximale pour une taille fixée des données initiales, on parle de *complexité en moyenne* lorsque l'on mesure le nombre moyen d'opérations nécessaires et enfin de *complexité dans le meilleur cas* pour parler du cas le plus favorable. Nous ne considérerons pas ce dernier cas dans la suite de ce cours car il n'a pas beaucoup d'intérêt en pratique.

Nous avons jusqu'à maintenant uniquement abordé la complexité dite *temporelle*, i.e. une mesure du temps de calcul nécessaire pour résoudre un problème. Dans certains cas, il est également intéressant d'étudier la complexité *spatiale* d'un algorithme, i.e. une mesure de la quantité de mémoire nécessaire à son exécution. Toutes les remarques faites précédemment s'applique donc de manière semblable.

1.2 Exemples d'algorithmes de complexité différente

Afin de se faire une idée plus concrète de ce qu'est la complexité d'un algorithme, nous allons aborder différents problèmes et proposer divers algorithmes de complexités temporelle et spatiale différentes.

1.2.1 Suite de Fibonacci

La suite $(F_n)_{n\geq 0}$ des nombres de Fibonacci est définie par la récurrence suivante :

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

Le problème que l'on se pose est de calculer le n-ième nombre F_n de Fibonacci (en fait, nous ne calculerons pas ici les nombres de Fibonacci dont la taille croît rapidement mais plutôt leur valeur modulo la représentation des entiers en machine). Un premier algorithme très simple consiste à programmer récursivement une fonction qui calcule F_n en fonction de F_{n-1} et F_{n-2} :

```
int fibo1(int n)
{
  if (n<=1) return 1;
  else return fibo1(n-1)+fibo1(n-2);
}
```

Afin d'estimer la complexité d'un tel calcul, évaluons le nombre d'appels C_n à la fonction `fibo1` effectués pour calculer F_n. On a $C_0 = C_1 = 1$ et $C_n = 1 + C_{n-1} + C_{n-2}$ pour tout $n \geq 2$. Une telle récurrence peut se résoudre en posant $D_n = (C_n + 1)/2$; la suite $\{D_n\}_{n\in\mathbb{N}}$ vérifie $D_0 = D_1 = 1$ et $D_n = D_{n-1} + D_{n-2}$, i.e. exactement les mêmes relations que la suite de Fibonacci. Par conséquent, $C_n = 2F_n - 1$.

On peut d'autre part résoudre la récurrence linéaire définissant les nombres de Fibonacci de manière exacte ; en utilisant une technique d'algèbre classique, on résout tout d'abord l'équation $x^2 = x + 1$ issue de la formule de récurrence. Soit $x_+ = \frac{1+\sqrt{5}}{2}$ et $x_- = \frac{1-\sqrt{5}}{2}$ les deux racines. On sait alors que F_n s'écrit, en fonction de n, sous la forme $F_n = \lambda {x_+}^n + \mu {x_-}^n$, les valeurs des paramètres λ et μ étant déterminés au moyen des valeurs initiales F_0 et F_1 de la suite. On obtient finalement

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right)$$

On en déduit que la complexité du calcul de la suite de Fibonacci par la fonction `fibo1` est $\Theta(\omega^n)$ où $\omega = (1+\sqrt{5})/2$ est le nombre d'or. Une telle fonction croît extrêmement vite ; l'algorithme ne pourra donc être utilisé que pour de très petites valeurs de n.

Une observation attentive de l'algorithme précédent montre que le temps de calcul est très important car les valeurs F_k pour $k < n$ sont très souvent recalculées. Une idée simple consiste alors à stocker les F_k successifs dans un tableau. On obtient la fonction suivante :

```
int fibo2(int n)
{
  int *t;
  int i;
  int resultat;
```

```
  t=(int *)malloc((n+1)*sizeof(int));
  t[0]=1; t[1]=1;
  for(i=2;i<=n;i++) t[i]=t[i-1]+t[i-2];
  resultat=t[n];
  free(t);
  return resultat;
}
```

Une telle fonction a une complexité linéaire en n car on ne calcule qu'une seule fois les valeurs F_k pour $k < n$. Par contre, la complexité spatiale, i.e. la quantité de mémoire nécessaire est elle-aussi linéaire en n, ce qui peut être rapidement un facteur limitant en pratique. De plus, il est inutile pour calculer F_n de stocker toutes les valeurs intermédiaires ; seules deux valeurs consécutives de la suite doivent être simultanément mémorisées. On obtient la fonction suivante, de complexité toujours linéaire en terme de temps de calcul mais cette fois-ci ne nécessitant plus qu'une quantité de mémoire constante, indépendante de n, ce que l'on note habituellement $O(1)$.

```
int fibo3(int n)
{
  int f1,f2,t,i;

  f1=1; f2=1;
  for(i=2;i<=n;i++)
    {
      t=f2;
      f2=f1+f2;
      f1=t;
    }
  return f2;
}
```

Peut-on faire mieux ? Ceci peut sembler surprenant mais la réponse est affirmative. On peut en effet remarquer l'égalité suivante :

$$\begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix}$$

ce qui s'écrit encore

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

Par conséquent, le calcul de F_n se ramène à la mise à la puissance $n-1$ de la matrice $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$. Or, tout comme pour calculer $a \times b$ le meilleur algorithme ne consiste pas à effectuer $a + a + a + \ldots a$, on dispose d'algorithmes pour calculer a^b de manière beaucoup plus intelligente que $a \times a \times \ldots a$. L'idée consiste à généraliser le cas des puissances de la forme 2^k : $a^{2^k} = \left(\ldots\left((a^2)\right)^2\right)^2\ldots\right)^2$. Par exemple, $a^{22} = (((a^2)^2 \times a)^2 \times a)^2$ (pour plus de détails, voir l'exercice 1.4).

Voici un exemple d'implémentation ne nécessitant de $\Theta(\log(n))$ opérations :

```
int fibo4(int n)
{
  int M00,M01,M10,M11;
  int P00,P01;
  int a;
  M00=1; M01=1; M10=1; M11=0;
  P00=1; P01=0;
  while (n>0)
    {
      if (n%2!=0)
        {
          a=P00*M00+P01*M10;
          P01=P00*M01+P01*M11;
          P00=a;
        }
      a=M01*M01;
      M01=M01*(M00+M11);
      M10=M01;
      M00=M00*M00+a;
      M11=M11*M11+a;
      n=n/2;
    }
  return P00;
}
```

En termes d'efficacité pratique, on peut mesurer le temps de calcul nécessaire pour calculer F_n avec les quatre algorithmes que nous venons de voir. On obtient la table suivante :

n	40	5.10^7	2.10^8	2.10^9
`fibo1(n)`	31 s	calcul irréalisable		
`fibo2(n)`	0 s	18 s	`Segmentation fault`	
`fibo3(n)`	0 s	4 s	19 s	3 min 15 s
`fibo4(n)`	0 s	0 s	0 s	0 s

La fonction `fibo1`, de complexité exponentielle, est incapable de calculer F_n dés que n dépasse seulement 50. Les fonction `fibo2` et `fibo3` semblent bien nécessiter un temps de calcul linéaire en n mais `fibo2` utilise beaucoup de mémoire, ce qui limite les valeurs de n que l'on peut atteindre à quelques millions (à comparer aux quelques millions d'octets qui composent la mémoire d'un ordinateur moderne). Finalement, `fibo4` confirme une efficacité bien supérieure aux précédentes fonctions ; il est pratiquement impossible d'en mesurer le temps de calcul, tant elle est rapide, même pour des valeurs gigantesques de n.

Ce premier exemple illustre de manière spectaculaire le gain de temps de calcul qui peut être réalisé au moyen d'une analyse algorithmique poussée du problème à résoudre. Pour diviser par 1000 un temps de calcul, il n'est donc pas toujours nécessaire de disposer de 1000 fois plus de machines !

1.2.2 Problème du tri

Considérons maintenant le problème très classique du tri de tableaux d'entiers. On mesure habituellement la complexité de ces algorithmes au nombre de comparaisons nécessaires pour trier entièrement un tableau de n entiers. La complexité dans le cas le pire est obtenue pour la configuration initiale nécessitant le plus de comparaisons (en général un tableau trié en sens inverse ou même parfois simplement bien trié). La complexité en moyenne se calcule en moyennant sur toutes les distributions initiales possibles, i.e. sur les $n!$ permutations.

Considérons trois algorithmes de tri : les tri *par insertion* décrit page 14, le tri dit *rapide*, étudié page 26, et enfin le tri *fusion* (p. 25). Le tableau suivant résume les complexité dans le cas le pire et en moyenne de ces trois algorithmes.

Algorithme	cas le pire	en moyenne
par Insertion	$\theta(n^2)$	$\theta(n^2)$
Rapide	$\theta(n^2)$	$\theta(n \log n)$
Fusion	$\theta(n \log n)$	$\theta(n \log n)$

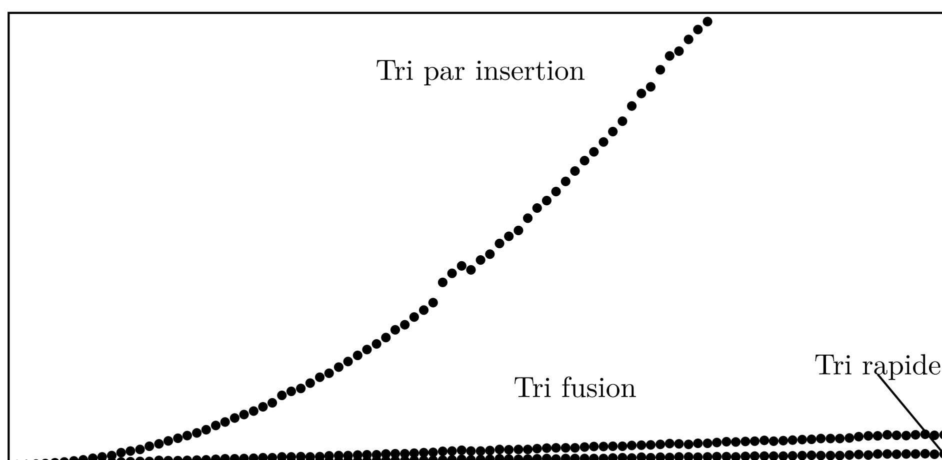

FIGURE 1.1 – Complexité moyenne expérimentale

Une question naturelle est de savoir si une telle mesure est en accord avec l'efficacité pratique de ces algorithmes. En effet, on a dit que l'on mesurait la complexité d'un algorithme de tri au nombre de comparaisons nécessaires mais il existe bien d'autre instructions dont le temps de calcul est négligé. Les figures 1.1, 1.2, 1.3 et 1.4 représente des courbes de temps de calcul (en ordonnée) en fonction du nombre n d'entiers à trier (en abscisse). On remarque qu'en moyenne (figures 1.1 et1.2), le tri par insertion semble bien nécessiter un temps de calcul quadratique alors que les courbes correspondant aux deux autres tris sont bien apparemment linéaires (il est très difficile de distinguer un fonction linéaire $\Theta(n)$ d'une fonction *quasi-linéaire* $\Theta(n \log n)$). Le tri rapide apparaît trois fois plus rapide que le tri fusion.

Lorsque l'on considère le temps de calcul nécessaire pour trier un tableau déjà ordonné mais en sens inverse (figure 1.4), on voit que le tri fusion demeure quasi-

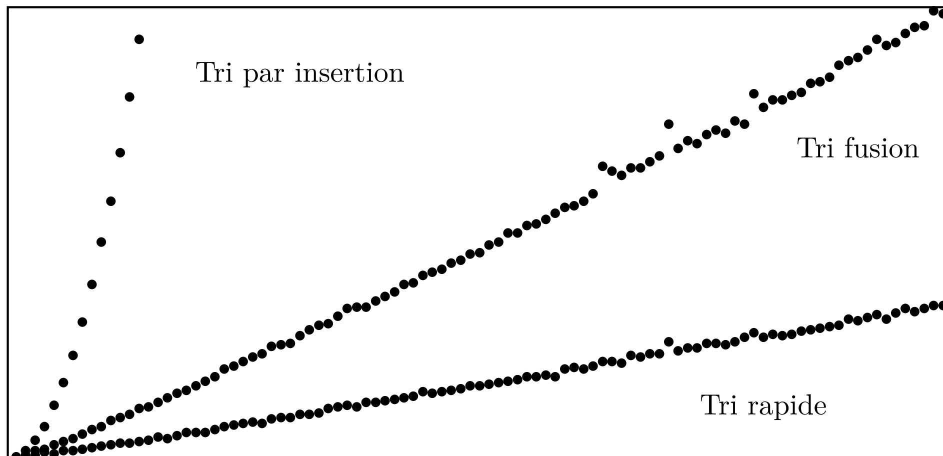

FIGURE 1.2 – Complexité moyenne expérimentale (agrandissement)

linéraire alors que le tri rapide devient étonnement peu efficace (deux fois plus lent que le tri par insertion). Ceci confirme l'analyse de complexité dans le pire cas.

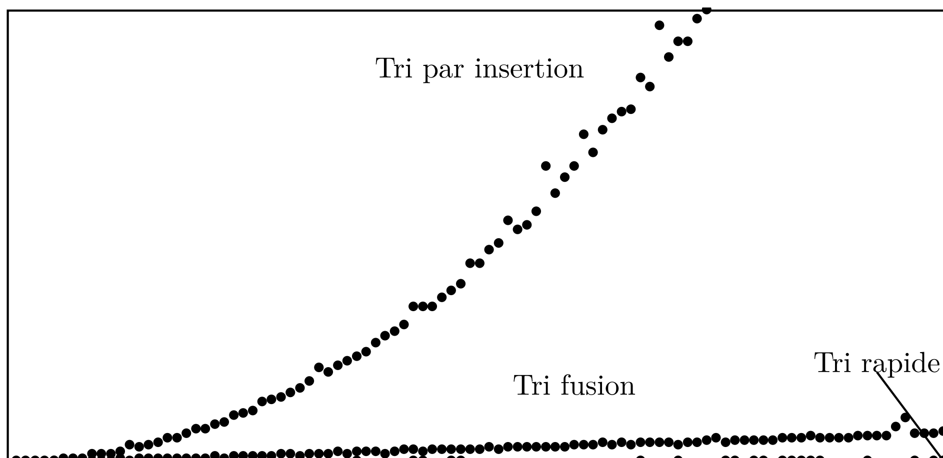

FIGURE 1.3 – Complexité expérimentale dans le cas d'un tableau déjà trié

1.3 Tri par insertion

Intéressons nous maintenant à la méthode à employer pour calculer la complexité d'un algorithme. Pour cela, nous prenons l'exemple du tri par insertion, le tri fusion et le tri rapide étant étudiés dans le prochain chapitre.

1.3.1 Description de l'algorithme

Un algorithme de tri très simple consiste à procéder comme lorsque l'on trie des cartes à jouer à la main. On procède comme suit : on prend une carte, puis une deuxième qu'on compare à la première et qu'on place là où il faut, puis une

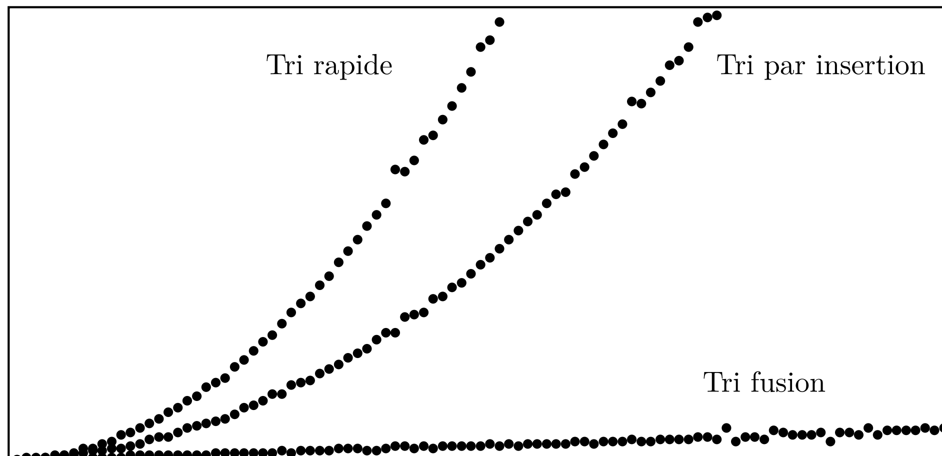

FIGURE 1.4 – Complexité expérimentale dans le cas d'un tableau inversement trié

troisième qu'on compare aux deux autres et qu'on insère au bon endroit, et ainsi de suite jusqu'à ce que l'on ait classé toutes les cartes.

Pour être plus explicite, nous donnons ci-après une version en pseudo-code de la procédure TRI-PAR-INSERTION, prenant en paramètre un tableau A de taille n, dont les indices varient entre 1 et n, et retournant le même tableau trié. Pour notre pseudo-code, les conventions seront les suivantes dans toute la suite de ce document :

— l'indentation indique une structure de bloc ; on n'utilise pas de délimiteurs comme les accolades en C ;
— les mots-clés se comprennent d'eux-mêmes, la flèche ← représentant l'affectation.

Algorithme 1.1 TRI-PAR-INSERTION(A)
1 **pour** $i \leftarrow 2$ **à** n **faire**
2   *clé* $\leftarrow A[i]$
3     $j \leftarrow i - 1$
4   **tant que** $j > 0$ **et** $A[j] >$ *clé* **faire**
5         $A[j+1] \leftarrow A[j]$
6         $j \leftarrow j - 1$
7     $A[j+1] \leftarrow$ *clé*

On s'assurera intuitivement que cette procédure renvoie bien le tableau A trié. Donnons quelques explications : i est l'indice de l'élément en cours d'insertion que l'on stocke dans la variable *clé*. On cherche à insérer la clé dans la partie du tableau déjà triée : $A[1, \ldots, i-1]$. Pour cela, la boucle **tant que** permet de décaler les éléments un à un vers la droite jusqu'à ce que l'emplacement correct pour la clé soit trouvé, entre deux valeurs bien ordonnées, l'un plus petite et l'autre plus grande ou bien en première position (cas $j = 0$) si toutes les valeurs précédentes sont plus grandes que la clé.

1.3.2 Implémentation en C

Voici une implémentation en C du tri par insertion :

```
void Tri_par_Insertion(int *A, int n)
{
  int i,j;
  int cle;

  for(i=1;i<n;i++)
    {
      cle=A[i];
      j=i-1;
      while ((j>=0) && (A[j]>cle))
        {
          A[j+1]=A[j];
          j=j-1;
        }
      A[j+1]=cle;
    }
}
```

Remarque : On notera l'importance d'écrire des programmes des plus clairs possibles. Le C permet en effet l'écriture de code syntaxiquement correct mais totalement illisible. On peut par exemple réécrire la procédure précédente comme suit, mais ceci est **très fortement déconseillé** pour des raisons évidentes :

```
Tri_pas_beau(int *A, int n){
int i=1,j=0,cle=*(A+1);
for(;i<n;A[j+1]=cle,j=i++,cle=A[i])
while ((j>=0) && (A[j]>cle)) A[j+1]=A[j--];}
```

1.3.3 Complexité

Étudions-en à présent la complexité du tri par insertion : l'idée est d'évaluer le temps d'exécution ou encore le nombre d'opérations nécessitées par notre algorithme en fonction d'une certaine mesure de la taille des entrées. Dans notre cas, celle-ci sera simplement la taille du tableau A, soit n, et on va tenter de se faire une idée du nombre C_n de tests de comparaison ($A[j] > clé$) requis. Pour une valeur de i fixée, on effectue au plus i comparaisons, j variant de $i-1$ à 0 dans le pire cas ; pour toute la procédure, on en déduit que :

$$C_n \leq 1 + 2 + \cdots + n - 1 = \frac{n(n-1)}{2}$$

ce qui nous donne le comportement asymptotique de notre algorithme (lorsque n devient "grand") : le tri par insertion est en $O(n^2)$.

Dans le cas le pire, obtenu pour un tableau initialement inversement trié, le tri par insertion est en fait non seulement en $O(n^2)$, mais même en $\Theta(n^2)$; on dit qu'il est quadratique, et plus précisément, en $n^2/2$.

Complexité en moyenne

Il est également intéressant d'effectuer l'analyse *en moyenne* de notre algorithme, c'est-à-dire pour toutes les entrées possibles. Il faut donc se donner une

distribution de probabilités sur celles-ci. Dans le cas qui nous préoccupe, on va supposer que tout tableau de taille n est isomorphe à une permutation de l'intervalle $[1, n]$, i.e. une bijection de l'intervalle $[1, n]$ sur lui-même, que l'on note $\sigma = (a_1 \ a_2 \ \cdots \ a_n)$ si $a_i = \sigma(i)$ pour tout $i \in [1, n]$. On suppose de plus que la distribution des permutations est équidistribuée. On note S_n l'ensemble de ces permutations. On définit une *inversion* d'une permutation $\sigma = (a_1 \ a_2 \ \cdots \ a_n) \in S_n$ comme un couple (a_i, a_j) tel que $i < j$ et $a_i > a_j$, et on note $I(\sigma)$ le nombre d'inversions que présente σ. Par exemple, pour $n = 5$ et $\sigma = (3\ 2\ 1\ 5\ 4)$, $I(\sigma) = 4$ ($3 > 2$, $3 > 1$, $2 > 1$ et $5 > 4$).

Il n'est pas très difficile de se convaincre que pour une permutation σ donnée, le nombre $c_i(\sigma)$ de comparaisons nécessaires pour insérer a_i dans la séquence de ses prédécesseurs est égal à :

$$c_i(\sigma) = 1 + \text{Card}\left(\{j \in [1, i-1], \ \ a_j > a_i\}\right)$$

ce qui entraîne que le nombre total $C(\sigma)$ de comparaisons requises pour σ est :

$$C(\sigma) = \sum_{i=2}^{n} c_i(\sigma) = n - 1 + \sum_{i=2}^{n} \text{Card}\left(\{j \in [1, i-1], \ \ a_j > a_i\}\right) = n - 1 + I(\sigma)$$

Or, d'après l'hypothèse d'équiprobabilité de la distribution, on en déduit que :

$$C_n = \frac{1}{n!} \sum_{\sigma \in S_n} C(\sigma) = n - 1 + \frac{1}{n!} \sum_{\sigma \in S_n} I(\sigma)$$

Remarquant alors que pour toute permutation $\sigma = (a_1 \ a_2 \ \cdots \ a_n) \in S_n$, son image "renversée" $\sigma' = (a_n \ a_{n-1} \ \cdots \ a_1)$ est telle que :

$$I(\sigma) + I(\sigma') = \frac{n(n-1)}{2}$$

on en déduit que :

$$C_n = n - 1 + \frac{1}{2} \times \frac{n(n-1)}{2} = \frac{n^2 + 3n - 4}{4}$$

En conclusion, en moyenne, le tri par insertion est en $n^2/4$, et donc encore en $\Theta(n^2)$, comme dans le cas le pire. De ce fait, le tri par insertion est une méthode de tri dite élémentaire, facile à programmer mais qui n'est pas satisfaisante dans le cas général.

Il existe beaucoup d'autres algorithmes, tels que le *tri par sélection*, le *tri à bulles*, etc... qui ont également une complexité en $\Theta(n^2)$ dans les cas moyen et pire, et dont l'étude est laissée en exercices (1.2 et 1.3).

1.4 Qu'est ce qu'un algorithme efficace ?

Tous ces exemples nous mènent à une question naturelle mais délicate : qu'est ce qu'un algorithme efficace ? On admet généralement que des algorithmes de complexité logarithmique ($\Theta(\log n)$), linéaire ($\Theta(n)$) ou quasi-linéaire ($\Theta(n \log n)$) sont efficaces, i.e. rapides dans la pratique, même pour des données initiales de grande

taille. Les algorithmes de complexité polynomiale ($\Theta(n^2)$, $\Theta(n^3)$,...) peuvent encore être considérés efficaces à condition que l'exposant ne soit pas trop grand. Enfin, les algorithme de complexité exponentielle ($\Theta(2^n)$, $\Theta(\exp n)$, $\Theta(n!)$) sont définitivement inefficaces et de peu d'intérêt pratique. Un tel panorama est cependant à nuancer au cas par cas, certains algorithmes exponentiels étant utilisables en pratique jusqu'à des tailles de données suffisantes en pratique alors que certains algorithmes polynomiaux sont impossibles à mettre en œuvre. Il est important de bien garder à l'esprit que l'on ne s'intéresse qu'à une mesure asymptotique de la complexité des algorithmes.

Une seconde question est comment choisir entre deux algorithmes résolvant le même problème ? On souhaite bien entendu en général avoir un algorithme le plus efficace possible mais plusieurs critères doivent être considérés, selon l'application envisagée.

— **complexité en moyenne** : le premier critère est bien entendu la complexité en moyenne. Il faut cependant être prudent et se demander si cette mesure correspond bien au cas pratique considéré. En effet, certaines hypothèses notamment d'équiprobabilité sur les données initiales, sont nécessaires pour calculer la complexité en moyenne d'un algorithme. De telles hypothèses peuvent ne pas être vérifiées en pratique, par exemple si l'on souhaite trier des tableaux qui sont presque déjà bien ordonnés.

— **complexité dans le pire cas** : suivant l'application, la complexité dans le pire cas peut être purement anecdotique ou d'une importance cruciale. Que dire en effet d'un algorithme ultra-rapide en général mais qui va nécessité un temps de calcul très important pour quelques données initiales particulières ? On a par exemple vu que le tri fusion est moins efficace que le tri rapide en général mais que ce dernier devient très mauvais pour certains tableaux initialement triés à l'envers. Par conséquent, pour des applications en temps réel pour lesquelles on exige avant tout que la durée des calculs ne dépasse pas certaines bornes, le tri fusion sera sûrement bien plus adapté.

— **facilité d'implémentation** : la théorie de la complexité nous fournit de très nombreux exemples d'algorithmes très difficiles mais qui améliorant sensiblement la complexité d'algorithmes simples. La multiplication matricielle de matrices carrées d'ordre n peut se réaliser très simplement en $\Theta(n^3)$ opérations en utilisant l'algorithme que l'on emploie *à la main*. Il existe cependant d'autres algorithmes qui améliorent cette complexité au prix de constructions très difficiles et par conséquent posant d'importants problèmes d'implémentation. Dans la pratique, entre un algorithme simple et facile à programmer et un algorithme plus efficace mais nécessitant un travail d'implémentation énorme, on choisira bien souvent la facilité, pour des raisons de coût mais aussi de fiabilité, un programme simple et court ayant beaucoup plus de chances d'être correct !

— **efficacité pratique** : comme nous l'avons dit, la notation Θ masque les coefficients multiplicateurs constants. Entre deux algorithmes de complexité différente, nous apprenons donc seulement qu'il existe une limite à partir de laquelle un des deux algorithmes devient nécessairement plus efficace que l'autre. Cependant, nous n'apprenons pas grand chose sur la valeur pratique de cette limite. Pour des tailles de données réellement manipulées,

il peut donc être intéressant d'utiliser un algorithme de complexité a priori plus mauvaise qu'un autre.

Le choix d'un algorithme plutôt qu'un autre reste donc fortement dépendant de l'application considérée. Les mesures théoriques de complexité peuvent cependant très souvent apporter des arguments décisifs. Enfin, toute solution pratique est envisageable ; on peut par exemple utiliser simultanément plusieurs algorithmes en attendant que le plus rapide trouve ou bien choisir dynamiquement l'algorithme au vue des données initiales particulières que l'on traite. Ainsi, on peut très bien utiliser un tri rapide lorsque le tableau à trier dépasse 10 éléments mais simplement un tri par insertion pour de petits tableaux de moins de 10 éléments. On peut aussi arrêter la récurrence plus tôt dans le tri fusion et traiter de même les petits tableaux avec un tri quadratique.

