Éléments d'Algorithmique
Ensta - in101

Françoise Levy-dit-Vehel & Matthieu Finiasz

Année 2009-2010

Table des matières

1 Complexité **1**
- 1.1 Définitions . . . 1
  - 1.1.1 Comparaison asymptotique de fonctions . . . 1
  - 1.1.2 Complexité d'un algorithme . . . 2
  - 1.1.3 Complexité spatiale, complexité variable . . . 3
- 1.2 Un seul problème, quatre algorithmes . . . 6
- 1.3 Un premier algorithme pour le tri . . . 9

2 Récursivité **11**
- 2.1 Conception des algorithmes . . . 11
- 2.2 Problème des tours de Hanoï . . . 12
- 2.3 Algorithmes de tri . . . 15
  - 2.3.1 Le tri fusion . . . 15
  - 2.3.2 Le tri rapide . . . 17
  - 2.3.3 Complexité minimale d'un algorithme de tri . . . 21
- 2.4 Résolution d'équations de récurrence . . . 22
  - 2.4.1 Récurrences linéaires . . . 22
  - 2.4.2 ★ Récurrences de partitions . . . 24
- 2.5 ★ Compléments . . . 28
  - 2.5.1 ★ Récursivité terminale . . . 28
  - 2.5.2 ★ Dérécursification d'un programme . . . 28
  - 2.5.3 ★ Indécidabilité de la terminaison . . . 30

3 Structures de Données **35**
- 3.1 Tableaux . . . 35
  - 3.1.1 Allocation mémoire d'un tableau . . . 36
  - 3.1.2 ★ Complément : allocation dynamique de tableau . . . 39
- 3.2 Listes chaînées . . . 40
  - 3.2.1 Opérations de base sur une liste . . . 41
  - 3.2.2 Les variantes : doublement chaînées, circulaires... . . . 43
  - 3.2.3 Conclusion sur les listes . . . 44
- 3.3 Piles & Files . . . 44
  - 3.3.1 Les piles . . . 45

3.3.2 Les files 46

4 Recherche en table **49**
4.1 Introduction 49
4.2 Table à adressage direct 50
4.3 Recherche séquentielle 51
4.4 Recherche dichotomique 53
4.5 Tables de hachage 55
4.6 Tableau récapitulatif des complexités 57

5 Arbres **59**
5.1 Préliminaires 59
5.1.1 Définitions et terminologie 59
5.1.2 Premières propriétés 61
5.1.3 Représentation des arbres 62
5.2 Utilisation des arbres 63
5.2.1 Évaluation d'expressions & parcours d'arbres 64
5.2.2 Arbres Binaires de Recherche 66
5.2.3 Tas pour l'implémentation de files de priorité 72
5.2.4 Tri par tas 77
5.3 Arbres équilibrés 78
5.3.1 Rééquilibrage d'arbres 78
5.3.2 Arbres AVL 79

6 Graphes **85**
6.1 Définitions et terminologie 85
6.2 Représentation des graphes 87
6.2.1 Matrice d'adjacence 87
6.2.2 Liste de successeurs 88
6.3 Existence de chemins & fermeture transitive 89
6.4 Parcours de graphes 92
6.4.1 Arborescences 93
6.4.2 Parcours en largeur 93
6.4.3 Parcours en profondeur 97
6.5 Applications des parcours de graphes 99
6.5.1 Tri topologique 99
6.5.2 ⋆ Calcul des composantes fortement connexes 100
6.5.3 Calcul de chemins optimaux 102

7 Recherche de motifs **109**
7.1 Définitions 109
7.2 L'algorithme de Rabin-Karp 110
7.3 Automates pour la recherche de motifs 112

7.3.1 Automates finis 113
7.3.2 Construction d'un automate pour la recherche de motifs 114
7.3.3 ⋆ Reconnaissance d'expression régulières 116

Chapitre 1

Complexité

La notion de complexité est centrale en algorithmique : c'est grâce à elle que l'on peut définir ce qu'est un bon algorithme. La recherche d'algorithmes ayant une complexité plus petite que les meilleurs algorithmes connus est un thème de recherche important dans toutes les branches de l'informatique et des mathématiques appliquées. Dans ce chapitre nous voyons comment est définie cette notion de complexité.

1.1 Définitions

1.1.1 Comparaison asymptotique de fonctions

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 :

$$\forall n > n_0, \quad 0 \leq f(n) \leq c \times g(n).$$

Inversement, quand $g(n) = O(f(n))$, on utilise la notation :

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

et, quand on a à la fois les propriétés (1.1) et (1.2) :

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

Plus formellement, $f(n) = \Theta(g(n))$ si :

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

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 :

- $n^2 + 3n + 1 = \Theta(n^2) = \Theta(50n^2)$,
- $n/\log(n) = O(n)$,
- $50n^{10} = O(n^{10,01})$,
- $2^n = O(\exp(n))$,
- $\exp(n) = O(n!)$,
- $n/\log(n) = \Omega(\sqrt{n})$,
- $\log_2(n) = \Theta(\log(n)) = \Theta(\ln(n))$.

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

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

1.1.2 Complexité d'un algorithme

On appelle complexité d'un algorithme est le nombre asymptotique « d'opérations de base » qu'il doit effectuer en fonction de la taille de l'entrée qu'il a à traiter. Cette complexité est indépendante de la vitesse de la machine sur laquelle est exécuté l'algorithme : la vitesse de la machine (ou la qualité de l'implémentation) peut faire changer le temps d'exécution d'une opération de base, mais ne change pas le nombre d'opérations à effectuer. Une optimisation qui fait changer le nombre d'opérations de base (et donc la complexité) doit être vue comme un changement d'algorithme.

Étant donnée la définition précédente, il convient donc de définir convenablement ce qu'est une opération de base et comment l'on mesure la taille de l'entrée avant de pouvoir parler de la complexité d'un algorithme. Prenons par exemple le code suivant, qui calcule la somme des carrés de 1 à n :

```
1 unsigned int sum_of_squares(unsigned int n) {
2   int i;
3   unsigned int sum = 0;
4   for (i=1; i<n+1; i++) {
5     sum += i*i;
6   }
7   return sum;
8 }
```

Pour un tel algorithme on considère que la taille de l'entrée est n et on cherche à compter le nombre d'opérations de base en fonction de n. L'algorithme effectue des multiplications et des additions, donc il convient de considérer ces opérations comme opérations de base. Il y a au total n multiplications et n additions, donc l'algorithme a une complexité $\Theta(n)$. Le choix de l'opération de base semble ici tout à fait raisonnable car dans un processeur

moderne l'addition et la multiplication sont des opérations qui prennent $\Theta(1)$ cycles pour être effectués.

Imaginons maintenant que l'on modifie l'algorithme pour qu'il puisse manipuler des grands entiers (plus grands que ce que le processeur peut manipuler d'un seul coup) : le coût d'une addition ou d'une multiplication va augmenter quand la taille des entiers va augmenter, et ces opérations n'ont alors plus un coût constant. Il convient alors de changer d'opération de base et de considérer non plus des multiplications/additions complexes mais des opérations binaires élémentaires. Le coût de l'addition de deux nombre entre 0 et n est alors $\Theta(\log n)$ (le nombre de bits nécessaires pour écrire n) et le coût d'une multiplication (en utilisant un algorithme basique) est $\Theta((\log n)^2)$. Calculer la somme des carrés de 1 à n quand n devient grand a donc une complexité $\Theta(n(\log n)^2)$ opérations binaires.

Comme vous le verrez tout au long de ce poly, selon le contexte, l'opération de base à choisir peut donc changer, mais elle reste en général l'opération la plus naturelle.

Ordres de grandeurs de complexités. Jusqu'ici nous avons parlé de complexité asymptotique des algorithmes, mais pour une complexité donnée, il est important d'avoir une idée des ordres de grandeurs des paramètres que l'algorithme pourra traiter. Un algorithme exponentiel sera souvent trop coûteux pour de grandes entrées, mais pourra en général très bien traiter des petites entrées. Voici quelques chiffres pour se faire une idée :

- effectuer 2^{30} (environ un milliard) opérations binaires sur un PC standard prend de l'ordre de la seconde.
- le record actuel de puissance de calcul fourni pour résoudre un problème donné est de l'ordre de 2^{60} opérations binaires.
- aujourd'hui en cryptographie, on considère qu'un problème dont la résolution nécessite 2^{80} opérations binaires est impossible à résoudre.
- une complexité de 2^{128} opérations binaires sera *a priori* toujours inatteignable d'ici quelques dizaines d'années (c'est pour cela que l'on utilise aujourd'hui des clefs de 128 bits en cryptographie symétrique, alors qu'à la fin des années 70 les clefs de 56 bits du DES semblaient suffisamment solides).

Dans la pratique, un algorithme avec une complexité $\Theta(n)$ pourra traiter des entrées jusqu'à $n = 2^{30}$ en un temps raisonnable (cela dépend bien sûr des constantes), et un algorithme cubique (de complexité $\Theta(n^3)$), comme par exemple un algorithme basique pour l'inversion d'une matrice $n \times n$, pourra traiter des données jusqu'à une taille $n = 2^{10} = 1024$. En revanche, inverser une matrice $2^{20} \times 2^{20}$ nécessitera des mois de calcul à plusieurs milliers de machines (à moins d'utiliser un meilleur algorithme...).

1.1.3 Complexité spatiale, complexité variable

Complexité spatiale. Jusqu'ici, la seule complexité dont il a été question est la complexité temporelle : le temps mis pour exécuter un algorithme. Cependant, il peut aussi être intéressant de mesurer d'autres aspects, et en particulier la *complexité spatiale* : la taille mémoire nécessaire à l'exécution d'un algorithme. Comme pour la complexité temporelle,

seule nous intéresse la complexité spatiale asymptotique, toujours en fonction de la taille de l'entrée. Des exemples d'algorithmes avec des complexités spatiales différentes seront donnés dans la section 1.2.

Il est à noter que la complexité spatiale est nécessairement toujours inférieure (ou égale) à la complexité temporelle d'un algorithme : on suppose qu'une opération d'écriture en mémoire prend un temps $\Theta(1)$, donc écrire une mémoire de taille M prend un temps $\Theta(M)$.

Comme pour la complexité temporelle, des bornes existent sur les complexités spatiales atteignables en pratique. Un programme qui utilise moins de 2^{20} entiers (4Mo sur une machine 32-bits) ne pose aucun problème sur une machine standard, en revanche 2^{30} est à la limite, 2^{40} est difficile à atteindre et 2^{50} est à peu près hors de portée.

⋆ Des Modèles de Mémoire Alternatifs

Vous pouvez être amenés à rencontrer des modèles de mémoire alternatifs dans lesquels les temps d'accès/écriture ne sont plus $\Theta(1)$ *mais dépendent de la complexité spatiale totale de l'algorithme. Supposons que la complexité spatiale soit* $\Theta(M)$.
Dans un système informatique standard, pour accéder à une capacité mémoire de taille M*, il faut pouvoir adresser l'ensemble de cet espace. cela signifie qu'une adresse mémoire (un pointeur en C), doit avoir une taille minimale de* $\log_2 M$ *bits, et lire un pointeur a donc une complexité* $\Theta(\log M)$*. Plus un algorithme utilise de mémoire, plus le temps d'accès à cette mémoire est grand. Dans ce modèle, la complexité temporelle d'un algorithme est toujours au moins égale à* $\Theta(M \log M)$.
Certains modèles vont encore plus loin et prennent aussi en compte des contraintes physiques : dans des puces mémoires planaires (le nombre de couches superposées de transistors est $\Theta(1)$*) comme l'on utilise à l'heure actuelle, le temps pour qu'une information circule d'un bout de la puce mémoire jusqu'au processeur est au moins égal à* $\Theta(\sqrt{M})$*. Dans ce modèle on peut borner la complexité temporelle par* $\Theta(M\sqrt{M})$*. La constante (l'inverse de la vitesse de la lumière) dans le* Θ *est en revanche très petite.*
Dans ce cours, nous considérerons toujours le modèle mémoire standard avec des temps d'accès constants en $\Theta(1)$.

Complexité dans le pire cas. Comme nous avons vu, la complexité d'un algorithme s'exprime en fonction de la taille de l'entrée à traiter. Cependant, certains algorithmes peuvent avoir une complexité très variable pour des entrées de même taille : factoriser un nombre va par exemple dépendre plus de la taille du plus grand facteur que de la taille totale du nombre.

Une technique d'étude des performances d'un tel algorithme à complexité variable consiste à examiner la complexité en temps du *pire cas*. Le temps de calcul dans le pire cas pour les entrées x de taille n fixée est défini par

$$T(n) = \sup_{\{x,\, |x|=n\}} T(x).$$

$T(n)$ fournit donc une borne supérieure sur le temps de calcul sur n'importe quelle

entrée de taille n. Pour calculer $T(n)$, on néglige les constantes dans l'analyse, afin de pouvoir exprimer comment l'algorithme dépend de la taille des données. Pour cela, on utilise la notation O ou Θ, qui permet de donner un ordre de grandeur sans caractériser plus finement. La complexité dans le pire cas est donc indépendante de l'implémentation choisie.

Complexité moyenne. Une autre façon d'étudier les performances d'un algorithme consiste à en déterminer la complexité *moyenne* : c'est la moyenne du temps de calcul sur toutes les données d'une taille n fixée, en supposant connue une distribution de probabilité $(p(n))$ sur l'ensemble des entrées de taille n :

$$T_m(n) = \sum_{x,\,|x|=n} p_n(x)T(x).$$

On remarque que $T_m(n)$ n'est autre que l'espérance de la variable aléatoire « temps de calcul ». Le cas le plus aisé pour déterminer $T_m(n)$ est celui où les données sont équidistribuées dans leur ensemble de définition : on peut alors dans un premier temps évaluer le temps de calcul $T(x)$ de l'algorithme sur une entrée x choisie aléatoirement dans cet ensemble, le calcul de la moyenne des complexités sur toutes les entrées de l'ensemble s'en déduisant facilement [1].

Lorsque la distribution $(p(n))$ n'est pas la distribution uniforme, la complexité moyenne d'un algorithme est en général plus difficile à calculer que la complexité dans le pire cas ; de plus, l'hypothèse d'uniformité peut ne pas refléter la situation pratique d'utilisation de l'algorithme.

⋆ **Analyse en moyenne.** L'analyse en moyenne d'un algorithme consiste à calculer le nombre moyen (selon une distribution de probabilité sur les entrées) de fois que chaque instruction est exécutée, en le multipliant par le temps propre à chaque instruction, et en faisant la somme globale de ces quantités. Elle est donc dépendante de l'implémentation choisie.

Trois difficultés se présentent lorsque l'on veut mener à bien un calcul de complexité en moyenne : d'abord, l'évaluation précise du temps nécessaire à chaque instruction peut s'avérer difficile, essentiellement à cause de la variabilité de ce temps d'une machine à l'autre.

Ensuite, le calcul du nombre moyen de fois que chaque instruction est exécutée peut être délicat, les calculs de borne supérieure étant généralement plus simples. En conséquence, la complexité en moyenne de nombreux algorithmes reste inconnue à ce jour.

Enfin, le modèle de données choisi n'est pas forcément représentatif des ensembles de données rencontrés en pratique. Il se peut même que, pour certains algorithmes, il n'existe pas de modèle connu.

1. Cette hypothèse d'équidistribution rend en effet le calcul de la complexité sur une donnée choisie aléatoirement représentatif de la complexité *pour toutes les données* de l'ensemble.

Pour ces raisons, le temps de calcul dans le pire cas est bien plus souvent utilisé comme mesure de la complexité.

1.2 Un seul problème, quatre algorithmes de complexités très différentes

Pour comprendre l'importance du choix de l'algorithme pour résoudre un problème donné, prenons l'exemple du calcul des nombres de Fibonacci définis de la manière suivante :

$$F_0 = 0,\ F_1 = 1,\ \text{et, } \forall n \geq 2,\ F_n = F_{n-1} + F_{n-2}.$$

Un premier algorithme pour calculer le n-ième nombre de Fibonacci F_n reprend exactement la définition par récurrence ci-dessus :

```
1 unsigned int fibo1(unsigned int n) {
2   if (n < 2) {
3     return n;
4   }
5   return fibo1(n-1) + fibo1(n-2);
6 }
```

L'algorithme obtenu est un algorithme récursif (*cf.* chapitre 2). Si l'on note C_n, le nombre d'appels à `fibo1` nécessaires au calcul de F_n, on a, $C_0 = C_1 = 1$, et, pour $n \geq 2$,

$$C_n = 1 + C_{n-1} + C_{n-2}.$$

Si l'on pose $D_n = (C_n + 1)/2$, on observe que D_n suit exactement la relation de récurrence définissant la suite de Fibonacci (seules les conditions initiales diffèrent).

La résolution de cette récurrence linéaire utilise une technique d'algèbre classique (*cf.* chapitre 2) : on calcule les solutions de l'équation caractéristique de cette récurrence - ici $x^2 - x - 1 = 0$ - qui sont $\phi = \frac{1+\sqrt{5}}{2}$ et $\bar{\phi} = \frac{1-\sqrt{5}}{2}$. D_n s'écrit alors

$$D_n = \lambda\phi^n + \mu\bar{\phi}^n.$$

On détermine λ et μ à l'aide des conditions initiales. On obtient finalement

$$D_n = \frac{1}{\sqrt{5}}(\phi^{n+1} - \bar{\phi}^{n+1}),\ n \geq 0.$$

Étant donné que $C_n = 2D_n - 1$, la complexité - en termes de nombre d'appels à `fibo1` - du calcul de F_n par cet algorithme est donc $\Theta(\phi^n)$, c'est-à-dire exponentiel en n.

On observe que l'algorithme précédent calcule plusieurs fois les valeurs F_k, pour $k < n$, ce qui est bien évidemment inutile. Il est plus judicieux de calculer les valeurs F_k, $2 \leq k \leq n$ à partir des deux valeurs F_{k-1} et F_{k-2}, de les stocker dans un tableau, et de retourner F_n.

```
1  unsigned int fibo2(unsigned int n) {
2    unsigned int* fib = (unsigned int*) malloc((n+1)*sizeof(unsigned int));
3    int i,res;
4    fib[0] = 0;
5    fib[1] = 1;
6    for (i=2; i<n+1; i++) {
7      fib[i] = fib[i-1] + fib[i-2];
8    }
9    res = fib[n];
10   free(fib);
11   return res;
12 }
```

On prend ici comme unité de mesure de complexité le temps de calcul d'une opération d'addition, de soustraction ou de multiplication. La complexité de l'algorithme `Fibo2` est alors $\Theta(n)$, *i.e.* linéaire en n (en effet, la boucle `for` comporte $n-1$ itérations, chaque itération consistant en une addition). La complexité est donc drastiquement réduite par rapport à l'algorithme précédent : le prix à payer ici est une complexité en espace linéaire ($\Theta(n)$ pour stocker le tableau `fib`).

On remarque maintenant que les $n-2$ valeurs F_k, $0 \leq k \leq n-3$ n'ont pas besoin d'être *stockées* pour le calcul de F_n. On peut donc revenir à une complexité en espace en $\Theta(1)$ en ne conservant que les deux dernières valeurs courantes F_{k-1} et F_{k-2} nécessaires au calcul de F_k : on obtient le troisième algorithme suivant :

```
1  unsigned int fibo3(unsigned int n) {
2    unsigned int fib0 = 0;
3    unsigned int fib1 = 1;
4    int i;
5    for (i=2; i<n+1; i++) {
6      fib1 = fib0 + fib1;
7      fib0 = fib1 - fib0;
8    }
9    return fib1;
10 }
```

Cet algorithme admet toutefois encore une complexité en temps de $\Theta(n)$.

Un dernier algorithme permet d'atteindre une complexité *logarithmique* en n. Il est basé sur l'écriture suivante de (F_n, F_{n-1}), pour $n \geq 2$:

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix}.$$

En itérant, on obtient :

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}.$$

Ainsi, calculer F_n revient à mettre à la puissance $(n-1)$ la matrice $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$.

La complexité en temps de l'algorithme qui en découle est ici $\Theta(\log(n))$ multiplications de matrices carrées 2×2 : en effet, c'est le temps requis par l'algorithme d'exponentiation « square-and-multiply » (*cf.* TD 01) pour calculer la matrice. L'espace nécessaire est celui du stockage de quelques matrices carrées 2×2, soit $\Theta(1)$. L'algorithme `fibo4` peut s'écrire ainsi :

```
1  unsigned int fibo4(unsigned int n) {
2    unsigned int res[2][2];
3    unsigned int mat_tmp[2][2];
4    int i = 0;
5    unsigned int tmp;
6
7    /* on initialise le résultat avec la matrice */
8    res[0][0] = 1;
9    res[0][1] = 1;
10   res[1][0] = 1;
11   res[1][1] = 0;
12
13   /* cas particulier n==0 à traiter tout de suite */
14   if (n == 0) {
15     return 0;
16   }
17
18   /* on doit trouver le un le plus à gauche de n-1 */
19   tmp = n-1;
20   while (tmp != 0) {
21     i++;
22     tmp = tmp >> 1;
23   }
24   /* on décrémente i car la première multiplication
25      a déjà été faite en initialisant res           */
26   i--;
27
28   while (i > 0) {
29     /* on élève au carré */
30     mat_tmp[0][0] = res[0][0]*res[0][0] + res[0][1]*res[1][0];
31     mat_tmp[0][1] = res[0][0]*res[0][1] + res[0][1]*res[1][1];
32     mat_tmp[1][0] = res[1][0]*res[0][0] + res[1][1]*res[1][0];
33     mat_tmp[1][1] = res[1][0]*res[0][1] + res[1][1]*res[1][1];
34     /* on regarde la valeur du ième bit de n-1
35        pour savoir si on doit faire une multiplication */
36     if (((n-1) & (1<<(i-1))) != 0) {
37       res[0][0] = mat_tmp[0][0] + mat_tmp[0][1];
38       res[0][1] = mat_tmp[0][0];
39       res[1][0] = mat_tmp[1][0] + mat_tmp[1][1];
40       res[1][1] = mat_tmp[1][0];
41     } else { /* on replace la matrice dans res */
42       res[0][0] = mat_tmp[0][0];
```

```
43        res[0][1] = mat_tmp[0][1];
44        res[1][0] = mat_tmp[1][0];
45        res[1][1] = mat_tmp[1][1];
46      }
47      i--;
48    }
49    return res[0][0];
50  }
```

Ainsi, une étude algorithmique préalable du problème de départ conduit à une réduction parfois drastique de la complexité de sa résolution, ce qui a pour conséquence de permettre d'atteindre des tailles de paramètres inenvisageables avec un algorithme irréfléchi... La TABLE 1.1 ci-dessous illustre les temps de calcul des quatre algorithmes précédents.

	n	40	2^{25}	2^{28}	2^{31}
`fibo1`	$\Theta(\phi^n)$	31s	calcul irréalisable		
`fibo2`	$\Theta(n)$	0s	18s	`Segmentation fault`	
`fibo3`	$\Theta(n)$	0s	4s	25s	195s
`fibo4`	$\Theta(\log(n))$	0s	0s	0s	0s

TABLE 1.1 – *Complexités et temps d'exécution de différents algorithmes pour le calcul de la suite de Fibonacci.*

Comme on le voit dans la TABLE 1.1, l'algorithme `fibo4` qui a la meilleure complexité est beaucoup plus rapide que les autres, et peut traiter des valeurs de n plus grandes. L'algorithme `fibo1` prend très vite un temps trop élevé pour pouvoir être utilisé. Une autre limitation de `fibo1` que l'on ne voit pas dans le tableau est la limite liée au nombre maximal de niveaux de récursivité autorisés dans un programme : en C cette valeur est souvent autour de quelques dizaines de milliers [2]. On ne peut donc pas avoir plus que ce nombre d'appels récursifs imbriqués au sein d'un algorithme. Pour l'algorithme `fibo2` la limite ne vient pas du temps de calcul, mais de la taille mémoire nécessaire : quand on essaye d'allouer un tableau de 2^{28} entiers (soit 1Go de mémoire d'un seul bloc) le système d'exploitation n'arrive pas à satisfaire notre requête et n'alloue donc pas de mémoire, et puisque l'on ne teste pas si la mémoire a bien été allouée avant d'écrire dans notre tableau, un problème à l'allocation se traduit immédiatement par une erreur de segmentation mémoire à l'écriture.