1.5 Exercices

Exercice 1.1 *Programmer en C le tri par insertion. Vérifier expérimentalement, en mesurant le nombre de comparaisons nécessaires pour trier des tableaux de n entiers choisis au hasard, la complexité quadratique de ce tri.*

Quelle est, expérimentalement, la complexité si les tableaux sont déjà en grande partie triés ? (on peut par exemple choisir un tableau rangé et permuter au hasard une fraction des données afin d'étudier la complexité en fonction de ce paramètre).

Exercice 1.2 *L'algorithme de* tri par sélection *d'un tableau de taille n procède de la manière suivante : on en cherche le plus petit élément que l'on place en tête en l'échangeant avec la première entrée du tableau ; on cherche alors le plus petit élément parmi les $n-1$ suivants, qu'on échange avec la deuxième entrée du tableau, et ainsi de suite jusqu'à ce que le tableau soit complètement trié.*

Voici deux pseudo-code implémentant le tri par sélection, le premier de manière récursive et le second de manière itérative.

Algorithme 1.2 Tri-Sélection-Récursif(A, p)

1 **si** $p < n$ **alors**
2 $\quad q \leftarrow$ Sélectionner(A, p)
3 $\quad$ Échanger$(A[p], A[q])$
4 $\quad$ Tri-Sélection-Récursif$(A, p+1)$
5 Sélectionner(A, p)
6 $\quad min \leftarrow p$
7 $\quad$ **pour** $i \leftarrow p+1$ **à** n **faire**
8 $\quad\quad$ **si** $A[i] < A[min]$ **alors** $min \leftarrow i$
9 $\quad$ **return** min

Algorithme 1.3 Tri-Sélection-Itératif(A)

1 **pour** $p \leftarrow 1$ **à** $n-1$ **faire**
2 $\quad q \leftarrow$ Sélectionner(A, p)
3 $\quad$ Échanger$(A[p], A[q])$

4 Sélectionner(A, p)
5 $\quad min \leftarrow p$
6 $\quad$ **pour** $i \leftarrow p+1$ **à** n **faire**

7 **si** $A[i] < A[min]$ **alors** $min \leftarrow i$
8 **return** *min*

Donner une version en C de ce tri. Quelle en est la complexité dans le meilleur et le pire des cas, en termes de comparaisons et d'échanges d'éléments ?

Exercice 1.3 *Le* tri à bulles *d'un tableau consiste à le parcourir en échangeant toute paire d'éléments consécutifs présentant une inversion, et à recommencer jusqu'à ce que le tableau soit trié — on notera qu'après k parcours, les k plus grands éléments du tableau sont en bonne place.*

Donner une version en C de ce tri. Quelle en est la complexité dans le cas le pire et en moyenne, en termes de nombre de comparaisons d'éléments ?

Exercice 1.4 *Afin de calculer efficacement* a^b *avec b un entier positif, l'idée des algorithmes dits* d'exponentiation binaire *est de décomposer b en base 2 :*

$$b = \sum_{i=0}^{k} b_i \times 2^i \quad \textit{avec} \quad b_i \in \{0, 1\}$$

On peut ensuite remarquer que

$$a^b = a^{\sum_{i=0}^{k} b_i \times 2^i} = \prod_{i=0}^{k} \left(a^{(2^i)}\right)^{b_i}$$

Afin de calculer a^b*, il suffit donc d'être capable de calculer les* $a^{(2^i)}$ *qui peuvent s'obtenir récursivement par élévation successive au carré.*

En déduire une fonction C capable de calculer a^b *en* $\Theta(\log b)$ *opérations.*

Chapitre 2

Récursivité

Avant d'aborder le cœur même de ce qui constitue ce cours, nous allons nous pencher sur une technique essentielle de programmation : la *récursivité*. Plus qu'une technique, c'est aussi une théorie mathématique très élaborée que nous ne ferons qu'effleurer ici.

Informellement, une fonction récursive est une fonction qui s'appelle elle-même. L'intérêt d'un tel concept est qu'il correspond bien à l'idée intuitive qu'on se fait d'un procédé de calcul automatique, qui consiste à réappliquer une même procédure sur des objets similaires (quoique éventuellement de taille différente). Dans les langages de programmation, cette idée peut s'exprimer à l'aide de boucles, ou encore à l'aide de fonctions récursives. En fait, on verra que de nombreux algorithmes s'écrivent plus naturellement de cette dernière façon.

2.1 Fonctions numériques récursives

En mathématique, on a souvent tendance à définir des objets, et en particulier des fonctions, sans en préciser clairement une représentation, et encore moins une méthode pour en calculer la ou les valeurs. Cette situation est d'ailleurs un peu paradoxale, quand on sait les exigences de rigueur propres à cette discipline, puisqu'on peut avoir ainsi l'impression de raisonnements rigoureux fondés sur des prémisses floues. Par exemple, reprenons l'une des définitions de la *factorielle* qu'on trouve régulièrement dans les manuels scolaires :

$$n! = 1 \times 2 \times \cdots \times n$$

Si l'on voit fort bien de quoi il s'agit, on peut tout de même s'interroger sur le sens exact des points de suspension : en aucun cas, cette formule ne fournit un modèle calculatoire explicite des valeurs de la fonction. Une définition alternative de la factorielle fait intervenir un opérateur produit :

$$n! = \prod_{i=1}^{n} i$$

Cette formule peut sembler plus claire, mais cela reste un leurre, car il faut alors connaître la définition de l'opérateur produit $\prod_{i=1}^{n}$, et on est de nouveau ramené à l'emploi de points de suspension.

Une telle exigence de rigueur risque d'apparaître un peu excessive, même pour un informaticien (c'est-à-dire quelqu'un préoccupé par la nécessité de toujours pouvoir représenter les objets qu'il manipule, et surtout de pouvoir effectuer des calculs automatiques sur eux). En effet, on n'aura aucun mal à transcrire les définitions précédentes en un programme permettant de calculer les valeurs de la factorielle, à l'aide d'une boucle **pour** ou **tant que**. Le problème est que cette transcription reste guidée par l'intuition du programmeur : elle ne saurait se faire de manière automatique, et rien ne garantit qu'elle soit toujours réalisable, à partir de n'importe quelle définition non totalement explicite.

Il est pourtant possible de donner une définition non ambiguë de la factorielle par une formule de récurrence :

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

On peut alors transcrire cette formule directement, c'est-à-dire en quelque sorte sans réfléchir, en une fonction récursive de n'importe quel langage de programmation — ce qui donne par exemple en C :

```
int factorielle(int n)
{
  if (n == 0)
    return 1;
  else
    return n * factorielle(n - 1);
}
```

Il peut encore exister des définitions de fonctions qui, tout en n'étant pas ambiguës, ne fournissent pas de procédé explicite de calcul des valeurs. Considérons l'exemple du *plus grand commun diviseur* (pgcd) de deux entiers naturels n et m. Cette définition n'indique (presque) rien sur la façon de le déterminer en pratique ; en la suivant tant bien que mal, on pourrait songer à calculer l'ensemble des diviseurs de n et l'ensemble des diviseurs de m, puis à en prendre l'intersection, pour enfin en extraire le plus grand élément. On voit bien qu'on n'a pas alors de transcription immédiate en un algorithme, puisqu'il faut encore spécifier plus finement chacune des précédentes actions (comment calculer l'intersection, par exemple ?). La complexité de cette méthode serait de plus très mauvaise, et ce n'est pas ainsi que l'on opère en pratique : on utilise l'algorithme d'Euclide qui se déduit du théorème suivant :

$$\forall (n,m) \in \mathbb{N}^2,\ n \geq m,\ \text{pgcd}(n,m) = \begin{cases} n & \text{si } m = 0 \\ \text{pgcd}(m, n \bmod m) & \text{sinon} \end{cases}$$

dont la traduction en fonction récursive est immédiate. On notera qu'on a ici quelque chose d'un peu plus général qu'une simple formule de récurrence reliant une valeur indicée par n à une valeur indicée par $n-1$ (puisque les indices correspondants sont le couple (n,m) et le couple $(m, n \bmod m)$).

L'idée qu'on essaie en fait d'illustrer à l'aide de ces exemples est la suivante : si l'on veut donner une définition parfaitement explicite d'une fonction opérant sur les entiers qui permette d'en calculer les valeurs de manière totalement automatique, un bon moyen de le faire est d'utiliser une formule de récurrence, ou

plus généralement une formule de récursion, qu'on pourra transcrire directement en une fonction récursive.

Cette idée est encore bien plus générale à la vérité, car on admet à l'heure actuelle que la notion même de calcul effectif — c'est-à-dire, réalisable sur un ordinateur grâce à un algorithme — coïncide avec celle de récursivité : tout calcul effectif sur un objet représentable en machine peut s'exprimer à l'aide d'une fonction récursive. L'objet en question, qui peut être autre chose qu'un entier (par exemple, un n-uplet d'entiers), se définissant lui-même récursivement, comme on le verra un peu plus loin. Cette idée est connue sous le nom de *thèse de Church*[1]. Pour plus de renseignements sur la théorie mathématique de la récursivité (et la calculabilité), on se référera par exemple au chapitre consacré au sujet dans *Handbook of Mathematical Logic* (voir [2, pages 527–566]) ou bien à [10].

Pour ce qui nous concerne, on se contentera de retenir qu'on peut tout programmer à l'aide fonctions récursives, sans boucles **pour** ou **tant que**. Bien sûr, cela ne signifie pas qu'il faille l'employer systématiquement : tout dépendra du langage employé et de l'algorithme choisi. Certains algorithmes s'écriront plus naturellement avec une boucle **pour**, d'autres dont nous verrons de nombreux exemples dans la suite, avec des procédures récursives.

En pratique, lorsqu'on voudra écrire une fonction ou une procédure récursive, on pourra se remémorer qu'il (ne) s'agit (que) d'une généralisation d'une formule de récurrence $u_n = f(u_{n-1})$ sur les entiers : généralisation d'une part parce que la formule pourra relier non seulement u_n à u_{n-1}, mais encore à toute une famille $(u_{h_i(n)})_{1\leq i\leq k}$, les h_i étant des fonctions récursives, et d'autre part parce que les indices pourront être autre chose que des entiers. Et comme dans toute définition par une formule de récurrence, on songera bien à ne jamais oublier le cas d'arrêt, c'est-à-dire l'équivalent de u_0 (sans quoi le calcul ne pourra terminer ou conduira à une erreur).

2.1.1 Suite de Fibonacci

Nous avons déjà abordé en détail le calcul des nombres de Fibonacci dans la section 1.2.1. Il ne faudrait cependant pas en conclure que l'approche récursive est totalement inefficace et que toute bonne implémentation doit nécessairement l'éviter. En effet, la complexité exponentielle de `fibo1` provient du recalcul de valeurs intermédiaires un très grand nombre de fois. On peut cependant réécrire la définition récursive de la suite de Fibonacci de la manière suivante :

$$(F_{n+1}, F_n) = \begin{cases} (1,1) & \text{si } n = 0 \\ f(F_n, F_{n-1}) & \text{sinon, avec } \; f(a,b) = (a+b, a) \end{cases}$$

Cette approche fournit la fonction suivante qui calcule F_n en réponse à l'appel `fibo5(n,1,1)` :

```
int fibo5(int n, int a, int b)
{
  if (n==0) return b;
  return fibo5(n-1,a+b,a);
}
```

1. De Alonzo Church, logicien américain qui le premier l'a exprimée en 1935.

On remarquera que la complexité en temps de cette fonction est linéaire puisque le calcule de F_n nécessite $n + 1$ appels à `fibo5`. L'évaluation de la complexité en espace est plus subtile. En effet, à chaque appel récursif de la fonction, de nouvelles variables sont créées pour stocker `n`, `a` et `b`. Dans la pratique, si l'on demande le calcul d'un très grand nombre de Fibonacci et que l'on surveille l'évolution de la quantité de mémoire utilisée par le programme, on voit que cette dernière évolue régulièrement au cours du temps.

Il ne faut cependant pas s'arrêter à ce constat négatif ; les compilateurs modernes disposent d'optimiseurs très performants, capables de réduire considérablement la complexité pratique des algorithmes. Ainsi, le même programme compilé en précisant que l'on souhaite optimiser l'exécutable au maximum, grâce à la commande

```
gcc fibo5.c -o fibo5 -O9
```

fournit un programme n'utilisant qu'une quantité fixe de mémoire, indépendante de n. En conclusion, il faut donc se méfier des capacités remarquables des compilateurs modernes !

2.1.2 Fonction d'Ackermann

On définit la fonction d'Ackermann de la façon suivante :

$$\begin{cases} Ack(0, n) = n + 1 & \forall n \in \mathbb{N} \\ Ack(m, 0) = Ack(m - 1, 1) & \forall n \in \mathbb{N}^* \\ Ack(m, n) = Ack(m - 1, Ack(m, n - 1)) & \forall (m, n) \in (\mathbb{N}^*)^2 \end{cases}$$

A priori, il n'est pas évident que pour deux entiers quelconques m et n le calcul se termine. Pour le démontrer, considérons l'ordre lexicographique sur les paires (m, n), i.e. $(m_1, n_1) \leq (m_2, n_2)$ si $m_1 < m_2$ ou ($m_1 = m_2$ et $n_1 \leq n_2$). On voit que le calcul de $Ack(m, n)$ nécessite deux appels récursifs à la fonction mais avec les paramètres lexicographiquement strictement plus petits. Le calcul finit donc par s'arrêter mais on n'a pas à priori d'idée sur le nombre d'itérations nécessaires.

En fait, la fonction d'Ackermann est un exemple classique de fonction qui croît très rapidement. On peut en effet montrer que :

— $Ack(0, n) = n + 1 = \Theta(n)$,
— $Ack(1, n) = n + 2 = \Theta(n)$,
— $Ack(2, n) = 2n + 3 = \Theta(n)$,
— $Ack(3, n) = 2^{n+3} - 3 = \Theta(2^n)$,
— $Ack(4, n) \approx 2^{2^{2^{2\cdots}}}$ avec ainsi de l'ordre de n niveaux d'exposant successifs.

Ainsi $Ack(5, 1)$ est déjà un nombre gigantesque, supérieur à 10^{80}.

Il faut donc se méfier des mécanismes récursifs qui peuvent dans certains cas générer des objets mathématiquement bien définis mais impossible à calculer, ne serait-ce qu'à cause de leur taille.

2.2 Procédures récursives

Plus généralement, la récursivité permet non seulement de calculer des fonctions mais surtout de résoudre des problèmes en ramenant la recherche de solutions

pour une instance de taille donnée à la recherche de solutions pour des instances de plus petites tailles. Le tri d'un tableau peut ainsi être ramené au tri de tableau plus petits, comme avec les algorithme de tri fusion et de tri rapide que nous allons étudier, et ce récursivement jusqu'à obtenir des tableaux d'au plus un élément, trivialement triés.

2.2.1 Tri fusion

Algorithme

Le *tri fusion* consiste à couper le tableau à trier en deux sous-tableaux de même taille, puis à appeler récursivement le tri sur les deux sous-tableaux ainsi obtenus, qu'on fusionne une fois triés à l'aide d'une fonction de complexité linéaire. En voici la version en pseudo-code :

Algorithme 2.1 TRI-FUSION(A, p, r)

1 **si** $p < r$ **alors**
2 $\quad q \leftarrow \lfloor (p+r)/2 \rfloor$
3 $\quad$ TRI-FUSION(A, p, q)
4 $\quad$ TRI-FUSION$(A, q+1, r)$
5 $\quad$ FUSIONNER(A, p, q, r)

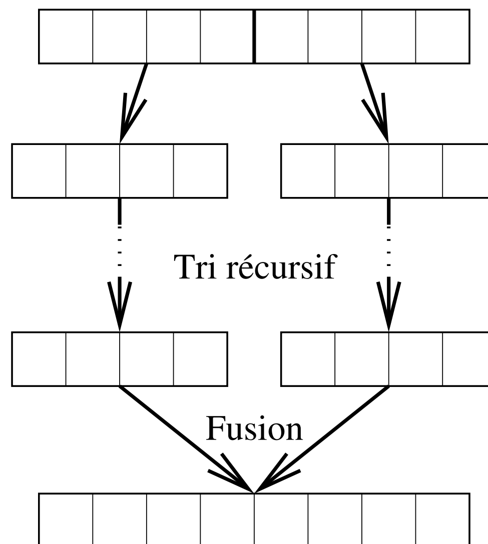

FIGURE 2.1 – Représentation graphique du tri fusion

La fonction FUSIONNER qui fusionne deux tableaux triés en un seul est laissée à titre d'exercice (voir exercice 2.4). On notera qu'à la différence du tri par insertion déjà vu et du tri rapide décrit dans la prochaine section, on a besoin d'un tableau annexe : le tri ne se fait pas *en place*. En pratique, on s'en sert en présence de listes et pour de gros fichiers (sur disques).

Le tri fusion, comme tous les tris récursifs, suit l'approche dite *diviser pour régner*, qui est une méthode de conception d'algorithmes consistant à diviser un problème en plusieurs sous-problèmes similaires mais plus simples à résoudre et ce

récursivement jusqu'à obtenir des problèmes élémentaires comme trier un tableau vide ou composé d'un unique élément.

Complexité

La complexité du tri fusion en moyenne est identique à la complexité dans le cas le pire car l'exécution de l'algorithme ne dépend pas du tableau à trier. Par conséquent, l'algorithme a un temps d'exécution constant, ce qui peut être très important pour certaines applications. On laisse a titre d'exercice (voir 2.4) le soin au lecteur de démontrer que la complexité est en $\Theta(n \log n)$.

2.2.2 Tri rapide

L'algorithme de tri rapide (*quicksort*), proposé par C. A. R. Hoare en 1960, est structurellement récursif, et l'un des plus couramment utilisés dans les applications informatiques.

Algorithme

L'idée est la suivante : choisissant un élément pivot au hasard dans le tableau, on partitionne celui-ci en deux sous-tableaux tels que les éléments du premier soient plus petits que le pivot, et les éléments du second plus grands, et on appelle récursivement la procédure de tri sur ces sous-tableaux. Le cas d'arrêt est obtenu quand les tableaux sont de taille 0 ou 1, et donc naturellement triés. On fait donc appel à une procédure de partitionnement PARTITIONNER qui constitue la partie plus complexe de l'algorithme :

Algorithme 2.2 TRI-RAPIDE(A, p, r)

```
1  si p < r alors
2      q ← PARTITIONNER(A, p, r)
3      TRI-RAPIDE(A, p, q − 1)
4      TRI-RAPIDE(A, q + 1, r)
5  PARTITIONNER(A, p, r)
6      clé ← A[p]
7      q ← p
8      pour i ← p + 1 à r faire
9          si A[i] < clé alors
10             q ← q + 1
11             ÉCHANGER(A[q], A[i])
12     ÉCHANGER(A[q], A[p])
13     retourner q
```

On se convaincra sur des exemples que cet algorithme trie bel et bien le tableau A quand on l'appelle avec $p = 1$ et $r = n$. La fonction PARTITIONNER choisit $A[p]$ comme pivot (clé), puis fait évoluer les pointeurs q et i de manière à ce qu'à tout moment, tous les $A[j]$ soient inférieurs à la clé pour $j \in]p, q]$ et supérieurs à la clé pour $j \in]q, i[$.

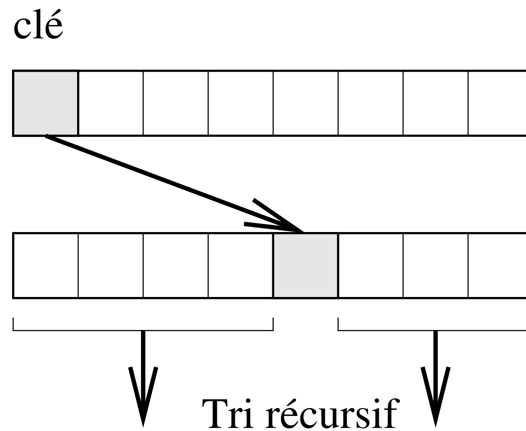

FIGURE 2.2 – Représentation graphique du tri rapide

Implémentation en C

```
int Partitionner(int *A, int p, int r)
{
  int t,i,q;
  q=p;
  for(i=p+1;i<=r;i++)
    if (A[i]<A[p])
      {
        q++;
        t=A[i]; A[i]=A[q]; A[q]=t;
      }
  t=A[p]; A[p]=A[q]; A[q]=t;
  return q;
}

void Tri_Rapide(int *A, int p, int r)
{
  int q;
  if (p<r)
    {
      q=Partitionner(A,p,r);
      Tri_Rapide(A,p,q-1);
      Tri_Rapide(A,q+1,r);
    }
}
```

Complexité

Intéressons-nous maintenant à la complexité de notre algorithme : soit C_n le nombre de comparaisons du type $A[j] \leq$ clé requises par l'exécution de la procédure. On a trivialement : $C_0 = C_1 = 0$. On peut encore remarquer que la fonction PARTITIONNER nécessite $\Theta(n)$ comparaisons.

Si le tableau est déjà trié (dans l'ordre croissant ou décroissant), le partitionnement se fait de manière complètement déséquilibré : à partir d'un tableau de taille n, on obtient un sous-tableau de taille 1 et un autre de taille $n-1$, et il n'est pas très difficile de prouver qu'alors le tri rapide est en $\Theta(n^2)$.

Toutefois, en moyenne, le résultat est bien meilleur. Pour le prouver, nous allons nous inspiré de la démonstration de [9]. La fonction PARTITIONNER(A, p, r) effectue exactement $r - p$ comparaisons. Pour trier un tableau de n éléments, le nombres de comparaisons C_n est donc de $n - 1$ pour l'appel à PARTITIONNER auquel il faut ajouter le coût des appels récursifs, i.e. C_{q-p} et C_{r-q}. En supposant que q est uniformément distribué entre p et r, ceci nous donne :

$$C_n = n - 1 + \frac{1}{n}\sum_{q=1}^{n}(C_{q-1} + C_{n-q})$$

En toute rigueur, l'hypothèse de répartition uniforme du pivot est quelque peu péremptoire ; elle n'est réellement assurée que pour des versions *stochastiques* de l'algorithme, où le pivot est choisi aléatoirement à chaque appel de la procédure. En pratique, ces versions présentent l'avantage supplémentaire d'éviter presque toujours d'avoir un comportement dans le pire des cas.

Par un argument de symétrie, on peut réécrire la formule précédente sous la forme :

$$C_n = n - 1 + \frac{2}{n}\sum_{q=1}^{n} C_{q-1}$$

En multipliant les deux côtés par n, on obtient :

$$nC_n = n(n-1) + 2\sum_{q=1}^{n} C_{q-1}$$

puis en soustrayant la formule équivalente pour $n - 1$:

$$nC_n - (n-1)C_{n-1} = n(n-1) - (n-1)(n-2) + 2C_{n-1}$$

d'où :

$$nC_n = (n+1)C_{n-1} + 2(n-1)$$

et donc en divisant par $n(n+1)$:

$$\frac{C_n}{n+1} = \frac{C_{n-1}}{n} + \frac{2(n-1)}{n(n+1)} = \frac{C_{n-2}}{n-1} + \frac{2(n-2)}{n(n-1)} + \frac{2(n-1)}{n(n+1)} = \cdots = \sum_{k=1}^{n}\frac{2(k-1)}{k(k+1)}$$

D'où finalement :

$$C_n = (n+1)\sum_{k=1}^{n}\frac{2(k-1)}{k(k+1)}$$

Un logiciel de calcul formel fournit le développement asymptotique suivant

$$C_n = (2\,\gamma + 2\ln(n) - 4)\,n + 1 + 2\,\gamma + 2\ln(n) + \frac{5}{6}\,\frac{1}{n} - \frac{1}{6}\,\frac{1}{n^2} + \frac{1}{60}\,\frac{1}{n^3} + \mathrm{O}(\frac{1}{n^4})$$

on en déduit que la complexité en moyenne du tri rapide est en $\Theta(n \log n)$.

2.2.3 Complexité des algorithmes de tri

Il faut bien se rendre compte qu'un algorithme en $\Theta(n \log n)$ est quasi-linéaire, et donc bien plus rapide qu'un algorithme en $\Theta(n^2)$ pour des données de grande taille (n de l'ordre de quelques dizaines en pratique). Ainsi, pour $n = 10^3$, $n \log_2 n$ vaut environ 10^4, et $n^2 = 10^6$; pour $n = 10^6$, $n \log_2 n \approx 2.10^7$ et $n^2 = 10^{12}$. Donc le tri rapide ou le tri fusion doivent être employés, et non pas le tri par insertion, dés que le nombre d'éléments à trier dépasse quelques centaines.

Pour en conclure avec les algorithmes de tri et leur complexité, on peut enfin se demander s'il est possible de faire mieux : la réponse est négative si l'on ne dispose pas d'hypothèse supplémentaire. En effet, on peut démontrer le résultat suivant :

Proposition 2.1 *La complexité moyenne en termes de nombre de comparaisons de tout algorithme de tri est en* $\Omega(n \log(n))$.

On peut en effet modéliser tout algorithme de tri n'utilisant que des comparaisons à l'aide d'un **arbre de décision**, comme illustré dans la figure 2.3.

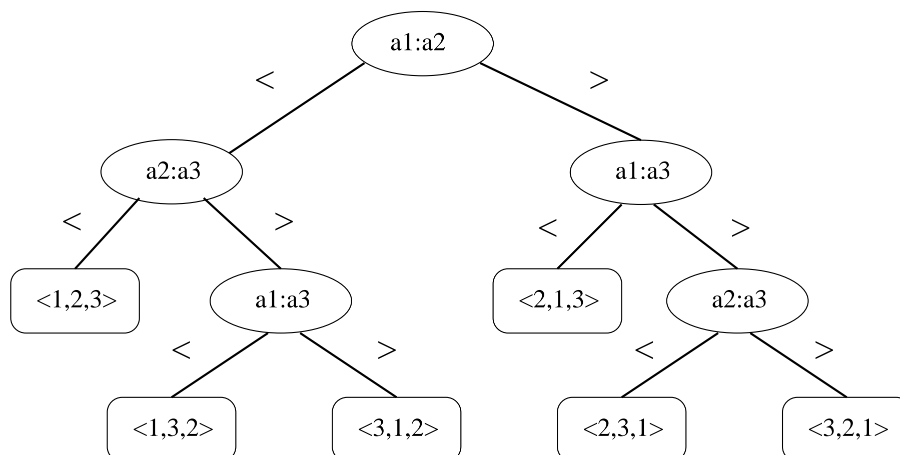

FIGURE 2.3 – Arbre de décision d'un algorithme de tri

L'arbre de décision a au moins $n!$ feuilles (une par permutation). Soit h sa hauteur. Dans le pire cas, il faut h comparaisons pour trier un tableau de n éléments. On a donc $n! \leq 2^h$ et par conséquent

$$h \geq \log n! > \log\left(\left(\frac{n}{e}\right)^n\right) = n \log n - n \log e$$

Finalement, on obtient $h = \Omega(n \log(n))$; il faut nécessairement au moins de l'ordre de $n \log(n)$ comparaisons, en moyenne, pour trier un tableau de n éléments. Notons cependant que sous certaines hypothèses, par exemple concernant la distribution des objets à trier, il est possible de faire mieux et notamment d'atteindre une complexité linéaire.

2.2.4 Tours de Hanoï

Le jeu des tours de Hanoï est un casse-tête qui se présente sous la forme d'un ensemble de trois piquets sur lesquels on peut enfiler des rondelles au nombre de

n et qui sont toutes de diamètre différent.

Initialement, les rondelles se trouvent toutes sur l'un des piquets dans l'ordre décroissant des diamètres, c'est-à-dire que la plus grande est tout en bas et la plus petite tout en haut. Le but du jeu est de transférer toutes les rondelles sur un piquet de destination choisi parmi les deux piquets vides, sachant qu'on ne peut en déplacer qu'une seule à la fois d'un sommet d'une pile vers un autre piquet, et qu'il est interdit de poser une rondelle sur une plus petite qu'elle-même.