1.3 Un premier algorithme pour le tri

Nous abordons un premier exemple d'algorithme permettant de trier des éléments munis d'une relation d'ordre (des entiers par exemple) : il s'agit du tri par insertion, qui modélise notre façon de trier des cartes à jouer. Voici le code d'un tel algorithme triant un tableau d'entier `tab` de taille `n` :

2. Le nombre maximal de niveaux de récursivité est de l'ordre de 2^{18} avec la version 4.1.2 de gcc sur une gentoo 2007.0, avec un processeur Intel Pentium D et les options par défaut.

```
1  void insertion_sort(int* tab, int n) {
2    int i,j,tmp;
3    for (i=1; i<n; i++) {
4      tmp = tab[i];
5      j = i-1;
6      while ((j > 0) && (tab[j] > tmp)) {
7        tab[j+1] = tab[j];
8        j--;
9      }
10     tab[j+1] = tmp;
11   }
12 }
```

Le principe de cet algorithme est assez simple : à chaque étape de la boucle `for`, les i premiers éléments de `tab` sont triés par ordre croissant. Quand $i = 1$ c'est vrai, et à chaque fois que l'on augmente i de 1, la boucle `while` se charge de replacer le nouvel élément à sa place en le remontant élément par élément vers les petits indice du tableau : si le nouvel élément est plus petit que celui juste avant, on inverse les deux éléments que l'on vient de comparer et on recommence jusqu'à avoir atteint la bonne position.

Cet algorithme a un coût très variable en fonction de l'état initial du tableau :

- pour un tableau déjà trié, l'algorithme va simplement parcourir tous les éléments et les comparer à l'élément juste avant eux, mais ne fera jamais d'échange. Le coût est alors de n comparaisons.
- pour un tableau trié à l'envers (le cas le pire), l'algorithme va remonter chaque nouvel élément tout au début du tableau. Il y a alors au total exactement $\frac{n(n-1)}{2}$ comparaisons et échanges. L'algorithme de tri par insertion à donc une complexité $\Theta(n^2)$ dans le pire des cas.
- en moyenne, il faudra remonter chaque nouvel élément sur la moitié de la longueur, donc $\frac{i-1}{2}$ comparaisons et échange par nouvel élément. Au final la complexité en moyenne du tri par insertion est $\Theta(n^2)$, la même que le cas le pire (on perd le facteur $\frac{1}{2}$ dans le Θ).

Généralisations du tri par insertion. Il existe plusieurs variantes du tri par insertion visant à améliorer sa complexité en moyenne. Citons en particulier le tri de Shell (*cf.* http://fr.wikipedia.org/wiki/Tri_de_Shell) qui compare non pas des éléments voisins, mais des éléments plus distants afin d'optimiser le nombre de comparaisons nécessaires.

Chapitre 2

Récursivité

Les définitions récursives sont courantes en mathématiques. Nous avons vu au chapitre précédent l'exemple de la suite de Fibonacci, définie par une relation de récurrence. En informatique, la notion de récursivité joue un rôle fondamental. Nous voyons dans ce chapitre la puissance de la récursivité au travers essentiellement de deux algorithmes de tri ayant les meilleures performances asymptotiques pour des algorithmes génériques. Nous terminons par une étude des solutions d'équations de récurrence entrant en jeu lors de l'analyse de complexité de tels algorithmes.

2.1 Conception des algorithmes

Il existe de nombreuses façons de concevoir un algorithme. On peut par exemple adopter une approche *incrémentale* ; c'est le cas du tri par insertion : après avoir trié le sous-tableau `tab[0]...tab[j-1]`, on insère l'élément `tab[j]` au bon emplacement pour produire le sous-tableau trié `tab[0]`...`tab[j]`.

Une approche souvent très efficace et élégante est l'approche *récursive* : un algorithme récursif est un algorithme défini en référence à lui-même (c'est la cas par exemple de l'algorithme `fibo1` vu au chapitre précédent). Pour éviter de boucler indéfiniment lors de sa mise en oeuvre, il est nécessaire de rajouter à cette définition une *condition de terminaison*, qui autorise l'algorithme à ne plus être défini à partir de lui-même pour certaines valeurs de l'entrée (pour `fibo1`, nous avons par exemple défini `fibo1(0)=0` et `fibo1(1)=1`).

Un tel algorithme suit généralement le paradigme « diviser pour régner » : il sépare le problème en plusieurs sous-problèmes semblables au problème initial mais de taille moindre, résout les sous-problèmes de façon récursive, puis combine toutes les solutions pour produire la solution du problème initial. La méthode diviser pour régner implique trois étapes à chaque niveau de la récursivité :

- **Diviser** le problème en un certain nombre de sous-problèmes. On notera $D(n)$ la complexité de cette étape.
- **Régner** sur les sous-problèmes en les résolvant de manière récursive (ou directe si le sous-problème est suffisamment réduit, *i.e.* une condition de terminaison est atteinte).

On notera $R(n)$ la complexité de cette étape.
– **Combiner** les solutions des sous-problèmes pour trouver la solution du problème initial. On notera $C(n)$ la complexité de cette étape.

Lien avec la récursivité en mathématiques. On rencontre en général la notion de récursivité en mathématiques essentiellement dans deux domaines : les preuves par récurrence et les suites récurrentes. La conception d'algorithmes récursifs se rapproche plus des preuves par récurrence, mais comme nous allons le voir, le calcul de la complexité d'un algorithme récursif se rapproche beaucoup des suites récurrentes.

Lors d'une preuve par récurrence, on prouve qu'une propriété est juste pour des conditions initiales, et on étend cette propriété à l'ensemble du domaine en prouvant que la propriété est juste au rang n si elle est vrai au rang $n-1$. Dans un algorithme récursif la condition de terminaison correspond exactement au conditions initiales de la preuve par récurrence, et l'algorithme va ramener le calcul sur une entrée à des calculs sur des entrées plus petites.

⚠ Attention !

Lors de la conception d'un algorithme récursif il faut bien faire attention à ce que tous les appels récursifs effectués terminent bien sur une condition de terminaison. Dans le cas contraire l'algorithme peut partir dans une pile d'appels récursifs infinie (limitée uniquement par le nombre maximum d'appels récursifs autorisés). De même, en mathématique, lors d'une preuve par récurrence, si la condition récurrente ne se ramène pas toujours à une condition initiale, la preuve peut être fausse.

2.2 Problème des tours de Hanoï

Le problème des tours de Hanoï peut se décrire sous la forme d'un jeu (*cf.* Figure 2.1) : on dispose de trois piquets numérotés 1,2,3, et de n rondelles, toutes de diamètre différent. Initialement, toutes les rondelles se trouvent sur le piquet 1, dans l'ordre décroissant des diamètres (elle forment donc une pyramide). Le but du jeu est de déplacer toutes les rondelles sur un piquet de destination choisi parmi les deux piquets vides, en respectant les règles suivantes :

– on ne peut déplacer qu'une seule rondelle à la fois d'un sommet de pile vers un autre piquet ;
– on ne peut pas placer une rondelle au-dessus d'une rondelle de plus petit diamètre.

Ce problème admet une résolution récursive élégante qui comporte trois étapes :

1. déplacement des $n-1$ rondelles supérieures du piquet origine vers le piquet intermédiaire par un appel récursif à l'algorithme.
2. déplacement de la plus grande rondelle du piquet origine vers le piquet destination.
3. déplacement des $n-1$ rondelles du piquet intermédiaire vers le piquet destination par un appel récursif à l'algorithme.

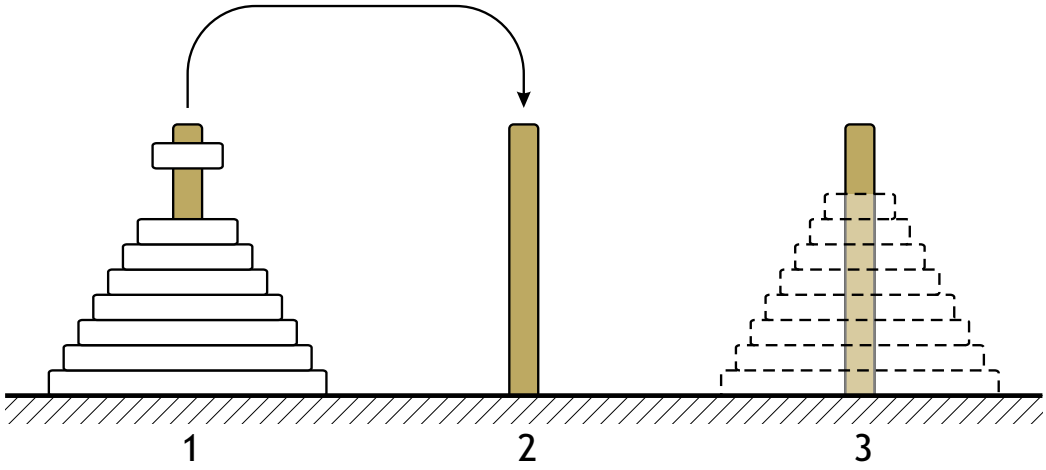


FIGURE 2.1 – *Le jeu des tours de Hanoï. Déplacement d'une rondelle du piquet 1 vers le piquet 2. En pointillés : l'objectif final.*

Cet algorithme, que l'on appelle Hanoï, suit l'approche diviser pour régner : la phase de division consiste toujours à diviser le problème de taille n en deux sous-problèmes de taille $n-1$ et 1 respectivement. La phase de règne consiste à appeler récursivement l'algorithme Hanoï sur le sous-problème de taille $n-1$. Ici, il y a deux « séries » d'appels récursifs par sous-problème de taille $n-1$ (étapes 1. et 3.). La phase de combinaison des solutions des sous-problèmes est inexistante ici. On donne ci-après le programme C implémentant l'algorithme Hanoï :

```
1  void Hanoi(int n, int i, int j) {
2    int intermediate = 6-(i+j);
3    if (n > 0) {
4      Hanoi(n-1,i,intermediate);
5      printf("Mouvement du piquet %d vers le piquet %d.\n",i,j);
6      Hanoi(n-1,intermediate,j);
7    }
8  }
9  int main(int argc, char* argv[]) {
10   Hanoi(atoi(argv[1]),1,3);
11 }
```

On constate qu'ici la condition de terminaison est $n = 0$ (l'algorithme ne fait des appels récursifs que si $n > 0$) et pour cette valeur l'algorithme ne fait rien. Le lecteur est invité à vérifier que l'exécution de `Hanoi(3,1,3)` donne :

```
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
```

Complexité de l'algorithme. Calculer la complexité d'un algorithme récursif peut parfois sembler compliqué, mais il suffit en général de se ramener à une relation définissant une suite récurrente. C'est ce que l'on fait ici. Soit $T(n)$ la complexité (ici, le nombre de déplacements de disques) nécessaire à la résolution du problème sur une entrée de taille n par l'algorithme Hanoï. En décomposant la complexité de chaque étape on trouve : l'étape de division ne coûte rien, l'étape de règne coûte le prix de deux mouvements de taille $n-1$ et l'étape de combinaison coûte le mouvement du grand disque, soit 1. On a $T(0) = 0$, $T(1) = 1$, et, pour $n \geq 2$,

$$T(n) = D(n) + R(n) + C(n) = 0 + 2 \times T(n-1) + 1.$$

En développant cette expression, on trouve

$$T(n) = 2^n T(0) + 2^{n-1} + \ldots + 2 + 1,$$

soit

$$T(n) = 2^{n-1} + \ldots + 2 + 1 = 2^n - 1.$$

Ainsi, la complexité de cet algorithme est exponentielle en la taille de l'entrée. En fait, ce caractère exponentiel *ne dépend pas* de l'algorithme en lui-même, mais est *intrinsèque* au problème des tours de Hanoï : montrons en effet que le nombre minimal min_n de mouvements de disques à effectuer pour résoudre le problème sur une entrée de taille n est exponentiel en n. Pour cela, nous observons que le plus grand des disques doit nécessairement être déplacé au moins une fois. Au moment du premier mouvement de ce grand disque, il doit être seul sur un piquet, son piquet de destination doit être vide, donc tous les autres disques doivent être rangés, dans l'ordre, sur le piquet restant. Donc, avant le premier mouvement du grand disque, on aura dû déplacer une pile de taille $n-1$. De même, après le dernier mouvement du grand disque, on devra déplacer les $n-1$ autres disques ; ainsi, $min_n \geq 2min_{n-1}+1$. Or, dans l'algorithme Hanoï, le nombre de mouvements de disques vérifie exactement cette égalité (on a $T(n) = min_n$). Ainsi, cet algorithme est optimal, et la complexité exponentielle est intrinsèque au problème.

⋆ Complexité Spatiale des Appels Récursifs

Notons que l'implémentation que nous donnons de l'algorithme `Hanoï` *n'est pas standard : l'algorithme est en général programmé à l'aide de trois piles (cf. section 3.3.1), mais comme nous n'avons pas encore étudié cette structure de donnée, notre algorithme se contente d'afficher les opérations qu'il effectuerait normalement. En utilisant des piles, la complexité spatiale serait $\Theta(n)$ (correspondant à l'espace nécessaire pour stocker les n disques). Ici il n'y a pas d'allocation mémoire, mais la complexité spatiale est quand même $\Theta(n)$: en effet, chaque appel récursif nécessite d'allouer de la mémoire (ne serait-ce que pour conserver l'adresse de retour de la fonction) qui n'est libérée que lorsque la fonction appelée récursivement se termine. Ici le programme utilise jusqu'à n appels récursifs imbriqués donc sa complexité spatiale est $\Theta(n)$. Il n'est pas courant de prendre en compte la complexité spatiale d'appels récursifs imbriqués, car en général ce nombre d'appels est toujours relativement faible.*

2.3 Algorithmes de tri

Nous continuons ici l'étude de méthodes permettant de trier des données indexées par des clefs (ces clefs sont munies d'une relation d'ordre et permettent de conduire le tri). Il s'agit de réorganiser les données de telle sorte que les clefs apparaissent dans un ordre bien déterminé (alphabétique ou numérique le plus souvent).

Les algorithmes simples (comme le tri par insertion, mais aussi le tri par sélection ou le tri à bulle) sont à utiliser pour des petites quantités de données (de l'ordre de moins de 100 clefs), ou des données présentant une structure particulière (données complètement ou presque triées, ou comportant beaucoup de clefs identiques).

Pour de grandes quantités de données aléatoires, ou si l'algorithme doit être utilisé un grand nombre de fois, on a plutôt recours à des méthodes plus sophistiquées. Nous en présentons deux ici : le tri fusion et le tri rapide. Tous deux sont de nature récursive et suivent l'approche diviser pour régner.

2.3.1 Le tri fusion

Cet algorithme repose sur le fait que fusionner deux tableaux triés est plus rapide que de trier un grand tableau directement. Supposons que l'algorithme prenne en entrée un tableau de n éléments dans un ordre quelconque. L'algorithme commence par diviser le tableau des n éléments en deux sous-tableaux de $\frac{n}{2}$ éléments chacun (étape diviser). Les deux sous-tableaux sont triés de manière récursive[1] en utilisant toujours le tri fusion (régner) ; ils sont ensuite fusionnés pour produire le tableau trié (combiner).

L'algorithme de tri fusion (*merge sort* en anglais) du tableau d'entiers `tab` entre les indices p et r est le suivant :

```
1 void merge_sort(int* tab, int p, int r) {
2   int q;
3   if (r-p > 1) {
4     q = (p+r)/2;
5     merge_sort(tab,p,q);
6     merge_sort(tab,q,r);
7     merge(tab,p,q,r);
8   }
9 }
```

La procédure fusion (la fonction `merge` décrite ci-dessous) commence par recopier les deux sous-tableaux triés `tab[p]`...`tab[q-1]` et `tab[q]`...`tab[r-1]` « dos-à-dos[2] » dans un tableau auxiliaire `tmp`. L'intérêt de cette façon de recopier est que l'on n'a alors pas besoin de rajouter de tests de fin de sous-tableaux, ni de case supplémentaire contenant le symbole ∞ par exemple à chacun des sous-tableaux. Ensuite, `merge` remet dans `tab` les éléments du tableau `tmp` trié, mais ici le tri a une complexité linéaire ($\Theta(n)$) puisque `tmp` provient

1. La récursion s'arrête lorsque les sous-tableaux sont de taille 1, donc trivialement triés.
2. Ainsi, `tmp` est le tableau [`tab[p]`,...,`tab[q-1]`,`tab[r-1]`,...,`tab[q]`].

de deux sous-tableaux déjà triés ; le procédé est alors le suivant : on part de $k = p$, $i = p$ et $j = r$, et on compare `tmp[i]` avec `tmp[j]`. On met le plus petit des deux dans `tab[k]` ; si c'est `tmp[i]`, on incrémente i, sinon, on décrémente j ; dans tous les cas, on incrémente k. On continue jusqu'à ce que k vaille r.

```
1  void merge(int* tab, int p, int q, int r) {
2    int* tmp = (int*) malloc((r-p)*sizeof(int));
3    int i,j,k;
4    for (i=p; i<q; i++) {
5      tmp[i-p] = tab[i];
6    }
7    for (i=q; i<r; i++) {
8      tmp[r-p-1-(i-q)] = tab[i];
9    }
10   i=p;
11   j=r-1;
12   for (k=p; k<r; k++) {
13     if (tmp[i] < tmp[j]) {
14       tab[k] = tmp[i];
15       i++;
16     } else {
17       tab[k] = tmp[j];
18       j--;
19     }
20   }
21   free(tmp);
22 }
```

Ainsi, au début de chaque itération de la boucle pour k, les $k - p$ éléments du sous-tableau `tab[p]`...`tab[k-1]` sont triés. Pour trier le tableau `tab` de taille n, on appelle `merge_sort(tab,0,n)`.

Note sur la Réallocation de Mémoire

L'algorithme `merge_sort` *tel qu'il est écrit ne gère pas bien sa mémoire. En effet, chaque appel à la fonction* `merge` *commence par allouer un tableau* `tmp` *de taille* $r - p$*, et les opérations d'allocation mémoire sont des opération relativement coûteuses (longues à exécuter en pratique). Allouer une mémoire de taille* n *a une complexité* $\Theta(n)$*, et réallouer de la mémoire à chaque appel de* `merge` *ne change donc pas la complexité de l'algorithme. En revanche, en pratique cela ralentit beaucoup l'exécution du programme. Il serait donc plus efficace d'allouer une fois pour toutes un tableau de taille* n *au premier appel de la fonction de tri et de passer ce tableau en argument à toutes les fonctions afin qu'elles puissent l'utiliser. Cela demande cependant d'ajouter une fonction supplémentaire (celle que l'utilisateur va appeler en pratique), qui alloue de la mémoire avant d'appeler la fonction récursive* `merge_sort`*.*

Complexité du tri fusion. Évaluons à présent la complexité en temps de l'algorithme `merge_sort`, en termes de nombre de comparaisons de clefs. On note $T(n)$, cette complexité pour une entrée de taille n.

La phase de division ne nécessite aucune comparaison. La phase de règne requiert deux fois le temps de l'algorithme `merge_sort` sur un tableau de taille deux fois moindre, *i.e.* $2 \times T(\frac{n}{2})$. La phase de recombinaison est la procédure `merge`. Lorsqu'appelée avec les paramètres (p, q, r), elle nécessite $r - p$ comparaisons. On a donc :

$$T(n) = D(n) + R(n) + C(n) = 0 + 2 \times T(\frac{n}{2}) + n.$$

Supposons d'abord que n est une puissance de 2, soit $n = 2^k$. Pour trouver la solution de cette récurrence, on construit un *arbre de récursivité* : c'est un arbre binaire dont chaque nœud représente le coût d'un sous-problème individuel, invoqué à un moment de la récursion. Le noeud racine contient le coût de la phase de division+recombinaison au niveau n de la récursivité (ici n), et ses deux sous-arbres représentent les coûts des sous-problèmes au niveau $\frac{n}{2}$. En développant ces coûts pour $\frac{n}{2}$, on obtient deux sous-arbres, et ainsi de suite jusqu'à arriver au coût pour les sous-problèmes de taille 1. Partant de $n = 2^k$, le nombre de niveaux de cet arbre est exactement $\log(n) + 1 = k + 1$ (la hauteur de l'arbre est k). Pour trouver la complexité $T(n)$, il suffit à présent d'additionner les coûts de chaque noeud. On procède en calculant le coût total par niveau : le niveau de profondeur 0 (racine) a un coût total égal à n, le niveau de profondeur 1 a un coût total égal à $\frac{n}{2} + \frac{n}{2}$,... le niveau de profondeur i pour $0 \leq i \leq k - 1$ a un coût total de $2^i \times \frac{n}{2^i} = n$ (ce niveau comporte 2^i nœuds). Le dernier niveau correspond aux 2^k sous-problèmes de taille 1 ; aucun ne contribue à la complexité en termes de nombre de comparaisons, donc le coût au niveau k est nul. Ainsi, chaque niveau, sauf le dernier, contribue pour n à la complexité totale. Il en résulte que $T(n) = k \times n = n \log(n)$.

Lorsque n n'est pas nécessairement une puissance de 2, on encadre n entre deux puissances de 2 consécutives : $2^k \leq n < 2^{k+1}$. La fonction $T(n)$ étant croissante, on a $T(2^k) \leq T(n) \leq T(2^{k+1})$, soit $k2^k \leq T(n) \leq (k+1)2^{k+1}$. Comme $\lfloor \log(n) \rfloor = k$, on obtient une complexité en $\Theta(n \log(n))$.

Remarques :

- Le tri fusion ne se fait pas « en place[3] » : en effet, la procédure `merge_sort` nécessite un espace mémoire supplémentaire sous la forme d'un tableau (`tmp`) de taille n.
- Le calcul de complexité précédent est indépendant de la distribution des entiers à trier : le tri fusion s'exécute en $\Theta(n \log(n))$ pour toutes les distributions d'entiers. La complexité en moyenne est donc égale à la complexité dans le pire cas.

2.3.2 Le tri rapide

Le deuxième algorithme de tri que nous étudions suit également la méthode diviser pour régner. Par rapport au tri fusion, il présente l'avantage de trier « en place ». En

3. Un tri se fait « en place » lorsque la quantité de mémoire supplémentaire - la cas échéant - est une petit constante (indépendante de la taille de l'entrée).

revanche, son comportement dépend de la distribution de son entrée : dans le pire cas, il possède une complexité quadratique, cependant, ses performances en moyenne sont en $\Theta(n \log(n))$, et il constitue souvent le meilleur choix en pratique [4]. Le fonctionnement de l'algorithme sur un tableau `tab` à trier entre les indices p et r est le suivant :

```
1 void quick_sort(int* tab, int p, int r) {
2   int q;
3   if (r-p > 1) {
4     q = partition(tab,p,r);
5     quick_sort(tab,p,q);
6     quick_sort(tab,q+1,r);
7   }
8 }
```

La procédure `partition` détermine un indice q tel que, à l'issue de la procédure, pour $p \leq i \leq q-1$, `tab[i]` $\leq$ `tab[q]`, et pour $q+1 \leq i \leq r-1$, `tab[i]` $>$ `tab[q]` (les sous-tableaux `tab[i]`...`tab[q-1]` et `tab[q+1]`...`tab[r-1]` n'étant pas eux-mêmes triés). Elle utilise un élément `x = tab[p]` appelé *pivot* autour duquel se fera le partitionnement.

```
1 int partition(int* tab, int p, int r) {
2   int x = tab[p];
3   int q = p;
4   int i,tmp;
5   for (i=p+1; i<r; i++) {
6     if (tab[i] <= x) {
7       q++;
8       tmp = tab[q];
9       tab[q] = tab[i];
10      tab[i] = tmp;
11    }
12  }
13  tmp = tab[q];
14  tab[q] = tab[p];
15  tab[p] = tmp;
16  return q;
17 }
```

À la fin de chacune des itérations de la boucle `for`, le tableau est divisé en trois parties :

$$\forall i,\ p \leq i \leq q,\ \texttt{tab[i]} \leq x$$

$$\forall i,\ q+1 \leq i \leq j-1,\ \texttt{tab[i]} > x$$

4. L'ordre de grandeur de complexité en termes de nombre de comparaisons, mais aussi de nombre d'opérations élémentaires (affectations, incrémentations) est le même que pour le tri fusion, mais les constantes cachées dans la notation Θ sont plus petites.

Pour $j \leq i \leq r$, les clefs `tab[i]` n'ont pas de lien fixé avec le pivot x (éléments non encore traités).

Si le test en ligne 6 est vrai, alors l'élément en position j est inférieur ou égal à x, donc on le décale le plus à gauche possible (lignes 8 à 10) ; mais on incrémente d'abord q (ligne 7) de façon à pouvoir insérer cet élément entre `tab[p]` et `tab[q]` strictement. À l'issue de la boucle `for` sur j, tous les éléments de `tab[p]`...`tab[r-1]` inférieurs ou égaux à x ont été placés à gauche de `tab[q]` (et à droite de `tab[p]`) ; il ne reste donc plus qu'à mettre x à la place de `tab[q]` (lignes 13 à 15).

Complexité du tri rapide. La complexité de l'algorithme `quick_sort` dépend du caractère équilibré ou non du partitionnement, qui lui-même dépend de la valeur du pivot choisi au départ. En effet, si $x =$ `tab[p]` est inférieur à tous les autres éléments du tableau, la procédure `partition` découpera le tableau initial en deux sous-tableaux extrêmement déséquilibrés : l'un de taille 0 et l'autre de taille $n-1$. En particulier si le tableau est déjà trié (ou inversement trié), cette configuration surviendra à chaque appel récursif. Dans ce cas, le temps d'exécution $T(n)$ de l'algorithme satisfait la récurrence

$$T(n) = T(n-1) + D(n),$$

(la procédure `quick_sort` sur une entrée de taille 0 ne nécessitant pas de comparaison) où $D(n)$ est le coût de la procédure `partition` sur une entrée de taille n. Il est clair que `partition`$(\texttt{tab}, p, r)$ nécessite $r-p-1$ comparaisons, donc $D(n) = n-1$. L'arbre récursif correspondant à $T(n) = T(n-1) + n - 1$ est de profondeur $n-1$, le niveau de profondeur i ayant un coût de $n-(i+1)$. En cumulant les coûts par niveau, on obtient

$$T(n) = \sum_{i=0}^{n-1} n - (i+1) = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}.$$

Ainsi, la complexité de l'algorithme *dans le pire cas* est en $\Theta(n^2)$, *i.e.* pas meilleure que celle du tri par insertion.

Le cas le plus favorable est celui où la procédure `partition` découpe toujours le tableau courant en deux sous-tableaux de taille presque égale ($\lfloor \frac{n}{2} \rfloor$ et $\lceil \frac{n}{2} \rceil - 1$, donc toujours inférieure ou égale à $\frac{n}{2}$). Dans ce cas, la complexité de l'algorithme est donné par

$$T(n) = 2 \times T(\tfrac{n}{2}) + n - 1.$$

Cette récurrence est similaire à celle rencontrée lors de l'étude du tri fusion. La solution est $T(n) = (n-1)\log(n)$ si n est une puissance de 2, soit $T(n) = \Theta(n \log(n))$ pour tout n.

Nous allons maintenant voir que cette complexité est aussi celle du cas moyen. Pour cela, il est nécessaire de faire une hypothèse sur la distribution des entiers en entrée de l'algorithme. On suppose donc qu'ils suivent une distribution aléatoire uniforme. C'est le résultat de `partition` qui, à chaque appel récursif, conditionne le découpage en sous-tableaux et donc la complexité. On ne peut pas supposer que `partition` fournit un indice q uniformément distribué dans $[p, ..., r-1]$: en effet, on choisit toujours comme pivot le

premier élément du sous-tableau courant, donc on ne peut faire d'hypothèse d'uniformité quant à la distribution de cet élément *tout au long de l'algorithme*[5]. Pour pouvoir faire une hypothèse raisonnable d'uniformité sur q, il est nécessaire de modifier un tant soit peut `partition` de façon à choisir non plus le premier élément comme pivot, mais un élément aléatoirement choisi dans le sous-tableau considéré. La procédure devient alors :

```
1 int random_partition(int* tab, int p, int r) {
2   int i,tmp;
3   i = (double) rand()/RAND_MAX * (r-p) + p;
4   tmp = tab[p];
5   tab[p] = tab[i];
6   tab[i] = tmp;
7   partition(tab,p,r);
8 }
```

L'échange a pour effet de placer le pivot `tab[i]` en position p, ce qui permet d'exécuter `partition` normalement).

En supposant à présent que l'entier q retourné par `random_partition` est uniformément distribué dans $[p, r-1]$, on obtient la formule suivante pour $T(n)$:

$$T(n) = n - 1 + \frac{1}{n}\sum_{q=1}^{n}(T(q-1) + T(n-q)).$$

En effet, $\sum_{q=1}^{n}(T(q-1)+T(n-q))$ représente la somme des complexités correspondant aux n découpages possibles du tableau `tab`$[0, ..., n-1]$ en deux sous-tableaux. La complexité moyenne $T(n)$ est donc la moyenne de ces complexités. Par symétrie, on a :

$$T(n) = n - 1 + \frac{2}{n}\sum_{q=1}^{n}T(q-1),$$

ou

$$nT(n) = n(n-1) + 2\sum_{q=1}^{n}T(q-1).$$

En soustrayant à cette égalité la même égalité au rang $n-1$, on obtient une expression faisant seulement intervenir $T(n)$ et $T(n-1)$:

$$nT(n) = (n+1)T(n-1) + 2(n-1),$$

ou

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2(n-1)}{n(n+1)}.$$

5. Et ce, même si les entiers en entrée sont uniformément distribués : en effet, la configuration de ces entiers dans le tableau est modifiée d'un appel de `partition` à l'autre.

En développant la même formule pour $T(n-1)$ et en la réinjectant ci-dessus, on obtient :

$$\frac{T(n)}{n+1} = \frac{T(n-2)}{n-1} + \frac{2(n-2)}{(n-1)n} + \frac{2(n-1)}{n(n+1)}.$$

En itérant ce processus :

$$\frac{T(n)}{n+1} = \sum_{k=2}^{n} \frac{2(k-1)}{k(k+1)}, \text{ ou } T(n) = (n+1)\sum_{k=2}^{n} \frac{2(k-1)}{k(k+1)}.$$

Pour n suffisamment grand, on a

$$\sum_{k=2}^{n} \frac{(k-1)}{k(k+1)} \approx \sum_{k=1}^{n} \frac{1}{k}, \text{ et } \sum_{k=1}^{n} \frac{1}{k} \approx \int_1^n \frac{dx}{x} = \ln(n).$$

Ainsi, $T(n) \approx 2(n+1)\ln(n)$ et on retrouve bien une complexité en $\Theta(n\log(n))$ pour *le cas moyen*.

2.3.3 Complexité minimale d'un algorithme de tri

Nous pouvons nous demander si les complexités en $\Theta(n\log(n))$ obtenues ci-dessus sont les meilleures que l'on puisse espérer pour un algorithme de tri générique (*i.e.* qui ne fait aucune hypothèse sur les données à trier). Pour répondre à cette question, nous allons calculer une borne inférieure sur la complexité (en termes de nombre de comparaisons) de tout algorithme de tri par comparaison. Précisons tout d'abord que l'on appelle *tri par comparaison* un algorithme de tri qui, pour obtenir des informations sur l'ordre de la séquence d'entrée, utilise seulement des comparaisons. Les algorithmes de tri fusion et rapide vus précédemment sont des tris par comparaison. La borne calculée est valable dans le pire cas (contexte général de la théorie de la complexité).

Tout algorithme de tri par comparaison peut être modélisé par un *arbre de décision* (un arbre binaire comme défini dans le chapitre 5). Chaque comparaison que l'algorithme effectue représente un nœud de l'arbre, et en fonction du résultat de la comparaison, l'algorithme peut s'engager soit dans le sous-arbre gauche, soit dans le sous-arbre droit. L'algorithme fait une première comparaison qui est la racine de l'arbre, puis s'engage dans l'un des deux sous-arbres fils et fait une deuxième comparaison, et ainsi de suite... Quand l'algorithme s'arrête c'est qu'il a fini de trier l'entrée : il ne fera plus d'autres comparaisons et une feuille de l'arbre est atteinte. Chaque ordre des éléments d'entrée mène à une feuille différente, et le nombre de comparaisons à effectuer pour atteindre cette feuille est la profondeur de la feuille dans l'arbre. L'arbre explorant toutes les comparaisons, il en résulte que le nombre ℓ de ses feuilles pour des entrées de taille n est au moins égal à $n!$, cardinal de l'ensemble de toutes les permutations des n positions.

Soit h, la hauteur de l'arbre. Dans le pire cas, il est nécessaire de faire h comparaisons pour trier les n éléments (autrement dit, la feuille correspondant à la permutation correcte

se trouve sur un chemin de longueur[6] h). Le nombre de feuilles d'un arbre binaire de hauteur h étant au plus 2^h, on a

$$2^h \geq \ell \geq n!, \text{ soit } \ h \geq \log(n!).$$

La formule de Stirling

$$n! = \sqrt{2\pi n}(\frac{n}{e})^n(1 + \Theta(\frac{1}{n})),$$

donne

$$\log(n!) > \log((\frac{n}{e})^n) = n\log(n) - n\log(e),$$

et par suite, $h = \Omega(n \log(n))$. Nous énonçons ce résultat en :

Théorème 2.3.1. *Tout algorithme de tri par comparaison nécessite* $\Omega(n \log(n))$ *comparaisons dans le pire cas.*

Les tris fusion et rapide sont donc des tris par comparaison asymptotiquement optimaux.

2.4 Résolution d'équations de récurrence

L'analyse des performances d'un algorithme donne en général des équations où le temps de calcul, pour une taille des données, est exprimé en fonction du temps de calcul pour des données de taille moindre. Il n'est pas toujours possible de résoudre ces équations. Dans cette section, nous donnons quelques techniques permettant de trouver des solutions exactes ou approchées de récurrences intervenant classiquement dans les calculs de coût.

2.4.1 Récurrences linéaires

Commençons par rappeler la méthode de résolution des récurrences linéaires homogènes à coefficients constants. Il s'agit d'équations du type :

$$u_{n+h} = a_{h-1}u_{n+h-1} + \ldots + a_0 u_n, \ a_0 \neq 0, \ h \in \mathbb{N}^*, \ n \in \mathbb{N}.$$

Cette suite est entièrement déterminée par ses h premières valeurs $u_0, \ldots, u_{h-1}$ (la suite est *d'ordre* h). Son polynôme caractéristique est :

$$G(x) = x^h - a_{h-1}x^{h-1} - \ldots - a_1 x - a_0.$$

Soit $\omega_1, \ldots, \omega_r$, ses racines, et soit n_i, la multiplicité de ω_i, $1 \leq i \leq r$. En considérant la série génératrice formelle associée à u_n, soit $U(X) = \sum_{n=0}^{\infty} u_n X^n$, et en multipliant cette

6. Toutes les feuilles ne se trouvant pas sur le dernier niveau.

série par le polynôme $B(X) = X^h G(1/X)$ (polynôme réciproque de $G(X)$), on obtient un polynôme $A(X)$ de degré au plus $h-1$. L'expression :

$$U(X) = \frac{A(X)}{B(X)}$$

montre alors que $U(X)$ est en fait une série rationnelle, *i.e.* une fraction rationnelle. Sa décomposition en éléments simples donne l'expression suivante [7] pour le terme général de la suite :

$$u_n = \sum_{i=1}^{r} P_i(n)\omega_i^n,$$

où $P_i(n)$ est un polynôme de degré au plus n_i. Une expression de cette forme pour u_n est appelée *polynôme exponentiel*.

La suite de Fibonacci $F_n = F_{n-1} + F_{n-2}$, avec $u_0 = 0$ et $u_1 = 1$, est un exemple de telle récurrence, traité dans le chapitre 1. On a $F_n = \frac{1}{\sqrt{5}}(\phi^n - \bar{\phi}^n)$, où $\phi = \frac{1+\sqrt{5}}{2}$. La suite de Fibonacci intervient dans le calcul de coût de nombreux algorithmes. Considérons par exemple l'algorithme d'Euclide de calcul du pgcd de deux entiers x et y non tous les deux nuls (on suppose $x \geq y$) :

```
1 int euclide(int x, int y) {
2   if (y == 0) {
3     return x;
4   } else {
5     /* en C, x modulo y s'écrit x % y */
6     return euclide(y,x % y);
7   }
8 }
```

Notons $n(x, y)$, le nombre de divisions avec reste effectuées par l'algorithme. Alors $n(x, y) = 0$ si $y = 0$, et $n(x, y) = 1 + n(y, x \mod y)$ sinon. Pour évaluer le coût de l'algorithme (en fonction de x), nous allons d'abord prouver que, pour $x > y \geq 0$,

$$n(x, y) = k \Rightarrow x \geq F_{k+2}.$$

Pour $k = 0$, on a $x \geq 1 = F_2$, et, pour $k = 1$, on a $x \geq 2 = F_3$. Supposons à présent $k \geq 2$, et la propriété ci-dessus vraie pour tout $j \leq k-1$, *i.e.* si $n(u, v) = j$, alors $u \geq F_{j+2}$, pour $u, v \in \mathbb{N}$, $u \geq v > 0$. Supposons $n(x, y) = k$, et considérons les divisions euclidiennes :

$$x = qy + z,\ 0 \leq z < y,\ \ y = q'z + u,\ 0 \leq u < z.$$

On a $n(y, z) = k-1$ donc $y \geq F_{k+1}$; de même, $z \geq F_k$ et par suite $x \geq F_{k+1} + F_k = F_{k+2}$. D'autre part, comme $\bar{\phi}^n \leq 1,\ \forall n \in \mathbb{N}$, on a $F_n \geq \frac{1}{\sqrt{5}}(\phi^n - 1)$. D'où $\sqrt{5}x + 1 \geq \phi^{k+2}$. Donc, pour $x > y \geq 0$,

$$n(x, y) \leq \log_\phi(\sqrt{5}x + 1) - 2,$$

7. Le détail des calculs peut être trouvé dans *Éléments d'algorithmique*, de Beauquier, Berstel, Chrétienne, éd. Masson.

autrement dit, $n(x, y) = O(\log(x))$.

Il existe de nombreux autres types d'équations de récurrence. Parmi eux, les récurrences linéaires *avec second membre* constituent une classe importante de telles récurrences ; elles peuvent être résolues par une technique similaire à celle présentée pour les équations homogènes. Des méthodes adaptées existent pour la plupart des types de récurrences rencontrées ; une présentation de ces différentes techniques peut être trouvée dans le livre *Éléments d'algorithmique* de Berstel *et* al. Nous allons dans la suite nous concentrer sur les équations de récurrence que l'on rencontre typiquement dans les algorithmes diviser pour régner.

2.4.2 ⋆ Récurrences de partitions

Nous considérons ici des relations de récurrence de la forme :

$$\begin{array}{rcl} T(1) & = & d \\ T(n) & = & aT(\frac{n}{b}) + f(n),\ n > 1. \end{array} \tag{2.1}$$

où $a \in \mathbb{R}^{*+}$, $b \in \mathbb{N}$, $b \geq 2$.

Notre but est de trouver une expression de $T(n)$ en fonction de n. Dans l'expression ci-dessus, lorsque $\frac{n}{b}$ n'est pas entier, on l'interprète par $\lfloor \frac{n}{b} \rfloor$ ou $\lceil \frac{n}{b} \rceil$.

La récurrence (2.1) correspond au coût du calcul effectué par un algorithme récursif du type diviser pour régner, dans lequel on remplace le problème de taille n par a sous-problèmes, chacun de taille $\frac{n}{b}$. Le temps de calcul $T(n)$ est donc $aT(\frac{n}{b})$ auquel il faut ajouter le temps $f(n)$ nécessaire à la combinaison des solutions des problèmes partiels en une solution du problème total. En général, on évalue une *borne supérieure* sur le coût $\tau(n)$ de l'algorithme, à savoir $\tau(1) = d$, et $\tau(n) \leq a\tau(\frac{n}{b}) + f(n)$, $n > 1$. On a donc $\tau(n) \leq T(n)$ pour tout n, et donc la fonction T constitue une majoration du coût de l'algorithme.

Lors de l'analyse de la complexité du tri fusion, nous avons vu une méthode de résolution d'équations de type (2.1), dans le cas où $a = b = 2$: nous avons construit un arbre binaire dont les nœuds représentent des coûts, tel que la somme des nœuds du niveau de profondeur i de l'arbre correspond au coût total des 2^i sous-problèmes de taille $\frac{n}{2^i}$. Pour a et b quelconques, l'arbre construit est a-aire, de hauteur $\log_b(n)$, le niveau de profondeur i correspondant au coût des a^i sous-problèmes, chacun de taille $\frac{n}{b^i}$, $0 \leq i \leq \log_b(n)$. Comme pour le tri fusion, l'addition des coûts à chaque niveau de l'arbre donne le coût total $T(n)$ pour une donnée de taille n. Cette méthode est appelée *méthode de l'arbre récursif*. Elle donne lieu à une expression générique pour $T(n)$ en fonction de n.

Lemme 2.4.1. *Soit $f : \mathbb{N} \to \mathbb{R}^+$, une fonction définie sur les puissances exactes de b, et soit $T : \mathbb{N} \to \mathbb{R}^+$, la fonction définie par*

$$\begin{array}{rcl} T(1) & = & d \\ T(n) & = & aT(\frac{n}{b}) + f(n),\ n = b^p,\ p \in \mathbb{N}^*, \end{array} \tag{2.2}$$

où $b \geq 2$ est entier, $a > 0$ et $d > 0$ sont réels. Alors

$$T(n) = \Theta(n^{\log_b(a)}) + \sum_{i=0}^{\log_b(n)-1} a^i f(\tfrac{n}{b^i}) \tag{2.3}$$

Preuve. L'expression ci-dessus provient exactement de l'addition des coûts à chaque niveau de l'arbre récursif correspondant à la récurrence (2.2) : l'arbre est a-aire, de hauteur $p = \log_b(n)$; le niveau de profondeur i correspond au coût des a^i sous-problèmes, chacun de taille $\frac{n}{b^i}$; le coût total du niveau i, $0 \leq i \leq \log_b(n) - 1$, est donc $a^i f(\frac{n}{b^i})$. Le dernier niveau ($i = \log_b(n)$) a un coût de $a^p d$. Ainsi :

$$T(n) = a^p d + \sum_{i=0}^{p-1} a^i f(\tfrac{n}{b^i})$$

On a :

$$a^p = a^{\log_b(n)} = b^{\log_b(n)\log_b(a)} = \Theta(b^{p\log_b(a)}) = \Theta(n^{\log_b(a)}),$$

d'où $a^p d = \Theta(n^{\log_b(a)})$. □

Dans le cas où $f(n) = n^k$, le temps de calcul est donné par le théorème suivant.

Théorème 2.4.1. *Soit $T : \mathbb{N} \to \mathbb{R}^+$, une fonction croissante, telle qu'il existe des entiers $b \geq 2$, et des réels $a > 0$, $c > 0$, $d > 0$, $k \geq 0$, pour lesquels*

$$\begin{array}{rcl} T(1) & = & d \\ T(n) & = & aT(\frac{n}{b}) + cn^k, \ \ n = b^p, \, p \in \mathbb{N}^*. \end{array} \tag{2.4}$$

Alors :

$$T(n) = \begin{cases} \Theta(n^k) & si\ a < b^k \\ \Theta(n^k \log_b(n)) & si\ a = b^k \\ \Theta(n^{\log_b(a)}) & si\ a > b^k. \end{cases} \tag{2.5}$$

Preuve. On suppose d'abord que n est une puissance de b, soit $n = b^p$, pour un entier $p \geq 1$. Par le lemme précédent,

$$T(n) = \theta(n^{\log_b(a)}) + cn^k \sum_{i=0}^{p-1} (\frac{a}{b^k})^i.$$

Notons $\gamma(n) = \sum_{i=0}^{p-1} (\frac{a}{b^k})^i$.
Si $\frac{a}{b^k} = 1$, $\gamma(n) = p \sim \log_b(n)$ et donc $T(n) = a^p d + cn^k \gamma(n) = \Theta(n^k \log_b(n))$.
Supposons à présent $\frac{a}{b^k} \neq 1$.

$$\gamma(n) = \sum_{i=0}^{p-1} (\frac{a}{b^k})^i = \frac{1 - (\frac{a}{b^k})^p}{1 - (\frac{a}{b^k})}.$$

Si $\frac{a}{b^k} < 1$, alors $\lim_{p\to\infty} (\frac{a}{b^k})^p = 0$ donc $\gamma(n) \sim \frac{1}{1-(a/b^k)}$, et $T(n) = \Theta(n^k)$.
Si $\frac{a}{b^k} > 1$,

$$\gamma(n) = \frac{(\frac{a}{b^k})^p - 1}{(\frac{a}{b^k}) - 1} = \alpha((\frac{a}{b^k})^p - 1),$$

avec $\alpha = \frac{1}{a/b^k - 1}$.

$$\frac{\gamma(n) - \alpha(\frac{a}{b^k})^p}{\alpha(\frac{a}{b^k})^p} = \frac{-1}{\alpha(\frac{a}{b^k})^p} \text{ et } \lim_{p\to\infty} \frac{-1}{\alpha(\frac{a}{b^k})^p} = 0,$$

donc

$$\gamma(n) \sim \alpha(\tfrac{a}{b^k})^p,$$

et

$$cn^k\gamma(n) \sim \alpha c a^p = \Theta(n^{\log_b(a)}) \text{ d'où } T(n) = \Theta(n^{\log_b(a)}).$$

Maintenant, soit n suffisamment grand, et soit $p \in \mathbb{N}^*$, tel que $b^p \leq n < b^{p+1}$. On a $T(b^p) \leq T(n) \leq T(b^{p+1})$. Or, $g(bn) = \Theta(g(n))$ pour chacune des trois fonctions g intervenant au second membre de (2.5) ; d'où $T(n) = \Theta(g(n))$. □

Remarques :

1. On compare k à $\log_b(a)$. Le comportement asymptotique de $T(n)$ suit $n^{\max(k,\log_b(a))}$, sauf pour $k = \log_b(a)$, auquel cas on multiplie la complexité par un « facteur correctif » $\log_b(n)$.
2. Ce théorème couvre les récurrences du type (2.1), avec $f(n) = cn^k,\ c > 0,\ k \geq 0$. Si $f(n) = O(n^k)$, le théorème reste valable en remplaçant les Θ par des O. Mais la borne obtenue peut alors ne pas être aussi fine [8] que celle obtenue en appliquant directement la formule (2.3) (*i.e.* la méthode de l'arbre récursif).