Pour résoudre ce problème, on utilise la procédure récursive suivante qui opère en trois temps (voir figure 2.4) :

1. on déplace les $n - 1$ rondelles supérieures vers le piquet intermédiaire à l'aide d'un appel récursif à notre procédure ;
2. on fait transiter la plus grande rondelle du piquet d'origine vers le piquet de destination ;
3. on déplace enfin les $n - 1$ rondelles du piquet intermédiaire vers le piquet de destination, au-dessus de la plus grande.

L'implémentation en C se fait très simplement de la manière suivante :

```
#include <stdio.h>

void Hanoi(int n, int i, int j)
{
  int intermediaire=6-(i+j);
  if (n>0)
    {Hanoi(n - 1, i, intermediaire);
     printf("Mouvement du piquet %d vers le piquet %d\n",i,j);
     Hanoi(n - 1, intermediaire, j);
    }}

int main(int argc, char * args[])
{
  Hanoi(atoi(args[1]),1,3);
}
```

Ce qui nous donne l'exécution suivante pour 3 rondelles :

```
>Hanoi 3
Mouvement du piquet 1 vers le piquet 3
Mouvement du piquet 1 vers le piquet 2
Mouvement du piquet 3 vers le piquet 2
Mouvement du piquet 1 vers le piquet 3
Mouvement du piquet 2 vers le piquet 1
Mouvement du piquet 2 vers le piquet 3
Mouvement du piquet 1 vers le piquet 3
```

2.3 Complément : définitions inductives

On a vu que les fonctions effectivement calculables sur les entiers se définissent de manière récursive ; on va voir maintenant que c'est le cas aussi des entiers eux-

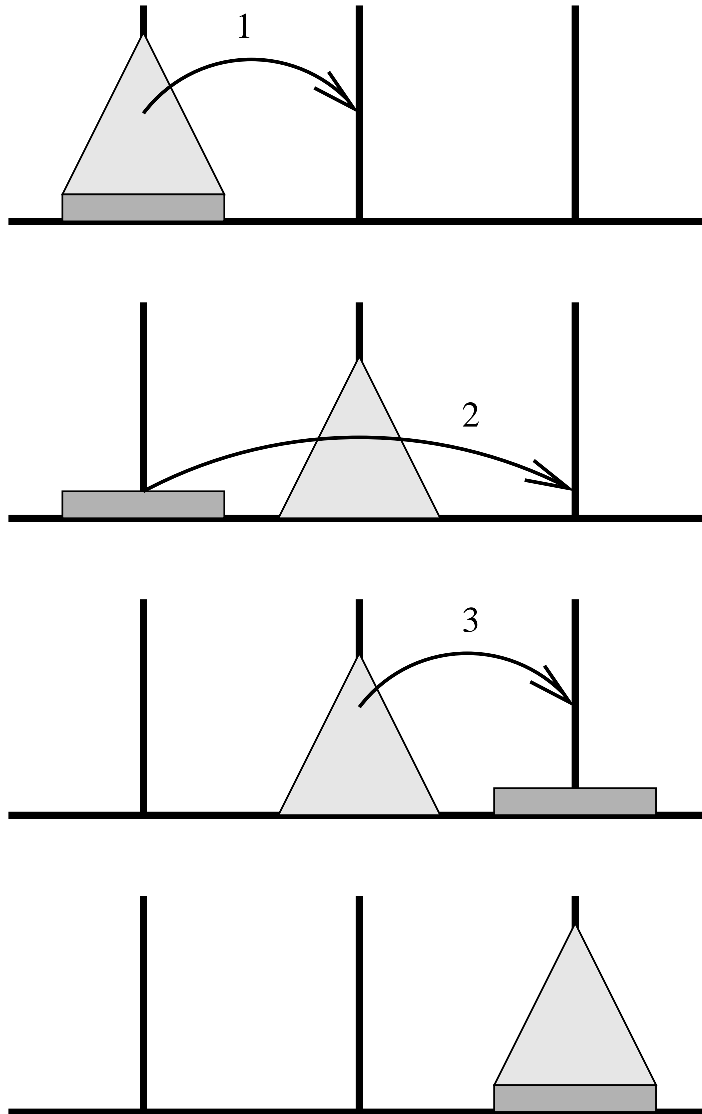

Figure 2.4 – Résolution récursive du problème des tours de Hanoï

mêmes. Ceci constituera notre premier exemple de définition *inductive* ou récursive d'un objet mathématique.

Un pensum très amusant à donner à des étudiants en mathématique est de leur demander de définir des objets aussi élémentaires que les nombres entiers naturels et l'addition s'y rapportant. En règle générale, on n'obtient pas de réponse tellement plus satisfaisante qu'un début d'énumération : 0, 1, 2, 3, 4... Et l'on retrouve nos fameux points de suspension déjà décriés. Cela ne dit rien en effet sur la façon de les représenter et ne nous donne pas plus un critère objectif permettant de savoir si quelque chose est un entier.

La réponse réside une fois encore dans la récursivité. Sans rentrer dans le détail de la théorie et de ses justifications, on peut définir les entiers naturels à l'aide des deux symboles de fonction 0 et S, le premier étant 0-aire, c'est-à-dire sans argument (autrement dit, c'est une constante), le second unaire, c'est-à-dire

admettant un et un seul argument, comme suit :

1. 0 est un entier naturel ;
2. si x est un entier naturel, alors $S(x)$ est un entier naturel.

Il faut encore préciser que rien d'autre que ceci n'est un entier naturel. On peut alors écrire la liste des premiers entiers naturels qu'on aura pu ainsi construire :

$$0, S(0), S(S(0)), S(S(S(0))), S(S(S(S(0)))), \ldots$$

qu'on appelle et note conventionnellement 0, 1, 2, 3 et 4. Cette définition n'est pas autre chose que la définition "bâton" des entiers que l'on apprend à l'école primaire (et qu'on s'empresse d'oublier par la suite) — un symbole S étant un bâton !

Le mérite de cette définition est de nous donner les moyens de progresser, et ainsi de nous permettre de définir l'addition de la manière la plus primitive possible (au sens propre comme au sens figuré). Une fois encore, on va faire appel à une définition récursive, qui va se décomposer en plusieurs sous-cas naturellement induits par la structure des objets auxquels la fonction s'applique. Afin d'en faciliter la compréhension, on notera $\text{somme}(n, m)$ la somme des deux entiers n et m.

1. $\text{somme}(n, 0) = n$;
2. $\text{somme}(n, S(m)) = \text{somme}(S(n), m)$.

Cela revient à dire que pour additionner deux entiers représentés par des tas de bâtons, on fait passer un à un les bâtons d'un tas vers l'autre.

L'intérêt de ces deux définitions est qu'on peut les transcrire telles quelles en un objet représentable et une fonction récursive d'un langage de programmation. Il est bien clair qu'on ne le fait pas en pratique, pas plus à la main que dans un ordinateur, car cette représentation des entiers est connue pour son inefficacité : on utilisera plutôt une représentation binaire ou décimale, avec un algorithme plus compliqué pour effectuer l'addition.

Toutefois, en toute logique, puisqu'il faut bien définir les entiers avant de les représenter, on a bel et bien besoin de ces définitions pour asseoir les fondements des mathématiques. En y ajoutant la définition de la multiplication et l'axiome de récurrence, on obtient une axiomatisation de l'arithmétique, appelée arithmétique de Péano (du nom du mathématicien italien l'ayant posée en 1889) sur laquelle s'appuie l'essentiel des mathématiques : il fallait bien enfin définir les objets sur lesquels on travaillait depuis si longtemps...

Pour fixer encore plus les idées, nous allons conclure par une dernière définition inductive d'objets ayant un très grand intérêt pratique : les *formules du calcul propositionnel* ou *propositions*. Il s'agit des formules les plus simples de la logique, c'est-à-dire celles où les seules variables autorisées sont booléennes, et sans aucun quantificateur. Avant de se préoccuper de leur donner un sens ou de les évaluer, il faut les définir, autrement dit préciser quelle est la *syntaxe* correcte de ces formules. Par analogie, si l'on considère un langage d'expressions arithmétiques, on dira que $3x + 4$ ou $-2xy^{2x}$ sont des expressions bien formées (syntaxiquement correctes), au contraire de $x5 - /2+$, par exemple : on pourra donc leur donner un sens, et les évaluer pour certaines valeurs des variables. Il en est de même pour les propositions, et une façon simple de dire ce qu'est une proposition bien

formée est d'utiliser une définition inductive. On se donne donc un ensemble infini dénombrable de variables $A, B, C, \ldots$, un symbole de fonction unaire $\neg$, les quatre symboles d'opération binaire $\vee$, $\wedge$, $\Rightarrow$, $\Leftrightarrow$, ainsi que les symboles de parenthèses (et), et on pose alors :

1. $A, B, C, \ldots$ sont toutes des formules du calcul propositionnel — dans ce cas précis, on parlera de *formules atomiques* ou d'*atomes* ;
2. si ϕ et ψ sont des formules du calcul propositionnel, alors les cinq expressions $(\neg\phi), (\phi \vee \psi), (\phi \wedge \psi), (\phi \Rightarrow \psi), (\phi \Leftrightarrow \psi)$ sont aussi des formules du calcul propositionnel.

Et il faut ajouter que rien d'autre n'est une formule du calcul propositionnel. Les cinq symboles $\neg$, $\vee$, $\wedge$, $\Rightarrow$, $\Leftrightarrow$ s'appellent les connecteurs logiques de négation ("non"), de disjonction ("ou"), de conjonction (" et"), d'implication ("implique") et d'équivalence ("est équivalent à"), respectivement. Donnons tout de suite quelques exemples de formules :

$$A, (\neg A), (A \wedge B), ((\neg A) \wedge B), (((\neg A) \wedge B) \vee (\neg(B \Leftrightarrow C))), \ldots$$

En pratique, il existe des règles supplémentaires permettant d'utiliser moins de parenthèses, mais cela n'a pas d'importance pour nous. On pourrait également inclure les deux constantes booléennes (Vrai et Faux) dans notre définition, mais cela ne changerait pratiquement rien.

Maintenant que l'on a bien défini la syntaxe de ces formules, on peut leur donner un sens et par exemple les évaluer pour une valeur de vérité donnée des variables propositionnelles (c'est-à-dire des formules atomiques) qu'elles contiennent. Une variable propositionnelle peut prendre pour valeur uniquement vrai ou faux ; si on affecte une certaine valeur à chaque variable d'une proposition, on peut alors en déterminer inductivement sa valeur de vérité. Par exemple, supposant A vraie et B fausse, on en déduit que $((A \wedge (\neg B)) \vee B)$ est alors vraie... On imagine fort bien que l'algorithme d'évaluation correspondant sera récursif et se décomposera en sous-cas induits par notre définition des propositions.

L'intérêt pratique du calcul propositionnel est grand, même s'il peut sembler très simple à première vue (il est certain que les formules couramment employées en mathématique sont bien plus complexes, puisqu'elles font intervenir des variables non booléennes et des quantificateurs). En effet, il peut servir à coder de nombreux problèmes combinatoires, ou encore à représenter les circuits électroniques. Les questions qui se posent généralement sont alors de savoir si une formule est une *tautologie* (c'est-à-dire est vraie pour toutes les valeurs de vérité de ses atomes) ou encore si elle est *satisfiable* (c'est-à-dire est vraie pour au moins une valeur de vérité de ses atomes). Y répondre permet par exemple de vérifier qu'un circuit électronique satisfait bel et bien ses spécifications (un additionneur additionne, un multiplieur multiplie, etc.) ; malheureusement, on ne connaît pas à ce jour d'algorithme efficace pour ce faire...

2.4 Exercices

Exercice 2.1 *Que calculent les fonctions récursives suivantes ?*

1. ```
   int foo(int x)
   {
     return foo(x);
   }
   ```
2. ```
   int Mac_Carthy(int n)
   {
     if (n > 100)
       return n - 10;
     else
       return Mac_Carthy(Mac_Carthy(n + 11));
   }
   ```
3. ```
   int Syracuse(int n)
   {
     if (n <= 1)
       return 1;
     else if (n % 2 == 0)
       return Syracuse(n / 2);
     else
       return Syracuse((3 * n + 1) / 2);
   }
   ```
4. ```
   int g(int m, int n)
   {
     if (m==0) return 1;
     else
       return g(m-1,g(m,n));
   }
   ```

Exercice 2.2 *Programmer en C une fonction récursive qui calcule le pgcd de deux entiers naturels à l'aide de l'algorithme d'Euclide.*

Démontrer par récurrence la propriété suivante : si $a > b$ et si l'appel de $\mathrm{pgcd}(a, b)$ *effectue $k \geq 1$ appels récursifs, alors $a \geq F_{k+1}$ et $b \geq F_k$, où F_k est le k-ième nombre de Fibonacci.*

On déduit de cette propriété le théorème de Lamé :

> **Théorème de Lamé.** *Pour tout entier $k \geq 1$, si $a > b \geq 0$ et $b < F_k$, alors l'invocation de* $\mathrm{pgcd}(a, b)$ *engendre moins de k appels récursifs.*

Montrer que cette borne supérieure est la meilleure possible (toujours dans le cas le pire, bien entendu).

Exercice 2.3 *Programmer en C le tri rapide sur les tableaux d'entiers.*

Exercice 2.4 *Programmer en C le tri par fusion sur les tableaux d'entiers. Calculer la complexité en terme de nombre de comparaisons, et ce pour n de la forme particulière $n = 2^k$ puis pour un n quelconque.*

Chapitre 3

Structures de données élémentaires

En règle générale, résoudre un problème donné à l'aide d'un ordinateur revient à déterminer un algorithme opérant sur une *structure de données* bien choisie. En d'autres termes, il faut structurer le codage de l'information de manière adéquate. Nous allons dans ce chapitre en étudier quelques exemples élémentaires.

3.1 Tableaux

Dans les langages de programmation de haut niveau, on dispose d'un certain nombre de structures de données prédéfinies, ou d'outils qui vont permettre d'en construire de plus complexes. Commençons donc par revenir sur l'une des plus simples, à savoir les tableaux – à une ou plusieurs dimensions.

Trois cas peuvent se présenter en pratique :

1. si l'on souhaite coder un ensemble d'au plus n données de même type, l'entier n étant connu lorsque l'on écrit le programme. L'utilisation d'un tableau est alors toute indiquée car facile d'emploie et efficace (des structures complexes comme les arbres peuvent cependant être nécessaires pour des raison de complexité en temps). Un tableau de 100 entiers se déclarera donc tout simplement comme une variable, globale ou locale selon le cas :

   ```
   int t[100];
   ```

   l'accès à la 47-ième case se fera tout simplement en faisant référence à `t[46]`, les éléments d'un tableau étant numérotés à partir de 0 en C.

2. dans d'autres cas, on ne connaît pas la quantité de données à stocker au moment de la compilation du programme. Il convient alors de gérer soi-même la réservation de place en mémoire nécessaire. On utilise pour cela la fonction `malloc` et le fait qu'en C une variable de type tableau de `truc` est en fait un pointeur de `truc` contenant l'adresse du début de l'espace mémoire réservé. Ainsi, pour créer un tableau de n entiers (`int`), il faut déclarer une variable de type pointeur d'entier, réserver un espace mémoire suffisant pour stocker n entiers, i.e. `sizeof(int)*n` octets, et enfin stocker l'adresse de cet espace dans le pointeur d'entier. Ainsi, la déclaration

suivante permet de définir une tableau de n entiers, la valeur de n étant fournie lors de l'exécution par l'utilisateur :

```
{
  int *t;
  int n;
  printf("Valeur de n ? ");
  scanf("%d",&n);
  t=(int *)malloc(sizeof(int)*n);
...
  free(t);
}
```

On notera la fonction `free` permettant la libération de la mémoire allouée par `malloc` lorsque le tableau `t` n'est plus utile. Une fois alloué, le tableau `t` se manipule exactement comme dans le cas précédent.

3. Le dernier cas pouvant se produire est celui d'un nombre d'éléments à mémoriser qui varie au cours de l'exécution du programme sans que l'on puisse borner raisonnablement le nombre maximal d'éléments à stocker simultanément. On doit alors employer des structures de donnée plus complexe que les tableaux, telles que les listes chaînées que nous étudierons dans le reste de ce chapitre. La gestion de la mémoire est réalisée élément par élément, ce qui apporte une très grande souplesse d'utilisation mais qui rend plus compliquée l'écriture des programmes et l'accès à l'information.

D'autre part, il faut être prudent lorsque l'on emploie des tableaux à cause de leur complexité en terme de mémoire. Ainsi, pour travailler sur des objets dont la taille est connue à l'avance tels que des matrices carrées d'ordre n, l'emploi du tableau s'impose tout naturellement, du moins dans la mesure où l'on ne dispose d'aucune information supplémentaire sur les matrices en question — si l'on savait par exemple que celles-ci sont creuses, les choses seraient alors totalement différentes. Supposons que l'on veuille coder un polynôme à une indéterminée à coefficients dans un anneau donné, tel que l'anneau des entiers relatifs : connaissant à l'avance une borne supérieure D à son degré, on peut employer un tableau monodimensionnel de taille D. Mais cette méthode n'est pas toujours utilisable. Ainsi, si l'on veut maintenant coder des polynômes à n indéterminées dont on supposera les degrés partiels relativement à toutes leurs indéterminées inférieurs à D — c'est-à-dire tels que leurs n degrés en tant que polynômes à une indéterminée restent inférieurs à D — il va nous falloir réserver un espace mémoire égal à D^n fois la taille nécessaire au stockage d'un seul coefficient. Ce qui nous donne, pour n égal à 6 et D à 100, un espace mémoire égal à 10^{12} fois cette taille. Autant dire que cela n'est guère réaliste, alors même que les valeurs choisies pour n et D sont pourtant tout à fait raisonnables ; et en effet, à la compilation du programme C suivant :

```
#define D 100
main()
{
   int polynome[D][D][D][D][D][D];
}
```

on obtient le message d'erreur :

```
  In function ‘main’:
 5: size of array ‘polynome’ is too large
```

Or, il est bien clair qu'on n'a pas besoin d'un tel espace mémoire pour représenter par exemple :

$$X_1^{34}X_5^{21} - 3X_2X_3^4X_6^{13} + 121X_1X_6^{79}$$

ce qui suggère bien que notre décision d'utiliser le tableau comme structure de données était peu judicieuse.

Plus généralement, on peut être incapable de fournir une borne supérieure à la taille des objets auxquels on s'intéresse ; on va alors faire appel à des structures de données dites *dynamiques*[1], c'est-à-dire qui vont pouvoir croître ou diminuer en taille au cours de l'exécution du programme. Les listes constituent notre premier exemple de telles structures de données.

3.2 Listes chaînées

3.2.1 Définition

De manière informelle, la *liste* est une structure de données similaire à un tableau, en ce sens qu'il s'agit d'un arrangement linéaire d'éléments, mais elle en diffère par le fait qu'on accède aux éléments non pas à l'aide d'un indice, mais en en parcourant tous les prédécesseurs, et que sa taille n'est pas fixée initialement, et peut (en théorie du moins) devenir aussi grande que l'on veut à l'exécution.

Un peu plus formellement, on définit la liste comme un objet récursif de la manière suivante : une liste d'éléments de type T est ou bien la *liste vide* NIL, ou bien une liste obtenue en rajoutant en tête d'une liste un élément x de type T, et ce à l'aide d'un *constructeur* de liste CONS :

$$\textit{liste} = \textsc{Nil} \vee \textsc{Cons}(x, \textit{liste})$$

Concrètement, les listes d'éléments de type T sont tout simplement :

$$\begin{aligned}
&\textsc{Nil},\\
&\textsc{Cons}(z, \textsc{Nil}),\\
&\textsc{Cons}(y, \textsc{Cons}(z, \textsc{Nil})),\\
&\textsc{Cons}(x, \textsc{Cons}(y, \textsc{Cons}(z, \textsc{Nil}))),\\
&\vdots
\end{aligned}$$

où x, y et z sont des éléments de type T. En pratique, on utilise une notation abrégée : par exemple, en Standard ML, la liste d'entiers :

$$\textsc{Cons}(1, \textsc{Cons}(2, \textsc{Cons}(3, \textsc{Nil})))$$

est représentée par $[1, 2, 3]$, et la liste vide NIL par $[\,]$. On utilisera ce type d'abréviation dans la suite. On pourra ainsi écrire :

$$\textsc{Cons}(1, [1, 2, 1]) = [1, 1, 2, 1]$$

Sur les listes opèrent deux autres fonctions de base, à savoir :

1. En informatique, l'adjectif *dynamique* fait référence à ce qui passe lors de l'exécution du programme, tandis que l'adjectif *statique* correspond à la phase de compilation.

— TÊTE, qui retourne le premier élément d'une liste ; par exemple :

$$\text{TÊTE}([1, 2, 3, 4]) = 1$$

— QUEUE, qui retourne la liste privée de son élément de tête ; par exemple :

$$\text{QUEUE}([1, 2, 3, 4]) = [2, 3, 4]$$

On notera que ces deux fonctions sont partielles : elles ne sont pas définies pour NIL (et doivent alors retourner une erreur).

On peut se servir des listes pour de très nombreux usages. Elles sont ainsi tout à fait appropriées pour coder les polynômes ; par exemple, la liste d'entiers $[0, 1, 0, -2, 4]$ représentera le polynôme à une indéterminée X :

$$4X^4 - 2X^3 + X$$

De même, on les emploiera pour coder les grands nombres, et aussi comme composants de base de structures de données plus complexes (pour coder les matrices creuses, par exemple).

On remarquera toutefois que les listes ne présentent pas que des avantages sur les tableaux : l'accès au k-ième élément d'une liste nécessite un parcours le long de k nœuds (ou $k - 1$ appels à la fonction QUEUE), tandis que lire `t[k]` ne requiert qu'une seule opération, et s'exécute donc en temps constant.

Pour conclure, mentionnons qu'il existe encore d'autres types de listes un peu plus compliquées, telles que :

— les *listes doublement chaînées*, où chaque nœud pointe non seulement vers son successeur (champ `suivant` de la structure `struct noeud`), mais aussi vers son prédécesseur (voir figure 3.1) ;

— les *listes circulaires*, dont le dernier élément pointe vers le premier (voir figure 3.2) ;

— les *listes circulaires doublement chaînées* ;

— etc.

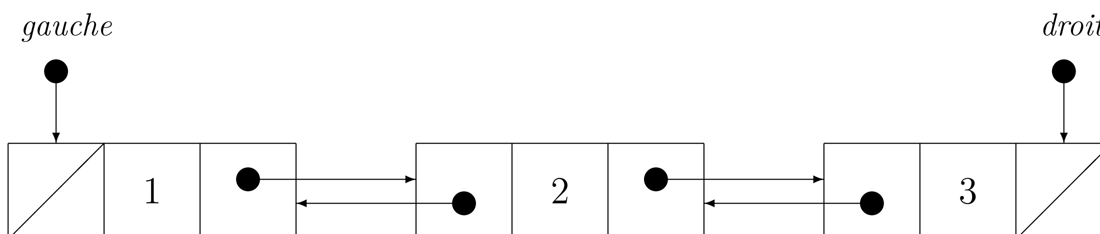

FIGURE 3.1 – Représentation graphique de la liste doublement chaînée $[1, 2, 3]$

3.2.2 implémentation en C

La liste est une structure de données prédéfinie dans certains langages de programmation, tels que LISP [2], ML, PROLOG, etc. Dans des langages comme PAS-

2. La liste est même la structure de données fondamentale dans ce langage, dont le nom est un acronyme dérivé de *LISt Processing*.

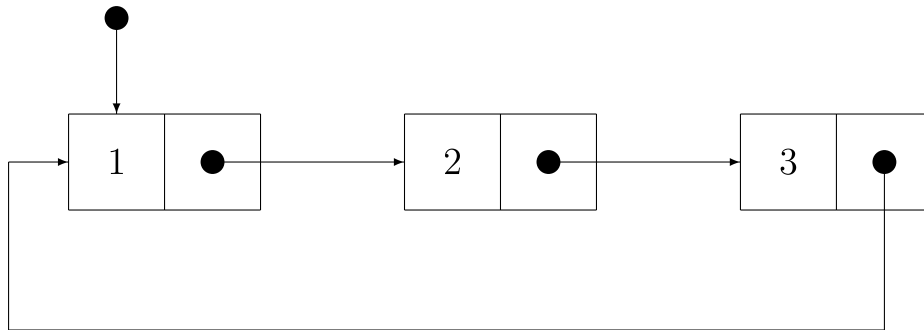

FIGURE 3.2 – Représentation graphique de la liste circulaire $[1, 2, 3]$

CAL, JAVA ou C, il faut l'implémenter soi-même. En C, on peut définir le type `Liste` des listes d'éléments de type `int` comme suit, à l'aide de pointeurs :

```
struct cellule{
  int val;
  struct cellule *suiv;
};

typedef struct cellule *Liste;
```

On parle alors de *liste chaînée* pour faire référence à ce type d'implémentation basée sur les pointeurs. Cette notion correspond intuitivement à la représentation de la figure 3.3 où chaque flèche représente un pointeur.

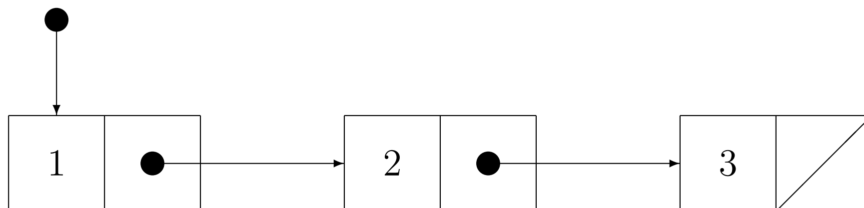

FIGURE 3.3 – Représentation graphique de la liste chaînée $[1, 2, 3]$

Nous venons de définir un nouveau type, `Liste`, qui est formellement un pointeur de `struct cellule`. Il est important de bien comprendre que lorsque les cellules qui constituent une liste sont correctement chaînées entre elles, il suffit d'avoir un pointeur vers la première cellule pour être ensuite capable d'accéder à tous les éléments.

La liste vide est simplement codée par un pointeur vers la constante prédéfinie `NULL`. De plus, le dernier pointeur sera lui aussi affecté à cette valeur particulière afin de bien indiquer la fin d'une liste. Les fonctions de base `cons`, `tete` et `queue` s'écrivent alors de la manière suivante :

```
Liste cons(int v, Liste L)
{
```

```
  Liste nouv;
  nouv=(Liste) malloc(sizeof(struct cellule));
  nouv->val=v;
  nouv->suiv=L;
  return nouv;
}

int tete(Liste L)
{
  if (L==NULL) {printf("tete(NULL)\n"); exit(-1);}
  return L->val;
}

Liste queue(Liste L)
{
  if (L==NULL) {printf("queue(NULL)\n"); exit(-1);}
  return L->suiv;
}
```

Seule la fonction `cons` mérite quelques explications ; elle prend en argument un entier `v` que l'on souhaite ajouter en tête de la liste préexistante `L`. Pour cela on crée une nouvelle structure cellule (“`nouv=(Liste) malloc(sizeof(struct cellule))`”) où l'on place `v` (“`nouv->val=v`”) et que l'on fait pointer sur `L` (“`nouv->suiv=L`”). Finalement la nouvelle liste est simplement un pointeur sur la nouvelle cellule (“`return nouv`”).

3.2.3 Opérations courantes sur les listes

Nous allons maintenant décrire comment réaliser les principales fonctions de manipulation de listes. Pour certaines d'entre elles, nous considérerons deux implémentations différentes, l'une dite *fonctionnelle* fondée sur une programmation récursive et l'autre dite *itérative*, par opposition avec la première approche et utilisant des boucles pour et tant que.. Selon les cas, une méthode est souvent plus indiquée que l'autre, comme nous allons nous en rendre compte.