Par exemple, si l'on a une récurrence de la forme :

$$\begin{aligned} T(1) &= 0 \\ T(n) &= 2T(n/2) + n\log(n),\ \ n = 2^p,\ p \in \mathbb{N}^*, \end{aligned}$$

la fonction $f(n) = n\log(n)$ vérifie $f(n) = O(n^{3/2})$. Le théorème précédent donne $T(n) = O(n^{3/2})$ ($a = b = 2$, $k = 3/2$).

Supposons d'abord que n est une puissance de 2, soit $n = 2^p$. Par l'expression (2.3) :

$$T(n) = \sum_{i=0}^{p-1} 2^i \times \frac{n}{2^i}\log(\frac{n}{2^i}) = \sum_{i=0}^{p-1} n\log(\frac{n}{2^i}) = np\log(n) - \sum_{i=0}^{p-1} i = np\log(n) - \frac{p(p-1)}{2},$$

soit

$$T(n) = n(\log(n))^2 - \frac{\log(n)(\log(n)-1)}{2}.$$

Ainsi, $T(n) = \Theta(n(\log(n))^2)$. On obtient le même résultat lorsque n n'est pas une puissance de 2 en encadrant n entre deux puissances consécutives de 2.

En revanche, la récurrence

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2T(n/4) + n^2\sqrt{n},\ \ n = 4^p,\ p \in \mathbb{N}^*, \end{aligned} \tag{2.6}$$

8. Elle dépend de la finesse de l'approximation de f comme $O(n^k)$.

est bien adaptée au théorème précédent, avec $a = 2$, $b = 4$, $c = 1$, $k = 3/2$. On a $a < b^k = 8$ donc $T(n) = \Theta(n^{3/2})$.

Enfin, de façon plus générale, nous pouvons énoncer le théorème suivant.

Théorème 2.4.2. *Soit $T : \mathbb{N} \to \mathbb{R}^+$, une fonction croissante telle qu'il existe des entiers $b \geq 2$, des réels $a > 0$, $d > 0$, et une fonction $f : \mathbb{N} \to \mathbb{R}^+$ pour lesquels*

$$\begin{array}{rcl} T(1) & = & d \\ T(n) & = & aT(n/b) + f(n), \ \ n = b^p, \ p \in \mathbb{N}^*. \end{array} \tag{2.7}$$

Supposons de plus que

$$f(n) = cn^k(\log_b(n))^q$$

pour des réels $c > 0$, $k \geq 0$ et q. Alors :

$$T(n) = \begin{cases} \Theta(n^k) & si\ a < b^k\ \ et\ q = 0 \\ \Theta(n^k \log_b(n)^{1+q}) & si\ a = b^k\ \ et\ q > -1 \\ \Theta(n^k \log_b(\log_b(n))) & si\ a = b^k\ \ et\ q = -1 \\ \Theta(n^{\log_b(a)}) & si\ a = b^k\ \ et\ q < -1 \\ \Theta(n^{\log_b(a)}) & si\ a > b^k. \end{cases} \tag{2.8}$$

Preuve. Soit $n = b^p$. La formule (2.3) donne

$$T(n) = \Theta(n^{\log_b(a)}) + \sum_{i=0}^{p-1} a^i f(b^{p-i}).$$

Soit $\tilde{T}(n) = \sum_{i=0}^{p-1} a^i f(b^{p-i})$.

$$\tilde{T}(n) = \sum_{i=1}^{p} a^{p-i} f(b^i) = c \sum_{i=1}^{p} a^{p-i} b^{ik} (\log_b(b)^i)^q = ca^p \sum_{i=1}^{p} (\frac{b^k}{a})^i i^q,$$

d'où [9], en notant $\gamma(n) = \sum_{i=1}^{p} (\frac{b^k}{a})^i i^q$:

$$T(n) = \Theta(n^{\log_b(a)} \gamma(n)).$$

Si $a = b^k$, alors

$$\gamma(n) = \begin{cases} \Theta(p^{1+q}) = \Theta((\log_b(n))^{1+q}) & \text{si } q > -1 \\ \Theta(\log_b(p)) = \Theta(\log_b(\log_b(n))) & \text{si } q = -1 \\ \Theta(1) & \text{si } q < -1. \end{cases}$$

(Ces estimations sont obtenues en approchant $\gamma(n)$ par $\int_1^p x^q dx$ quand $p \to \infty$).

Si $a > b^k$, alors $\gamma(n) = \Theta(1)$. Si $a < b^k$ et $q = 0$, on a $\gamma(n) = \Theta(n^k / n^{\log_b(a)})$.

Lorsque n n'est pas une puissance de b, on procède comme dans la preuve du théorème précédent. □

9. *cf.* preuve du lemme 2.4.1.

2.5 ⋆ Compléments

2.5.1 ⋆ Récursivité terminale

La *récursivité terminale* (*tail recursivity* en anglais) est un cas particulier de récursivité : il s'agit du cas où un algorithme récursif effectue son appel récursif comme toute dernière instruction. C'est le cas par exemple de l'algorithme d'Euclide vu page 23 : l'appel récursif se fait dans le `return`, et aucune opération n'est effectuée sur le résultat retourné par l'appel récursif, il est juste passé au niveau du dessus. Dans ce cas, le compilateur a la possibilité d'optimiser l'appel récursif : au lieu d'effectuer l'appel récursif, récupérer le résultat à l'adresse de retour qu'il a fixé pour l'appel récursif, et recopier le résultat à sa propre adresse de retour, l'algorithme peut directement choisir de redonner sa propre adresse de retour à l'appel récursif. Cette optimisation présente l'avantage de ne pas avoir une pile récapitulant les appels récursifs aux différentes sous-fonctions : l'utilisation de récursion terminale supprime toute limite sur le nombre d'appels récursifs imbriqués.

Bien sûr, pour que la récursion terminale présente un intérêt, il faut qu'elle soit gérée par le compilateur. C'est le cas par exemple du compilateur Caml, ou (pour les cas simples de récursivité terminale) de `gcc` quand on utilise les options d'optimisation `-O2` ou `-O3`.

⚠ Attention !

Si l'appel récursif n'est pas la seule instruction dans le `return`*, il devient impossible pour le compilateur de faire l'optimisation : par exemple, un algorithme se terminant par une ligne du type* `return n*recursif(n-1)` *ne fait pas de récursivité terminale.*

2.5.2 ⋆ Dérécursification d'un programme

Il est *toujours* possible de supprimer la récursion d'un programme afin d'en obtenir une version itérative. C'est en fait ce que fait un compilateur lorsqu'il traduit un programme récursif en langage machine. En effet, pour tout appel de procédure, un compilateur engendre une série générique d'instructions : placer les valeurs des variables locales et l'adresse de la prochaine instruction sur la pile, définir les valeurs des paramètres de la procédure et aller au début de celle-ci ; et de même à la fin d'une procédure : dépiler l'adresse de retour et les valeurs des variables locales, mettre à jour les variables et aller à l'adresse de retour. Lorsque l'on veut « dérécursifier » un programme, la technique employée par le compilateur est la technique la plus générique. Cependant, dans certains cas, il est possible de faire plus simple, même si l'utilisation d'une pile est presque toujours nécessaire. Nous donnons ici deux exemples de dérécursification, pour l'algorithme d'Euclide, et pour un parcours d'arbre binaire (*cf.* chapitre 5).

Dérécursification de l'algorithme d'Euclide. L'écriture la plus simple de l'algorithme d'Euclide est récursive :

```
1 int euclide(int x, int y) {
2   if (y == 0) {
3     return x;
4   } else {
5     /* en C, x modulo y s'écrit x % y */
6     return euclide(y,x % y);
7   }
8 }
```

Toutefois, comme nous l'avons vu dans la section précédente, il s'agit ici de récursion terminale. Dans ce cas, de la même façon que le compilateur arrive à se passer de pile, nous pouvons aussi nous passer d'une pile, et récrire le programme avec une simple boucle `while`. On conserve la même condition de terminaison (sauf que dans la boucle `while` il s'agit d'une condition de continuation qu'il faut donc inverser), les deux variables `x` et `y`, et on se contente de mettre les bonnes valeurs dans `x` et `y` à chaque tour de la boucle. Cela donne la version itérative de l'algorithme d'Euclide :

```
1 int iterative_euclide(int x, int y) {
2   int tmp;
3   while (y != 0) {
4     /* une variable temporaire est nécessaire pour
5        "échanger" les valeurs x et y               */
6     tmp = x;
7     x = y;
8     y = tmp % y;
9   }
10  return x;
11 }
```

Dérécursification d'un parcours d'arbre. Prenons l'algorithme récursif de parcours préfixe d'arbre binaire suivant :

```
1 void depth_first_traversing(node n) {
2   if ( n != NULL ) {
3     explore(n);
4     depth_first_traversing(n->left);
5     depth_first_traversing(n->right);
6   }
7   return;
8 }
```

Ici *`node`* est une structure correspondant à un nœud de l'arbre. Cette structure contient les deux fils du nœud `left` et `right` et toute autre donnée que peut avoir à contenir le nœud. La fonction `explore` est la fonction que l'on veut appliquer à tous les nœuds de l'arbre (cela peut-être une comparaison dans le cas d'une recherche dans l'arbre).

Ici nous sommes en présence d'une récursion double, il est donc nécessaire d'utiliser une pile pour gérer l'ensemble des appels récursifs imbriqués. On suppose donc qu'une pile est implémentée (*cf.* section 3.3.1) et que les fonction `push` et `pop` nous permettent respectivement d'ajouter ou de récupérer un élément dans cette pile. On suppose que la fonction `stack_is_empty` renvoie 1 quand la pile est vide, 0 autrement. Ce qui va rendre cette dérécursification plus facile que le cas général est qu'ici les différents appels à la fonction sont indépendants les uns des autres : la fonction `explore` ne renvoie rien, et l'on n'utilise pas son résultat pour modifier la façon dont le parcours va se passer. On obtient alors le code itératif suivant :

```
1  void iterative_depth_first_traversing(node n) {
2    node current;
3    push(n);
4    while ( !stack_is_empty() ) {
5      current = pop();
6      if (current != NULL) {
7        explore(current);
8        push(current->right);
9        push(current->left);
10     }
11   }
12   return;
13 }
```

Le but est que chaque nœud qui est mis sur la pile soit un jour exploré, et que tous ses fils le soient aussi. Il suffit donc de mettre la racine sur la pile au départ, et ensuite, tant que la pile n'est pas vide d'explorer les nœuds qui sont dessus et à chaque fois d'ajouter leurs fils. Il faut faire attention à ajouter les fils dans l'ordre inverse des appels récursifs pour que le parcours se fasse bien dans le même ordre. En effet, le dernier nœud ajouté sur la pile sera le premier exploré ensuite.

2.5.3 ⋆ Indécidabilité de la terminaison

Un exemple de preuve de terminaison. Comme nous l'avons vu, dans un algorithme récursif, il est indispensable de vérifier que les conditions de terminaison seront atteintes pour tous les appels récursifs, et cela quelle que soit l'entrée. La façon la plus simple de prouver que c'est le cas est de définir une « distance aux conditions de terminaison » et de montrer que cette distance est strictement décroissante lors de chaque appel récursif [10]. Prenons par exemple le cas d'un calcul récursif de coefficients binomiaux (basé sur la construction du triangle de Pascal) :

10. Dans le cas ou cette distance est discrète (si par exemple cette distance est toujours entière), une décroissance stricte est suffisante, mais ce n'est pas le cas si la distance est continue (ce qui n'arrive jamais en informatique !).

```
1 int binomial(int n, int p) {
2   if ((p==0) || (n==p)) {
3     return 1;
4   }
5   return binomial(n-1,p) + binomial(n-1,p-1);
6 }
```

Ici la condition de terminaison est double : l'algorithme s'arrête quand l'un des deux bords du triangle de Pascal est atteint. Pour prouver que l'un des bords est atteint on peut utiliser la mesure $D : (n, p) \mapsto D(n, p) = p \times (n - p)$. Ainsi, on est sur un bord quand la mesure vaut 0 et on a $D(n-1, p) < D(n, p)$ et $D(n-1, p-1) < D(n, p)$. Donc la distance décroît strictement à chaque appel récursif. Cela prouve donc que cet algorithme termine.

Notons toutefois que cette façon de calculer les coefficients binomiaux est très mauvaise, il est bien plus rapide d'utiliser un algorithme itératif faisant le calcul du développement en factoriels du coefficient binomial.

Un exemple d'algorithme sans preuve de terminaison. Jusqu'à présent, tous les algorithmes récursifs que l'on a vu terminent, et on peut de plus prouver qu'ils terminent. Cependant dans certains cas, aucune preuve n'est connue pour dire que l'algorithme termine. Dans ce cas on parle *d'indécidabilité de la terminaison* : pour une entrée donnée, on ne peut pas savoir si l'algorithme va terminer, et la seule façon de savoir s'il termine est d'exécuter l'algorithme sur l'entrée en question, puis d'attendre qu'il termine (ce qui peut bien sûr ne jamais arriver). Regardons par exemple l'algorithme suivant :

```
1 int collatz(int n) {
2   if (n==1) {
3     return 0;
4   } else if ((n%2) == 0) {
5     return 1 + collatz(n/2);
6   } else {
7     return 1 + collatz(3*n+1);
8   }
9 }
```

Cet algorithme fait la chose suivante :

- on part d'un entier n
- si cet entier est pair on le divise par 2
- s'il est impair on le multiplie par 3 et on ajoute 1
- on recommence ainsi jusqu'à atteindre 1
- l'algorithme renvoie le nombre d'étapes nécessaires avant d'atteindre 1

La conjecture de Collatz (aussi appelée conjecture de Syracuse [11]) dit que cet algorithme termine toujours. Cependant, ce n'est qu'une conjecture, et aucune preuve n'est connue. Nous somme donc dans un contexte où, pour un entier donné, la seule façon de savoir si l'algorithme termine est d'exécuter l'algorithme et d'attendre : la terminaison est indécidable.

Un exemple d'algorithme pour lequel on peut prouver que la terminaison est indécidable. Maintenant nous arrivons dans des concepts un peu plus abstraits : avec la conjecture de Collatz nous avions un algorithme pour lequel aucune preuve de terminaison n'existe, mais nous pouvons aussi imaginer un algorithme pour lequel il est possible de prouver qu'il ne termine pas pour certaines entrées, mais décider *a priori* s'il termine ou non pour une entrée donnée est impossible. La seule façon de savoir si l'algorithme termine est de l'exécuter et d'attendre, sachant que pour certaines entrées l'attente sera infinie.

L'algorithme en question est un prouveur automatique pour des propositions logiques. Il prend en entrée un proposition A et cherche à prouver soit A, soit non-A. Il commence par explorer toutes les preuves de longueur 1, puis celles de longueur 2, puis 3 et ainsi de suite. Si la proposition A est décidable, c'est-à-dire que A ou non-A admet une preuve, cette preuve est de longueur finie, et donc notre algorithme va la trouver et terminer. En revanche, nous savons que certaines propositions sont indécidables : c'est ce qu'affirme le théorème d'incomplétude de Gödel (*cf.* http://en.wikipedia.org/wiki/Kurt_Godel, suivre le lien *Gödel's incompleteness theorems*). Donc nous savons que cet algorithme peut ne pas terminer, et nous ne pouvons pas décider s'il terminera pour une entrée donnée.

Nous pouvons donc prouver que la terminaison de cet algorithme est indécidable, contrairement à l'exemple précédent ou la terminaison était indécidable uniquement parce qu'on ne pouvait pas prouver la terminaison.

Le problème de l'arrêt de Turing. Dès 1936, Alan Turing s'est intéressé au problème de la terminaison d'un algorithme, qu'il a formulé sous la forme du *problème de l'arrêt* (*halting problem* en anglais) :

> *Étant donnée la description d'un programme et une entrée de taille finie, décider si le programme va finir ou va s'exécuter indéfiniment pour cette entrée.*

Il a prouvé qu'aucun algorithme générique ne peut résoudre ce problème pour tous les couples programme/entrée possibles. Cela signifie en particulier qu'il existe des couples programme/entrée pour lesquels le problème de l'arrêt est indécidable.

La preuve de Turing est assez simple et fonctionne par l'absurde. Supposons que l'on ait un algorithme `halt_or_not(A,i)` prenant en entrée un algorithme `A` et une entrée `i` et qui retourne vrai si `A(i)` termine, et faux si `A(i)` ne termine pas. On crée alors l'algorithme suivant :

11. Pour plus de détails sur cette conjecture, allez voir la page http://fr.wikipedia.org/wiki/Conjecture_de_Collatz, ou la version anglaise un peu plus complète http://en.wikipedia.org/wiki/Collatz_conjecture.

```
1 void halt_err(program A) {
2   if (halt_or_not(A,A)) {
3     while (1) {
4       printf("Boucle infinie\n");
5     }
6   }
7   return;
8 }
```

Cet algorithme `halt_err` va donc terminer uniquement si `halt_or_not(A,A)` renvoie faux, sinon, il part dans une boucle infinie. Maintenant si on appel `halt_err(halt_err)`, le programme ne termine que si `halt_or_not(halt_err,halt_err)` renvoie faux, mais s'il termine cela signifie que `halt_or_not(halt_err,halt_err)` devrait renvoyer vrai. De même, si l'appel à `halt_or_not(halt_err,halt_err)` renvoie vrai, `halt_err(halt_err)` va tourner indéfiniment, ce qui signifie que `halt_or_not(halt_err,halt_err)` aurait du renvoyer faux. Nous avons donc une contradiction : un programme ne peut donc pas résoudre le problème de l'arrêt pour tous les couples programme/entrée.

Chapitre 3

Structures de Données

L'informatique a révolutionné le monde moderne grâce à sa capacité à traiter de grandes quantités de données, des quantités beaucoup trop grandes pour être traitées à la main. Cependant, pour pouvoir manipuler efficacement des données de grande taille il est en général nécessaire de bien les structurer : tableaux, listes, piles, arbres, tas, graphes... Une multitude de structures de données existent, et une multitude de variantes de chaque structure, chaque fois mieux adaptée à un algorithme en particulier. Dans ce chapitre nous voyons les principales structures de données nécessaires pour optimiser vos premiers algorithmes.

3.1 Tableaux

Les tableaux sont la structure de donnée la plus simple. Ils sont en général implémentés nativement dans la majorité des langages de programmation et sont donc simples à utiliser. Un tableau représente une zone de mémoire consécutive d'un seul bloc (*cf.* FIGURE 3.1) ce qui présente à la fois des avantages et des inconvénients :

- la mémoire est en un seul bloc consécutif avec des éléments de taille constante à l'intérieur (la taille de chaque élément est défini par le type du tableau), donc il est très facile d'accéder au $i^{\text{ème}}$ élément du tableau. L'instruction `tab[i]` se contente de prendre l'adresse mémoire sur laquelle pointe `tab` et d'y ajouter i fois la taille d'un élément.
- il faut fixer la taille du tableau avant de commencer à l'utiliser. Les systèmes d'exploitation modernes gardent une trace des processus auxquels les différentes zones de mémoire appartiennent : si un processus va écrire dans une zone mémoire qui ne lui appartient pas (une zone que le noyau ne lui a pas alloué) il y a une erreur de segmentation. Quand vous programmez, avant d'écrire (ou de lire) à la case i d'un tableau il est nécessaire de vérifier que i est plus petit que la taille allouée au tableau (car le compilateur ne le vérifiera pas pour vous).

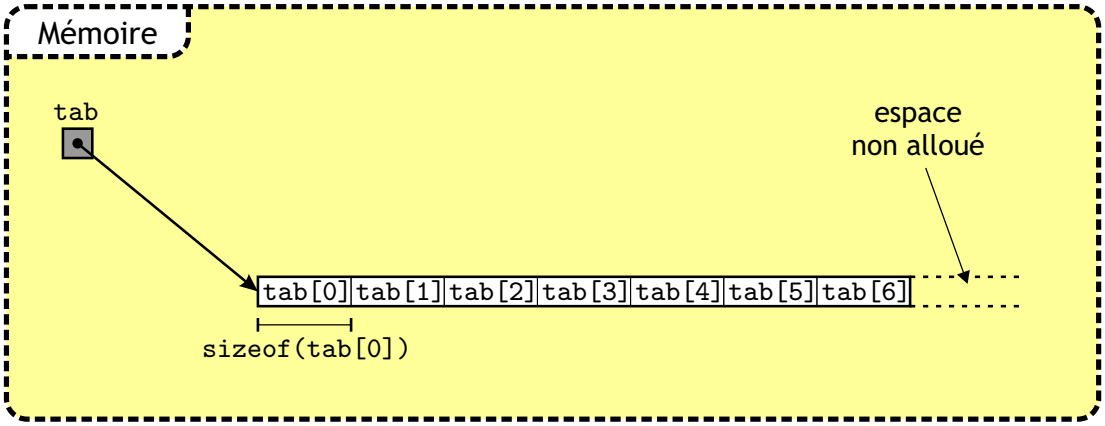


FIGURE 3.1 – *Représentation en mémoire d'un tableau simple.*

3.1.1 Allocation mémoire d'un tableau

Il existe deux façons d'allouer de la mémoire à un tableau.

- la plus simple permet de faire de l'allocation *statique*. Par exemple `int tab[100];` qui va allouer un tableau de 100 entiers pour `tab`. De même `int tab2[4][4];` va allouer un tableau à deux dimensions de taille 4×4. En mémoire ce tableau bidimensionnel peut ressemblera à ce qu'on voit dans la FIGURE 3.2. Attention, un tableau alloué statiquement ne se trouve pas dans la même zone mémoire qu'un tableau alloué avec l'une des méthodes d'allocation dynamique : ici il se trouve dans la même zone que toutes les variables de type `int`. On appelle cela de l'allocation statique car on ne peut pas modifier la taille du tableau en cours d'exécution.
- la deuxième façon utilise soit la commande `new` (syntaxe C++), soit la commande `malloc` (syntaxe C) et permet une allocation dynamique (dont la taille dépend des entrées par exemple). L'allocation du tableau s'écrit alors `int* tab = new int[100];` ou `int* tab = (int*) malloc(100*sizeof(int));`. En revanche cette technique ne permet pas d'allouer directement un tableau à deux dimensions. Il faut pour cela effectuer une boucle qui s'écrit alors :

```
1  int i;
2  int** tab2;
3  tab2 = (int**) malloc(4*sizeof(int*));
4  for (i=0; i<4; i++) {
5    tab2[i] = (int*) malloc(4*sizeof(int));
6  }
7  /* ou en utilisant new */
8  tab2 = new int*[4];
9  for (i=0; i<4; i++) {
10   tab2[i] = new int[4];
11 }
```

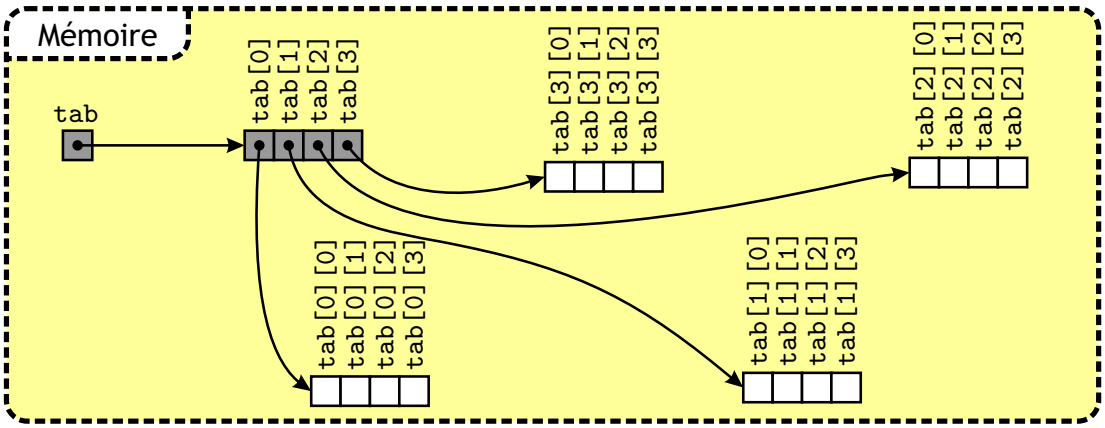


FIGURE 3.2 – *Représentation en mémoire d'un tableau à deux dimensions.*

L'utilisation de `malloc` (ou `new` en C++) est donc beaucoup plus lourde, d'autant plus que la mémoire allouée ne sera pas libérée d'elle même et l'utilisation de la commande `free` (ou `delete` en C++) sera nécessaire, mais présente deux avantages :

- le code est beaucoup plus proche de ce qui se passe réellement en mémoire (surtout dans le cas à deux dimensions) ce qui permet de mieux se rendre compte des opérations réellement effectuées. En particulier, une commande qui paraît simple comme `int tab[1000][1000][1000];` demande en réalité d'allouer un million de fois 1000 entiers, ce qui est très long et occupe 4Go de mémoire.
- cela laisse beaucoup plus de souplesse pour les allocations en deux dimensions (ou plus) : contrairement à la notation simplifiée, rien n'oblige à avoir des tableaux carrés !

Pour les cas simples, la notations `[100]` est donc suffisante, mais dès que cela devient compliqué, l'utilisation de `malloc` devient nécessaire.

Exemple d'allocation non carrée. Le programme suivant permet de calculer tous les coefficients binomiaux de façon récursive en utilisant la construction du triangle de Pascal. La méthode récursive simple vue au chapitre précédent (*cf.* page 30) est très inefficace car elle recalcule un grand nombre de fois les même coefficients. Pour l'améliorer, on utilise un tableau bidimensionnel qui va servir de cache : chaque fois qu'un coefficient est calculé on le met dans le tableau, et chaque fois que l'on a besoin d'un coefficient on regarde d'abord dans le tableau avant de la calculer. C'est ce que l'on appelle la *programmation dynamique*. Le point intéressant est que le tableau est triangulaire, et l'utilisation de `malloc` (ou `new`) permet de ne pas allouer plus de mémoire que nécessaire (on gagne un facteur 2 sur l'occupation mémoire ici).

```
1 int** tab;
2
3 int binomial(int n, int p) {
4   if (tab[n][p] == 0) {
```

```
5     if ((p==0) || (n==p) {
6       tab[n][p] = 1;
7     } else {
8       tab[n][p] = binomial(n-1,p) + binomial(n-1,p-1);
9     }
10  }
11  return tab[n][p];
12 }
13
14 int main(int argc, char** argv) {
15   int i;
16   tab = (int**) malloc(33*sizeof(int*));
17   for (i=0; i<34; i++) {
18     tab[i] = (int*) calloc((i+1),sizeof(int));
19   }
20   for (i=0; i<34; i++) {
21     binomial(33,i);
22   }
23
24   /* insérer ici les instructions qui
25      utilisent la table de binomiaux   */
26
27   for (i=0; i<33; i++) {
28     free(tab[i]);
29   }
30   free(tab);
31 }
```

On utilise donc la variable globale `tab` comme table de cache et la fonction binomiale est exactement la même qu'avant, mis à part qu'elle vérifie d'abord dans le cache si la valeur a déjà été calculée, et qu'elle stocke la valeur avant de la retourner. On arrête ici le calcul de binomiaux à $n = 33$ car au-delà les coefficients binomiaux sont trop grands pour tenir dans un `int` de 32 bits. Notez ici l'utilisation de `calloc` qui alloue la mémoire et l'initialise à 0, contrairement à `malloc` qui laisse la mémoire non initialisée. La fonction `calloc` est donc plus lente, mais l'initialisation est nécessaire pour pouvoir utiliser la technique de mise en cache. Cette technique de programmation dynamique est très utilisée quand l'on veut programmer vite et efficacement un algorithme qui se décrit mieux de façon récursive qu'itérative. Cela sera souvent le cas dans des problèmes combinatoires ou de dénombrement.

LIBÉRATION DE LA MÉMOIRE ALLOUÉE

Notez la présence des `free` *à la fin de la fonction* `main` *: ces* `free` *ne sont pas nécessaire si le programme est terminé à cet endroit là car la mémoire est de toute façon libérée par le système d'exploitation quand le processus s'arrête, mais si d'autres instructions doivent suivre, la mémoire allouée sera déjà libérée. De plus, il est important de prendre l'habitude de toujours libérer de la mémoire allouée (à chaque* `malloc` *doit correspondre un* `free`*) car cela permet de localiser plus facilement une* fuite de mémoire *(de la mémoire allouée mais non libérée, typiquement dans une boucle) lors du debuggage.*

3.1.2 ⋆ Complément : allocation dynamique de tableau

Nous avons vu que la structure de tableau a une taille fixée avant l'utilisation : il faut toujours allouer la mémoire en premier. Cela rend cette structure relativement peu adaptée aux algorithmes qui ajoutent dynamiquement des données sans que l'on puisse borner à l'avance la quantité qu'il faudra ajouter. Pourtant, les tableaux présentent l'avantage d'avoir un accès instantané à la $i^{\text{ème}}$ case, ce qui peut être très utile dans certains cas. On a alors envie d'avoir des tableaux de taille variable. C'est ce qu'implémente par exemple la classe `Vector` en java (sauf que la classe Vector le fait mal...).

L'idée de base est assez simple : on veut deux fonctions `insert` et `get(i)` qui permettent d'ajouter un élément à la fin du tableau et de lire le $i^{\text{ème}}$ élément en temps constant (en $O(1)$ en moyenne). Il suffit donc de garder en plus du tableau `tab` deux entiers qui indiquent le nombre d'éléments dans le tableau, et la taille totale du tableau (l'espace alloué). Lire le $i^{\text{ème}}$ élément peut se faire directement avec `tab[i]` (on peut aussi implémenter une fonction `get(i)` qui vérifie en plus que l'on ne va pas lire au-delà de la fin du tableau). En revanche, ajouter un élément est plus compliqué :

- soit le nombre d'éléments dans le tableau est strictement plus petit que la taille totale et il suffit d'incrémenter le nombre d'éléments et d'insérer le nouvel élément,
- soit le tableau est déjà rempli et il faut réallouer de la mémoire. Si on veut conserver un accès en temps constant il est nécessaire de conserver un tableau d'un seul bloc, il faut donc allouer un nouvel espace mémoire, plus grand que le précédent, y recopier le contenu de `tab`, libérer la mémoire occupée par `tab`, mettre à jour `tab` pour pointer vers le nouvel espace mémoire et on est alors ramené au cas simple vu précédemment.

Cette technique marche bien, mais pose un problème : recopier le contenu de `tab` est une opération longue et son coût dépend de la taille totale de `tab`. Recopier un tableau de taille n a une complexité de $\Theta(n)$. Heureusement, cette opération n'est pas effectuée à chaque fois, et comme nous allons le voir, il est donc possible de conserver une complexité en moyenne de $O(1)$.

La classe `Vector` de java permet à l'initialisation de choisir le nombre d'éléments à ajouter au tableau à chaque fois qu'il faut le faire grandir. Créer un `Vector` de taille n en augmentant de t à chaque fois va donc nécessiter de recopier le contenu du tableau une

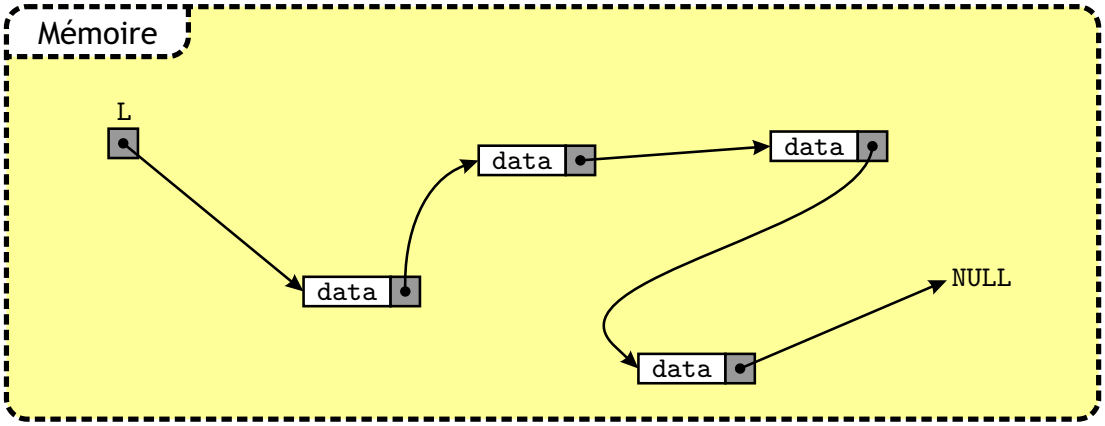


FIGURE 3.3 – *Représentation en mémoire d'une liste simplement chaînée.*

fois toute les t insertions. La complexité pour insérer n éléments est donc :

$$K = n + \sum_{i=1}^{\lfloor \frac{n}{t} \rfloor} t \times i \simeq t \times \frac{\frac{n}{t}(\frac{n}{t}+1)}{2} + n = \Theta(n^2).$$

Donc en moyenne, la complexité de l'insertion d'un élément est $\frac{K}{n} = \Theta(n)$. C'est beaucoup trop !

La bonne solution consiste à *doubler la taille du tableau à chaque réallocation* (ou la multiplier par n'importe quelle constante plus grande que 2). Ainsi, pour insérer n éléments dans le tableau il faudra avoir recopié une fois $\frac{n}{2}$ éléments, le coup d'avant $\frac{n}{4}$, celui d'avant $\frac{n}{8}$... Au total la complexité est donc :

$$K = n + \sum_{i=0}^{\lceil \log_2 n \rceil} 2^i \simeq 2n = \Theta(n).$$

Ce qui donne en moyenne une complexité par insertion de $\frac{K}{n} = \Theta(1)$. Il est donc possible de conserver toutes les bonnes propriétés des tableaux et d'ajouter une taille variable sans rien perdre sur les complexités asymptotiques.

La réallocation dynamique de tableau a donc un coût assez faible si elle est bien faite, mais elle nécessite en revanche d'utiliser plus de mémoire que les autres structures de données : un tableau contient toujours de l'espace alloué mais non utilisé, et la phase de réallocation nécessite d'allouer en même temps le tableau de taille n et celui de taille $\frac{n}{2}$.

3.2 Listes chaînées

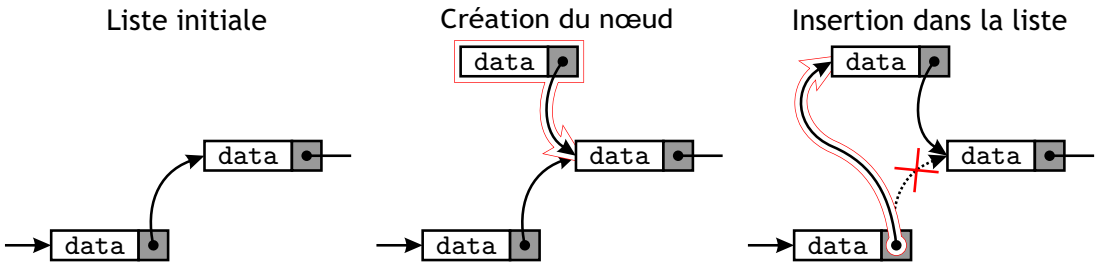
Contrairement à un tableau, une liste chaînée n'est pas constituée d'un seul bloc de mémoire : chaque élément (ou nœud) de la liste est alloué indépendamment des autres et contient d'une part des données et d'autre part un pointeur vers l'élément suivant (*cf.*

FIGURE 3.3). Une liste est donc en fait simplement un pointeur vers le premier élément de la liste et pour accéder à un autre élément, il suffit ensuite de suivre la chaîne de pointeurs. En général le dernier élément de la liste pointe vers `NULL`, ce qui signifie aussi qu'une liste vide est simplement un pointeur vers `NULL`. En C, l'utilisation de liste nécessite la création d'une structure correspondant à un nœud de la liste. Le type `list` en lui même doit ensuite être défini comme un pointeur vers un nœud. Cela donne le code suivant :

```
1 struct cell {
2   /* insérer ici toutes les données que doit contenir un noeud */
3   cell* next;
4 };
5 typedef cell* list;
```

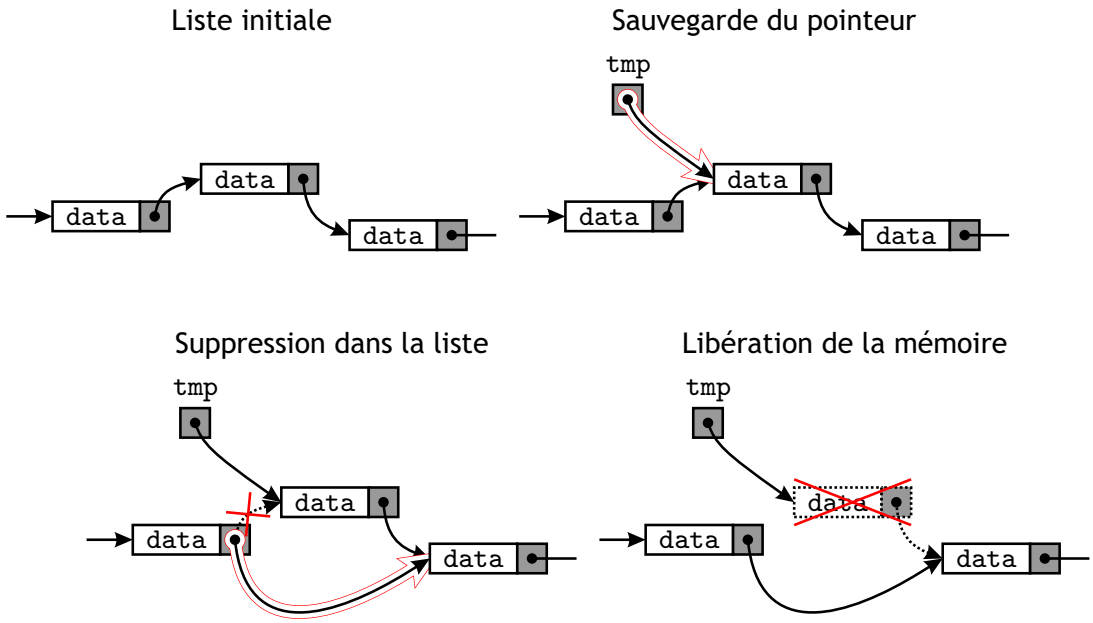
3.2.1 Opérations de base sur une liste

Insertion. L'insertion d'un élément à la suite d'un élément donné se fait en deux étapes illustrées sur le dessin suivant :



On crée le nœud en lui donnant le bon nœud comme nœud suivant. Il suffit ensuite de faire pointer l'élément après lequel on veut l'insérer vers ce nouvel élément. Le dessin représente une insertion en milieu de liste, mais en général, l'ajout d'un élément à une liste se fait toujours par le début : dans ce cas l'opération est la même, mais le premier élément de liste est un simple pointeur (sans champ `data`).

Suppression. Pour supprimer un nœud c'est exactement l'opération inverse, il suffit de faire attention à bien sauvegarder un pointeur vers l'élément que l'on va supprimer pour pouvoir libérer la mémoire qu'il utilise.



Ici encore, le dessin représente une suppression en milieux de liste, mais le cas le plus courant sera la suppression du premier élément d'une liste.

Parcours. Pour parcourir une liste on utilise un « curseur » qui pointe sur l'élément que l'on est en train de regarder. On initialise le curseur sur le premier élément et on suit les pointeurs jusqu'à arriver sur `NULL`. Il est important de ne jamais perdre le pointeur vers le premier élément de la liste (sinon les éléments deviennent définitivement inaccessibles) : c'est pour cela que l'on utilise une autre variable comme curseur. Voila le code C d'une fonction qui recherche une valeur (passée en argument) dans une liste d'entiers, et remplace toutes les occurrences de cette valeur par des 0.

```
1  void cleaner(int n, list L) {
2    list cur = L;
3    while (cur != NULL) {
4      if (cur->data == n) {
5        cur->data = 0;
6      }
7      cur = cur->next;
8    }
9    return;
10 }
```

Notons qu'ici, la liste `L` est passée en argument, ce qui crée automatiquement une nouvelle variable locale à la fonction `cleaner`. Il n'était donc pas nécessaire de créer la variable `cur`. Cela ayant de toute façon un coût négligeable par rapport à un appel de fonction, il n'est pas gênant de prendre l'habitude de toujours avoir une variable dédiée pour le curseur.

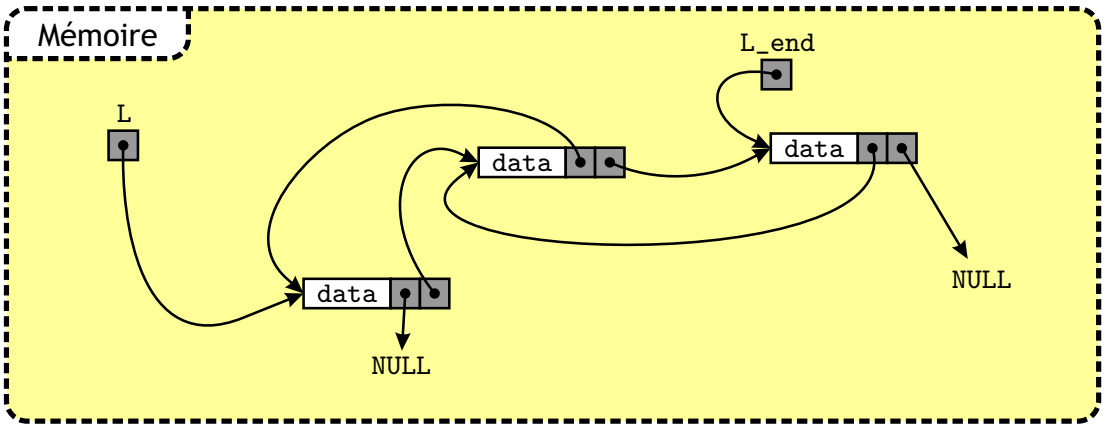


FIGURE 3.4 – *Représentation en mémoire d'une liste doublement chaînée.*

3.2.2 Les variantes : doublement chaînées, circulaires...

La structure de liste est une structure de base très souvent utilisée pour construire des structures plus complexes comme les piles ou les files que nous verrons à la section suivante. Selon les cas, un simple pointeur vers l'élément suivant peut ne pas suffire, on peut chercher à avoir directement accès au dernier élément... Une multitude de variations existent et seules les plus courantes sont présentées ici.

Listes doublement chaînées. Une liste doublement chaînée (*cf.* FIGURE 3.4) possède en plus des listes simples un pointeur vers l'élément précédent dans la liste. Cela s'accompagne aussi en général d'une deuxième variable `L_end` pointant vers le dernier élément de la liste et permet ainsi un parcours dans les deux sens et un ajout simple d'éléments en fin de liste. Le seul surcoût est la présence du pointeur en plus qui ajoute quelques opérations de plus à chaque insertion/suppression et qui occupe un peu d'espace mémoire.

Listes circulaires. Les listes circulaires sont des listes chaînées (simplement ou doublement) dont le dernier élément ne pointe pas vers `NULL`, mais vers le premier élément (*cf.* FIGURE 3.5). Il n'y a donc plus de réelle notion de début et fin de liste, il y a juste une position courante indiquée par un curseur. Le problème est qu'une telle liste ne peux jamais être vide : afin de pouvoir gérer ce cas particulier il est nécessaire d'utiliser ce que l'on appelle une sentinelle.

Sentinelles. Dans une liste, une sentinelle est un nœud particulier qui doit pouvoir être reconnaissable en fonction du contenu de son champ `data` (une méthode classique est d'ajouter un entier au champ `data` qui est non-nul uniquement pour la sentinelle) et qui sert juste à simplifier la programmation de certaines listes mais ne représente pas un réel élément de la liste. Une telle sentinelle peut avoir plusieurs usages :

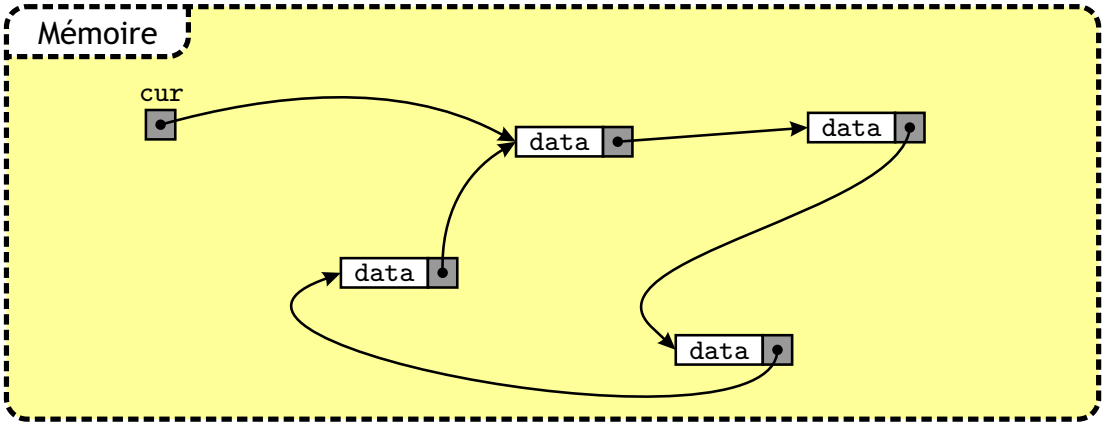


FIGURE 3.5 – *Représentation en mémoire d'une liste circulaire simplement chaînée.*

- représenter une liste circulaire vide : il est possible de décider qu'une liste circulaire vide sera représentée en mémoire par une liste circulaire à un seul élément (qui est donc son propre successeur) qui serait une sentinelle.
- terminer une liste non-circulaire : il peut être pratique d'ajouter une sentinelle à la fin de toute liste chaînée pour que lorsque la liste est vide des fonctions comme « retourner le premier élément » ou « retourner le dernier élément » aient toujours quelque chose à renvoyer. Dans la plupart des cas on peut leur demander de retourner `NULL`, mais une sentinelle peut rendre un programme plus lisible.