Impression des éléments d'une liste

```
void print_Liste1(Liste L) // version ITERATIVE
{
  Liste P=L;
  while (P!=NULL)
    {printf("%d ",P->val);
     P=P->suiv;}
  printf("\n");
}


void print_Liste2(Liste L) // version FONCTIONNELLE
{
```

```
  if (L==NULL) printf("\n");
  else
    {printf("%d ",tete(L));
     print_Liste2(queue(L));}
}
```

Recherche d'un élément dans une liste

```
int element1(int v, Liste L) // version ITERATIVE
{
  Liste P=L;
  while (P!=NULL)
    {if (P->val==v) return 1;
     P=P->suiv;}
  return 0;
}

int element2(int v, Liste L) // version FONCTIONNELLE
{
  if (L==NULL) return 0;
  return ((v==tete(L))||element2(v,queue(L)));
}
```

calcul de la longueur d'un liste

```
int length(Liste L) // version FONCTIONNELLE
{
  if (L==NULL) return 0;
  else return 1+length(queue(L));
}
```

Insertion d'un élément à la k-ième place d'une liste

```
Liste ajoute1(int v, int k, Liste L) // version FONCTIONNELLE
{
  if (k==0) return cons(v,L);
  return cons(tete(L),ajoute1(v,k-1,queue(L)));
}

void ajoute2(int v, int k, Liste *L) // version ITERATIVE
{
  int i;
  Liste P,nouv;

  if (k==0) *L=cons(v,*L);
  else
    {if (k>length(*L)) {printf("Ajoute impossible\n"); exit(-1);}
     P=*L;
     for(i=0;i<k-1;i++)
```

```
       P=P->suiv;
     nouv=(Liste) malloc(sizeof(struct cellule));
     nouv->val=v;
     nouv->suiv=P->suiv;
     P->suiv=nouv;}
}
```

Suppression de toutes les occurrences d'un élément dans une liste

```
Liste enleve1(int v, Liste L) // version FONCTIONNELLE
{
  if (L==NULL) return NULL;
  if (tete(L)==v) return enleve1(v,queue(L));
  return cons(tete(L),enleve1(v,queue(L)));
}
```

La fonction précédente a l'avantage d'être simple à écrire mais elle présente un inconvénient lié au style de programmation récursif. En effet, chaque appel à la fonction `cons` crée une nouvelle cellule. Au cours de l'exécution du programme, il y a donc beaucoup de cellules qui sont créées puis en quelque sorte perdues car devenues inaccessibles en mémoire. Certains langages modernes comme JAVA sont capables de repérer les structures allouées par `malloc` et qui ne servent plus à rien ; en C ce n'est pas le cas et par conséquent il faut indiquer explicitement quelles structures sont devenues obsolètes, au moyen de la commande `free`.

La fonction suivante montre comment libérer les cellules qui ne servent plus, afin de ne pas saturer la mémoire de données inutiles. On peut se représenter la commande `free` comme l'inverse de la commande `malloc` ; cette dernière réserve un quantité d'octets spécifiées et renvoie un pointeur alors que `free` prend comme unique argument ce pointeur et libère la mémoire associée. On remarquera qu'il n'est pas nécessaire de fournir la taille du bloc à libérer à la commande `free` ; en effet, le système d'allocation de mémoire tient à jour une "cartographie" de la mémoire lui permettant notamment de savoir la taille d'un bloc débutant à une adresse donnée.

```
void enleve2(int v, Liste *L) // version ITERATIVE
{ Liste P;
  if (*L!=NULL)
    {if ((*L)->val==v)
       {P=*L;
        *L=(*L)->suiv;
        free(P);
        enleve2(v,L);}
     else
       enleve2(v,&((*L)->suiv));}
}
```

Concaténation de deux listes

```
Liste concat(Liste L1,Liste L2) // version FONCTIONNELLE
```

```
{
  if (L1==NULL) return L2;
  else return cons(tete(L1),concat(queue(L1),L2));
}
```

Inversion de l'ordre des éléments d'une liste

```
Liste renverse1(Liste L)
{
  if (L==NULL) return L;
  else return concat(renverse1(queue(L)),cons(tete(L),NULL));
}
```

Cette fonction inverse une liste en $\Theta(n^2)$ opérations si n est la longueur de la liste car on fait de l'ordre de n appels récursifs à `renverse1` qui exécute à chaque fois une concaténation de liste. La fonction suivante permet d'inverser une liste en seulement $\Theta(n)$ opérations :

```
Liste renverse2(Liste L, Liste Acc)
{
  if (L==NULL) return Acc;
  else return renverse2(queue(L),cons(tete(L),Acc));
}
```

Suppression de la totalité d'une liste

```
void freeliste(Liste *L)
{
  if ((*L)!=NULL)
    {
      freeliste((*L)->suiv);
      free(*L);
      *L=NULL;
    }
}
```

Il faut bien faire attention dans la fonction précédente à l'ordre des instructions `freeliste` et `free`, l'accès à un cellule n'étant plus possible dés lors qu'on l'a libérée.

3.3 Piles et files

Nous présentons ici brièvement deux structures de données fort similaires aux listes et employées intensivement en informatique : les piles et les files. Comme les listes, ce sont des structures de données dynamiques, linéaires, et elles sont caractérisées par la donnée des quatre fonctions ou valeurs suivantes :

— un objet (pile ou file) vide, homologue de Nil ;
— un constructeur qui insère un élément (de type T) en tête de la pile ou la file, tout comme Cons ;

— une fonction, similaire à TÊTE, et qui retourne l'élément de tête ;
— une fonction de suppression d'un élément, telle que QUEUE, et qui fait justement toute la différence entre les piles et les files : la stratégie de suppression est en effet LIFO dans le cas des piles, c'est-à-dire *Last In, First Out* (dernier entré, premier sorti) : l'élément inséré le plus récemment est supprimé, alors qu'elle est FIFO dans le cas des files, soit *First In, First Out* (premier entré, premier sorti) : l'élément le plus ancien est candidat à la suppression.

Les piles sont donc tout à fait isomorphes aux listes, si ce n'est qu'on a traditionnellement tendance à les imaginer verticales, et qu'on donne aux opérations d'insertion, de lecture de la tête et de suppression d'un élément les noms de EMPILER, SOMMET, DÉPILER (respectivement PUSH, TOP et POP en anglais). Leur codage par des listes (ou des listes chaînées) s'effectue de manière immédiate.

Leur importance en informatique provient de leur utilisation systématique pour gérer les environnements d'exécution des fonctions : lors de l'exécution d'un programme, l'appel d'une fonction entraîne la création d'un environnement temporaire (variables locales), stocké en mémoire au sommet d'une pile (dite *pile d'exécution*), et qui sera dépilé en sortie de cette fonction.

Quant aux files, ce sont en fait des files d'attente, semblables à ce qu'on trouve — ou plutôt ce qu'on devrait trouver — à un guichet quelconque, et elles sont par exemple employées pour stocker des messages ou des processus en attente de traitement. Leur codage par des listes est un peu plus subtil et moins efficace que pour les piles : en effet, par un codage naïf, les fonctions d'insertion et de suppression ne pourront être toutes deux en temps constant — l'une des deux effectuera un balayage complet de la liste. En pratique, on aura intérêt à utiliser des listes chaînées munies d'un pointeur vers le dernier élément, comme indiqué sur la figure 3.4. Nous verrons dans le chapitre sur les graphes des applications importantes des files d'attente.

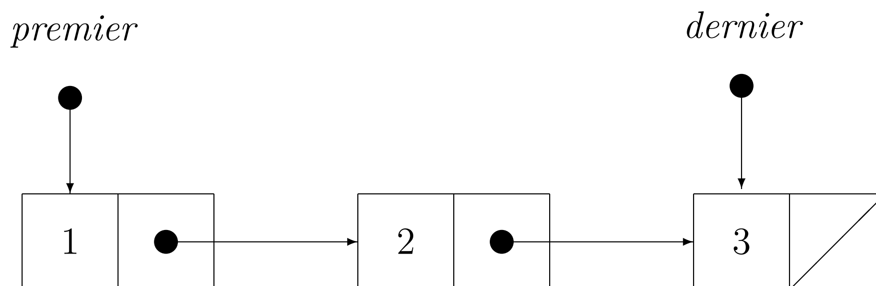

FIGURE 3.4 – Représentation d'une file à l'aide d'une liste chaînée

3.4 Application : évaluation des expressions arithmétiques préfixées

Afin d'illustrer par un exemple concret les notions, très proches, de liste et de pile, considérons le problème suivant : on souhaite écrire un programme capable

d'évaluer une expression arithmétique fournie en notation *préfixée* et ne contenant que des entiers et les symboles d'addition, de soustraction et de multiplication (noté `x` pour des raisons techniques). L'écriture préfixée consiste à indiquer l'opérateur avant ses opérandes. Ainsi $1+2$ sera noté "`+ 1 2`", $3\times(4+5)$ s'écrit "`x 3 + 4 5`" et $3\times4+5$ devient "`+ x 3 4 5`". On notera que la notation préfixée supprime toute ambiguïté concernant les priorités entre opérateurs et que par conséquent les parenthèses sont inutiles.

Afin d'évaluer une expression, nous allons stocker les différents symboles qui la composent dans une pile. L'algorithme d'évaluation est récursif ; si le sommet de la pile contient un entier alors on le retire de la pile et le résultat de l'évaluation est cet entier. Si le sommet de la pile est un opérateur, on le dépile, on évalue une première fois la pile restante (ce qui aura pour effet d'évaluer et de retirer de la pile tout le premier terme), on refait de même pour évaluer et dépiler le second terme et enfin on renvoie le résultat.

Prenons l'exemple de l'expression $(2+3)\times 4 - 4\times 2$, notée "`- x + 2 3 4 x 4 2`". Si on empile ces symboles, on a le premier signe de soustraction au sommet de la pile. On le retire et on évalue récursivement la pile deux fois de suite. Si le programme fonctionne correctement, il va une première fois opérer sur la pile "`x + 2 3 4 x 4 2`", en extraire la première sous expression bien formée ("`x + 2 3 4`") et renvoyer son évaluation, i.e. 20. on va ensuite de nouveau évaluer et dépiler une expression complète, ici le reste de la pile, i.e. "`x 4 2`". Le résultat 8 est renvoyé. Le programme calcule donc finalement $20 - 8 = 12$.

Le programme suivant implémente cet algorithme. La pile est constituée de cellule contenant un entier `typ` indiquant si l'on a affaire à un opérateur ou à un entier, ainsi qu'un entier `val` et un caractère `op` qui ne seront bien entendu pas utilisés simultanément.

On notera que la majorité des fonctions reçoivent les variables de type `PILE` par adresse afin de pouvoir les modifier. On notera également l'emploi de la fonction `free` lorsque l'on dépile un élément de la pile afin de libérer la mémoire précédemment allouée par `malloc`. Notez enfin la possibilité d'imprimer une expression contenue dans une pile de manière classique (notation dite *infixée*) en utilisant un algorithme récursif très semblable à celui utilisé pour l'évaluation (voir la fonction `printinfixe`).

```
#include <stdio.h>
#include <stdlib.h>

#define typval 1
#define typop 2

struct pile
 {int typ;
  int val;
  char op;
  struct pile *suiv;};

typedef struct pile *PILE;
```

```
void empileval(int v, PILE *P)
 {PILE nouv;
  nouv=(PILE)malloc(sizeof(struct pile));
  nouv->typ=typval;
  nouv->val=v;
  nouv->suiv=*P;
  *P=nouv;}

void empileop(char oper, PILE *P)
 {PILE nouv;
  nouv=(PILE)malloc(sizeof(struct pile));
  nouv->typ=typop;
  nouv->op=oper;
  nouv->suiv=*P;
  *P=nouv;}

void depile(PILE *P)
 {PILE temp;
  temp=*P;
  *P=(*P)->suiv;
  free(temp);}

int typsommet(PILE P)
 {return P->typ;}

int valsommet(PILE *P)
 {int v=(*P)->val;
  depile(P);
  return v;}

char opsommet(PILE *P)
 {char oper=(*P)->op;
  depile(P);
  return oper;}

void printpile(PILE P)
 {if (P!=NULL)
    {if (typsommet(P)==typval)
       printf("%d ",P->val);
     else
       printf("%c ",P->op);
     printpile(P->suiv);}}

void empilesymbole(char *st, PILE *P)
 {if ((st[0]=='+')||(st[0]=='-')||(st[0]=='x'))
    empileop(st[0],P);
  else
    empileval(atoi(st),P);}
```

```
int evaluepile(PILE *P)
{
  if (typsommet(*P)==typval)
    return valsommet(P);
  else
    if (typsommet(*P)==typop)
      switch (opsommet(P))
       {case '+':return evaluepile(P)+evaluepile(P);
        case '-':return evaluepile(P)-evaluepile(P);
        case 'x':return evaluepile(P)*evaluepile(P);}
}

void printinfixe(PILE *P)
{
  char oper;
  if (typsommet(*P)==typval)
    printf("%d",valsommet(P));
  else
    if (typsommet(*P)==typop)
      {oper=opsommet(P);
       printf("(");
       printinfixe(P);
       printf("%c",oper);
       printinfixe(P);
       printf(")");}
}

int main(int argc, char *args[])
{
  int i;
  PILE P=NULL;
  for(i=argc-1;i>0;i--)
    empilesymbole(args[i],&P);
  printf("Etat pile : ");
  printpile(P);
  printf("\nExpression: ");
  printinfixe(&P);
  for(i=argc-1;i>0;i--)
    empilesymbole(args[i],&P);
  printf("\nEvaluation: %d\n",evaluepile(&P));
  exit(0);
}
```

Voici un exemple d'exécution de ce programme :

```
>evalu - x + 2 3 4 x 4 2
Etat pile : - x + 2 3 4 x 4 2
```

```
Expression: (((2+3)x4)-(4x2))
Evaluation: 12
```

3.5 Exercices

Exercice 3.1 *Expliquer comment coder les listes, piles et files de taille bornée au moyen de tableaux.*

Exercice 3.2 *Implémenter des files au moyen de listes chaînées comme indiquer dans la section 3.3.*

Exercice 3.3 *Expliquer comment représenter les polynômes à plusieurs indéterminées à l'aide de listes. Programmer en C les principales opérations sur les polynômes.*

Exercice 3.4 *Reprendre et compléter le programme d'évaluation d'expressions infixées afin de gérer les fonctions (*sin, cos,*…). Introduire des tests permettant de gérer d'éventuelles erreurs de syntaxe dans l'expression préfixée (par exemple, indiquer que* `+ 1 2 3` *n'est pas valide).*

Chapitre 4

Recherche en table

Un problème de base auquel sont confrontés les informaticiens est de retrouver aussi vite que possible l'information rangée en mémoire. Ce problème se pose typiquement lors de la consultation d'un annuaire électronique ou d'une *table des symboles*[1] générée par un compilateur. Nous allons ici aborder ce sujet en présentant divers algorithmes de recherche dans une table.

4.1 Adressage direct

En premier lieu, commençons par définir ce qu'est une *table* : il s'agit d'une structure de données isomorphe à un ensemble de couples (*clé*, *information*) tel que chaque *clé* n'apparaisse qu'une fois dans la table, et sur lequel opèrent les trois fonctions suivantes :

— une fonction de *recherche* d'un élément, qui retourne l'information associée à toute clé de la table, ou un indicateur de recherche infructueuse si la clé n'est pas dans le table ;

— une fonction d'*insertion* d'un élément, qui insère dans la table un couple (*clé*, *information*), ou bien se contente de modifier l'information associée à la clé si celle-ci y est déjà présente ;

— une fonction de *suppression* d'un élément, qui enlève de la table le couple (*clé*, *information*) associé à la clé passée en paramètre si celle-ci s'y trouve.

Bien évidemment, on disposera également d'une fonction de création de la table.

On notera qu'une table est isomorphe au graphe d'une fonction, puisqu'à une clé ne peut être associée qu'une seule information. Plutôt que de table, on parle encore parfois de *dictionnaire*.

Il existe plusieurs manières d'implémenter les tables, dont aucune n'est optimale en général : tout dépend en fait de la situation à laquelle on est confronté. Lorsque l'univers de clés, que l'on notera dorénavant U, est relativement petit, on peut employer la méthode fort simple de l'*adressage direct*. L'idée est la suivante : on spécifie une bijection h de l'ensemble U des clés sur l'intervalle $[0, m-1]$ des m premiers entiers, où m est le cardinal de l'ensemble U. La table sera codée par un tableau T de taille m (et d'indice variant entre 0 et $m-1$), les éléments du tableau T fournissant l'information correspondante. Plus précisément, pour toute

1. Il s'agit de la table qui contient toutes les informations utiles associées à chaque identificateur rencontré dans le programme que l'on compile.

clé $\in U$, $T[h(\textit{clé})]$ indiquera si la clé est bel et bien présente dans la table, et quelle est alors l'information associée. On peut coder ceci de différentes façons :

- à l'aide d'un tableau T de pointeurs vers des éléments de type *information*, qui seront nuls si la clé correspondante n'est pas dans la table ;
- en employant un tableau dont les éléments sont des enregistrements à deux champs : l'un booléen indiquant la présence de la clé, l'autre fournissant l'information ;
- en utilisant en même temps un tableau de booléens et un tableau d'éléments de type *information*.

Les opérations de recherche, d'insertion et de suppression d'un élément sont alors triviales ; par exemple dans le premier cas, on aura :

Algorithme 4.1 TABLE-À-ADRESSAGE-DIRECT

```
1  RECHERCHE-ADRESSAGE-DIRECT(T, clé)
2      retourner T[h(clé)]

3  INSERTION-ADRESSAGE-DIRECT(T, clé, information)
4      T[h(clé)] ← information

5  SUPPRESSION-ADRESSAGE-DIRECT(T, clé)
6      T[h(clé)] ← NIL
```

Ces trois opérations s'effectuent clairement en temps constant (on dit encore : en $O(1)$), à condition bien sûr que la fonction h se calcule elle-même en temps constant. En pratique, elle sera choisie suffisamment simple pour que cela ne pose pas de problème, comme par exemple dans les deux cas suivants :

- l'univers U des clés est exactement l'intervalle $[0, m-1]$, et h est donc l'identité ;
- U est l'ensemble des identificateurs constitués d'une lettre suivie d'un chiffre décimal (tels que A0, B7 ou T6), pour lequel on laisse en exercice la spécification d'une bijection h de U sur $[0, 259]$.

La seule opération un peu coûteuse est la création de la table, qui s'effectue en temps linéaire : on aura à réaliser m initialisations.

Cette méthode est néanmoins loin de constituer la panacée, notamment si l'ensemble U des clés est très grand. Par exemple, si les clés sont des noms de huit lettres, $\mathrm{Card}(U)$ vaut 26^8, soit environ 2.10^{11} ; la table est donc impossible à stocker en mémoire et difficilement sur disque. Nous allons dorénavant nous affranchir de l'hypothèse de petitesse du cardinal de U.

4.2 Recherche séquentielle

On supposera ici que U peut être aussi grand que l'on veut, mais que le nombre n de clés susceptibles de se trouver en même temps dans la table reste relativement petit. On peut alors choisir une représentation très simple de la table comme un ensemble de couples (*clé*, *information*), dont on parcourra les éléments rangés linéairement de manière *séquentielle* lors de la recherche d'une information.

Plus précisément, une première implémentation consiste à se donner un tableau de taille m, où m est choisi tel qu'il soit toujours (espéré) supérieur au nombre n d'entrées présentes en table ; les éléments du tableau sont des couples (*clé*, *information*) — on peut également travailler avec deux tableaux. Les n entrées correspondront alors aux n premiers indices du tableau.

Rechercher un élément consistera alors à parcourir le tableau jusqu'à trouver la clé ou la fin de la partie pertinente du tableau (les n premiers éléments). Une telle recherche va donc nécessiter :

— i comparaisons en cas de recherche fructueuse, si l'élément recherché est situé au i-ème rang dans le tableau : i peut donc valoir toute valeur comprise entre 1 et n ; en supposant alors que tous les éléments sont également susceptibles de faire l'objet d'une requête, le nombre moyen de comparaisons nécessaires sera donc égal à :

$$\frac{1}{n}(1 + 2 + \cdots + n) = \frac{n+1}{2}$$

— $n + 1$ comparaisons en cas de recherche infructueuse.

L'opération d'insertion va elle-même faire appel à une recherche préalable, afin d'insérer le nouvel élément à l'endroit correct, c'est-à-dire s'il existe dans la table un élément présent avec la même clé, en lieu et place de celui-ci, et sinon à la suite des n premiers éléments (on n'omettra pas de tester que la table n'est pas pleine). On retrouve donc une complexité comparable à celle de l'opération de recherche. L'opération de suppression d'un élément est assez similaire : elle va consister en une recherche suivie en cas de succès d'un décalage des éléments situés plus loin dans la table, ou d'un remplacement de l'élément à détruire par le dernier présent dans la table. Bref, les trois opérations sont toujours linéaires en fonction du nombre de clés dans la table, et c'est pour cela qu'on appelle également cette méthode recherche *linéaire*.

On peut alternativement implémenter la recherche séquentielle en table à l'aide d'une liste de couples (*clé*, *information*) : on n'a pas alors de problème de limitation de taille, et les trois opérations de base se programment très naturellement. On parle dans ce cas de liste *associative*.

Enfin, notons qu'il est possible de gérer la table de manière un peu plus subtile si l'on dispose d'informations supplémentaires sur la fréquence d'accès à certaines clés : on aura intérêt à placer au tout début les plus fréquemment lues.

4.3 Recherche dichotomique

La méthode de recherche précédente n'est guère utilisable si le nombre de clés présentes en table est élevé, un temps d'exécution linéaire devenant alors beaucoup trop long pour de nombreuses applications. On peut cependant améliorer grandement cette complexité en conservant une même représentation de la table par un tableau, mais en imposant que les clés soient rangées dans l'ordre croissant (il faut bien sûr que U soit muni d'une relation d'ordre total).

Dans ces conditions, on peut effectuer une recherche dite *dichotomique* ou *binaire* d'un élément dans la table. L'idée est tout simplement d'appliquer l'approche *diviser pour régner* : on compare la clé recherchée avec celle située au milieu du

tableau [2] ; si on l'a trouvée, c'est terminé, sinon on recommence récursivement la recherche sur la moitié adéquate du tableau (inférieure ou supérieure). Plus formellement, cela nous donne le pseudo-code suivant :

Algorithme 4.2 RECHERCHE-DICHOT(T, v, p, r)
1 **si** $p > r$ **alors retourner** *recherche_infructueuse*
2 $q \leftarrow \lfloor (p+r)/2 \rfloor$
3 **si** $v = T[q].\mathit{clé}$ **alors retourner** $T[q].\mathit{information}$
4 **si** $v < T[q].\mathit{clé}$
5   **alors retourner** RECHERCHE-DICHOT$(T, v, p, q-1)$
6   **sinon retourner** RECHERCHE-DICHOT$(T, v, q+1, r)$

T représentant le tableau trié, v la clé recherchée, p et r les indices extrêmes du sous-tableau dans lequel on fait la recherche.

Il nous reste maintenant à analyser la complexité de cet algorithme, c'est-à-dire à évaluer le nombre total d'appels récursifs à la fonction RECHERCHE-DICHOT nécessaires pour trouver l'information, ou encore le nombre de comparaisons du type $v < T[q].\mathit{clé}$ effectuées en tout. Si $n = r - p + 1$ est la taille de la table, il n'est pas trop malaisé de se convaincre que ce nombre est majoré par C_n qui satisfait la relation de récurrence suivante :

$$C_n = C_{\lfloor n/2 \rfloor} + 1$$

avec $C_1 = 1$. Si n est une puissance de 2 : $n = 2^k$, on en déduit :

$$C_{2^k} = C_{2^{k-1}} + 1 = C_{2^{k-2}} + 2 = \cdots = C_{2^0} + k = k + 1$$

et donc $C_n = \log_2 n + 1$. Si n n'est pas une puissance de 2, cette majoration reste valide, et on déduit finalement que le nombre d'appels récursifs est inférieur ou égal à $\log_2 n + 1$.

La recherche dichotomique est donc beaucoup plus rapide que la recherche linéaire. On notera toutefois que l'insertion ou la suppression d'un élément nécessitent le décalage d'environ $n/2$ éléments en table, en moyenne. Ainsi, cette méthode n'est réellement intéressante que lorsqu'on modifie peu la table triée. On verra toutefois au chapitre suivant une structure de données permettant de résoudre ce problème.

On peut enfin se demander si la recherche dichotomique mime notre façon intuitive de consulter un dictionnaire ou une encyclopédie. En y réfléchissant bien, on se rend compte que la réponse est négative : ainsi, on aura tendance à aller chercher plutôt vers la fin qu'au milieu une entrée commençant par la lettre V. Cette technique est applicable dans notre cas, lorsqu'on est capable de coder une fonction d'interpolation servant à "deviner" l'endroit approximatif où une clé doit se trouver dans une table donnée. On parle alors de *recherche par interpolation*.

Par exemple, si les clés sont numériques, on peut remplacer dans l'algorithme 4.2, à la ligne 2, l'expression $(p+r)/2$, qui fournit la médiane du sous-tableau compris entre les indices p et r, et dérive donc elle-même de :

$$p + \frac{1}{2}(r - p)$$

2. Précisément, il va s'agir de l'élément médian si le tableau a un nombre impair d'éléments, et de l'élément médian inférieur s'il en a un nombre pair.

par l'expression :

$$p + \frac{v - T[p].clé}{T[r].clé - T[p].clé}(r - p)$$

Bien sûr il ne faudrait pas oublier d'apporter d'autres modifications au code pour obtenir la fonction de recherche souhaitée.

On peut alors prouver que si les clés sont distribuées de façon *uniforme*, une recherche par interpolation ne requerra pas plus de $\log_2 \log_2 n + 1$ appels récursifs en moyenne. Comme $\log \log n$ est une fonction à croissance extrêmement lente ($\log_2 \log_2 10^9 < 5$), quasi-constante en pratique, on aura tout intérêt à choisir cette méthode pour travailler sur de très gros fichiers triés (ou quand l'on doit faire des accès externes très coûteux). Cependant, dès que la distribution cesse d'être uniforme, on perd cette complexité en $O(\log \log n)$.

Notons d'autre part que la recherche dichotomique est possible dans un tableau car il est très facile d'accéder à un élément situé "au milieu" d'un tableau. Par contre, si l'on souhaite utiliser des listes chaînées afin de réduire la complexité des opérations d'insertion et de suppression, on se rend compte qui n'est alors plus possible d'accéder ainsi au milieu de la liste sans en parcourir initialement la moitié. Cet exemple nous montre un des avantages des tableaux sur les listes dans certaines situations.

4.4 Tables de hachage

Pour conclure, nous allons présenter une dernière méthode de recherche en table très couramment utilisée par les informaticiens, et particulièrement efficace en pratique pour effectuer les trois opérations de recherche, d'insertion et de suppression d'un élément. L'idée est de s'inspirer de l'adressage direct, pour lequel ces trois opérations s'exécutent en $O(1)$.

Le problème que l'on rencontre avec l'adressage direct est que l'univers U des clés doit être très petit, afin de pouvoir être mis en bijection avec l'intervalle $[0, m-1]$ par h, tout en rendant possible la réservation d'un tableau de taille m. On peut néanmoins s'affranchir de cette condition en n'imposant plus que h soit une bijection mais seulement une surjection. On dit alors que h est une *fonction de hachage*, et la table une *table de hachage*. Le problème qui peut alors apparaître est le cas de deux clés c_1 et c_2 telles que $h(c_1) = h(c_2)$, ce qui s'appelle une *collision*. Il va sans dire que l'on peut même avoir une collision entre plus de deux clés. Une solution simple consiste à utiliser un tableau de *listes* de couples (*clé*, *information*), plutôt qu'un tableau de tels couples, comme indiqué dans la figure 4.1 (avec des listes chaînées).