On peut aussi envisager d'avoir plusieurs types de sentinelles, par exemple une qui marquerait un début de liste et une autre une fin de liste.

3.2.3 Conclusion sur les listes

Par rapport à un tableau la liste présente deux principaux avantages :

- il n'y a pas de limitation de longueur d'une liste (à part la taille de la mémoire)
- il est très facile d'insérer ou de supprimer un élément au milieu de la liste sans pour autant devoir tout décaler ou laisser un trou.

En revanche, pour un même nombre d'éléments, une liste occupera toujours un peu plus d'espace mémoire qu'un tableau car il faut stocker les pointeurs (de plus le système d'exploitation conserve aussi des traces de tous les segments de mémoire alloués pour pouvoir les désallouer quand le processus s'arrête) et l'accès au $i^{\text{ème}}$ élément de la liste coûte en moyenne $\Theta(n)$ pour une liste de longueur n.

3.3 Piles & Files

Les piles et les files sont des structures très proches des listes et très utilisées en informatique. Ce sont des structures de données dynamiques (comme les listes) sur lesquelles

on veut pouvoir exécuter deux instructions principales : ajouter un élément et retirer un élément (en le retournant afin de pouvoir lire son contenu). La seule différence entre la pile et la liste est que dans le cas de la pile l'élément que l'on retire est le dernier à avoir été ajouté (on parle alors de LIFO comme *Last In First Out*), et dans le cas de la file on retire en premier les élément ajoutés en premier (c'est un FIFO comme *First In First Out*). Les noms de pile et file ne sont bien sûr pas choisis au hasard : on peut penser à une pile de livres sur une table ou à une file d'attente à la boulangerie qui se comportent de la même façon.

3.3.1 Les piles

Les piles sont en général implémentées à l'aide d'une liste simplement chaînée, la seule différence est que l'on utilise beaucoup moins d'opérations : en général on ne cherche jamais à aller lire ce qui se trouve au fond d'une pile, on ne fait qu'ajouter et retirer des éléments. Une pile vide est alors représentée par une liste vide, ajouter un élément revient à ajouter un élément en tête de liste, et le retirer à retourner son contenu et à la supprimer de la liste. Voici en C ce que pourrait être l'implémentation d'une pile d'entiers :

```
1  list L;
2  void push(int n) {
3    cell* nw = (cell*) malloc(sizeof(cell));
4    nw->data = n;
5    nw->next = L;
6    L = nw;
7  }
8
9  int pop() {
10   int val;
11   cell* tmp;
12   /* on teste d'abord si la pile est vide */
13   if (L == NULL) {
14     return -1;
15   }
16   val = L->data;
17   tmp = L;
18   L = L->next;
19   free(tmp);
20   return val;
21 }
```

La fonction `push` ajoute un entier dans la pile et la fonction `pop` retourne le dernier entier ajouté à la pile et le retire de la pile. Pour l'utilisateur qui n'utilise que les fonctions `push` et `pop`, le fait que la pile est implémentée avec une liste `L` est transparent.

Exemples d'utilisation de piles. Les piles sont une structure de donnée très basique (il n'y a que deux opérations possibles), mais elles sont utilisées très souvent en informatique.

Depuis le début de ce poly nous avons déjà vu deux exemples d'utilisation :

- Les tours de Hanoï : programmer réellement les tours de Hanoï nécessite trois piles (une pour chaque piquet) contenant des entiers représentant les rondelles. Dans ces piles on peut uniquement ajouter une rondelle (c'est ce que fait `push`) ou retirer la rondelle du dessus (avec `pop`).
- Dans la dérécursification : lorsqu'un compilateur dérécursifie un programme, il utilise une pile pour stocker les appels récursifs. De façon plus général, tout appel à une fonction se traduit par l'ajout de plusieurs éléments (adresse de retour...) dans la pile qui sert à l'exécution du processus.

Une autre utilisation courante de pile est l'évaluation d'expressions mathématiques. La notation polonaise inverse (que l'on retrouve sur les calculatrice HP) est parfaitement adaptée à l'usage d'une pile : chaque nombre est ajouté à la pile et chaque opération prend des éléments dans la pile et retourne le résultat sur la pile. Par exemple, l'expression $5 \times (3+4)$ s'écrit en notation polonaise inverse $4\ 3\ +\ 5\ \times$ ce qui représente les opérations :

- `push(4)`
- `push(3)`
- `push(pop() + pop())`
- `push(5)`
- `push(pop() * pop())`

À la fin il reste donc juste 35 sur la pile.

3.3.2 Les files

Les files sont elles aussi en générale implémentées à l'aide d'une liste. En revanche, les deux opérations `push` et `pop` que l'on veut implémenter doivent l'une ajouter un élément en fin de liste et l'autre retirer un élément en début de liste (ainsi les éléments sortent bien dans le bon ordre : *First In, First Out*). Il est donc nécessaire d'avoir un pointeur sur la fin de la liste pour pouvoir facilement (en temps constant) y insérer des éléments. Pour cela on utilise donc une liste simplement chaînée, mais avec deux pointeurs : l'un sur le début et l'autre sur la fin. Si on suppose que `L` pointe vers le premier élément de la liste et `L_end` vers le dernier, voici comment peuvent se programmer les fonctions `push` et `pop` :

```
1  list L, L_end;
2  void push(int n) {
3    cell* nw = (cell*) malloc(sizeof(cell));
4    nw->data = n;
5    nw->next = NULL;
6    /* si la file est vide, il faut mettre
7       à jour le premier et le dernier élément */
8    if (L == NULL) {
9      L = nw;
10     L_end = nw;
11   } else {
12     L_end->next = nw;
```

```
13      L_end = nw;
14    }
15  }
16
17  int pop() {
18    int val;
19    cell* tmp;
20    /* on teste d'abord si la file est vide */
21    if (L == NULL) {
22      return -1;
23    }
24    val = L->data;
25    tmp = L;
26    L = L->next;
27    free(tmp);
28    return val;
29  }
```

Encore une fois, pour l'utilisateur de cette file, le fait que nous utilisons une liste est transparent.

Exemples d'utilisation de files. Les files sont utilisées partout où l'on a des données à traiter de façon asynchrone avec leur arrivée, mais où l'ordre de traitement de ces données est important (sinon on préfère en général utiliser une pile qui est plus légère à implémenter). C'est le cas par exemple pour le routage de paquets réseau : le routeur reçoit des paquets qui sont mis dans une file au fur et à mesure qu'ils arrivent, le routeur sort les paquets de cette file un par un et les traite en fonction de leur adresse de destination pour les renvoyer. La file sert dans ce cas de mémoire tampon et permet ainsi de mieux utiliser les ressources du routeur : si le trafic d'entrée est irrégulier il suffit grâce au tampon de pouvoir traiter un flux égal au débit d'entrée moyen, alors que sans tampon il faudrait pouvoir traiter un flux égal au débit maximum en entrée.

Chapitre 4

Recherche en table

Le problème auquel on s'intéresse dans ce chapitre est celui de la mise en œuvre informatique d'un dictionnaire ; autrement dit, il s'agit d'étudier les méthodes permettant de retrouver le plus rapidement possible une information en mémoire, accessible à partir d'une clef (par exemple, trouver une définition à partir d'un mot). La méthode la plus naturelle est l'implémentation par tableau (table à adressage direct : pour gérer un dictionnaire avec des mots jusqu'à 30 lettres on crée une table assez grande pour contenir les 26^{30} *mots possibles, et on place les définitions dans la case correspondant à chaque mot), mais bien que sa complexité temporelle soit optimale, cette méthode devient irréaliste en terme de complexité spatiale dès lors que l'on doit gérer un grand nombre de clefs. Une deuxième méthode consiste à ne considérer que les clefs effectivement présentes en table (recherche séquentielle) : on obtient alors des performances analogues aux opérations de base sur les listes chaînées, la recherche d'un élément étant linéaire en la taille de la liste. Une approche plus efficace est la recherche dichotomique, dans laquelle on utilise un tableau contenant les clefs triées (comme c'est le cas dans un vrai dictionnaire) ; la complexité de l'opération de recherche devient alors en* $\log(n)$*, est rend cette méthode très intéressante pour de gros fichiers auxquels on accède souvent mais que l'on modifie peu. Enfin, une quatrième méthode abordée est l'utilisation de tables de hachage, qui permet d'atteindre en moyenne les performances de l'adressage direct tout en s'affranchissant du nombre potentiellement très grand de clefs possible.*

4.1 Introduction

Un problème très fréquent en informatique est celui de la recherche de l'information stockée en mémoire. Cette information est en général accessible *via* une clef. La consultation d'un annuaire électronique est l'exemple type d'une telle recherche. Un autre exemple est celui d'un compilateur, qui doit gérer une table des symboles, dans laquelle les clefs sont les identificateurs des donnés à traiter. Les fonctionnalités [1] d'un dictionnaire sont exactement celles qu'il convient de mettre en oeuvre lorsque l'on souhaite gérer l'information en

1. Pour la création, l'utilisation et la mise à jour.

mémoire. En effet, les opérations que l'on désire effectuer sont :

- la recherche de l'information correspondant à une clef donnée,
- l'insertion d'une information à l'endroit spécifié par la clef,
- la suppression d'une information.

À noter que si la clef correspondant à une information existe déjà, l'insertion est une *modification* de l'information. On dispose donc de paires de la forme (clef,information), auxquelles on doit appliquer ces opérations. Pour cela, l'utilisation d'une structure de données dynamique s'impose.

Nous allons ici étudier les structures de données permettant l'implémentation efficace de dictionnaires. Nous supposerons que les clefs sont des entiers dans l'intervalle $[0, m-1]$, où m est un entier qui peut être très grand (en général il sera souvent très grand). Si les clefs ont des valeurs non entières, on peut appliquer une bijection de l'ensemble des clefs vers (un sous-ensemble de) $[0, m-1]$. Nous nous intéresserons à la complexité spatiale et temporelle des trois opérations ci-dessus, complexités dépendantes de la structure de données utilisée.

4.2 Table à adressage direct

La méthode la plus naturelle pour rechercher l'information est d'utiliser un tableau `tab` de taille m : `tab[i]` contiendra l'information de clef i. La recherche de l'information de clef i se fait alors en $\Theta(1)$ (il suffit de retourner `tab[i]`), tout comme l'insertion (qui est une simple opération d'affectation `tab[i] = info`) et la suppression (`tab[i] = NULL`, avec la convention que `NULL` correspond à une case vide). Le stockage du tableau nécessite un espace mémoire de taille $\Theta(m)$.

En général, une table à adressage direct ne contient pas directement l'information, mais des pointeurs vers l'information. Cela présente deux avantages : d'une part cela permet d'avoir une information par case de taille variable et d'autre part, la mémoire à allouer pour le tableau est m fois la taille d'un pointeur au lieu de m fois la taille d'une information. Dans un tableau d'informations (qui n'utiliserait pas de pointeurs) il faut allouer m fois la taille de l'information la plus longue, mais avec des pointeurs on n'alloue que m fois la taille d'un pointeur, plus la taille totale de toutes les informations. En pratique, si on appelle *`type_info`* le type d'une définition, on allouera la mémoire avec *`type_info**`* `tab = new` *`type_info*`*`[m]` pour un tableau avec pointeurs. Par convention `tab[i] = NULL` si la case est vide (aucune information ne correspond à la clef i), sinon `tab[i] = &info`.

Dans le cas où l'information est de petite taille, l'utilisation d'un tableau sans pointeurs peut toutefois être intéressante. Chaque case du tableau doit alors contenir deux champs : le premier champ est de type booléen et indique la présence de la clef (si la clef n'est pas présente en table c'est que l'information associée n'est pas disponible), l'autre est de type `type_info` et contient l'information proprement dite.

```
1 struct cell {
2   bool key_exists;
```

```
3   type_info info;
4 };
5 cell* tab = (cell*) malloc(m*sizeof(cell));
```

Toute ces variantes de structures de données conduisent à la complexité donnée ci-dessus.

Cette représentation en tableau (ou variantes) est bien adaptée lorsque m est petit, mais si m est très grand, une complexité spatiale linéaire devient rapidement irréaliste. Par exemple si l'on souhaite stocker tous les mots de huit lettres (minuscules sans accents), nous avons $m = 26^8 \simeq 2^{37.6}$ mots possibles, soit un espace de stockage de 100 Go rien que pour le tableau de pointeurs.

4.3 Recherche séquentielle

En général, l'espace des clefs possibles peut être très grand, mais le nombre de clefs présentes dans la table est bien moindre (c'est le cas pour les mots de huit lettres). L'idée de la recherche séquentielle [2] est de mettre les données (les couples (clef,info)) dans un tableau de taille n, où n est le nombre maximal [3] de clefs susceptibles de se trouver simultanément en table. On utilise aussi un indice p indiquant la première case libre du tableau. Les données sont insérées en fin de tableau (`tab[p]` reçoit (clef,info) et p est incrémenté). On effectue une recherche en parcourant le tableau *séquentiellement* jusqu'à trouver l'enregistrement correspondant à la clef cherchée, ou arriver à l'indice n de fin de tableau (on suppose que le tableau est entièrement rempli). Si l'on suppose que toutes les clefs ont la même probabilité d'être recherchées, le nombre moyen de comparaisons est :

$$\sum_{i=1}^{n} \frac{1}{n} \times i = \frac{n+1}{2}.$$

En effet, si l'élément cherché se trouve en `tab[i]`, on parcourra le tableau jusqu'à la position i (ce qui correspond à $i+1$ comparaisons). Les clefs susceptibles d'être cherchées étant supposées équiprobables, i peut prendre toute valeur entre 0 et $n-1$. En cas de recherche infructueuse, on a à effectuer n comparaisons. La complexité de l'opération de recherche est donc en temps $\Theta(n)$.

L'insertion d'un enregistrement s'effectue en temps $\Theta(1)$, sauf si la clef est déjà présente en table, auquel cas la modification doit être précédée d'une recherche de cette clef. L'opération de suppression nécessitant toujours une recherche préalable, elle s'effectue en temps $\Theta(n)$ (il est également nécessaire de décaler les éléments restant dans le tableau après suppression, ce qui a aussi une complexité $\Theta(n)$). La complexité de stockage du tableau est en $\Theta(n)$.

Si l'on dispose d'information supplémentaires sur les clefs, on peut améliorer la complexité des opérations ci-dessus, par exemple en plaçant les clefs les plus fréquemment lues en début de tableau.

2. On parle aussi de recherche linéaire.
3. Ou une borne supérieure sur ce nombre si le nombre maximal n'est pas connu d'avance.

Voici comment s'implémente les opérations de recherche, insertion, suppression dans une telle structure.

```
1  struct cell {
2    int key;
3    type_info info;
4  };
5  cell* tab; /* l'allocation doit être faite dans le main */
6  int p=0;
7
8  type_info search(int val) {
9    int i;
10   for (i=0; i<p; i++) {
11     if (tab[i].key == val) {
12       return tab[i].info;
13     }
14   }
15   printf("Recherche infructueuse.\n");
16   return NULL;
17 }
18
19 /* on suppose qu'une clef n'est jamais insérée deux fois */
20 void insert(cell nw) {
21   tab[p] = nw;
22   p++;
23 }
24
25 void delete(int val) {
26   int i,j;
27   for (i=0; i<p; i++) {
28     if (tab[i].key == val) {
29       for (j=i; j<p-1; j++) {
30         tab[j] = tab[j+1];
31       }
32       tab[p-1] = NULL;
33       p--;
34       return;
35     }
36   }
37   printf("Élément inexistant, suppression impossible.\n");
38 }
```

On peut également utiliser une liste chaînée à la place d'un tableau. Un avantage est alors que l'on n'a plus de limitation sur la taille. Les complexités de recherche et de suppression sont les mêmes que dans l'implémentation par tableau de taille n. L'insertion conserve elle aussi sa complexité de $\Theta(1)$ sauf que les nouveaux éléments sont insérés au début au lieu d'à la fin. La modification nécessite toujours un temps en $\Theta(n)$ (le coût d'une recherche). Asymptotiquement les complexités sont les mêmes, mais dans la pratique la recherche sera un peu plus lente (un parcours de tableau est plus rapide qu'un parcours

de liste) et la suppression un peu plus rapide (on n'a pas à décaler tous les éléments du tableau). Avec une liste, les opérations sont alors exactement les mêmes que celles décrites dans le chapitre sur les listes.

4.4 Recherche dichotomique

Une méthode efficace pour diminuer la complexité de l'opération de recherche est d'utiliser un tableau (ou une liste) de clefs *triées par ordre croissant* (ou décroissant). On peut alors utiliser l'approche « diviser-pour-régner » pour réaliser la recherche : on compare la clef `val` cherchée à celle située au milieu du tableau. Si elles sont égales, on retourne l'information correspondante. Sinon, si `val` est supérieure à cette clef, on recommence la recherche dans la partie supérieure du tableau. Si `val` est inférieure, on explore la partie inférieure. On obtient l'algorithme récursif suivant (on recherche entre les indices p (inclus) et r (exclus) du tableau) :

```
1  type_info dicho_search(cell* tab, int val, int p, int r) {
2    int q = (p+r)/2;
3    if (p == r) {
4      /* la recherche est infructueuse */
5      return NULL;
6    }
7    if (val == tab[q].key) {
8      return tab[q].info;
9    } else if (val < tab[q].key) {
10     return dicho_search(tab, val, p, q);
11   } else {
12     return dicho_search(tab, val, q+1, r);
13   }
14 }
```

Complexité de l'opération de recherche. La complexité est ici calculée en nombre d'appels récursifs à `dicho_search`, ou de manière équivalente en nombre de comparaisons de clefs à effectuer (même si l'algorithme comporte deux comparaisons, tant que ce nombre est constant il va disparaître dans le Θ de la complexité).

On note $n = r - p$, la taille du tableau et C_n, la complexité cherchée. Pour une donnée de taille n, on fait une comparaison de clef, puis (si la clef n'est pas trouvée) on appelle la procédure récursivement sur un donnée de taille $\lfloor \frac{n}{2} \rfloor$, donc $C_n \leq C_{\lfloor \frac{n}{2} \rfloor} + 1$. De plus, $C_1 = 1$. Si $n = 2^k$, on a alors

$$C_{2^k} \leq C_{2^{k-1}} + 1 \leq C_{2^{k-2}} + 2 \leq \ldots \leq C_{2^0} + k = k + 1,$$

autrement dit, $C_{2^k} = O(k)$.

Lorsque n n'est pas une puissance de deux, notons k l'entier tel que $2^k \leq n < 2^{k+1}$. C_n est une fonction croissante, donc

$$C_{2^k} \leq C_n \leq C_{2^{k+1}},$$

soit $k+1 \leq C_n \leq k+2$. Ainsi, $C_n = O(k) = O(\log(n))$. La complexité de l'opération de recherche avec cette représentation est donc bien meilleure qu'avec les représentations précédentes.

Malheureusement ce gain sur la recherche fait augmenter les coût des autres opérations. L'insertion d'un élément doit maintenant se faire à la bonne place (et non pas à la fin du tableau). Si l'on considère l'insertion d'un élément choisi aléatoirement, on devra décaler en moyenne $n/2$ éléments du tableau pour le placer correctement. Ainsi, l'opération d'insertion est en $\Theta(n)$. Le coût de l'opération de suppression lui ne change pas et nécessite toujours le décalage d'en moyenne $n/2$ éléments.

En pratique, cette méthode est à privilégier dans le cas où l'on a un grand nombre de clef sur lesquels on peut faire de nombreuses requêtes, mais que l'on modifie peu. On peut alors, dans une phase initiale, construire la table par une méthode de tri du type tri rapide.

Recherche par interpolation. La recherche dichotomique ne modélise pas notre façon intuitive de rechercher dans un dictionnaire. En effet, si l'on cherche un mot dont la première lettre se trouve à la fin de l'alphabet, on ouvrira le dictionnaire plutôt vers la fin. Une façon de modéliser ce comportement est réalisée par la recherche par interpolation. Elle consiste simplement à modifier l'algorithme précédent en ne considérant plus systématiquement le milieu du tableau, mais plutôt une estimation de l'endroit où la clef devrait se trouver dans le tableau. Soit encore `val`, la clef à rechercher. Étant donnée que le tableau est trié par ordre croissant des clefs, la valeur de `val` donne une indication de l'endroit où elle se trouve ; cette estimation est donnée par :

$$\mathtt{q} = \mathtt{p} + \frac{\mathtt{val} - \mathtt{tab[p]}}{\mathtt{tab[r-1]} - \mathtt{tab[p]}}(\mathtt{r} - \mathtt{p}).$$

Il s'avère que, dans le cas où les clefs ont une distribution aléatoire, cette amélioration réduit drastiquement le coût de la recherche, comme le montre le résultat suivant (que nous admettrons) :

Théorème 4.4.1. *La recherche par interpolation sur un ensemble de clefs distribuées aléatoirement nécessite moins de* $\log(\log(n)) + 1$ *comparaisons, sur un tableau de taille* n*.*

Attention, ce résultat très intéressant n'est valable que si les clef sont réparties uniformément dans l'ensemble des clefs possibles. Une distribution biaisée peut grandement dégrader cette complexité.

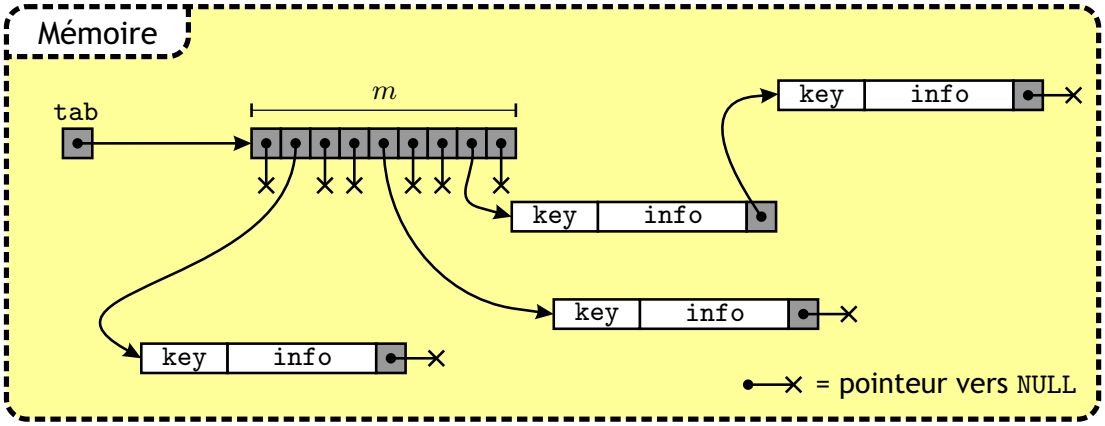


FIGURE 4.1 – *Représentation en mémoire d'une table de hachage. La huitième case du tableau contient une collision : deux clef qui ont la même valeur hachée.*

4.5 Tables de hachage

Lorsque l'ensemble des clefs effectivement stockées est beaucoup plus petit que l'espace Ω de toutes les clefs possibles, on peut s'inspirer de l'adressage direct - très efficace lorsque Ω est petit - pour implémenter un dictionnaire. Soit n le nombre de clefs réellement présentes. L'idée est d'utiliser une fonction surjective h de Ω dans $[0, m-1]$ (où on choisit en général $m \sim n$). Il suffit alors d'avoir une table de taille m (comme précédemment) et de placer une clef c dans la case $k = h(c)$ de cette table. La fonction h est appelée *fonction de hachage*, la table correspondante est appelée *table de hachage* et l'entier $k = h(c)$ la *valeur hachée* de la clef c. Les fonctions utilisées habituellement sont très rapides, et l'on peut considérer que leur temps d'exécution est en $\Theta(1)$ quelle que soit la taille de l'entrée.