Les opérations sur la table s'implémentent alors tout naturellement en se servant des opérations de recherche, d'insertion et de suppression d'un élément dans une liste (exercice 4.3). On se ramène en fait aux opérations correspondantes de recherche séquentielle sur les listes.

Analysons maintenant la complexité de la recherche : on suppose s'être donné un tableau de taille m, et que le nombre de clés en table est n. On définit alors le *facteur de remplissage* de la table comme étant $\rho = n/m$. Il est bon de noter que ρ peut varier entre 0 et l'infini, ou autrement dit, qu'il n'y a pas de limite

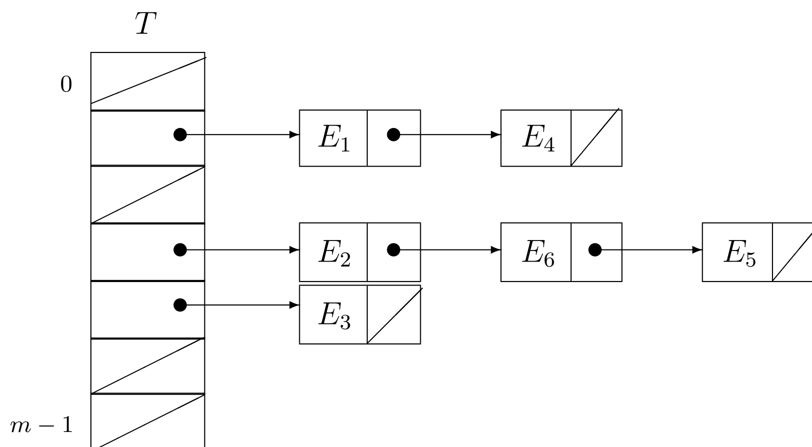

FIGURE 4.1 – Table de hachage avec listes de collisions

supérieure au nombre d'éléments susceptibles de se trouver en table, ce qui est un autre avantage de la méthode.

Dans le pire des cas, il y a collisions entre les n clés et on n'a qu'une seule liste de taille n à parcourir : on retombe ni plus ni moins sur la recherche séquentielle, et une complexité en $\Theta(n)$. En moyenne, par contre, on peut supposer disposer d'une fonction de hachage h qui va répartir les clés uniformément sur les m indices. Ce qui signifie donc que la longueur moyenne d'une liste sera égale à ρ, et on en déduit que la recherche d'un élément ne nécessitera pas plus de $1+\rho$ comparaisons en moyenne. Si donc on est capable d'estimer à l'avance la valeur de n, on peut alors dimensionner la table pour obtenir une recherche en temps constant bien déterminé.

Le dernier point qui reste à préciser est la fonction de hachage elle-même : quelle fonction choisir ? Il n'y a pas de méthode générale, et nous allons seulement citer deux exemples simples, mais qui donnent entière satisfaction en pratique.

On supposera tout d'abord que les clés appartiennent à l'ensemble des entiers naturels : si ce n'est pas le cas, il faudra en premier lieu spécifier une fonction injective de U dans cet ensemble. Pour le cas usuel des chaînes de caractères, il suffira de les réinterpréter comme des entiers codés dans une base particulière : souvent, on choisit le code ASCII des lettres dans la base 128. À partir de là, on peut prendre comme fonction de hachage le reste de la division euclidienne, soit :

$$h(c) = c \bmod m$$

où m est la taille de la table. Le choix de m n'est en général pas complètement indifférent : ainsi, si $m = 2^k$, ce hachage revient à ne considérer que les k bits de poids le plus faible de la clé, ce qui n'est pas forcément très judicieux en pratique. Une bonne solution consiste à prendre m premier et assez éloigné des puissances de deux.

Une autre fonction de hachage fréquemment utilisée est la suivante :

$$h(c) = \lfloor m(cx - \lfloor cx \rfloor) \rfloor$$

où x est une constante réelle comprise strictement entre 0 et 1. Cela revient à prendre la partie entière du produit de m (la taille de la table) par la partie

fractionnaire de cx. Le choix de x n'est pas non plus complètement indifférent ; pour des raisons mathématiques non triviales, on peut prouver que $x = (\sqrt{5}-1)/2$ est une valeur qui “marche bien” en pratique — on parle dans ce cas du hachage de Fibonacci.

4.5 Récapitulatif

En conclusion, le tableau suivant résume les complexité spatiales et temporelles des différentes méthodes de recherche en table. Nous avons de plus ajouté une méthode à base d'arbres binaires de recherche (ABR) qui sera vue dans le prochain chapitre.

Méthode utilisée	Complexité spatiale	Complexité de la recherche	Complexité de la insertion	Complexité de la suppression
Adressage direct	$\theta(m)$	$\theta(1)$	$\theta(1)$	$\theta(1)$
Recherche sequentielle	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(n)$
Recherche dichotomique	$\theta(n)$	$\theta(\log(n))$	$\theta(n)$	$\theta(n)$
Table de hachage	$\theta(m+n)$	$\theta(n/m)$	$\theta(1)$	$\theta(n/m)$
ABR (cas le pire)	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
ABR (en moyenne)	$\theta(n)$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$

4.6 Exercices

Exercice 4.1 *Programmer la recherche séquentielle en table :*

1. *implémentée au moyen d'une liste associative ;*
2. *implémentée à l'aide d'un tableau.*

Exercice 4.2 *Programmer en C la recherche dichotomique sur un tableau trié.*

Exercice 4.3 *Programmer en C une table de hachage et les opérations de recherche, d'insertion et de suppression d'un élément associées.*

Chapitre 5

Arbres

Nous allons étudier ici une structure de données plus élaborée que toutes celles vues jusqu'à présent : la structure d'*arbre*. On l'utilise très fréquemment en informatique. C'est par exemple un arbre que l'on emploie pour classer ses fichiers sous UNIX, qui sont eux-mêmes physiquement organisés comme des arbres.

5.1 Terminologie

Il existe plusieurs façons de définir un arbre, et de nombreux types d'arbres différents. Intuitivement, un arbre au sens informatique ressemble à un arbre généalogique, et la terminologie usitée en dérive d'ailleurs directement.

Informellement, un arbre est un ensemble de *nœuds* connectés entre eux de manière bien spécifique : chaque nœud pointe vers un ensemble éventuellement vide d'autres nœuds, qui en sont les *fils* (les *enfants*), et tous les nœuds sauf un ont un *père* et un seul ; le nœud sans père s'appelle la *racine*. Un dessin sera certainement plus explicite : la figure 5.1 nous montre un exemple d'arbre ; conventionnellement, la racine est toujours représentée en haut et les branches dirigées vers le bas. Ici, notre arbre est constitué de sept nœuds. Le nœud 5 en est la racine, et a deux fils : 1 et 3 ; le nœud 1 a un fils : 4, et le nœud 3 trois fils : 2, 6 et 7. Le père de 2 est 3, celui de 3 est 5.

Donnons encore quelques définitions en les illustrant sur l'exemple de la figure 5.1 : un nœud sans fils s'appelle une *feuille*, comme 4, 2, 6 et 7. Un nœud qui n'est pas une feuille est un *nœud interne* — ici 1, 3 et 5.

Quand il existe un chemin d'un nœud n_1 vers un nœud n_2, on dit que n_1 est un *ancêtre* de n_2, ou que n_2 est un *descendant* de n_1 ; sur notre exemple, 5 admet les sept nœuds de l'arbre comme descendants et seulement lui-même comme ancêtre ; les ancêtres de 2 sont 2, 3 et 5.

Un *sous-arbre* d'un arbre A est un arbre constitué de tous les descendants d'un nœud quelconque de A, qui en est la racine. Ainsi, les ensembles de nœuds $\{3, 2, 6, 7\}$ et $\{2\}$ forment des sous-arbres de l'arbre de la figure 5.1. Dans la suite, un fils désignera selon contexte indifféremment un nœud ou le sous-arbre dont ce nœud est la racine.

On appelle *profondeur* d'un nœud la longueur du chemin qui le lie à la racine : 5 est donc de profondeur nulle, 1 et 3 de profondeur un, 4, 2, 6 et 7 de profondeur deux. La *hauteur* d'un arbre est la profondeur maximale de ses nœuds (de ses

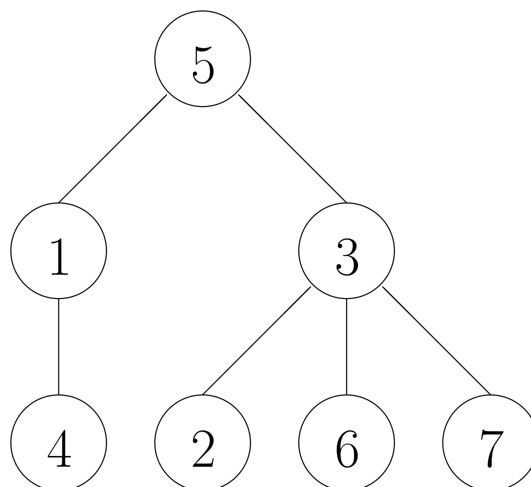

FIGURE 5.1 – Un exemple d'arbre

feuilles, donc) : ici, deux.

Le *degré* d'un nœud est égal au nombre de ses fils : 5 est de degré deux, 1 de degré un, 3 de degré trois, et les feuilles (4, 2, 6, 7) de degré nul. L'*arité* d'un arbre est le degré maximal de ses nœuds, soit trois sur notre exemple.

Dans notre présentation des arbres, nous n'avons pas précisé de structuration particulière de l'ensemble des fils d'un nœud. Lorsqu'il n'y en a effectivement pas, on parle plutôt d'*arborescence*. En général, dans un arbre, on considère que l'ensemble des fils de chaque nœud est ordonné [1]. On peut même imposer comme condition supplémentaire qu'ils soient numérotés par des entiers strictement positifs (distincts), et l'on parle alors d'*arbre numéroté*. Ce n'est pas la même chose, car on peut avoir des fils manquants pour certaines valeurs entières, comme on va le voir sur un exemple un peu plus loin.

Lorsqu'on a un arbre numéroté par des entiers tous compris dans l'intervalle $[1, k]$, pour un certain entier $k \geq 1$, on parle alors d'*arbre k-aire*. On notera qu'un arbre k-aire est d'arité inférieure ou égale à k. En particulier, lorsque k vaut 2, on est dans le cas très important en pratique des *arbres binaires*, dont un exemple est donné figure 5.2 : implicitement, le fils situé à gauche d'un nœud est numéroté par 1, et le fils situé à droite par 2. On dit encore : *fils gauche*, *fils droit*. La racine (5) admet donc 3 comme fils gauche, et 7 comme fils droit ; 7 a un fils droit : 8, et pas de fils gauche. On notera qu'en tant qu'arbre non numéroté, on ne serait pas capable de préciser si 8 est un fils gauche ou droit de 7.

Jusqu'à présent, on a dénoté les nœuds par des entiers sur nos exemples : en faisant cela, on a abusivement confondu le nœud et l'information qu'il contient, et qui s'appelle une *clé* [2]. Cela n'est pas toujours très judicieux, car des nœuds différents peuvent bien sûr contenir la même clé, comme on peut le voir sur l'exemple de la figure 5.2. Dans la suite, on prendra garde en faisant cette assimilation.

On va à présent énoncer une définition plus formelle des arbres, de nature récursive. De manière analogue aux listes, on peut se donner un objet vide : l'*arbre*

1. On parlera indifféremment d'arbre ou d'*arbre ordonné*.
2. Certains types d'arbres contiennent en fait plusieurs clés par nœud.

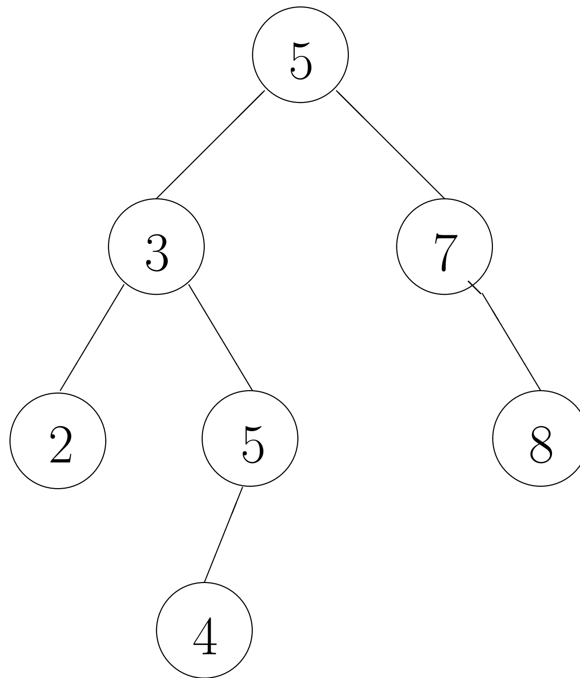

FIGURE 5.2 – Un arbre binaire

vide, et on définit alors un arbre à clés de type T comme étant ou bien l'arbre vide, ou bien un arbre construit en créant un nœud racine à partir d'un élément de type T dont il sera la clé et d'une liste d'arbres qui seront ses fils. On notera que cette définition est très générale puisqu'elle permet de représenter même les arbres numérotés : le i-ème fils d'un nœud sera absent si la liste de ses fils est de longueur inférieure strictement à i, ou si son i-ème élément est l'arbre vide.

Concrètement, si l'on note FEUILLE l'arbre vide, et BRANCHE le constructeur de nouveaux arbres, que l'on comparera au constructeur CONS sur les listes, en voici alors quelques spécimens :

$$\begin{array}{l} \text{FEUILLE}, \\ \text{BRANCHE}(x, [\,]), \\ \text{BRANCHE}(x, [\text{FEUILLE}, \text{FEUILLE}, \text{FEUILLE}]), \\ \text{BRANCHE}(x, [\text{BRANCHE}(y, [\text{FEUILLE}, \text{FEUILLE}]), \text{BRANCHE}(z, [\,])]) \\ \vdots \end{array}$$

où x, y et z sont des éléments de type T.

En particulier, on peut redéfinir les arbres k-aires, et en tout état de cause, les représenter comme étant les arbres dont les nœuds internes ont exactement k fils, et dont les feuilles sont l'arbre vide. De cette manière, l'arbre binaire de la figure 5.2 est le même que celui de la figure 5.3. On notera qu'en machine, on pourra alors utiliser un tableau de taille k, voire k champs différents (si k est petit), plutôt qu'une liste, pour coder les fils d'un nœud interne dans ce type d'arbre. Ainsi, dans le cas des arbres binaires, on peut employer un constructeur BRANCHE-BIN qui construit un nouvel arbre à partir d'une clé et de deux arbres binaires, pour nous

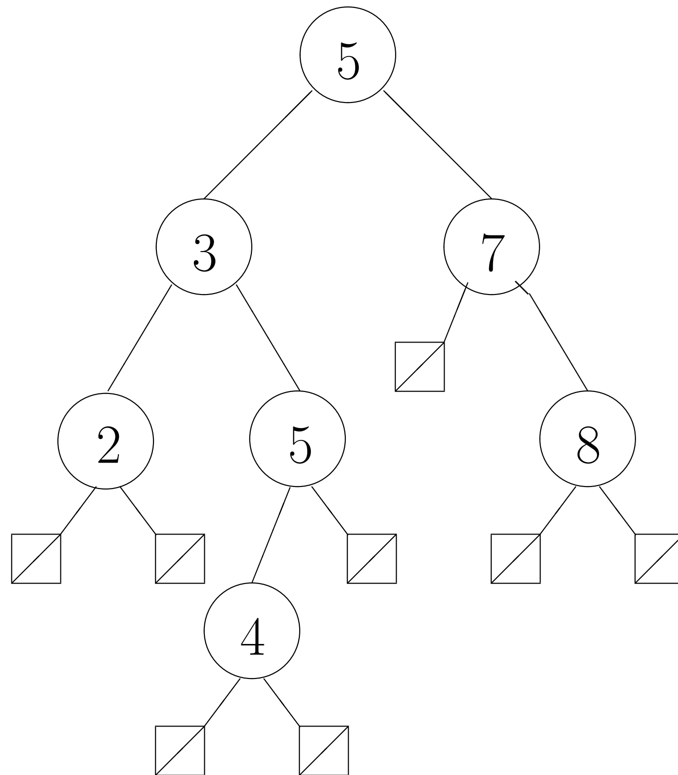

FIGURE 5.3 – Un arbre binaire doté de feuilles vides

donner par exemple :

$$\begin{array}{l}\textsc{Feuille},\\ \textsc{Branche-Bin}(x, \textsc{Feuille}, \textsc{Feuille}),\\ \textsc{Branche-Bin}(x, \textsc{Branche-Bin}(y, \textsc{Feuille}, \textsc{Feuille}), \textsc{Feuille})\\ \vdots\end{array}$$

où x et y sont des éléments de type T.

On notera également que la structure de liste se confond alors avec celle d'*arbre unaire*. Pour n'importe quel type d'arbre, on peut d'ailleurs définir l'équivalent des fonctions TÊTE et QUEUE sur les listes, qui extrait la clé contenue dans la racine, et renvoie le i-ème sous-arbre fils d'un arbre, respectivement (voir exercice 5.2).

En C, on implémente les arbres à l'aide de pointeurs, ce qui nous donne par exemple pour les arbres binaires la définition de type suivante :

```
typedef struct noeud {
  type_element cle;
  struct noeud *gauche;
  struct noeud *droit;
}*arbre_binaire;
```

Il existe encore d'autres possibilités de codage, telle l'analogue des listes doublement chaînées, où chaque nœud pointe non seulement vers ses fils, mais également vers son père, et nous en verrons une autre ultérieurement.

5.2 Parcours d'arbre

Les arbres servent à maintes choses en informatique, et en particulier, on les utilise pour représenter la structure effective des programmes. Plus précisément, la première tâche qu'effectue tout compilateur est de transformer un fichier source écrit dans un langage de programmation donné, qui n'est généralement ni plus ni moins qu'une suite de caractères, en un *arbre de syntaxe abstraite*, qui exprime le sens réel du programme : c'est ce qu'on appelle la phase d'*analyse syntaxique* du programme (*parsing* en anglais). Ainsi, toute expression arithmétique ou logique est transformée en un arbre ; par exemple, l'expression :

$$(4+3)\times 2 \tag{5.1}$$

sera représentée par l'arbre de la figure 5.4 (qui ne correspond pas à : $4+3\times 2$). Le programme tout entier est en fait transformé en un arbre : les affectations sont

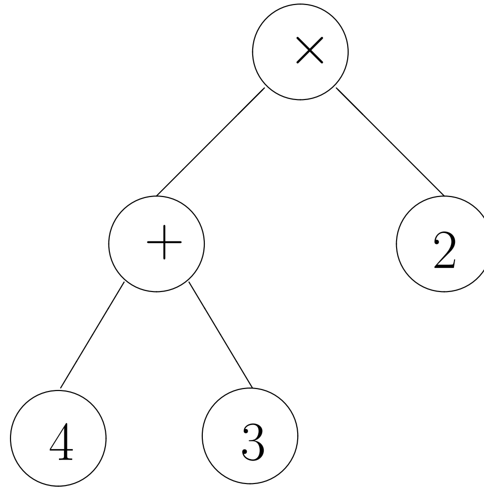

Figure 5.4 – Un exemple d'arbre d'expression

représentées par des nœuds de degré deux, les instructions conditionnelles **si ... alors ... sinon ...** par des nœuds de degré trois, etc.

On n'étudiera pas ici les algorithmes plutôt compliqués employés pour ce faire, mais ceux qui effectuent l'opération inverse : comment à partir d'un arbre d'expression réécrire celle-ci sous une forme plus lisible ? Ce qui revient en gros à parcourir l'arbre dans son intégralité et le bon ordre. Par "bon ordre", on veut dire un ordre qui permette une lecture naturelle d'une expression arithmétique. Lorsqu'on a affaire à des opérateurs d'arité (i.e. de nombre d'arguments) quelconque, deux notations s'imposent : la notation *préfixée*, déjà vue dans la section 3.4 et la notation *postfixée*. La première consiste à écrire l'opérateur avant ses arguments, et la seconde après. Par exemple, l'expression (5.1) s'écrit :

$$\times + 4\ 3\ 2$$

en notation préfixée (c'est-à-dire $(\times\ (+\ 4\ 3)\ 2)$), et :

$$4\ 3 + 2\ \times$$

en notation postfixée (ou encore ((4 3 +) 2 ×))), sans qu'il y ait ambiguïté dans l'un ou l'autre cas (et les parenthèses sont donc inutiles).

À partir de l'arbre de la figure 5.4, on peut retrouver ces expressions par un parcours récursif : dans le premier cas, il faut traverser chaque arbre en commençant par la racine, puis tous ses sous-arbres fils dans l'ordre, alors que dans le second cas, on commence par l'ensemble des fils et on termine par la racine. On parle de parcours *préfixé* et *postfixé* d'un arbre. Donnons le pseudo-code correspondant à celui-là pour les arbres binaires :

Algorithme 5.1 Parcours-Préfixé(A)
1 **si** $A \neq$ Feuille **alors**
2 imprimer Clé(A)
3 Parcours-Préfixé(Gauche(A))
4 Parcours-Préfixé(Droit(A))

Algorithme 5.2 Parcours-Postfixé(A)
1 **si** $A \neq$ Feuille **alors**
2 Parcours-Postfixé(Gauche(A))
3 Parcours-Postfixé(Droit(A))
4 imprimer Clé(A)

où les fonctions Clé, Gauche et Droit retournent respectivement la clé et les sous-arbres gauche et droit d'un arbre binaire.

Dans le cas des arbres binaires, il existe encore un autre parcours qui consiste à inspecter d'abord le sous-arbre gauche, puis la racine, et enfin le sous-arbre droit. On parle alors de parcours *infixé*. Pour un arbre d'expression ne contenant que des opérateurs binaires, on retrouve la notation *infixée* habituelle : ainsi, l'arbre de la figure 5.4 est traversé dans l'ordre :

$$4 + 3 \times 2$$

ce qui donne cette fois-ci une expression ambiguë si l'on ne suppose pas de prioritéentre opérateurs. Pour remédier à ce défaut, il faut de plus utiliser des parenthèses au moment opportun (voir exercice 5.4).

Algorithme 5.3 Parcours-Infixé(A)
1 **si** $A \neq$ Feuille **alors**
2 Parcours-Infixé(Gauche(A))
3 imprimer Clé(A)
4 Parcours-Infixé(Droit(A))

Il n'est pas difficile de se convaincre que tous ces parcours nécessitent une visite et une seule de chaque nœud, soit donc n appels récursifs pour un arbre de n nœuds.

Enfin, il existe d'autres types de parcours des arbres, tels le parcours en largeur d'abord, qui consiste à inspecter tous les nœuds de même profondeur avant d'aller voir ceux de profondeur plus élevée, mais nous n'en dirons rien de plus ici.

5.3 Arbres binaires de recherche

5.3.1 Définition

Nous allons ici aborder une autre utilisation importante des arbres, et qui va compléter le chapitre précédent, puisqu'il va s'agir de structurer l'information en mémoire pour pouvoir la retrouver très rapidement. L'idée est d'utiliser des arbres binaires afin d'obtenir un algorithme de recherche proche dans l'esprit de la recherche dichotomique mais pour lequel les opérations d'insertion et de suppression d'un élément soient elles aussi efficaces.

On va donc définir un *arbre binaire de recherche* comme un arbre binaire dont les clés appartiennent à un ensemble totalement ordonné et vérifient la propriété suivante :

> *Toute clé d'un nœud de l'arbre est supérieure ou égale à celles des descendants de son fils gauche, et inférieure ou égale à celle des descendants de son fils droit.*

L'arbre représenté sur la figure 5.2 est donc un arbre binaire de recherche. On a omis de préciser qu'une information pouvait être associée à une clé, mais cela va de soi pour une recherche en table ; de même, on pourra imposer que toute clé n'apparaît au plus qu'une fois.

On notera encore que tout sous-arbre d'un arbre binaire de recherche est lui-même un arbre binaire de recherche. Une dernière propriété remarquable (et caractéristique) des arbres binaires de recherche est qu'un parcours infixé en imprime les clés dans l'ordre croissant. On peut d'ailleurs en déduire un nouvel algorithme de tri dont la complexité se déduira de l'analyse ci-dessous.

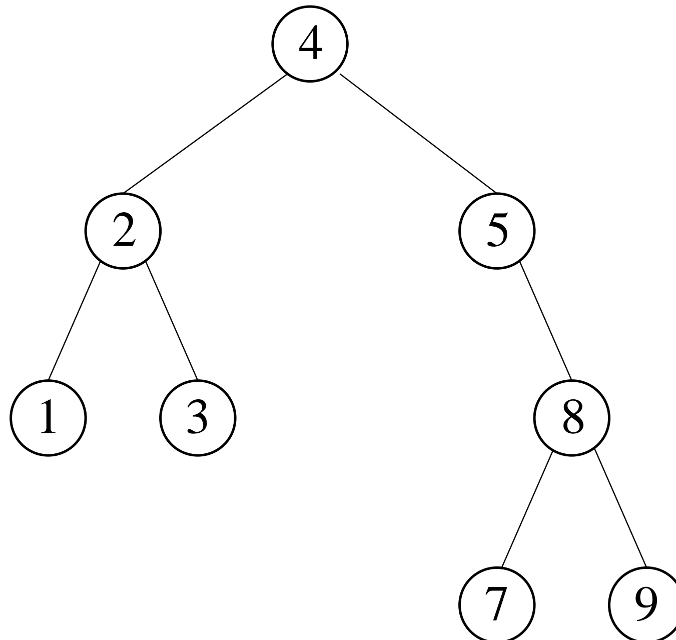

FIGURE 5.5 – Arbre binaire de recherche obtenu en ajoutant successivement les entiers 4, 5, 8, 9, 2, 3, 1 et 7 dans un arbre initialement vide

5.3.2 Opérations sur les arbres binaires de recherche

La **recherche d'une clé** dans un arbre binaire de recherche imite la recherche dichotomique : il suffit de comparer la clé à découvrir avec celle du nœud courant, et si on ne l'a pas trouvée, de répéter l'opération sur le sous-arbre droit ou gauche selon le cas. Ceci nous donne le pseudo-code suivant, pour un arbre A :

Algorithme 5.4 RECHERCHE-ARBRE(A, *clé*)
1 **si** A = FEUILLE **alors retourner** *recherche_infructueuse*
2 **si** *clé* = CLÉ(A) **alors retourner** INFORMATION(A)
3 **si** *clé* < CLÉ(A)
4 **alors retourner** RECHERCHE-ARBRE(GAUCHE(A), *clé*)
5 **sinon retourner** RECHERCHE-ARBRE(DROIT(A), *clé*)

La fonction RECHERCHE-ARBRE consiste à suivre un chemin descendant dans l'arbre jusqu'au succès ou à l'échec, qui sera prononcé lorsqu'on atteindra une feuille. Ce qui signifie que dans le pire des cas, le nombre d'appels récursifs (ou de comparaisons requises) sera en $\Theta(h)$, où h est la hauteur de l'arbre A. En cas de succès, la fonction INFORMATION retournera l'information associée à la clé.

L'**opération d'insertion** d'une clé consiste à suivre le bon chemin dans l'arbre jusqu'à arriver à une feuille qu'on remplace alors par un nouveau nœud interne pourvu de deux fils vides :

Algorithme 5.5 INSERTION-ARBRE(A, *clé*)
1 **si** A = FEUILLE
2 **alors** $A \leftarrow$ BRANCHE-BIN(*clé*, FEUILLE, FEUILLE)
3 **sinon si** *clé* $\leq$ CLÉ(A)
4 **alors** INSERTION-ARBRE(GAUCHE(A), *clé*)
5 **sinon** INSERTION-ARBRE(DROIT(A), *clé*)

La complexité de cette opération est clairement la même que celle de la recherche, à savoir $\Theta(h)$, si h est la hauteur de l'arbre A. On notera toutefois que cette opération ne correspond pas tout à fait à l'insertion en table, puisqu'elle permet d'insérer plusieurs fois la même clé. Si l'on veut éviter ceci, il faudra modifier l'algorithme précédent de manière adéquate, ce qui n'est vraiment pas difficile.