Le principal problème qui se pose est l'existence de plusieurs clefs possédant la même valeur hachée. Si l'on considère par exemple deux clefs c_1, c_2, telles que $h(c_1) = h(c_2)$, le couple (c_1, c_2) est appelé une *collision*. Étant donné que $|\Omega| \gg m$, les collisions sont nombreuses. Une solution pour gérer les collisions consiste à utiliser un tableau `tab` de listes (chaînées) de couples (clef,info) : le tableau est indicé par les entiers de 0 à $m-1$ et, pour chaque indice i, `tab[i]` est (un pointeur sur) la liste chaînée de tous les couples (clef,info) tels que h(clef) $= i$ (*i.e.* la liste de toutes les collisions de valeur hachée i). Une telle table de hachage est représentée en FIGURE 4.1. Les opérations de recherche, insertion, suppression sont alors les mêmes que lorsque l'on implémente une recherche séquentielle avec des listes, sauf qu'ici les listes sont beaucoup plus petites.

L'opération d'insertion a une complexité en $\Theta(1)$ (insertion de (clef,info) au début de la liste `tab[i]` avec $i = h$(clef)). Pour déterminer la complexité - en nombre d'éléments à examiner (comparer) - des autres opérations (recherche et suppression), on doit d'abord connaître la taille moyenne des listes. Pour cela, on définit $\rho = \frac{n}{m}$, le *facteur de remplissage* de la table, *i.e.* le nombre moyen d'éléments stockés dans une même liste. À noter que ρ peut prendre des valeurs arbitrairement grandes, *i.e.* il n'y a pas de limite *a priori* sur le nombre d'éléments susceptibles de se trouver en table.

Dans le pire cas, les n clefs ont toute la même valeur hachée, et on retombe sur la recherche séquentielle sur une liste de taille n, en $\Theta(n)$ (suppression en $\Theta(n)$ également). En revanche, si la fonction de hachage répartit les n clefs uniformément dans l'intervalle $[0, m-1]$ - on parle alors de *hachage uniforme simple* - chaque liste sera de taille ρ en moyenne, et donc la recherche d'un élément (comme sa suppression) nécessitera au plus $\rho+1$ comparaisons. Une estimation préalable de la valeur de n permet alors de dimensionner la table de façon à avoir une recherche en temps constant (en choisissant $m = O(n)$). La complexité spatiale de cette méthode est en $\Theta(m+n)$, pour stocker la table et les n éléments de liste.

⋆ **Choix de la fonction de hachage.** Les performances moyennes de la recherche par table de hachage dépendent de la manière dont la fonction h répartit (en moyenne) l'ensemble des clefs à stocker parmi les m premiers entiers. Nous avons vu que, pour une complexité moyenne optimale, h doit réaliser un hachage uniforme simple.

En général, on ne connaît pas la distribution initiale des clefs, et il est donc difficile de vérifier si une fonction de hachage possède cette propriété. En revanche, plus les clefs sont réparties uniformément dans l'espace Ω des clefs possibles, moins il sera nécessaire d'utiliser une « bonne » fonction de hachage. Si les clefs sont réparties de façon parfaitement uniforme, on peut choisir $m = 2^\ell$ et simplement prendre comme valeur hachée les ℓ premiers bits de la clef. Cela sera très rapide, mais la moindre régularité dans les clefs peut être catastrophique. Il est donc en général recommandé d'avoir une fonction de hachage dont la sortie dépend de tous les bits de la clef.

Si l'on suppose que les clefs sont des entiers naturels (si ce n'est pas le cas, on peut en général trouver un codage qui permette d'interpréter chaque clef comme un entier), la fonction modulo :

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

est rapide et possède de bonnes propriétés de répartition statistique dès lors que m est un nombre premier assez éloigné d'une puissance de 2 ; (si $m = 2^\ell$, cela revient à prendre ℓ bits de la clef).

Une autre fonction fréquemment utilisée est la fonction

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

où x est une constante réelle appartenant à $]0, 1[$. Le choix de $x = \frac{\sqrt{5}-1}{2}$ conduit à de bonnes performances en pratique (hachage de Fibonacci), mais le passage par des calculs flottants rend le hachage un peu plus lent.

4.6 Tableau récapitulatif des complexités

méthode	stockage	recherche	insertion	modif.	suppr.
adressage direct	$\Theta(m)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
recherche séquentielle	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
recherche dichotomique	$\Theta(n)$	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
recherche par interpolation	$\Theta(n)$	$\Theta(\log\log(n))$	$\Theta(\log\log(n))$	$\Theta(n)$	$\Theta(n)$
tables de hachage	$\Theta(m+n)$	$\Theta(\frac{n}{m})^*$	$\Theta(1)$	$\Theta(\frac{n}{m})^*$	$\Theta(\frac{n}{m})^*$
avec $m \simeq n$	$\Theta(n)$	$\Theta(1)^*$	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)^*$

Note : les complexités ci-dessus sont celles dans le pire cas, sauf celles suivies de * pour lesquelles il s'agit du cas moyen.

On constate qu'une table de hachage de taille bien adaptée et munie d'une bonne fonction de hachage peut donc être très efficace. Cependant cela demande d'avoir une bonne idée de la quantité de données à gérer à l'avance.

Chapitre 5

Arbres

La notion d'arbre modélise, au niveau des structures de données, la notion de récursivité pour les fonctions. Il est à noter que la représentation par arbre est très courante dans la vie quotidienne : arbres généalogiques, organisation de compétitions sportives, organigramme d'une entreprise... Dans ce chapitre, après avoir introduit la terminologie et les principales propriétés des arbres, nous illustrons leur utilisation à travers trois exemples fondamentaux en informatique : l'évaluation d'expressions, la recherche d'information, et l'implémentation des files de priorité. Nous terminons par la notion d'arbre équilibré, dont la structure permet de garantir une complexité algorithmique optimale quelle que soit la distribution de l'entrée.

5.1 Préliminaires

5.1.1 Définitions et terminologie

Un arbre est une collection (éventuellement vide) de nœuds et d'arêtes assujettis à certaines conditions : un nœud peut porter un nom et un certain nombre d'informations pertinentes (les données contenues dans l'arbre) ; une arête est un lien entre deux nœuds.

Une branche (ou chemin) de l'arbre est une suite de nœuds distincts, dans laquelle deux nœuds successifs sont reliés par une arête. La longueur d'une branche est le nombre de ses arêtes (ou encore le nombre de nœuds moins un). Un nœud est spécifiquement désigné comme étant la *racine* de l'arbre[1]. La propriété fondamentale définissant un arbre est alors que *tout nœud est relié à la racine par une et une seule branche*[2] (*i.e.* il existe un unique chemin d'un nœud donné à la racine).

Comme on peut le voir sur la FIGURE 5.1, chaque nœud possède un lien (descendant, selon la représentation graphique) vers chacun de ses fils (ou *descendants*), éventuellement un lien vers le vide, ou pas de lien si le nœud n'a pas de fils ; inversement, tout nœud - sauf

1. Si la racine n'est pas spécifiée, on parle plutôt d'arborescence.

2. Dans le cas ou certains nœuds sont reliés par plus d'une branche (ou pas de branche du tout) à la racine on parle alors de *graphe*.

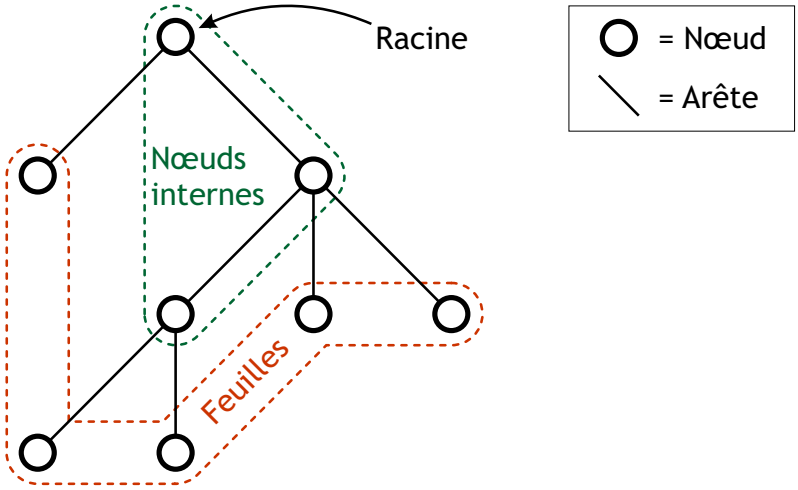


FIGURE 5.1 – *Représentation d'un arbre. On place toujours la racine en haut.*

la racine - possède un père (ou *ancêtre*) et un seul, qui est le nœud immédiatement au-dessus de lui dans la représentation graphique. Les nœuds sans descendance sont appelés *feuilles*, les nœuds avec descendance étant des *nœuds internes*. Tout nœud est la racine du sous-arbre constitué par sa descendance et lui-même.

Voici une liste du vocabulaire couramment utilisé pour les arbres :

- Une *clef* est une information contenue dans un nœud.
- Un ensemble d'arbres est une *forêt*.
- Un arbre est *ordonné* si l'ordre des fils de chacun de ses nœuds est spécifié. C'est généralement le cas dans les arbres que l'on va considérer dans la suite.
- Un arbre est de plus *numéroté* si chacun de ses fils est numéroté par un entier strictement positif (deux nœuds ne possédant pas le même numéro).
- Les nœuds d'un arbre se répartissent en *niveaux*. Le niveau d'un nœud est la longueur de la branche qui le relie à la racine.
- La *hauteur* d'un arbre est le niveau maximum de l'arbre (*i.e.* plus grande distance d'un nœud à la racine).
- La *longueur totale* d'un arbre est la somme des longueurs de tous les chemins menant des nœuds à la racine.
- Le *degré* d'un nœud est le nombre de fils qu'il possède.
- L'*arité* d'un arbre est le degré maximal de ses nœuds.
- Lorsque l'on a un arbre dont les fils de chaque nœud sont numérotés par des entiers tous compris dans l'intervalle $[1, \ldots, k]$, on parle d'arbre *k-aire*. Un arbre k-aire est donc un arbre numéroté, d'arité inférieure ou égale à k.
- Comme avec les sentinelles pour les listes, on peut définir un type particulier de nœud dit *nœud vide* : c'est un nœud « factice » dans le sens où il ne contient pas d'information ; il peut juste servir à remplir la descendance des nœuds qui contiennent moins de k fils.
- L'exemple le plus important d'arbre m-aire est l'*arbre binaire*. Chaque nœud possède deux fils et on parle alors de *fils gauche* et de *fils droit* d'un nœud interne. On peut

également définir la notion de fils gauche et fils droit d'une feuille : il suffit de représenter chacun d'eux par le nœud vide. De cette manière, tout nœud non vide peut être considéré comme un nœud interne.
– Un arbre binaire est *complet* si les nœuds remplissent complètement tous les niveaux, sauf éventuellement le dernier, pour lequel les nœuds apparaissent alors tous le plus à gauche possible (notez que l'arbre binaire de hauteur 0 est complet).

5.1.2 Premières propriétés

La meilleure définition des arbres est sans doute récursive : un arbre est soit l'arbre vide, soit un nœud appelé racine relié à un ensemble (éventuellement vide) d'arbres appelés ses fils.

Cette définition récursive se particularise trivialement au cas des arbres binaires comme suit : un arbre binaire est soit l'arbre vide, soit un nœud racine relié à un arbre binaire gauche (appelé sous-arbre gauche) et un arbre binaire droit (appelé sous-arbre droit). Un arbre binaire est évidemment un arbre ; mais réciproquement, tout arbre peut être représenté par un arbre binaire (*cf.* plus loin la représentation des arbres). Cette vision récursive des arbres nous permet de démontrer les propriétés suivantes.

Propriété 5.1.1. *Il existe une branche unique reliant deux nœuds quelconques d'un arbre.*

Propriété 5.1.2. *Un arbre de N nœuds contient $N-1$ arêtes.*

Preuve. C'est une conséquence directe du fait que tout nœud, sauf la racine, possède un père et un seul, et que chaque arête relie un nœud à son père. Il y a donc autant d'arêtes que de nœuds ayant un père, soit $N-1$. □

Propriété 5.1.3. *Un arbre binaire complet possédant N nœuds internes contient $N+1$ feuilles.*

Preuve. Pour un arbre binaire complet A, notons $f(A)$, son nombre de feuilles et $n(A)$, son nombre de nœuds internes. On doit montrer que $f(A) = n(A) + 1$. Le résultat est vrai pour l'arbre binaire de hauteur 0 (il est réduit à une feuille). Considérons un arbre binaire complet $A = (r, A_g, A_d)$, r désignant la racine de l'arbre, et A_g et A_d ses sous-arbres gauche et droit respectivement. Les feuilles de A étant celles de A_g et de A_d, on a $f(A) = f(A_g) + f(A_d)$; les nœuds internes de A sont ceux de A_g, ceux de A_d, et r, d'où $n(A) = n(A_g) + n(A_d) + 1$. A_g et A_d étant des arbres complets, la récurrence s'applique, et $f(A_g) = n(A_g) + 1$, $f(A_d) = n(A_d) + 1$. On obtient donc $f(A) = f(A_g) + f(A_d) = n(A_g) + 1 + n(A_d) + 1 = n(A) + 1$. □

Propriété 5.1.4. *La hauteur h d'un arbre binaire contenant N nœuds vérifie $h + 1 \geq \log_2(N+1)$.*

Preuve. Un arbre binaire contient au plus 2 nœuds au premier niveau, 2^2 nœuds au deuxième niveau,..., 2^h nœuds au niveau h. Le nombre maximal de nœuds pour un arbre binaire de hauteur h est donc $1 + 2 + \ldots + 2^h = 2^{h+1} - 1$, *i.e.* $N + 1 \leq 2^{h+1}$. □

Pour un arbre binaire complet de hauteur h contenant N nœuds, tous les niveaux (sauf le dernier sont complètement remplis. On a donc $N \geq 1 + 2 + \ldots + 2^{h-1} = 2^h - 1$, *i.e.* $\log_2(N+1) \geq h$. La hauteur d'un arbre binaire complet à N nœuds est donc toujours de l'ordre de $\log_2(N)$.

5.1.3 Représentation des arbres

Arbres binaires. Pour représenter un arbre binaire, on utilise une structure similaire à celle d'une liste chaînée, mais avec deux pointeurs au lieu d'un : l'un vers le fils gauche, l'autre vers le fils droit. On définit alors un nœud par :

```
1 struct node {
2   int key;
3   type_info info;
4   node* left;
5   node* right;
6 };
7 typedef node* binary_tree;
```

Arbres *k*-aires. On peut construire un arbre k-aire de façon très similaire (où k est un entier fixé une fois pour toute dans le programme) :

```
1 struct node {
2   int key;
3   type_info info;
4   node sons[k];
5 };
6 typedef node* k_ary_tree;
```

Arbres généraux. Dans le cas d'un arbre général il n'y a pas de limite sur le nombre de fils d'un nœud. Il faut donc utiliser une structure dynamique pour stocker tous les fils d'un même nœud. Pour cela on utilise une liste chaînée. Chaque nœud contient donc un pointeur vers son fils le plus à gauche (le premier élément de la liste), qui lui contient un pointeur vers sont frère juste à sa droite (la queue de la liste). Chaque nœud contient donc un pointeur vers un fils et un pointeur vers un frère. On obtient une structure de la forme :

```
1 struct node {
2   int key;
3   type_info info;
4   node* brother;
5   node* son;
6 };
7 typedef node* tree;
```

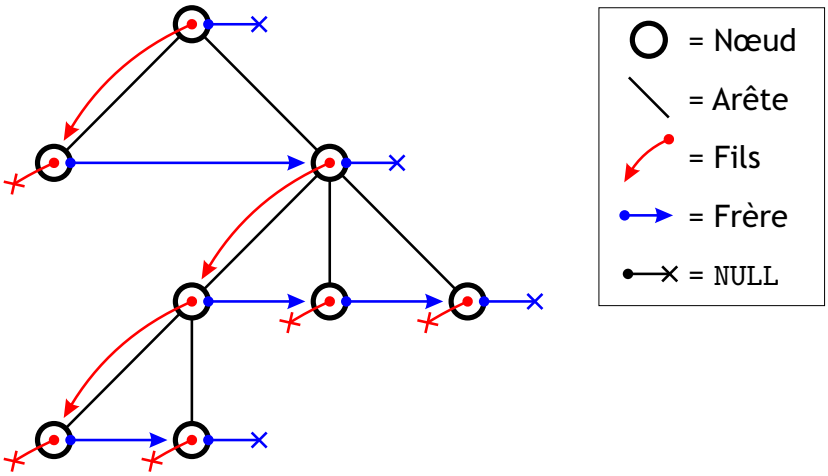


FIGURE 5.2 – *Arbre général : on accède aux fils d'un nœud en suivant une liste chaînée.*

Dans cette représentation, on voit que tout nœud possède deux liens : elle est donc identique à la représentation d'un arbre binaire. Ainsi, on peut voir un arbre quelconque comme un arbre binaire (avec, pour tout nœud, le lien gauche pointant sur son fils le plus à gauche, le lien droit vers son frère immédiatement à droite). Nous verrons cependant les modifications à apporter lors du parcours des arbres généraux.

REMONTER DANS UN ARBRE

Dans les applications où il est seulement nécessaire de remonter dans l'arbre, et pas de descendre, la représentation par lien-père d'un arbre est suffisante. Cette représentation consiste à stocker, pour chaque nœud, un lien vers son père. On peut alors utiliser deux tableaux pour représenter un tel arbre : on prend soin d'étiqueter les nœuds de l'arbre au préalable (de 1 à k s'il y a k nœuds). Le premier tableau `tab_info` *de taille k contiendra l'information contenue dans chaque nœud (*`tab_info[i]` *= info du nœud i), le deuxième tableau* `tab_father` *de taille k lui aussi contiendra les liens père, de telle sorte que* `tab_father[i]` *= clef du père du nœud i. L'information relative au père du nœud i se trouve alors dans* `tab_info[tab_father[i]]`*.*
Lorsqu'on doit juste remonter dans l'arbre, une représentation par tableaux est donc suffisante. On pourrait aussi utiliser une liste de tous les nœud dans laquelle chaque nœud contiendrait en plus un lien vers son père.

5.2 Utilisation des arbres

Il existe une multitude d'algorithmes utilisant des arbres. Souvent, l'utilisation d'arbres ayant une structure bien précise permet d'obtenir des complexités asymptotiques meilleures qu'avec des structures linéaires (liste ou tableau par exemple). Nous voyons ici quelques exemples typiques d'utilisation d'arbres.

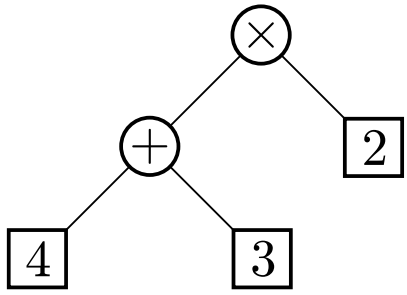


FIGURE 5.3 – *Un exemple d'arbre d'évaluation, correspondant à l'expression* $(4+3)\times 2$.

5.2.1 Évaluation d'expressions & parcours d'arbres

Nous nous intéressons ici à la représentation par arbre des expressions arithmétiques. Dans cette représentation, les nœuds de l'arbre sont les opérateurs, et les feuilles les opérandes. S'il n'y a que des opérateurs d'arité 2 (comme + ou ×) on obtient alors un arbre binaire. Si on considère des opérateurs d'arité supérieur on obtient un arbre général, mais on a vu que tout arbre est équivalent à un arbre binaire. On ne considère donc ici que des arbres binaires et des opérateurs d'arité deux, mais les algorithmes seront les mêmes pour des opérateurs d'arité quelconque.

On ne s'intéresse pas ici à la construction d'un arbre d'évaluation, mais uniquement à son évaluation. On part donc d'une arbre comme celui représenté sur la FIGURE 5.3. L'évaluation d'une telle expression se fait en *parcourant* l'arbre, c'est-à-dire en visitant chaque nœud de manière systématique. Il existe trois [3] façons de parcourir un arbre. Chacune correspond à une écriture différente de l'expression. Nous détaillons ci-après ces trois parcours, qui sont essentiellement récursifs, et font tous parti des parcours dits *en profondeur* (*depth-first* en anglais).

Parcours préfixe. Ce parcours consiste à visiter la racine d'abord, puis le sous-arbre gauche, et enfin le sous-arbre droit. Il correspond à l'écriture d'une expression dans laquelle les opérateurs sont placés avant les opérandes, par exemple $\times + 4\ 3\ 2$ pour l'expression de la FIGURE 5.3. Cette notation est aussi appelée notation polonaise. L'algorithme suivant effectue un parcours préfixe de l'arbre `A` et pour chaque nœud visité affiche son contenu (pour afficher au final $\times + 4\ 3\ 2$).

```
1 void preorder_traversal(tree A) {
2   if (A != NULL) {
3     printf("%d ", A->key);
```

3. Plus le parcours par niveau, appelé également *parcours en largeur* (*i.e.* parcours du haut vers le bas, en visitant tous les nœuds d'un même niveau de gauche à droite avant de passer au niveau suivant. Ce parcours n'est pas récursif, et est utilisé par exemple dans la structure de tas (*cf.* section 5.2.3), mais ne permet pas d'évaluer une expression.

```
4     preorder_traversal(A->left);
5     preorder_traversal(A->right);
6   }
7 }
```

Parcours infixe. Le parcours infixe correspond à l'écriture habituelle des expressions arithmétiques, par exemple $(4 + 3) \times 2$ (sous réserve que l'on rajoute les parenthèses nécessaires). Algorithmiquement, on visite d'abord le sous-arbre gauche, puis la racine, et enfin le sous-arbre droit :

```
1 void inorder_traversal(tree A) {
2   if (A != NULL) {
3     printf("(");
4     inorder_traversal(A->left);
5     printf("%d", A->key);
6     inorder_traversal(A->right);
7     printf(")");
8   }
9 }
```

On est obligé d'ajouter des parenthèses pour lever les ambiguïtés, ce qui rend l'expression un peu plus lourde : l'algorithme affichera $((4) + (3)) \times (2)$.

Parcours postfixe. Ce parcours consiste à visiter le sous-arbre gauche d'abord, puis le droit, et enfin la racine. Il correspond à l'écriture d'une expression dans laquelle les opérateurs sont placés après les opérandes, par exemple $4\ 3 + 2\times$ (c'est ce que l'on appel la notation polonaise inverse, bien connue des utilisateurs de calculatrices HP).

```
1 void postorder_traversal(tree A) {
2   if (A != NULL) {
3     postorder_traversal(A->left);
4     postorder_traversal(A->right);
5     printf("%d ", A->key);
6   }
7 }
```

Complexité. Pour les trois parcours ci-dessus, il est clair que l'on visite une fois chaque nœud, donc n appels récursifs pour un arbre à n nœuds, et ce quel que soit le parcours considéré. La complexité de l'opération de parcours est donc en $\Theta(n)$.

Cas des arbres généraux. Les parcours préfixes et postfixes sont aussi bien définis pour les arbres quelconques. Dans ce cas, la loi de parcours préfixe devient : visiter la racine, puis chacun des sous-arbres ; la loi de parcours postfixe devient : visiter chacun des

sous-arbres, puis la racine. Le parcours infixe ne peut en revanche pas être bien défini si le nombre de fils de chaque nœud est variable.