L'**opération de suppression** d'un élément dans un arbre binaire de recherche est un peu plus subtile : on se contentera d'en exposer le principe en français, laissant au lecteur le soin d'en déduire un algorithme complet. Supposant donc vouloir supprimer d'un arbre le nœud contenant une clé donnée, on commencera par la rechercher à l'aide de la fonction précédemment décrite dans l'algorithme 5.4, dans laquelle on aura remplacé INFORMATION(A) par A ligne 2, c'est-à-dire que la fonction de recherche retourne le sous-arbre dont la racine contient la clé cherchée plutôt que l'information associée. Si la recherche est fructueuse, on sera alors amené à distinguer trois cas :

1. si le nœud à supprimer a deux fils vides (tel 2 sur la figure 5.3), on le détruit alors purement et simplement, c'est-à-dire qu'on le remplace par l'arbre vide ;
2. s'il a un et un seul fils vide, on le remplace par son autre fils non vide (on remplacerait ainsi 7 par 8) ;

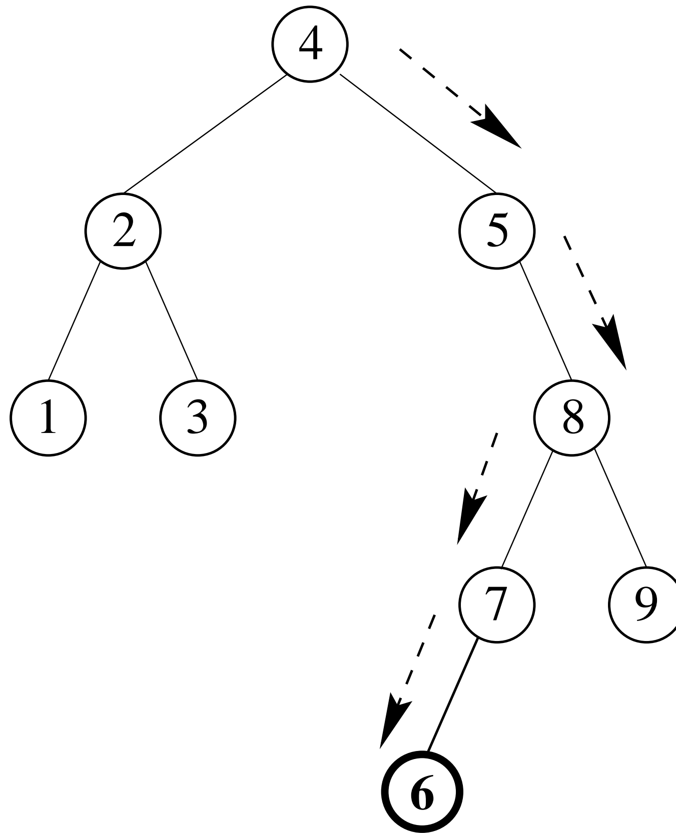

Figure 5.6 – Ajout de 6 dans l'arbre de la figure 5.5

3. sinon, on commence par calculer son *successeur* dans l'arbre, c'est-à-dire le nœud possédant la plus petite clé plus grande que la sienne : il n'est pas très difficile de se rendre compte qu'il s'agit du nœud de clé minimale de son fils droit ; on remplace alors sa clé par celle de son successeur, qu'on supprime finalement de l'arbre, ce qui est aisé car il ne peut avoir de fils gauche (on emploie donc la méthode précédente) ; ainsi, sur la figure 5.3, pour supprimer la racine, on calcule son successeur : le nœud 7, qu'on remplace par le nœud 8 après avoir mis la clé 7 dans la racine ; pour supprimer le nœud 3, on remplace le nœud 4 par l'arbre vide, et la clé 3 par 4.

Sans rentrer dans le détail de la chose, on peut prouver que cette opération s'exécute aussi en $\Theta(h)$, où h est la hauteur de l'arbre.

Il nous reste à présent à évaluer la complexité de nos trois opérations en fonction du nombre n de nœuds internes de l'arbre (ou de clés contenues dans l'arbre), qui fera office dorénavant de *taille* de l'arbre.

Il existe de toute évidence des arbres de taille n de hauteur égale à $n - 1$: il suffit que tous leurs nœuds internes aient au plus un fils non vide, et on a un arbre isomorphe à une liste. Dans ce cas, les trois opérations s'effectuent en $\Theta(n)$, et on n'a donc rien gagné sur la recherche séquentielle dans un tableau. Les arbres binaires de recherche ne sont donc pas intéressants dans le cas le pire.

Par contre, il existe aussi des arbres de taille $2^{h+1} - 1$ et de hauteur h : les arbres dont tous les niveaux de profondeur sont complètement remplis. En effet,

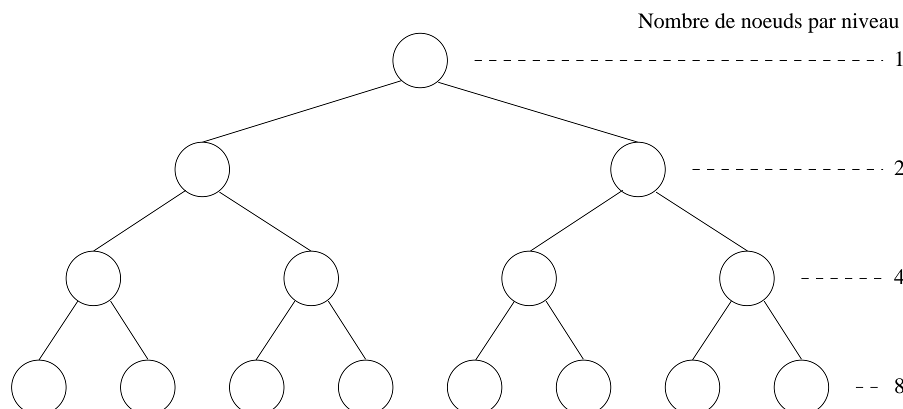

FIGURE 5.7 – Arbre binaire équilibré

on a alors un nœud de profondeur 0, deux nœuds de profondeur 1, quatre nœuds de profondeur 2, etc., soit 2^p nœuds de profondeur p, ce qui donne un nombre total de nœuds égal à :

$$2^0 + 2^1 + 2^2 + \cdots + 2^h = \frac{2^{h+1} - 1}{2 - 1} = 2^{h+1} - 1$$

Nos trois opérations sont donc en $\Theta(\log n)$ dans ce cas, ce qui est cette fois-ci bien meilleur que les recherches séquentielles et dichotomiques. Que se passe-t-il alors en moyenne ?

Pour répondre à cette question, on pourrait imaginer dénombrer les arbres binaires de recherche et en déterminer la hauteur moyenne, mais cela n'aurait guère de sens en pratique. Il est bien mieux d'évaluer la hauteur moyenne d'arbres construits aléatoirement à partir d'insertions et de suppressions de clés. Malheureusement, on sait encore fort peu de choses à l'heure actuelle sur ce problème. On se contentera donc d'énoncer sans démonstration le résultat connu sur les arbres binaires construits aléatoirement uniquement à partir d'insertions :

Proposition 5.1 *La hauteur moyenne d'un arbre binaire de recherche construit aléatoirement à partir de n clés distinctes est en $\Theta(\log n)$.*

Ceci justifie mathématiquement, quoique de manière partielle, l'emploi de cette structure de données.

En pratique, outre le fait que les suppressions auront tendance à faire croître sensiblement la hauteur moyenne de l'arbre, on peut également rencontrer le cas où l'ordre d'arrivée des clés n'a rien d'aléatoire (par exemple si elles arrivent dans l'ordre croissant ou décroissant), engendrant alors un arbre isomorphe à une liste, sur lequel les opérations de base sont toutes trois inefficaces.

Pour remédier à ce problème s'est fait jour l'idée d'employer la structure de données plus élaborée d'*arbre équilibré*, abordées dans la section 5.5.

5.4 Files de priorité et tas

Nous allons étudier ici une nouvelle structure de données : les *files de priorité*. Nous verrons qu'un certain type d'arbre permettra de les représenter de manière

extrêmement efficace.

5.4.1 Définition

Commençons par définir les files de priorité : intuitivement, il s'agit d'un ensemble d'éléments clients à qui sont attribués un *rang de priorité* avant qu'ils ne rentrent dans la file, et qui en sortiront précisément selon leur rang, à savoir que l'élément de rang le plus élevé sera servi en premier. En d'autres termes, il s'agit d'une structure de données sur laquelle opèrent les trois opérations suivantes :

— une fonction d'insertion d'un élément dans la file (avec son rang de priorité) ;
— une fonction qui retourne l'élément de plus haut rang ;
— une fonction qui supprime l'élément de plus haut rang de la file.

On disposera évidemment de plus d'un objet vide correspondant. On utilise cette structure de données par exemple pour planifier l'ordre d'exécution de tâches de priorités différentes dans un ordinateur.

Il est à noter que les piles et les files (cf. section 3.3) peuvent se concevoir comme des files de priorité pour lesquelles les éléments arriveraient tous dans l'ordre croissant ou décroissant, respectivement.

Une première manière naturelle de coder les files de priorité est d'employer justement une file d'attente classique, mais alors si l'insertion se fait en temps constant, la fonction qui retourne ou supprime l'élément de plus haut rang (on dira encore le *maximum* dans la suite) nécessite une recherche préalable de cet élément dans la file, ce qui prend en moyenne un temps en $\Theta(n)$, où n est le nombre d'éléments contenus dans la file.

De même, si l'on choisissait alternativement un tableau trié, la recherche du maximum serait alors en $O(1)$, mais l'insertion prendrait un temps $\Theta(n)$ en moyenne.

On va donc utiliser la structure de *tas* (*heap* en anglais), qui est un arbre vérifiant les deux propriétés suivantes :

1. c'est un arbre binaire *complet*, c'est-à-dire un arbre binaire dont tous les niveaux de profondeur sont complètement remplis, sauf éventuellement le dernier (celui de profondeur la plus élevée) où les nœuds sont néanmoins rangés le plus à gauche possible ;
2. la clé de tout nœud est supérieure ou égale à celles de ses descendants.

La figure 5.8 nous montre un exemple de tas (les clés sont à l'intérieur des cercles ; on reviendra plus loin sur les nombres situés à l'extérieur).

5.4.2 Opérations sur les tas

Montrons maintenant comment employer un tas pour coder une file de priorité. Tout d'abord, le codage de la fonction qui retourne le maximum est très simple : il suffit de retourner la clé contenue dans la racine (si le tas n'est pas vide). Elle s'exécute donc en temps constant.

En ce qui concerne la **fonction d'insertion d'un élément**, voici brièvement le principe : on commence par créer un nouveau nœud contenant la nouvelle clé qu'on insère en bonne place dans le tas, c'est-à-dire le plus à gauche possible sur

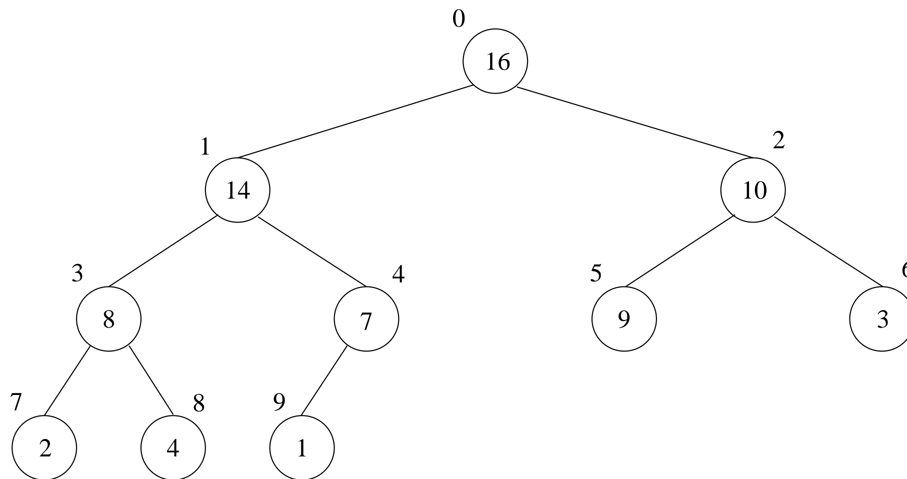

FIGURE 5.8 – Un exemple de tas

le niveau de profondeur le plus élevée (ou s'il est plein, à l'extrême gauche d'un nouveau niveau). On a donc bien toujours une structure d'arbre binaire complet, mais plus nécessairement un tas : il faut alors comparer la nouvelle clé avec celle de son père, les permuter si nécessaire, et recommencer le processus jusqu'à ce qu'il n'y ait plus d'échange à réaliser.

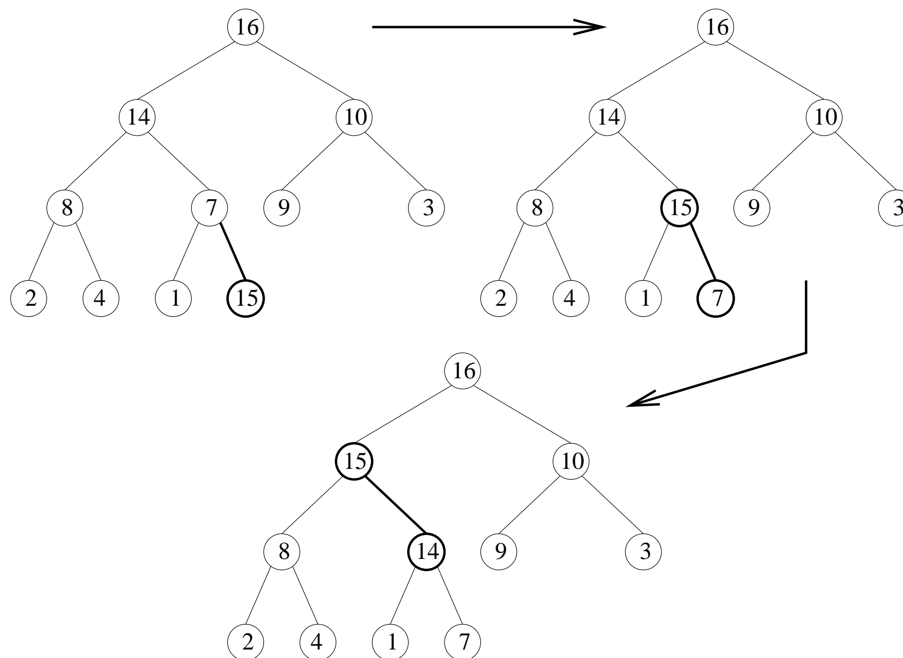

FIGURE 5.9 – Insertion d'un élément dans un tas

Sur notre exemple (voir figure 5.9), supposons vouloir insérer la nouvelle clé 15 : on crée un nœud la contenant qu'on insère comme fils droit de 7, puis l'on permute 7 et 15. On compare 15 et son nouveau père 14 ; il faut encore les échanger, puis comparant 15 à 16. On s'aperçoit alors que la clé est en bonne place et on a

bien reconstitué un tas.

La **suppression d'un élément** d'un tas non vide commence par l'extraction de la clé contenue dans la racine, qu'on remplace par la clé du nœud situé le plus à droite du niveau de profondeur maximal du tas, nœud que l'on supprime alors. Si l'on obtient toujours un arbre binaire complet, la propriété de tas n'est plus garantie ; on va la rétablir en effectuant cette fois-ci des permutations le long d'un chemin descendant. Pour ce faire, on compare la clé située à la racine avec ses fils, et on la permute avec le plus grand des trois (ou des deux) : s'il n'y a pas d'échange à faire, c'est terminé, sinon on recommence plus bas, jusqu'à ce qu'on n'ait pas de permutation à faire ou qu'on arrive à une feuille.

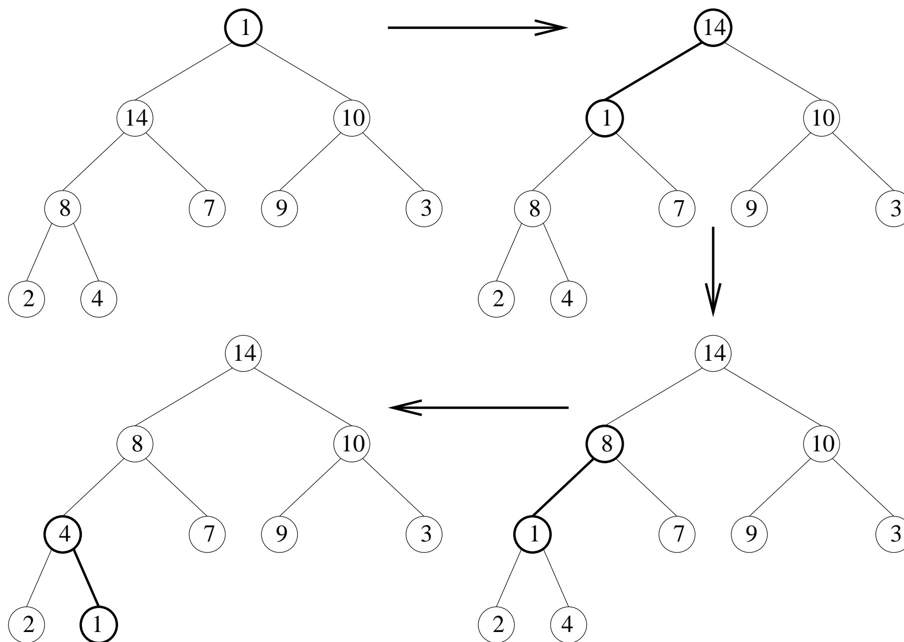

Figure 5.10 – Suppression d'un élément dans un tas

Sur l'exemple précédent (voir figure 5.10), on supprime donc la clé 16 qu'on remplace par la clé 1, et on supprime ce nœud. On compare alors 1 et ses fils : 14 et 10, on permute donc 1 et 14. Ensuite on compare 1 et ses nouveaux fils, 8 et 7, et l'on permute 1 et 8. On recommence la comparaison avec 2 et 4 ; 4 l'emporte et on l'échange avec 1, qui alors n'a plus de fils, et c'est donc terminé.

La complexité des procédures d'insertion et de suppression est en $O(h)$, où h est la hauteur du tas, puisque pour insérer l'on ne fait que remonter un chemin ascendant d'une feuille vers la racine (en s'arrêtant éventuellement avant), alors que pour supprimer on ne fait que suivre un chemin descendant. Or, la hauteur d'un tas de taille n est précisément égale à $\lfloor \log_2 n \rfloor$, et donc l'insertion requiert un temps $O(\log n)$. Ainsi les tas constituent une bonne structure de données pour coder les files de priorité.

i	0	1	2	3	4	5	6	7	8	9
$A[i]$	16	14	10	8	7	9	3	2	4	1

FIGURE 5.11 – Représentation d'un tas à l'aide d'un tableau

5.4.3 Implémentation

Les tas se codent bien sûr tout naturellement comme des arbres mais il se pose alors le problème d'accéder efficacement au père d'un nœud ainsi que le problème de trouver le noeud le plus à droite du dernier niveau. On préfère donc utiliser un codage, particulièrement efficace, à base de tableaux. En effet, on peut coder un arbre binaire complet par un tableau et un index qui en indique la taille (le nombre de nœuds de l'arbre), en numérotant tout simplement les nœuds par un parcours en largeur d'abord : sur l'exemple de la figure 5.8, les nombres extérieurs aux nœuds sont les indices du tableau correspondant, représenté sur la figure 5.11.

Avec cette représentation, le calcul des fils et du père d'un nœud est particulièrement simple : le fils gauche du nœud d'indice i, s'il existe, est situé à l'indice $2i + 1$, le fils droit à l'indice $2i + 2$, et le père à l'indice $\lfloor (i-1)/2 \rfloor$. La propriété de tas s'énonce donc alors, pour un tas de taille n représenté par un tableau A :

$$\forall i \in [1, n-1], A[i] \leq A[\lfloor (i-1)/2 \rfloor]$$

Les opérations précédentes s'exécutent sur machine de manière extrêmement efficace, et c'est ainsi qu'on code les tas en pratique.

5.5 Arbres équilibrés

Informellement, un arbre équilibré est un arbre dont les feuilles sont situées à peu près toujours à la même profondeur, et qui est donc le plus “rempli” possible. En fait, il n'existe pas un seul, mais de nombreux types différents d'arbres équilibrés : pour information, citons entre autres :

— les *arbres AVL*, des initiales de G. M. Adel'son-Vel'skiĭ et E. M. Landis qui en introduisirent le principe, qui constituèrent le premier exemple d'arbre équilibré, et qui sont tout bonnement des arbres binaires de recherche vérifiant de plus la propriété que les deux sous-arbres fils de tout nœud ont même hauteur, à 1 près ;

— les *arbres rouge et noir*, qui sont des arbres binaires de recherche dont les nœuds sont en plus coloriés (en rouge ou en noir), et tels que que toute feuille et tout fils d'un nœud rouge soit noir, et tels que tous les chemins reliant un nœud donné à une feuille contiennent le même nombre de nœuds noirs ;

— les *arbres 2–3*, les *arbres 2–3–4*, les *B-arbres*, qui sont des arbres de recherche dont l'arité est supérieure à deux ;

— etc...

Tous ces arbres équilibrés satisfont bien sûr la propriété attendue d'avoir une hauteur moyenne dépendant de manière logarithmique du nombre n de nœuds de l'arbre, la recherche d'une clé s'effectuant donc en temps $O(\log n)$. On peut

prouver qu'il en est également ainsi, en général, pour les opérations d'insertion et de suppression d'une clé ; toutefois, celles-ci sont bien plus compliquées que dans le cas des arbres binaires de recherche, car elles ne doivent pas détruire l'équilibre de l'arbre, et font donc intervenir des rotations d'arbre ou des éclatements de nœuds.

5.6 Exercices

Exercice 5.1 *Définisser en C un type pour la structure :*

1. *d'arbre k-aire ;*
2. *d'arbre d'arité quelconque.*

Exercice 5.2 *Programmer en C l'équivalent sur les arbres binaires des quatre fonctions de base opérant sur les listes.*

Exercice 5.3 *Programmer les fonctions qui retournent respectivement le nombre total de nœuds, de nœuds internes et de feuilles d'un arbre binaire.*

Exercice 5.4 *Programmer en C les parcours préfixé, postfixé et infixé (avec parenthésage) d'un arbre binaire.*

Exercice 5.5 *Écrire une fonction retournant la plus petite clé d'un arbre binaire de recherche. Quelle en est la complexité ?*

Exercice 5.6 *Programmer en C les fonctions d'insertion et de recherche dans un arbre binaire de recherche.*

Exercice 5.7 *Programmer en C la fonction de suppression d'un nœud dans un arbre binaire de recherche.*

Exercice 5.8 *Programmer en C les tas comme des tableaux et implémenter les opérations de files de priorité.*

Chapitre 6

Graphes

Les graphes sont des structures de données très générales, permettant de modéliser de nombreux problèmes. Ils posent de plus de très intéressants problèmes d'algorithmique dont nous allons aborder certains dans ce chapitre.

6.1 Définitions

Un *graphe orienté* G est par définition un couple (S, A) où S est ensemble fini de *sommets* et A un sous-ensemble du produit cartésien $S \times S$ appelé ensemble des arcs de G.

La figure 6.1 donne une représentation graphique du graphe $G = (S, A)$ avec $S = \{1, 2, 3, 4, 5, 6, 7\}$ et $A = \{(1, 2), (1, 4), (2, 5), (3, 5), (3, 6), (4, 2), (5, 4), (6, 6)\}$.

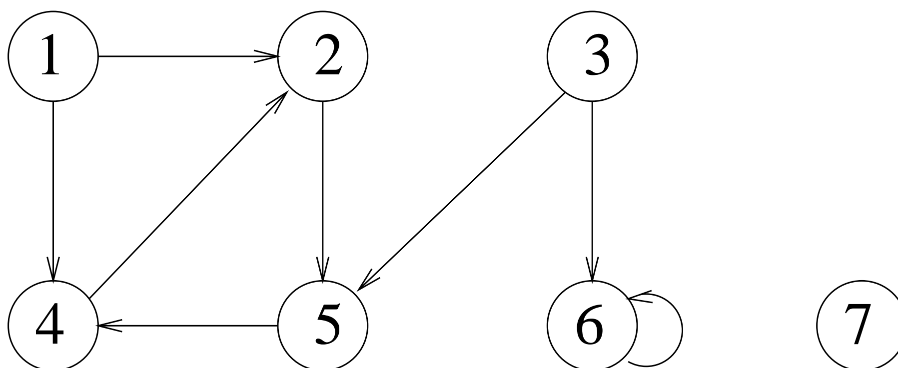

FIGURE 6.1 – Un exemple de graphe

Les graphes sont une structure de données d'emploi très courant car ils permettent de modéliser facilement toutes formes de réseaux mais aussi de résoudre de nombreux problèmes. C'est une structure très générale dont les listes et les arbres ne sont que des cas particuliers. On distingues diverses sortes de graphes, comme par exemple les *graphes non-orientés* dont les arcs (appelés *arêtes*) ne sont plus vus comme des couples de sommets mais plutôt comme des pairs de sommets. On peut de plus associer à un arc (ou une arête) une étiquette, par exemple un nombre, ce qui mène à la notion de *graphe pondéré*.

Une terminologie assez fournie est associée aux graphes. Nous ne définirons que de manière informelles les principaux termes employés. Un sommet S_1 *précède* S_2 si (S_1, S_2) est un arc du graphe. On dit aussi que S_2 est un *successeur* de S_1. On dit encore que S_1 est l'*origine* et S_2 l'*extrémité* de l'arc (S_1, S_2).

Un *chemin* est une suite d'arcs $a_1, \ldots a_k$ telle que pour tout $i \in [1, k-1]$, l'extrémité de a_i et l'origine de a_{i+1} sont confondues. L'origine de a_1 est l'origine du chemin et l'extrémité de a_k est également celle du chemin. La *longueur* d'un tel chemin est par définition l'entier k.

Un chemin dont les origines et les extrémités sont confondues est appelé un *cycle* du graphe. Un graphe sans cycle est dit *acyclique*.

On appelle *arborescence* de racine r un graphe dans lequel, pour tout sommet s, il existe un unique chemin reliant r à s. Un graphe formé d'arborescences non connectées est appelé une *forêt*.

On peut regrouper les sommets d'un graphe en utilisant diverses relations d'équivalence. Par exemple, la relation $x \equiv y$ si et seulement si il existe un chemin (en ne tenant pas compte de l'orientation des arcs) reliant x à y définit les *composantes connexes* du graphe alors que la relation $x \equiv y$ si et seulement si il existe un chemin reliant x à y et un chemin reliant y à x définit les *composantes fortement connexes*.

Dans l'exemple de la figure 6.1, il y a deux composantes connexes ($\{1, 2, 3, 4, 5, 6\}$ et $\{7\}$) et cinq composantes fortement connexes ($\{1\}$, $\{2, 4, 5\}$, $\{3\}$, $\{6\}$, $\{7\}$).

6.2 Représentation des graphes

Afin de représenter des graphes en mémoire, on dispose essentiellement de deux méthodes que nous allons maintenant exposer. Le choix entre l'une ou l'autre de ces deux techniques sera guidé en pratique par le type de graphe que l'on considère et notamment par le rapport entre le nombre de sommets et le nombre d'arcs.

6.2.1 Matrice d'adjacence

Une première manière de représenter un graphe est d'utiliser une *matrice d'adjacence*. Dans la suite, nous considérerons que les sommets sont au nombre de n et numérotés de 1 à n, comme dans figure 6.1.

> La *matrice d'adjacence* du graphe $G = ([1, n], A)$ est la matrice carrée d'ordre n telle que $M_{ij} = 1$ si $(i, j) \in A$ et $M_{ij} = 0$ autrement.

Par exemple, la matrice d'adjacence du graphe de la figure 6.1 est

$$
\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array}
\begin{array}{c}
\begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\
\begin{pmatrix}
0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
\end{array}
$$

La complexité spatiale d'une telle représentation est clairement en $\Theta(n^2)$. Si le graphe est peu dense, i.e. si le rapport entre le nombre d'arcs et le carré du nombre de sommets est faible, une telle représentation n'est donc pas du tout économique, voir même inutilisable dés que le nombre de sommets dépasse quelques centaines de milliers. Les propriétés algébriques remarquables de la matrice d'adjacence sont cependant très utiles dans de nombreux algorithmes. Une solution de compromis peut cependant consister à coder la matrice non pas au moyen d'un tableau binaire bi-dimensionnel mais plutôt avec une matrice creuse, i.e. une structure à base de listes chaînées.

6.2.2 Listes de successeurs

Une seconde façon de coder un graphe consiste à utiliser une *liste d'adjacence*.

> Une *liste d'adjacence* est un tableau $\text{Adj}[1, \ldots, n]$ de listes de sommets où la liste $\text{Adj}[i]$ contient la liste des successeurs de i.

Par exemple, la liste d'adjacence du graphe de la figure 6.1 est représentée dans la figure 6.2. La complexité spatiale d'une telle structure est en $\Theta(|S| + |A|)$, où $|S|$ représente le cardinal de S. Elle est donc optimale d'un point de vue théorique. Elle ne dispose cependant pas de propriétés algébriques, au contraire de la matrice d'adjacence.

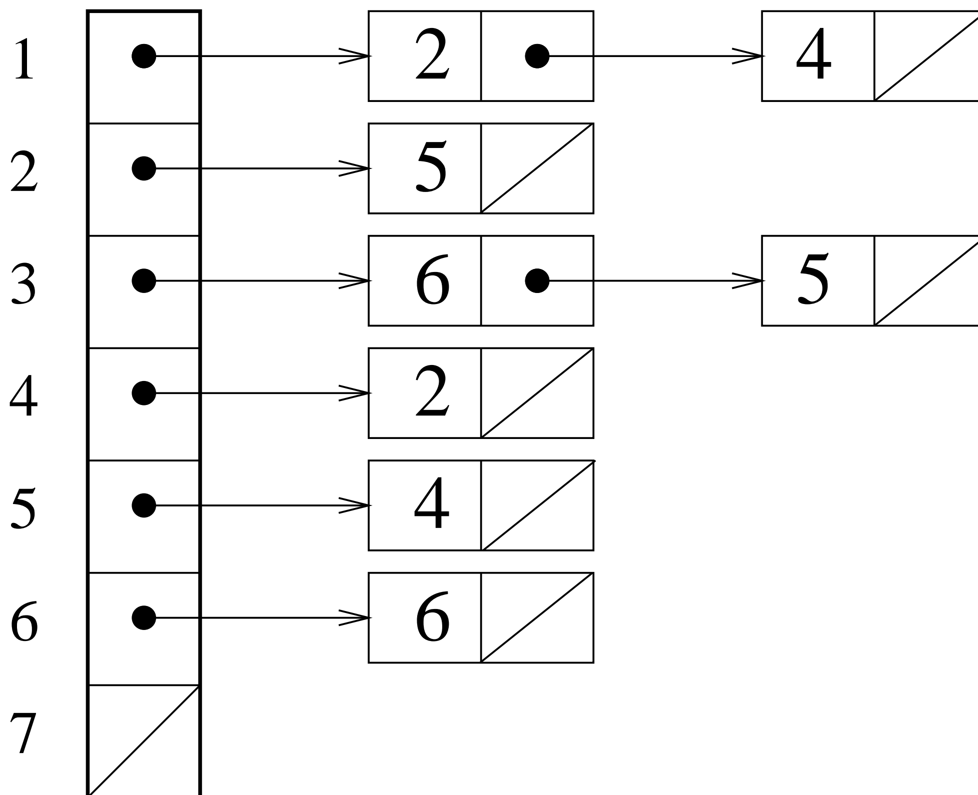

FIGURE 6.2 – Exemple de liste d'adjacence

6.3 Parcours de graphe

Comme pour les listes et les arbres, il est important de pouvoir parcourir les sommets d'un graphes selon certaines règles. Il existe deux grands types de parcours de graphe, les parcours en largeur d'abord et ceux en profondeur d'abord. Le but est de visiter chaque sommet dans un ordre bien déterminé. Pour cela, on colorie les sommets de trois couleurs différentes, par exemple en blanc, gris ou noir. Ces couleurs permettent de dater les visites et de créer une arborescence ou une forêt.

Les forêts étant des graphes très particuliers, nous allons les coder de manière spécifique grâce à un tableau $\pi[1, \ldots, n]$ d'éléments de $[1, n] \cup \{\text{VIDE}\}$ vérifiant :

$$\pi[i] = \begin{cases} \text{VIDE} & \text{si } i \text{ n'a pas de prédécesseur} \\ j & \text{si } j \text{ est le prédécesseur de } i \end{cases}$$

Notez que si j à un prédécesseur, ce dernier est nécessairement unique. On parle alors souvent de *père*, par analogie avec les arbres.

6.3.1 Parcours en largeur d'abord

Dans le parcours *en largeur d'abord*, un sommet s est fixé comme origine et l'on visite tous les sommets situés à distance k de s avant tous les sommets situés à distance $k+1$. Un système de datation (tableau $d[.]$ dans l'algorithme) contient la longueur du plus court chemin au sommet.

Nous allons décrire un algorithme utilisant une file d'attente afin de définir une *arborescence des plus courts chemins* du graphe. La complexité d'un tel algorithme est en $\Theta(|S| + |A|)$.

Algorithme 6.1 PARCOURS-LARG(G, s)

```
1  INITIALISER(G, s)
2  F ← ENFILER(s, FILE-VIDE)
3  tant que F ≠ FILE-VIDE faire
4      u ← PREMIER-ÉLÉMENT(F)
5      pour chaque v ∈ Adj[u] faire
6          si couleur[v] = BLANC alors
7              TRAITER(u, v)
8              ENFILER(v, F)
9      DÉFILER(F)
10     couleur[u] ← NOIR

11 INITIALISER(G, s)
12     pour chaque v ∈ G \ {s} faire
13         couleur[v] ← BLANC
14         d[v] ← +∞
15         π[v] ← VIDE
16     couleur[s] ← GRIS
17     d[s] ← 0
18     π[s] ← VIDE

19 TRAITER(u, v)
```

20     $couleur[v] \leftarrow$ Gris
21     $d[v] \leftarrow d[u] + 1$
22     $\pi[v] \leftarrow u$

La figure 6.3 fournit deux exemple de parcours en largeur d'abord du graphe de la figure 6.1, le premier au départ du sommet 1 et le second au départ de 3. On indique de plus la valeur de la date pour chaque somme, ceux qui ne sont pas accessibles de la racines étant par convention à distance infinie.

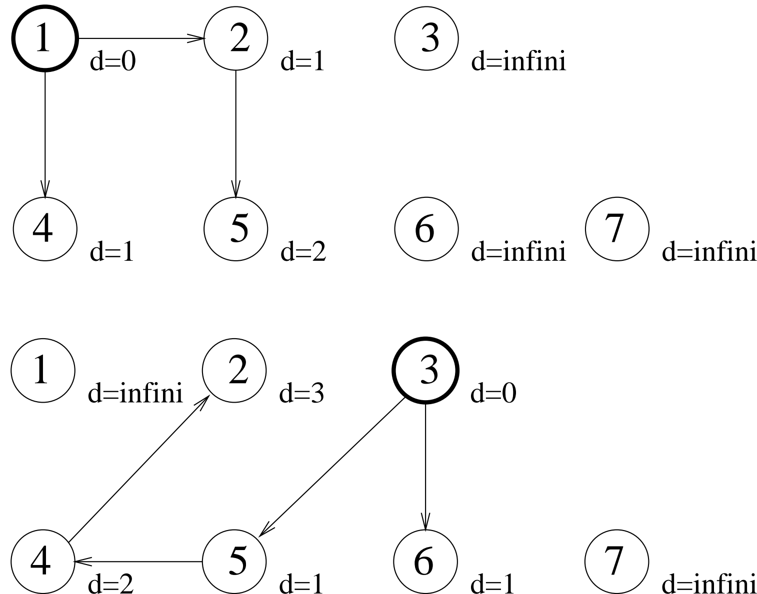

Figure 6.3 – Exemples de parcours en largeur d'abord (le sommet de départ est entouré en gras)

6.3.2 Parcours en profondeur d'abord

Le principe du parcours en profondeur d'abord est de visiter tous les sommets en allant d'abord le plus "profondément" possible dans le graphe. Un système de datation permet de mémoriser les dates de début et de fin de traitement de chaque sommet (tableau $d[.]$ et $f[.]$). Le but de l'algorithme décrit ci-dessous est de créer un forêt par un procédé récursif. La complexité de l'algorithme est en $\Theta(|S|+|A|)$.

Algorithme 6.2 Parcours-Prof(G)

1 **pour** chaque sommet $s \in G$ **faire**
2     $couleur[s] \leftarrow$ Blanc
3     $\pi[s] \leftarrow$ Vide
4 $date \leftarrow 0$
5 **pour** chaque sommet $s \in G$ **faire**
6   **si** $couleur[s] =$ Blanc **alors**
7   Parcours-À-Partir-De(s)

8 Parcours-À-Partir-De(s)

```
9        couleur[s] ← GRIS
10       date ← date + 1
11       d[s] ← date
12       pour chaque v ∈ Adj[s] faire
13           si couleur[v] = BLANC alors
14               π[v] ← s
15               PARCOURS-À-PARTIR-DE(v)
         date ← date + 1
16       f[s] ← date
17       couleur[s] ← NOIR
```

La figure 6.4 fournit un exemple de parcours en profondeur d'abord du graphe de la figure 6.1 en prenant les sommet dans un ordre “naturel”.

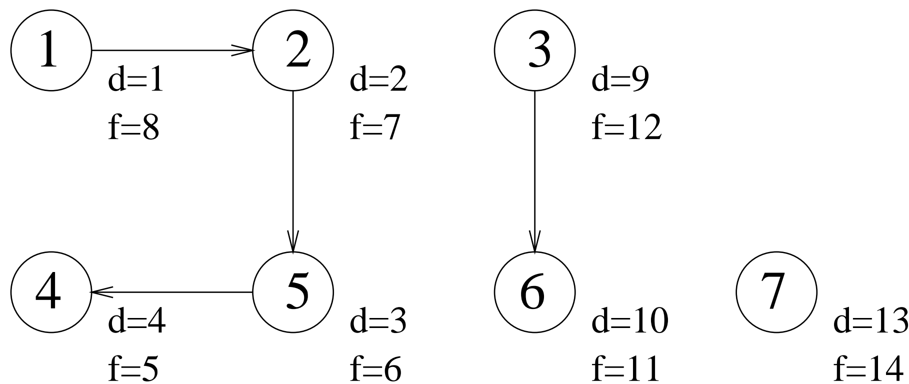

FIGURE 6.4 – Exemple de parcours en profondeur d'abord

6.4 Applications des parcours de graphe

6.4.1 Tri topologique

Considérons un graphe orienté acyclique, i.e. sans cycle. Un tel graphe représente des précédences, i.e. une relation d'ordre partiel. On peut par exemple représenter un ensemble de tâches (les sommets) telles que un arc relie une tache à l'autre si la première doit impérativement être réalisée avant la seconde. L'exemple de la figure 6.5 fournit un tel exemple de graphe utilisé au quotidien.

Le but du *tri topologique* est de fournir une liste de sommets respectant la relation d'ordre partiel. Un algorithme linéaire de tri topologique consiste :

1. à effectuer un parcours du graphe en profondeur d'abord ;
2. à retourner la liste des sommets dans l'ordre décroissant des dates de fin de traitement.

La complexité d'un tel algorithme est en $\Theta(|S| + |A|)$. Avec le graphe de la figure 6.5 on peut ainsi trouver l'arrangement linéaire présenté figure 6.6.

6.4.2 Calcul des composantes fortement connexes

Le calcul des composantes fortement connexes est essentiel pour *réduire* des problèmes de graphe. Le but est d'associé à tout graphe G un graphe orienté

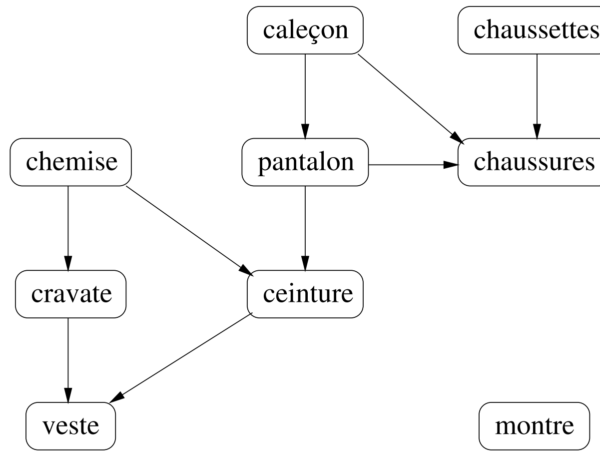

FIGURE 6.5 – Un problème pratique de tri topologique

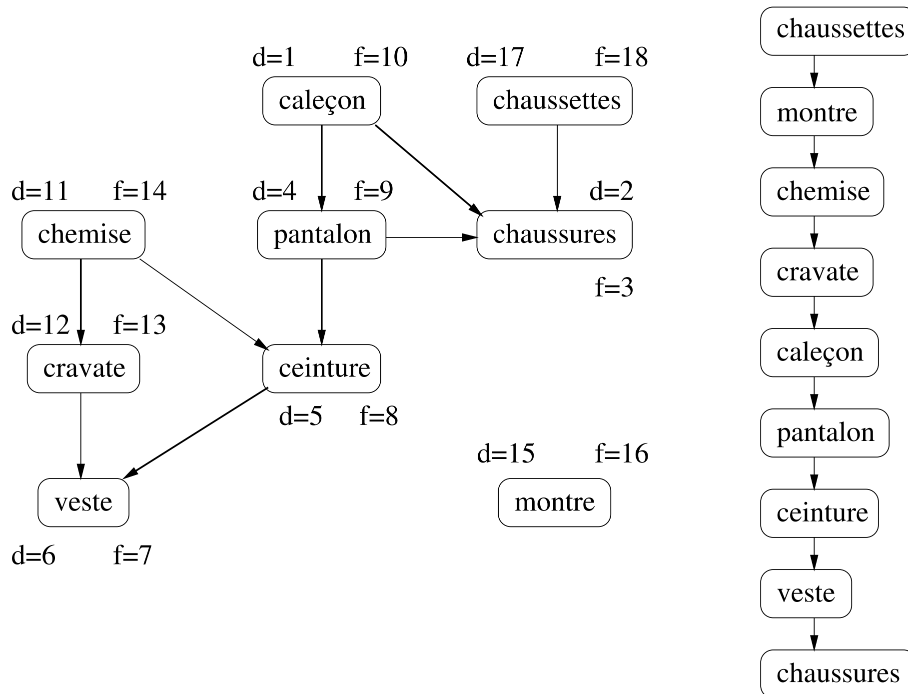

FIGURE 6.6 – Solution au problème du tri topologique

acyclique dont les sommets sont les composantes fortement connexes de G. On traite ensuite le problème sur chaque composante.

Voici un algorithme de calcul des composantes fortement connexes :

1. effectuer un parcours du graphe en profondeur d'abord ;
2. calculer le *graphe transposé*, i.e. le graphe obtenu en inversant le sens de tous les arcs ;
3. en effectuer un parcours en profondeur d'abord dans l'ordre décroissant des fins de traitement de l'étape 1 ;
4. chaque arborescence obtenue est une composante fortement connexe.

La complexité de cet algorithme est en $\Theta(|S| + |A|)$.

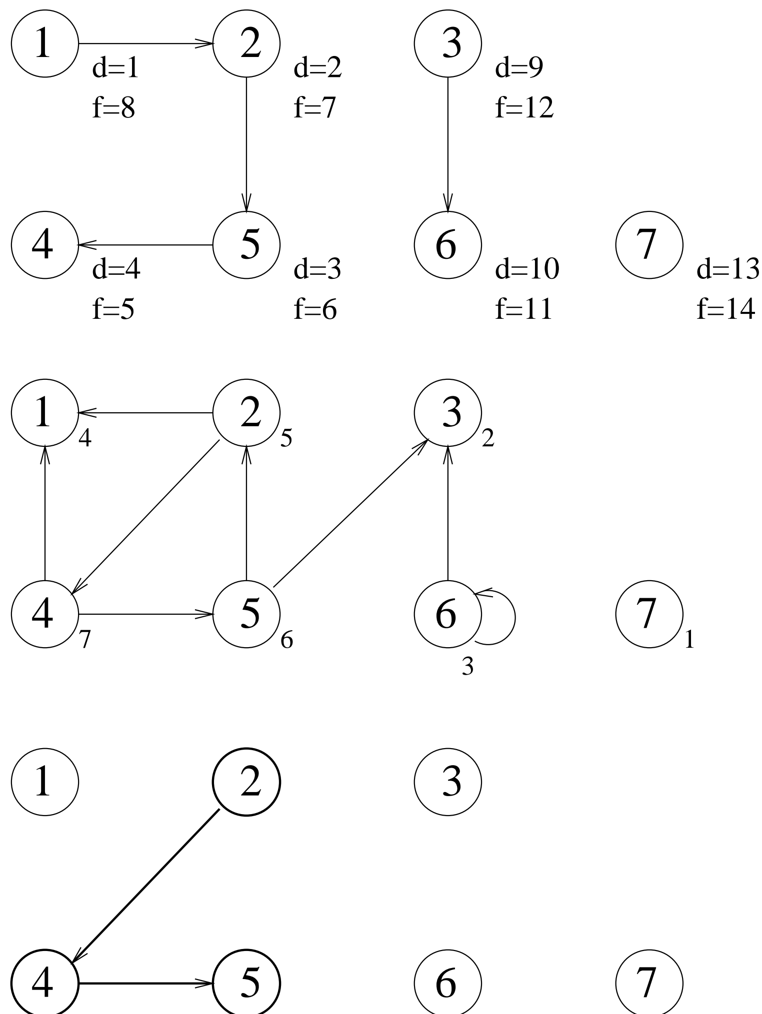

Figure 6.7 – Exemple de recherche de composantes fortement connexes

La figure 6.7 indique les 3 étapes de recherche de composantes fortement connexes. Dans la première on effectue un parcours en profondeur d'abord ; on calcule ensuite le graphe transposé et on effectue un second parcours en profondeur, en suivant l'ordre décroissant des dates de fin de traitement du premier

parcours (indiqué en bas à droite des sommets dans le graphe transposé). On retrouve les 5 composantes fortement connexes ($\{1\}$, $\{2,4,5\}$, $\{3\}$, $\{6\}$, $\{7\}$).

Certains pourront trouver bien complexe un algorithme résolvant un problème en apparence très simple. Lorsque l'on regarde un graphe, les composantes fortement connexes apparaissent *à l'oeil nu*, sans que leur détermination pose beaucoup de problème. Il faut cependant bien comprendre que la recherche de composantes fortement connexes s'applique en pratique à des graphes de taille colossale, le nombre de sommets pouvant largement dépasser plusieurs millions, par exemple pour des applications de conception de circuits électroniques. Il est clair que dans de tels cas un algorithme automatique, de complexité modérée, est particulièrement précieux.

6.5 Recherche de chemins les plus courts dans un graphe

6.5.1 Calcul des chemins à partir de la matrice d'adjacence

Nous avons déjà dit que la matrice d'adjacence, bien que coûteuse en terme de mémoire, dispose de propriétés très intéressantes ; voici la principale :

Proposition 6.1 *Soit $G = (S, A)$ un graphe de matrice d'adjacence M. Le nombre de chemins de longueur k reliant i à j est exactement $\left(M^k\right)_{ij}$.*

Preuve : La preuve de cette proposition s'effectue par récurrence sur k. Elle est en effet claire pour $k = 1$ car on obtient alors la définition même de la matrice d'adjacence, les chemins de longueur 1 étant les arcs élémentaires.

Supposons la propriété est vraie au rang $k-1$. Pour tous sommets i et j, un chemin de longueur k reliant i à j peut être décomposé en un chemin de longueur $k-1$ reliant i à un sommet s suivi d'un chemin de longueur 1 (un arc) reliant s à j. Le nombre de chemins de longueur k reliant i à j est donc égal à

$$\sum_{s=1}^{n} \left(M^{k-1}\right)_{is} \times M_{sj} = \left(M^k\right)_{ij}$$

en utilisant l'hypothèse de récurrence et la définition de la multiplication matricielle.

On obtient par conséquent la proposition suivante en considérant que s'il existe un chemin reliant i à j dans un graphe à n sommets, il en existe nécessairement un de longueur inférieure ou égale à n (il suffit de prendre un chemin quelconque et d'un supprimer tous les cycles, ce qui fournit un chemin passant au plus une fois par chaque sommet du graphe) :

Proposition 6.2 *Soit $G = (S, A)$ un graphe de matrice d'adjacence M. Il existe un chemin reliant i à j si et seulement si $N_{ij} \neq 0$, où $N = M + M^2 + \cdots + M^n$.*

Nous pouvons maintenant définir la *fermeture transitive* d'un graphe $G = (S, A)$ comme étant le graphe dont l'ensemble des sommets est S et l'ensemble des arcs A^* est défini par $(i, j) \in A^*$ si et seulement si il existe un chemin reliant i à j dans le graphe G (voir l'exemple de la figure 6.8).

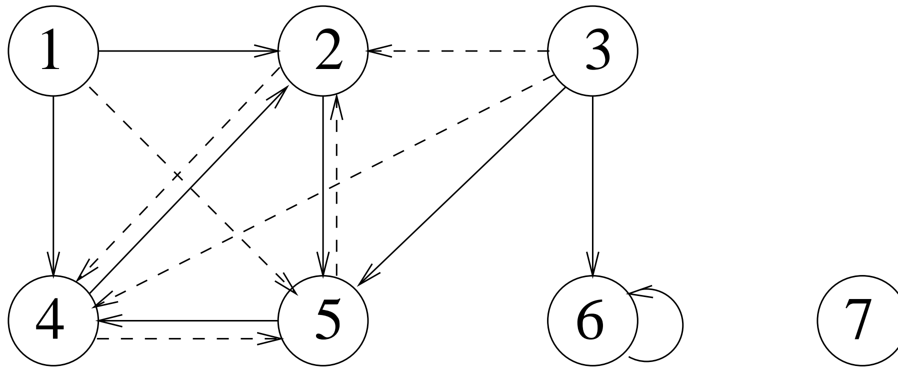

FIGURE 6.8 – Un exemple de fermeture transitive

La matrice d'adjacence M^* de la fermeture transitive G^* du graphe G peut être simplement calculé à partir de la matrice d'adjacence M de G de la manière suivante :

$$M^*_{ij} = \begin{cases} 1 & \text{si } N_{ij} \neq 0 \\ 0 & \text{sinon} \end{cases}$$

où $N = M^0 + M^1 + M^2 + \cdots + M^n$.

Ceci nous fournit un algorithme de complexité $\Theta(n^4)$. On peut faire mieux en utilisant un algorithme plus efficace de multiplication matricielle mais l'algorithme devient alors très complexe et quasiment inutilisable en pratique. Un algorithme efficace peut cependant être déduit de la procédure précédente ; il consiste à calculer les matrices $T^{(k)}$ définies par la relation de récurrence :

$$T^{(0)}_{ij} = M_{ij} \vee (i = j)$$

Pour $k \geq 1$:

$$T^{(k)}_{ij} = T^{(k-1)}_{ij} \vee \left(T^{(k-1)}_{ik} \wedge T^{(k-1)}_{kj} \right)$$

où $\vee$ et $\wedge$ désignent respectivement le *ou* et le *et* logiques.

Proposition 6.3 $M^* = T^{(n)}$

Nous obtenons ainsi un algorithme en $\Theta(n^3)$ pour le calcul de la fermeture transitive d'un graphe. En fait cet algorithme se généralise à de nombreux problèmes aussi nous en démontrerons la validité dans un cadre plus général dans la prochaine section.

Algorithme 6.3 FERMETURE-EFFICACE(G)

1 soit n le nombre de sommets de G
2 soit ℓ une matrice $n \times n$
3 **pour** $i \leftarrow 1$ **à** n **faire**
4   **pour** $j \leftarrow 1$ **à** n **faire**
5   **si** $i = j$ **alors**
6             $\ell_{ij} \leftarrow 1$
7   **sinon**
8             $\ell_{ij} \leftarrow M_{ij}$

9 **pour** $k \leftarrow 1$ **à** n **faire**
10 **pour** $i \leftarrow 1$ **à** n **faire**
11 **pour** $j \leftarrow 1$ **à** n **faire**
12 $\ell_{ij} \leftarrow \ell_{ij} \vee (\ell_{ik} \wedge \ell_{kj})$
13 **return** ℓ

6.5.2 Algorithme de Aho-Hopcroft-Ullman

On considère maintenant un graphe $G = (S, A)$ pondéré, i.e. auquel une valeur appelée *poids*, est associée à chaque arc. L'ensemble des sommets peut par exemple représenter les villes d'un pays et les arcs correspondre aux routes reliant ces villes. Le poids de chaque arc correspond alors à la longueur de la route correspondante.

On appellera *poids* d'un chemin la somme des poids des arcs qui le composent. On se pose alors le problème suivant : étant donné deux sommets, trouver le chemin de poids minimal reliant ces deux sommets, s'il existe. Dans l'exemple précédant, ceci revient à recherche le plus court chemin reliant deux villes quelconques.

Afin de résoudre ce problème, nous allons considérer le cas général où les poids des arcs font partie d'une structure algébrique $\mathcal{S}(\sqcup, \odot, \emptyset, \varepsilon)$, appelée *semi-anneau*, vérifiant les propriétés suivantes :

— $\sqcup$ est une loi de composition interne sur $\mathcal{S}$, commutative, associative, idempotente ($x \sqcup x = x$) et dont $\emptyset$ est un élément neutre. On impose de plus qu'étant donnée une séquence infinie dénombrable $s_1, s_2, \ldots$ d'éléments de $\mathcal{S}$, $s_1 \sqcup s_2 \sqcup \ldots$ appartienne encore à $\mathcal{S}$.
— $\odot$ est associative et ε est un élément neutre de $\odot$.
— $\emptyset$ est absorbant pour $\odot$ ($\emptyset \odot x = x \odot \emptyset = \emptyset$).
— pour tout $x \in \mathcal{S}$, on définit $x^* \in \mathcal{S}$ tel que

$$x^* = \varepsilon \sqcup x \sqcup x \odot x \sqcup x \odot x \odot x \sqcup \ldots$$

On peut par exemple vérifier que $\mathcal{S} = \{\mathbb{R}^+ \cup \infty\}(\min, +, \infty, 0)$ est bien un semi-anneau. Intuitivement, toujours en considérant le graphes des villes reliées par des routes, le poids d'un arc sera égal à la distance ($\in \mathbb{R}$) entre deux villes reliées directement par une route. La longueur d'un chemin est la somme (lois $\odot$) des poids des arcs qui le composent et la longueur du plus court chemin est bien le minimum (loi $\sqcup$) des longueurs des chemins existants. Dans cet exemple, la notation x^* ne sert pas car pour tout $x \in \mathcal{S}$, $x^* = 0$.

Considérons maintenant le problème suivant : soit $G = (S, A)$ un graphe, soit $S(\sqcup, \odot, \emptyset, \varepsilon)$ un semi-anneau, soit p une fonction de A dans S associant à tout arc un élément de $\mathcal{S}$. Si $(i, j) \notin A$, on pose $p(i, j) = \emptyset$. On étend de plus la fonction p aux chemins de la manière suivante : $p(s_1, s_2, \ldots s_k) = p(s_1, s_2) \odot p(s_2, s_3) \odot \ldots p(s_{k-1}, s_k$. On cherche à déterminer pour chaque couple (i, j) de sommets du graphe le coût optimal ℓ_{ij} des chemins reliant i à j relativement au semi-anneau $\mathcal{S}$:

$$\ell_{ij} = \bigsqcup_{chemin \in Chemins(i,j)} p(chemin)$$

Il est en général impossible d'énumérer l'ensemble des chemins reliant deux sommets car ils sont bien trop nombreux. Nous allons donc utiliser l'algorithme

de Aho, Hopcroft et Ullman pour calculer la matrice ℓ en temps $\Theta(n^3)$, où n est le nombre de sommets.

Algorithme 6.4 AHO-HOPCROFT-ULLMAN($G, \mathcal{S}, p$)

1 soit n le nombre de sommets de G
2 soit ℓ une matrice $n \times n$
3 **pour** $i \leftarrow 1$ **à** n **faire**
4   **pour** $j \leftarrow 1$ **à** n **faire**
5   **si** $i = j$ **alors**
6             $\ell_{ij} \leftarrow \varepsilon \sqcup p(i,j)$
7   **sinon**
8             $\ell_{ij} \leftarrow p(i,j)$
9 **pour** $k \leftarrow 1$ **à** n **faire**
10   **pour** $i \leftarrow 1$ **à** n **faire**
11   **pour** $j \leftarrow 1$ **à** n **faire**
12             $\ell_{ij} \leftarrow \ell_{ij} \sqcup (\ell_{ik} \odot (\ell_{kk})^* \odot \ell_{kj})$
13 **return** ℓ

Pour comprendre pourquoi cet algorithme résout bien le problème posé, notons $\ell_{ij}^{(0)}$ la matrice initiale définie par $\ell_{ij}^{(0)} = \varepsilon \sqcup p(i,j)$ si $i = j$ et $\ell_{ij}^{(0)} = p(i,j)$ sinon. Définissons récursivement la suite de matrices $\ell_{ij}^{(k)}$ par l'équation

$$\forall (i,j) \in [1,n]^2 \quad \ell_{ij}^{(k)} = \ell_{ij}^{(k-1)} \sqcup (\ell_{ik}^{(k-1)} \odot (\ell_{kk}^{(k-1)})^* \odot \ell_{kj}^{(k-1)})$$

Lemme : le coefficient $\ell_{ij}^{(k)}$ contient le coût minimal des chemins reliant i à j en passant uniquement par des sommets intermédiaires de numéro inférieur ou égal à k.

La preuve de ce lemme s'effectue par récurrence. La propriété est clairement vraie pour $k = 0$. Si la propriété est vérifiée au rang $k-1$, chaque chemin reliant i à j en passant par les sommets de rang inférieur ou égal à k peut soit ne pas passer par k, soit être décomposé en un chemin allant de i à k, d'éventuels cycles partant et arrivant en k et finalement un chemin de k à i. En admettant l'hypothèse de récurrence, ceci explique que le chemin de coût minimal soit obtenu par la formule de récurrence précédemment indiquée, $(\ell_{kk}^{(k-1)})^*$ traduisant le coût minimal d'un nombre quelconque de cycles partant et arrivant en k.

Enfin, une observation attentive de l'évolution de l'algorithme montre que l'on peut implémenter l'algorithme en n'utilisant qu'une unique matrice ℓ pour calculer les matrices $\ell^{(k)}$ successives.

Finalement, l'algorithme calcule $\ell_{ij}^{(n)}$, i.e. le coût minimal des chemins reliant i à j en passant par des sommets intermédiaires de numéro inférieur ou égal à n, c'est-à-dire par n'importe quel sommet. Notez qu'en l'état, l'algorithme calcul le coût minimal sans pour autant fournir le chemin qui le réalise. Ceci peut cependant être réaliser en mémorisant en permanence les chemins minimaux.

Le principal intérêt d'une formalisation du problème au moyen de semi-groupes est que le choix de ce dernier permet de résoudre divers problèmes. Nous avons déjà vu que $\mathcal{S} = \{\mathbb{R}^+ \cup \infty\}(\min, +, \infty, 0)$ correspond à la recherche des plus courts chemins. Si l'on choisit $\mathcal{S} = \{vrai, faux\}(et, ou, faux, vrai)$ en étiquetant chaque arc avec la valeur $vrai$, on retrouve l'algorithme de calcul de la fermeture transitive

exposé dans la précédente section. Si $\mathcal{S} = \{\mathbb{R}^+ \cup \infty\}(\max, \min, 0, \infty)$, on trouve les chemins de capacité maximale (pensez par exemple à un réseau de communication pour lequel le taux transmission d'un chemin est égal au minimum des taux de transmission de chaque liaison intermédiaire).

6.6 Exercices

Exercice 6.1 *Programmer en C la représentation des graphes au moyen de matrices d'adjacence et de listes de successeurs ainsi que les fonctions permettant de passer d'une représentation à l'autre.*

Exercice 6.2 *Coder en C une représentation des graphes au moyen de matrices d'adjacence "creuses" en utilisant des listes ou des listes de listes, ainsi que les opérations d'addition et de multiplication sur de telles matrices.*

Exercice 6.3 *Programmer l'algorithme de tri topologique sur les graphes.*

Exercice 6.4 *Programmer les algorithmes de calcul de composantes fortement connexes d'un graphe.*

Exercice 6.5 *On considère un graphe dont les sommets sont des monnaies et arcs pondérés par les taux de change entre monnaies. Écrire un programme recherchant un moyen de s'enrichir au moyen de changes successifs.*

Chapitre 7

Analyse syntaxique

Le terme d'analyse syntaxique regroupe l'étude des différents types de langages, ce terme étant pris dans un sens très large, du point de vue de leur syntaxe et pas de leur sens. Ainsi, dans le langage courant, la phrase "*chien voiture mange*" n'est pas syntaxiquement correcte alors que la phrase "*le chien mange une voiture*" respecte la syntaxe du langage mais n'est pas sémantiquement correcte. Les langages de programmation peuvent ainsi s'étudier du point de vue de la syntaxe et/ou de la sémantique. Typiquement, un programme syntaxiquement correct est un programme qui compile sans erreur alors qu'un programme est sémantiquement correct s'il compile mais surtout s'il fait ce qu'il est sensé faire. L'expérience courante montre que le fait de compiler sans erreur n'est en rien une garantie suffisante à la correction d'un programme !

On peut s'intéresser à des langages beaucoup plus simples, par exemple le langage des expressions mathématiques préfixées ou infixées, le langage des formules propositionnelles, le langage des expressions régulières... Dans ce chapitre, nous ne ferons qu'aborder un problème de base de l'analyse syntaxique, à savoir la recherche d'un mot dans un texte, ces deux termes étant ici très généraux. L'étude des langages peut en effet mener très loin et constitue en particulier un préliminaire fondamental à la conception de compilateurs.

7.1 Définitions

Un *alphabet* est par définition un simple ensemble fini de *symboles*. Un *mot* sur un alphabet Σ est une suite finie de symboles de Σ.

Si l'on considère par exemple l'alphabet latin $\Sigma = \{a, b, c, \dots, z\}$, les mots sont donc simplement des suites de lettres.

Pour des raisons techniques, on considère un mot *vide*, en général noté ε, qui ne contient aucun symbole. On peut définir plusieurs fonctions naturelles sur les mots, celle qui renvoie la longueur, i.e. le nombre de symboles, celle qui prend deux mots en argument et qui retourne leur concaténation,...

On dit encore qu'un mot est préfixe d'un autre si ce mot apparaît intégralement en tête de l'autre. De même, un mot est suffixe d'un autre s'il apparaît à la fin.

D'un point de vue pratique, un codage naturel des mots (qui sont de taille finie mais quelconque) est la liste mais en pratique on les code souvent dans des tableaux.

Venons en maintenant au problème qui nous intéresse, la recherche de motif dans un texte (en anglais *Pattern-matching*). Il consiste simplement à trouver toutes les occurrences d'un mot dans un texte.

On considère donc un texte T de longueur n codé par un tableau $T[1,\dots,n]$ d'éléments de Σ ainsi qu'un motif P de longueur m codé par un tableau $P[1,\dots,m]$ d'éléments de Σ. On recherche tout *décalage* s ($0 \leq s \leq n-m$) avec lequel P apparaît dans T, c'est-à-dire tel que :

$$\forall j \in [1,m], P[j] = T[s+j]$$

Ce problème, bien que très simple en apparence, a beaucoup d'intérêt pratique, et pas seulement pour rechercher des mots dans un traitement de texte. On peut par exemple penser à la recherche de motifs dans des bases de données génétiques ; l'alphabet utilisé est alors composé de quatre symboles notés A, T, G et C et les motifs recherchés peuvent par exemple correspondre à des codages d'acides aminés ou de précurseurs de transcription,...

7.1.1 Algorithme naïf de pattern-matching

Un premier algorithme, très simple mais peu efficace, est décrit ci-dessous.

Algorithme 7.1 RECH-MOTIF-NAÏVE(P,T)

1 **pour** $s \leftarrow 0$ **à** $n-m$ **faire**
2   **si** DÉCALAGE-VALIDE(P,T,s) **alors**
3   classer s parmi les décalages valides

4 DÉCALAGE-VALIDE(P,T,s) =
5     $\forall j \in [1,m], P[j] = T[s+j]$

La complexité dans le cas le pire est en $\Theta((n-m+1)\times m)$. Elle n'est clairement pas optimale car, intuitivement, l'information trouvée pour s est ignorée pour $s+1$.

7.1.2 Algorithme de Rabin-Karp

Afin d'obtenir un algorithme de complexité bien meilleure, on peut utiliser un astucieux codage des mots sous forme d'entiers. Soit d le nombre de symboles qui composent l'alphabet Σ. On définit les entiers suivants au moyen d'une bijection canonique entre les symboles et l'ensemble $[0,d[$:

— $p = \sum_{j=1}^{m} P[j]d^{m-j}$
— pour tout $s \in [0,n-m]$, $t_s = \sum_{j=1}^{m} T[s+j]d^{m-j}$

Les entiers ainsi définis peuvent être vus comme un codage en base d des mots. Notez que le calcule de t_{s+1} se fait simplement à partir de t_s en temps constant au moyen de l'équation de récurrence :

$$t_{s+1} = (t_s - T[s+1]\times h)\times d + T[s+m+1]$$

où $h = d^{m-1}$.

L'algorithme de recherche de motif de Rabin et Karp s'écrit alors de la manière suivante :

Algorithme 7.2 RABIN-KARP-PARTIC(P,T,d)

1 calcul de $h \leftarrow d^{m-1}$
2 calcul de p
3 calcul de $t \leftarrow t_0$
4 **pour** $s \leftarrow 0$ **à** $n - m - 1$ **faire**
5 **si** $p = t$ **alors** classer s valide
6 $t \leftarrow (t - T[s+1]h)d + T[s+m+1]$
7 **si** $p = t$ **alors** classer $n - m$ valide

Le calcul de la complexité de cet algorithme doit tenir compte du calcul de h (en $\Theta(\ln m)$), des calculs de p et t_0 (en $\Theta(m)$) et des $n-m$ calculs des t_s (en $\Theta(1)$). La complexité globale dans le cas le pire est donc en $\Theta(n+m)$, ce qui est optimal d'un point de vue asymptotique car pour rechercher un mot de longueur m dans un texte de n caractères il faut bien parcourir au moins une fois l'intégralité du motif et du texte, soit déjà $\Theta(n+m)$ opérations.

Remarquons cependant que si p et t_s sont grands, les hypothèses implicites que nous avons utilisé (l'exécution des opérations arithmétiques en temps constant) ne sont plus valables. Une solution consiste donc à faire les calculs modulo un entier q ($\geq m$). On obtient l'algorithme suivant :

Algorithme 7.3 Rabin-Karp(P, T, d, q)

1 calcul de $h \leftarrow d^{m-1} \bmod q$
2 calcul de $p \leftarrow p \bmod q$
3 calcul de $t \leftarrow t_0 \bmod q$
4 **pour** $s \leftarrow 0$ **à** $n - m - 1$ **faire**
5 **si** $p = t$ **alors**
6 **si** Décalage-Valide(P, T, s) **alors**
7 classer s valide
8 $t \leftarrow ((t - T[s+1]h)d + T[s+m+1]) \bmod q$
9 **si** $p = t$ **alors**
10 **si** Décalage-Valide(P, T, s) **alors**
11 classer $n - m$ valide

La complexité de ce nouvel algorithme redevient très mauvaise dans le cas le pire ($\Theta((n-m+1)m)$) mais s'il y a peu de décalages valides, ce qui est en général le cas, la complexité est bien optimale ($\Theta(n+m)$).

7.2 Pattern-matching à base d'automates finis

Un autre algorithme, très élégant, de recherche de motif nécessite l'introduction d'une notion fondamentale de l'informatique théorique, celle d'*automate*.

7.2.1 Définition des automates finis déterministes

Un automate fini déterministe, noté $(Q, q_0, F, \Sigma, \delta)$, est constitué de 5 éléments vérifiant les propriétés suivantes :

— Q est ensemble fini d'*états* ;
— $q_0 \in Q$ est un état particulier appelé *état initial* ;
— $F \subseteq Q$ est un ensemble d'états appelés *états finaux* ;

— Σ est un alphabet fini ;
— $\delta \; : Q \times \Sigma \mapsto Q$ est une fonction prenant en argument un état et un symbole de l'alphabet et qui fournit un nouvel état. On l'appelle *fonction de transition* de l'automate ; c'est elle qui en décrit l'évolution.

Une telle définition ne laisse pas bien comprendre ce qu'est réellement un automate. En fait, comme son nom l'indique, il faut le voir comme une machine pouvant prendre plusieurs états, dont l'état de départ est fixé (q_0), et dont l'évolution dépend de la fonction de transition et d'un mot sur l'alphabet Σ. On représente habituellement les automates sous la forme de graphes, celui de la figure 7.1 représentant l'automate suivant :

— $Q = \{1, 2, 3, 4\}$
— $q_0 = 1$
— $F = \{1, 2, 3\}$
— $\Sigma = \{0, 1\}$
— $\delta \; : Q \times \Sigma \mapsto Q$ définie ci-contre.

Q	Σ	Q
1	0	3
1	1	2
2	0	4
2	1	1
3	1	3
3	1	4
4	0	2
4	1	3

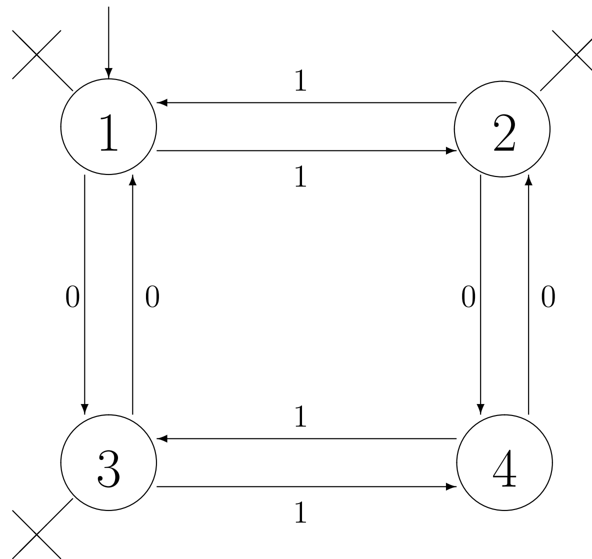

FIGURE 7.1 – Un exemple d'automate

Un automate M sert à *reconnaître* des mots. Pour cela, M démarre dans l'état initial q_0. On lit les caractères un à un et pour chaque caractère, on fait évoluer l'état de M qui passe de l'état q à q' en lisant le symbole x si et seulement si $\delta(q, x) = q'$. Le mot w est *reconnu* (ou *accepté*) si en fin de lecture, M se retrouve dans un état final $q \in F$, autrement le mot est *rejeté*.

Par exemple l'automate de la figure 7.1 accepte 11010 mais refuse 101001.

Nous allons utiliser des automates finis déterministes afin de rechercher un motif dans un texte ; le motif permettra de construire l'automate et le texte sera un mot à reconnaître.

7.2.2 Automate de recherche de motif

Pour tout motif P de longueur m, on peut construire un automate M qui le reconnaisse tel que :

- M ait $m+1$ états : $\{0, 1, \ldots, m\}$;
- 0 soit l'état initial ;
- m soit le seul état final.

Par exemple, avec l'alphabet $\Sigma = \{a, b\}$, l'automate de la figure 7.2 reconnaît le mot *aabba* et seulement lui.

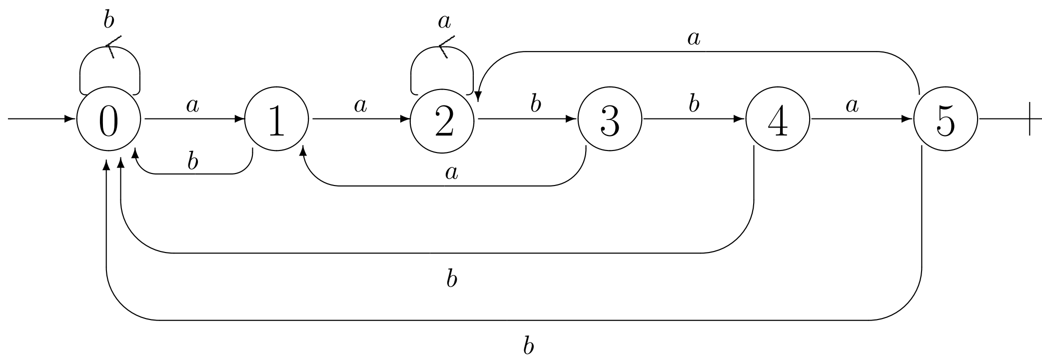

Figure 7.2 – Automate de recherche de motif

On en déduit un algorithme très simple, de complexité optimale $\Theta(n)$, permettant de rechercher un motif dans un texte, une fois construit l'automate correspondant au motif :

Algorithme 7.4 Rech-Automate(δ, T)

1 $q \leftarrow 0$
2 **pour** $s \leftarrow 1$ **à** n **faire**
3 $\quad q \leftarrow \delta(q, T[s])$
4 $\quad$ **si** $q = m$ **alors**
5 $\quad\quad$ classer $s - m$ valide

7.2.3 Calcul de l'automate de recherche

Le problème de la construction de l'automate de recherche n'est cependant pas trivial. Il est fabriqué à partir d'un "squelette" linéaire de m transitions correspondant au motif P. La difficulté réside dans la détermination de la fonction de transition δ.

Soit P_q le préfixe de longueur q de P. La fonction de transition δ doit être telle que $\delta(q, x)$ soit égal à la longueur du plus long préfixe de P également suffixe de la concaténation de P_q et x.

Ainsi dans l'exemple de la figure 7.2, $\delta(3, a)$ est égal à la longueur du plus long préfixe de *aabba* également suffixe de *aaba*, i.e. égal à 1. De même, $\delta(5, a)$ est égal à la longueur du plus long préfixe de *aabba* également suffixe de *aabbaa*, i.e. égal à 2 (le mot commun étant *aa*).

La complexité du calcul de l'automate de recherche est donc en $\Theta(m \times |\Sigma|)$, d'où une complexité globale du problème de la recherche de motif en $\Theta(n+m\times|\Sigma|)$. Cette approche est donc particulièrement efficace si l'on travail avec un alphabet de taille restreinte, ce qui est en général le cas (pensez à la recherche de séquence dans une base de donnée génétique).

7.3 Exercices

Exercice 7.1 *Programmer les algorithmes suivants de recherche de motif dans un texte et en comparer l'efficacité pratique :*

1. *l'algorithme naïf,*
2. *l'algorithme de Rabin-Karp,*
3. *l'algorithme à base d'automates finis.*

Chapitre 8

Conclusion : les limites de l'algorithmique

Les nombreux algorithmes étudiés dans ce cours sont efficaces, i.e. de complexité polynomiale en la taille des données à traiter. Il ne faudrait cependant pas en déduire que tous les problèmes posés à l'algorithmique peuvent être résolus aussi efficacement. Il existe en effet les problèmes dont on peut prouver la difficulté et même des problèmes pour lesquels il n'existe pas de moyen automatique de résolution. Afin de mieux comprendre pourquoi certains problèmes sont réellement difficiles, commençons par les classer en grandes familles.

Problèmes abstraits

Un *problème abstrait* est une relation entre une ensemble d'*instances* et un ensemble de *solutions*.

Par exemple, le problème de la recherche du plus court chemin entre deux sommets d'un graphe est composés d'instances du type (G, s_1, s_2) et de solutions, des suites de sommets de G.

Problèmes de décision

Un *problème de décision* est un problème dont l'ensemble des solutions possibles est simplement VRAI ou FAUX.

Par exemple l'existence d'un chemin de longueur k entre deux sommets d'un graphe est un problème de décision sur des instances du type (G, s_1, s_2, k).

La **classe de complexité P** est formée de l'ensemble des problèmes de décision solubles en temps polynomial : si n est la taille de l'instance, alors il existe un algorithme A résolvant le problème en $O(n^k)$, pour une certaine constante k.

Problèmes de validation

Un *problème de validation* a pour solution VRAI ou FAUX, mais dispose de deux arguments, une instance et un *certificat*.

Par exemple, la recherche d'un chemin de longueur k entre deux sommets d'un graphe est un problème de validation si en plus de l'instance du type (G, s_1, s_2, k) on fournit comme certificat un chemin de s_1 à s_2 de longueur k.

Un problème de validation est par conséquent au moins aussi simple que le problème de décision associé.

Certains problèmes de validation sont faciles à résoudre alors que le problème de décision associé est difficile. C'est par exemple le cas de la recherche de *cycle hamiltonien* dans un graphe non orienté G, i.e. d'un cycle élémentaire passant par tous les sommets de G. Connaissant un tel cycle, il est facile de vérifier qu'il passe bien par tous les sommets du graphe mais nous ne connaissons pas à l'heure actuelle d'algorithme capable de trouver un tel cycle en temps polynomial (et il n'en existe très probablement pas).

La **classe de complexité NP** est formée de l'ensemble des problèmes de validation solubles en temps polynomial.

La classe **P** est donc clairement incluse dans la classe **NP**. Le problème ouvert le plus important de la théorie de la complexité est cependant de démontrer que ces deux classes ne sont pas confondues, ce que l'on note en général

$$\mathbf{P} \stackrel{?}{=} \mathbf{NP}$$

La classe de complexité **NP** est assez surprenante. Il existe en effet des problèmes dits **NP**-complets qui sont en quelque sorte les plus durs de la classe. Plus précisément, la résolution de n'importe quel problème de la classe **NP** se ramène à la résolution de n'importe quel problème **NP**-complet. Par conséquent, la découverte d'un algorithme polynomial permettant de résoudre un seul problème **NP**-complet démontrerait l'égalité des classes **P** et **NP**.

L'exemple le plus classique de problème **NP**-complet provient de la logique ; il s'agit du problème de la satisfiabilité d'une formule logique.

Les formules du *calcul des propositions* sont définies de la manière suivante :

1. $x_1, x_2, \ldots \in \mathcal{F}$ (les variables simples sont des formules) ;
2. Si ϕ_1 et ϕ_2 sont des formules alors $(\neg\phi_1), (\phi_1 \vee \phi_2), (\phi_1 \wedge \phi_2), (\phi_1 \Rightarrow \phi_2), (\phi_1 \Leftrightarrow \phi_2)$ sont aussi des formules.

On dit qu'une formule ϕ est *satisfiable* s'il existe une valeur des variables telle que l'interprétation de ϕ vaille VRAI. Le problème, noté **SAT** de savoir si une formule est satisfiable est **NP**-complet. Il est facile d'écrire un algorithme de recherche de solution de complexité $\Theta(2^n)$, où n est le nombre de variables, mais on ne connaît pas d'algorithme polynomial en n pour résoudre **SAT**.

Il existe beaucoup d'autres problèmes **NP**-complets. Par exemple, étant donné un ensemble *fini* E d'entiers naturels et un entier naturel cible noté c, existe-t-il $F \subseteq E$ tel que $\sum_{i \in F} i = c$?

Citons encore le célèbre problème du *voyageur de commerce* : étant données n villes, quelle tournée choisir pour minimiser la distance globale à parcourir ?

Il existe bien d'autres classes de complexité et d'autres types de problèmes comme les problèmes d'approximation. L'étude de ces classes constitue l'activité principale des recherches en théorie de la complexité.

Problèmes indécidables

Comme nous le disions en préambule, il ne faudrait pas croire que tout problème est soluble de manière automatique à condition d'y *mettre le prix*, i.e. en

s'autorisant une complexité même gigantesque. Il existe en effet des problèmes dont on peut prouver l'*indécidabilité* comme par exemple celui de la terminaison des programmes.

Supposons qu'il existe un programme capable de décider simplement en lisant un code source d'un programme C si l'exécution de ce programme se terminera ou au contraire s'exécutera indéfiniment. Considérons alors le petit programme suivant utilisant une telle fonction booléenne notée TERMINAISON :

Algorithme 8.1 PREUVE-ABSURDE()

1 **tant que** TERMINAISON(PREUVE-ABSURDE) **faire**
  ▷ *On ne fait rien dans la boucle*

Si ce programme se termine, TERMINAISON doit renvoyer VRAI et par conséquent le programme est une *boucle folle* qui ne se terminera jamais. Inversement, si le programme ne se termine pas, il se termine… ! Ceci donne une idée de la démonstration d'indécidabilité du problème de l'arrêt.

Afin d'avoir une idée de la difficulté du problème de l'arrêt, considérons une fonction SUCCESSEUR qui parcourt tout les entiers dans $[1, +\infty)^3 \times [3, +\infty)$. Que dire alors de la terminaison du programme suivant :

Algorithme 8.2 FERMAT(x, y, z, n)

1 **tant que** $x^n + y^n \neq z^n$ **faire**
2 $(x, y, z, n) \leftarrow$ SUCCESSEUR(x, y, z, n)

Dés que l'on cherche à écrire des programmes de vérification automatique de programmes, on se heurte immédiatement à ce genre de problème. Il est en particulier malheureusement impossible de vérifier qu'un programme fait bien ce que l'on attend de lui même si certaines techniques permettent de faire des vérifications partielles.

Bibliographie

[1] Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, 1976.

[2] Jon Barwise, *Handbook of Mathematical Logic*, North-Holland Publishing Company, 1977.

[3] **Thomas Cormen, Charles Leiserson, Ronald Rivest, *Introduction à l'algorithmique*, Dunod, 2002.**

[4] G. H. Gonnet, R. Baeza-Yates, *Handbook of Algorithms and Data Structures In Pascal and C*, Second Edition, Addison-Wesley Publishing Company, 1990.

[5] Donald E. Knuth, *Fundamental Algorithms — The Art of Computer Programming*, vol. 1, Addison-Wesley Publishing Company, 1968.

[6] Donald E. Knuth, *Seminumerical Algorithms — The Art of Computer Programming*, vol. 2, Addison-Wesley Publishing Company, 1969.

[7] Donald E. Knuth, *Sorting and Searching — The Art of Computer Programming*, vol. 3, Addison-Wesley Publishing Company, 1973.

[8] William H. Press, Brian P. Flannery, Saul A. Teukolsky, William T. Vetterling, *Numerical Recipes in C : The Art of Scientific Computing*, Second Edition, Cambridge University Press, 1992.

[9] Robert Sedgewick, *Algorithms in C*, Addison-Wesley Publishing Company, 1990.

[10] Jacques Stern, *Fondements mathématiques de l'informatique*, McGraw-Hill, 1990.