Notons aussi que le parcours postfixe d'un arbre généralisé est équivalent au parcours infixe de l'arbre binaire équivalent (comme vu sur la FIGURE 5.2) et que le parcours préfixe d'un arbre généralisé est équivalant au parcours préfixe de l'arbre binaire équivalent.

Parcours en largeur. Le parcours en largeur (*breadth-first* en anglais) est très différent des parcours en profondeur car il se fait par niveaux : on visite d'abord tous les nœuds du niveau 0 (la racine), puis ceux du niveau 1... Un tel parcours n'est pas récursif mais doit être fait sur l'arbre tout entier. La technique la plus simple utilise deux files `A` et `B`. On initialise `A` avec la racine et `B` à vide. Puis à chaque étape du parcours on `pop` un nœud de la file `A`, on effectue l'opération que l'on veut dessus (par exemple afficher la clef), et on ajoute tous les fils de ce nœud à `B`. On recommence ainsi jusqu'à avoir vider `A`, et quand `A` et vide on inverse les files `A` et `B` et on recommence. On s'arrête quand les deux files sont vides.

À chaque étape la file `A` contient les nœuds du niveau que l'on est en train de parcourir, et la file `B` se remplit des nœuds du niveau suivant. On effectue donc bien un parcours par niveaux de l'arbre. Ce type de parcours n'est pas très fréquent sur des arbres et sera plus souvent rencontré dans le contexte plus général des graphes.

5.2.2 Arbres Binaires de Recherche

Nous avons vu dans le chapitre 4 comment implémenter un dictionnaire à l'aide d'une table. Nous avons vu que la méthode la plus efficace est l'utilisation d'une table de hachage. Cependant, pour qu'une telle table offre des performances optimales, il est nécessaire de connaître sa taille finale à l'avance. Dans la plus part des contextes, cela n'est pas possible. Dans ce cas, l'utilisation d'un arbre binaire de recherche est souvent le bon choix.

Un arbre binaire de recherche (ABR) est un arbre binaire tel que le sous-arbre gauche de tout nœud contienne des valeurs de clef strictement inférieures à celle de ce nœud, et son sous-arbre droit des valeurs supérieures ou égales. Un tel arbre est représenté en FIGURE 5.4. Cette propriété est suffisante pour pouvoir affirmer qu'un parcours *infixe* de l'arbre va visiter les nœuds dans l'ordre croissant des clefs. Un ABR est donc un arbre ordonné dans lequel on veut pouvoir facilement effectuer des opérations de recherche, d'insertion et de suppression, et cela sans perturber l'ordre.

Recherche dans un ABR. Pour chercher une clef dans un ABR on utilise naturellement une méthode analogue à la recherche dichotomique dans une table. Pour trouver un nœud de clef v, on commence par comparer v à la clef de la racine, notée ici r. Si $v < r$, alors on se dirige vers le sous-arbre gauche de la racine. Si $v = r$, on a terminé, et on retourne l'information associée à la racine. Si $v > v_r$, on considère le sous-arbre droit de la racine. On applique cette méthode récursivement sur les sous-arbres.

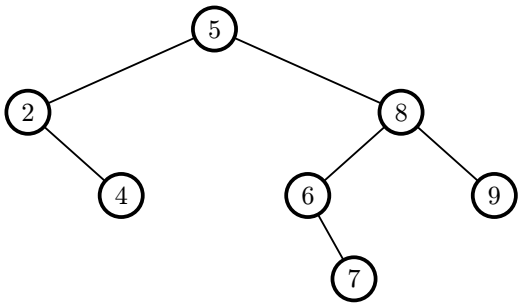


FIGURE 5.4 – *Un exemple d'arbre binaire de recherche (ABR).*

Il est à noter que la taille du sous-arbre courant diminue à chaque appel récursif. La procédure s'arrête donc toujours : soit parce qu'un nœud de clef v a été trouvé, soit parce qu'il n'existe pas de nœud ayant la clef v dans l'arbre, et le sous-arbre courant est alors vide. Cela peut se programmer de façon récursive ou itérative :

```
type_info ABR_search(int v, tree A) {
  if (A == NULL) {
    printf("Recherche infructueuse.\n");
    return NULL;
  }
  if (v < A->key) {
    return ABR_search(v, A->left);
  } else if (v == A->key) {
    printf("Noeud trouvé.\n");
    return A->info;
  } else {
    return ABR_search(v,A->right);
  }
}
```

```
type_info ABR_search_iter(int v, tree A) {
  node* cur = A;
  while (cur != NULL) {
    if (v < cur->key) {
      cur = cur->left;
    } else if (v == cur->key) {
      printf("Noeud trouvé.\n");
      return cur->info;
    } else {
      cur = cur->right;
    }
  }
  printf("Recherche infructueuse.\n");
  return NULL;
}
```

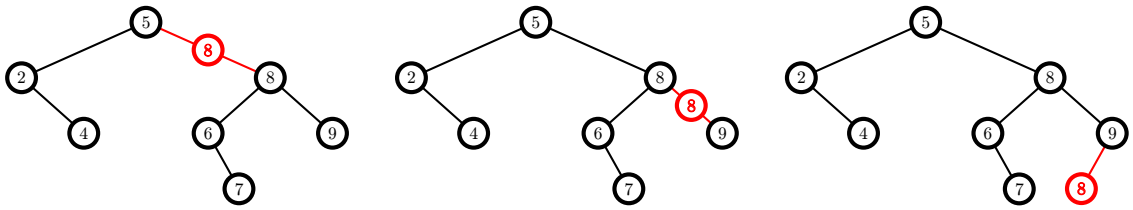


FIGURE 5.5 – *Insertion d'un nœud dont la clef est déjà présente dans un ABR. Trois emplacements sont possibles.*

Insertion dans un ABR. L'insertion dans un ABR n'est pas une opération compliquée : il suffit de faire attention à bien respecter l'ordre au moment de l'insertion. Pour cela, la méthode la plus simple et de parcourir l'arbre de la même façon que pendant la recherche d'une clef et lorsque l'on atteint une feuille, on peut insérer le nouveau nœud, soit à gauche, soit à droite de cette feuille, selon la valeur de la clef. On est alors certain de conserver l'ordre dans l'ABR. Cette technique fonctionne bien quand on veut insérer un nœud dont la clef n'est pas présente dans l'ABR. Si la clef est déjà présente deux choix sont possibles (*cf.* FIGURE 5.5) : soit on dédouble le nœud en insérant le nouveau nœud juste à côté de celui qui possède la même clef (en dessus ou en dessous), soit on descend quand même jusqu'à une feuille dans le sous-arbre droit du nœud.

Encore une fois, l'insertion peut se programmer soit de façon récursive, soit de façon itérative. Dans les deux cas, on choisit d'insérer un nœud déjà présent dans l'arbre comme s'il n'était pas présent : on l'insère comme fils d'une feuille (troisième possibilité dans la FIGURE 5.5).

```
1  void ABR_insert(int v, tree* A) {
2    if ((*A) == NULL) {
3      node* n = (node*) malloc(sizeof(node));
4      n->key = v;
5      n->left = NULL;
6      n->right = NULL;
7      (*A) = n;
8      return;
9    }
10   if (v < (*A)->key) {
11     ABR_insert(v, &((*A)->left));
12   } else {
13     ABR_insert(v, &((*A)->right));
14   }
15 }
```

Notons que afin de pouvoir modifier l'arbre `A` (il n'est nécessaire de le modifier que quand l'arbre est initialement vide) il est nécessaire de le passer en argument par pointeur. De ce fait, les appels récursifs utilisent une écriture un peu lourde `&((*A)->left)` :

– on commence par prendre l'arbre `*A`.

– on regarde son fils gauche `(*A)->left`
– et on veut pouvoir modifier ce fils gauche : on récupère donc le pointeur vers le fils gauche `&((*A)->left)` afin de pouvoir faire pointer ce pointeur vers un nouveau nœud.

```
1  void ABR_insert_iter(int v, tree* A) {
2    node** cur = A;
3    while ((*cur) != NULL) {
4      if (v < (*cur)->key) {
5        cur = &((*cur)->left);
6      } else {
7        cur = &((*cur)->right);
8      }
9    }
10   node* n = (node*) malloc(sizeof(node));
11   n->key = v;
12   n->left = NULL;
13   n->right = NULL;
14   (*cur) = n;
15   return;
16 }
```

Dans cette version itérative, on retrouve l'écriture un peu lourde de la version récursive avec `cur = &((*cur)->left)`. On pourrait être tenté de remplacer cette ligne par la ligne `(*cur) = (*cur)->left`, mais cela ne ferait pas ce qu'il faut ! On peut voir la différence entre ces deux commandes sur la FIGURE 5.6.

Suppression dans un ABR. La suppression de nœud est l'opération la plus délicate dans un arbre binaire de recherche. Nous avons vu qu'il est facile de trouver le nœud `n` à supprimer. En revanche, une fois le nœud `n` trouvé, si on le supprime cela va créer un trou dans l'arbre : il faut donc déplacer un autre nœud de l'arbre pour reboucher le trou. En pratique, on peut distinguer trois cas différents :

1. Si `n` ne possède aucun fils (`n` est une feuille), alors on peut le supprimer de l'arbre directement, *i.e.* on fait pointer son père vers `NULL` au lieu de `n`.
2. Si `n` ne possède qu'un seul fils : on supprime `n` de l'arbre et on fait pointer le père de `n` vers le fils de `n` (ce qui revient à remplacer `n` par son fils).
3. Si `n` possède deux fils, on commence par calculer le successeur de `n`, c'est-à-dire le nœud suivant `n` lorsque l'on énumère les nœuds avec un parcours infixe de l'arbre : le successeur est le nœud ayant la plus petite clef plus grande que la clef de `n`. Si on appelle `s` ce nœud, il est facile de voir que `s` sera le nœud le plus à gauche du sous-arbre droit de `n`. Ce nœud étant tout à gauche du sous-arbre, il a forcément son fils gauche vide. On peut alors supprimer le nœud `n` en le remplaçant par le nœud `s` (on remplace la clef et l'information contenue dans le nœud), et en supprimant le nœud `s` de son emplacement d'origine avec la méthode vue au cas 2 (ou au cas 1 si `s` possède deux fils vides). Une telle suppression est représentée sur la FIGURE 5.7.

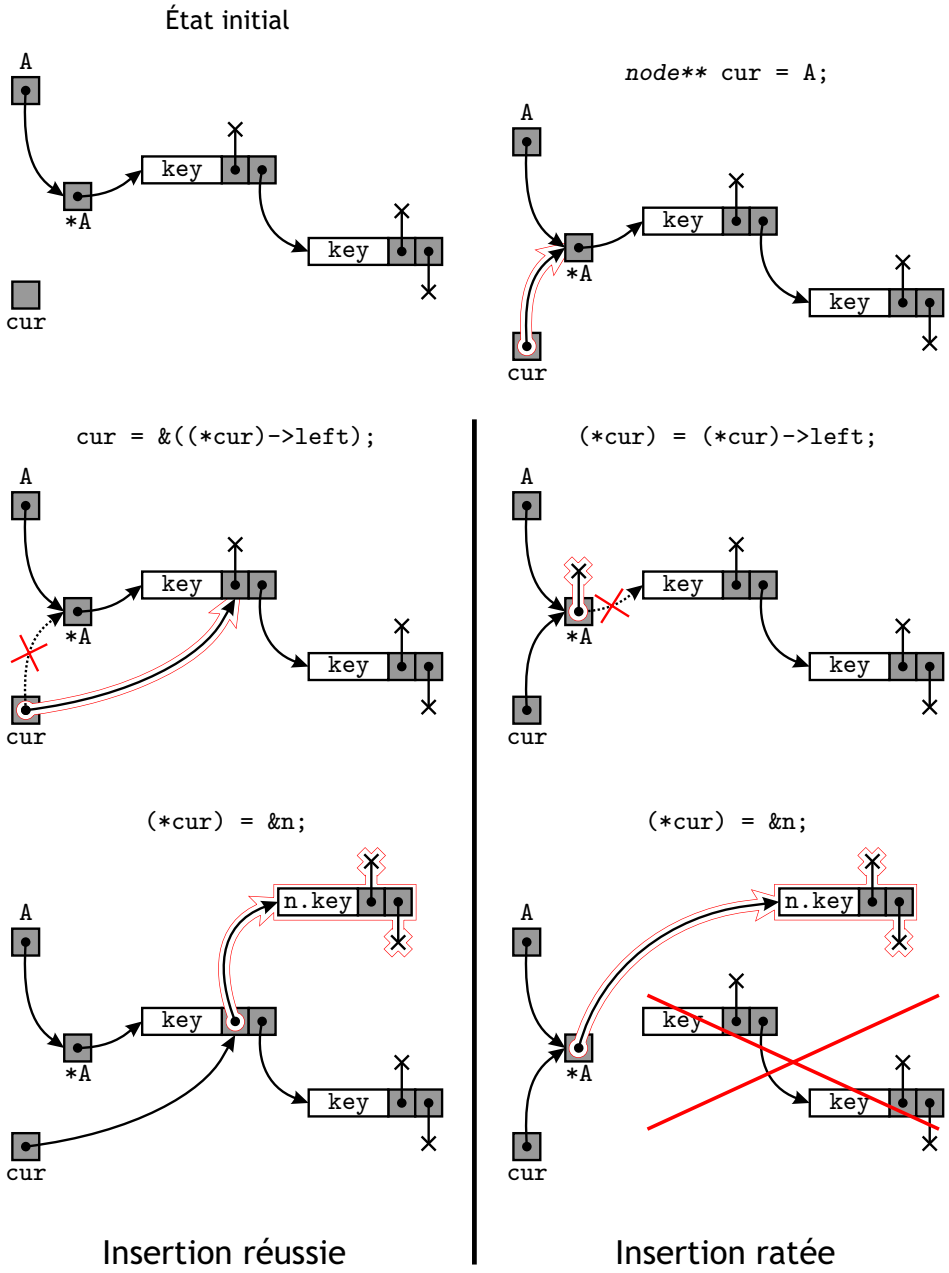


FIGURE 5.6 – *Ce qu'il se passe en mémoire lors de l'insertion d'un nœud* `n` *dans un ABR. On part d'un arbre ayant juste une racine et un fils droit et on doit insérer le nouveau nœud comme fils gauche de la racine. On initialise* `cur`*, puis on descend dans l'arbre d'une façon ou d'une autre : à gauche la bonne méthode qui marche, à droite la mauvaise. Dans les deux cas, on trouve un pointeur vers* `NULL` *et on insère le nouveau nœud* `n`*. Avec la méthode de droite l'arbre original est modifié et deux nœuds sont perdus.*

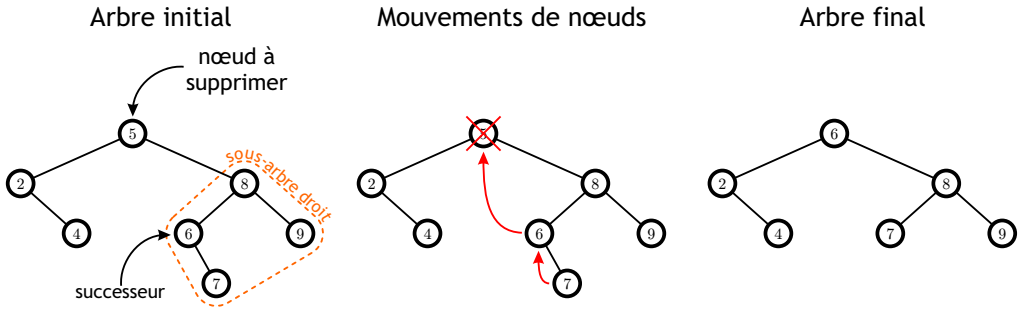


FIGURE 5.7 – *Suppression d'un nœud d'un ABR.*

En C, la version itérative de la suppression (la version récursive est plus compliquée et n'apporte pas grand chose) peut se programmer ainsi :

```
1  void ABR_del(int v, tree* A) {
2    node** cur = A;
3    node** s;
4    node* tmp;
5    while ((*cur) != NULL) {
6      if (v < (*cur)->key) {
7        cur = &((*cur)->left);
8      } else if (v > (*n)->key){
9        cur = &((*cur)->right);
10     } else {
11       /* on a trouvé le noeud à supprimer, on cherche son successeur */
12       s = &((*cur)->right);
13       while ((*s)->left != NULL) {
14         s = &((*s)->left);
15       }
16       /* on a trouvé le successeur */
17       tmp = (*s); /* on garde un lien vers le successeur */
18       (*s) = tmp->right;
19       tmp->left = (*cur)->left;
20       tmp->right = (*cur)->right;
21       (*cur) = tmp;
22       return;
23     }
24   }
25   printf("Noeud introuvable. Suppression impossible.");
26   return;
27 }
```

Complexité des différentes opérations. Soit n le nombre de nœuds de l'arbre, appelé aussi *taille* de l'arbre, et h la hauteur de l'arbre. Les algorithmes ci-dessus permettent de se convaincre aisément du fait que la complexité des opérations de recherche, d'insertion

et de suppression, sont en $\Theta(h)$: pour les version récursives, chaque appel fait descendre d'un niveau dans l'arbre et pour les version itératives chaque étape de la boucle `while` fait aussi descendre d'un niveau. Pour calculer la complexité, il faut donc connaître la valeur de h.

Il existe des arbres de taille n et de hauteur $n-1$: ce sont les arbres dont tous les nœuds ont au plus un fils non vide. Ce type d'arbre est alors équivalent à une liste, et les opérations ci-dessus ont donc une complexité analogue à celle de la recherche séquentielle (*i.e.* en $\Theta(h) = \Theta(n)$). Noter que l'on obtient ce genre de configuration lorsque l'on doit insérer n clefs dans l'ordre (croissant ou décroissant), ou bien par exemple les lettres A,Z,B,Y,C,X,... dans cet ordre.

En revanche, dans le cas le plus favorable, *i.e.* celui d'un arbre complet où tous les niveaux sont remplis, l'arbre a une hauteur environ égale [4] à $\log_2(n)$. Dans ce cas, les trois opérations ci-dessus ont alors une complexité en $\Theta(h) = \Theta(\log_2(n))$.

La question est maintenant de savoir ce qu'il se passe en moyenne. Si les clefs sont insérées de manière aléatoire, on a le résultat suivant :

Proposition 5.2.1. *La hauteur moyenne d'un arbre binaire de recherche construit aléatoirement à partir de n clefs distinctes est en $\Theta(\log_2(n))$.*

Ainsi, on a un comportement en moyenne qui est logarithmique en le nombre de nœuds de l'arbre (et donc proche du cas le meilleur), ce qui rend les ABR bien adaptés pour l'implémentation de dictionnaires. En outre, afin de ne pas « tomber » dans le pire cas, on dispose de méthodes de rééquilibrage d'arbres pour rester le plus près possible de la configuration d'arbre complet (*cf.* section 5.3).

Tri par Arbre Binaire de Recherche

Comme nous l'avons vu, le parcours infixe d'un ABR affiche les clefs de l'arbre dans l'ordre croissant. Ainsi, une méthode de tri se déduit naturellement de cette structure de données :
- *Soient n entiers à trier*
- *Construire l'ABR dont les nœuds ont pour clefs ces entiers*
- *Imprimer un parcours infixe de cet ABR.*

La complexité de cette méthode de tri est en $\Theta(n\log_2(n))$ ($\Theta(n\log_2(n))$ pour l'insertion de n clefs, chaque insertion se faisant en $\Theta(\log_2(n))$, plus $\Theta(n)$ pour le parcours infixe) en moyenne et dans le cas le plus favorable. Elle est en revanche en $\Theta(n^2)$ dans le pire cas (entiers dans l'ordre croissant ou décroissant).

5.2.3 Tas pour l'implémentation de files de priorité

Une file de priorité est un ensemble d'éléments, chacun muni d'un rang ou priorité (attribuée avant qu'il ne rentre dans la file). Un exemple fondamental de file est l'ordon-

4. On trouve dans ce cas 1 nœud de profondeur 0, 2 nœuds de profondeur 1, ..., 2^h nœuds de profondeur h et le nombre total de nœuds est alors $\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$.

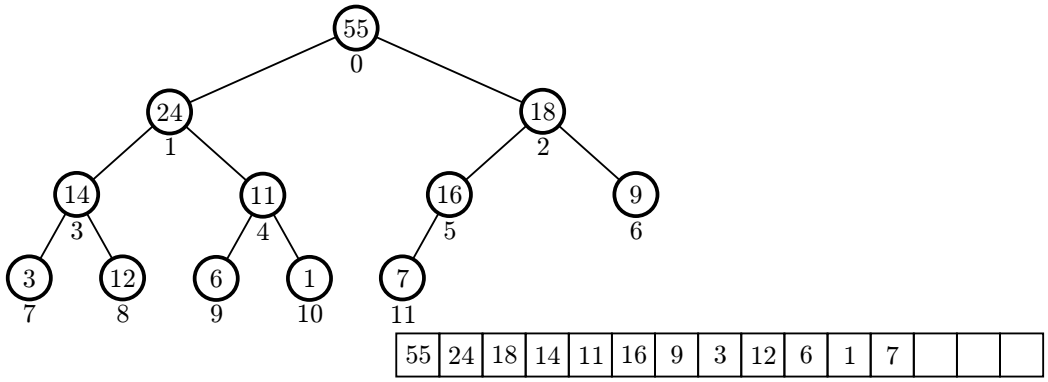


FIGURE 5.8 – *Représentation d'un tas à l'aide d'un tableau.*

nanceur d'un système d'exploitation qui doit exécuter en priorité les instructions venant d'un processus de priorité élevée. Les éléments sortiront de cette file précisément selon leur rang : l'élément de rang le plus élevé sortira en premier. Les primitives à implémenter pour mettre en oeuvre et manipuler des files de priorité sont :

- une fonction d'initialisation d'une file (création d'une file vide),
- une fonction d'insertion d'un élément dans la file (avec son rang),
- une fonction qui retourne l'élément de plus haut rang,
- une fonction de suppression de l'élément de plus haut rang.

Le codage d'une file de priorité peut être réalisé au moyen d'une liste : alors l'insertion se fait en temps constant, mais l'opération qui retourne ou supprime l'élément de plus haut rang nécessite une recherche préalable de celui-ci, et se fait donc en $\Theta(n)$, où n est le nombre d'éléments de la file. Autrement, on peut aussi choisir d'avoir une opération d'insertion plus lente qui insère directement l'élément à la bonne position en fonction de sa priorité (cette opération coûte alors $\Theta(n)$) et dans ce cas les opérations de recherche et suppression peuvent se faire en $\Theta(1)$.

Une méthode beaucoup plus efficace pour représenter une file de priorité est obtenue au moyen d'un arbre binaire de structure particulière, appelé *tas* (*heap* en anglais) ou (arbre) maximier. Un tas est un arbre binaire *complet*, qui possède en plus la propriété que *la clef de tout nœud est supérieure ou égale à celle de ses descendants*. Un exemple de tas est donné à la FIGURE 5.8.

La façon la plus naturelle d'implémenter un tas semble être d'utiliser une structure d'arbre similaire à celle utilisées pour les ABR. Pourtant l'implémentation la plus efficace utilise en fait un tableau. On commence par numéroter les nœuds de 0 (racine) à n-1 (nœud le plus à droite du dernier niveau) par un parcours par niveau. On place chaque nœud dans la case du tableau `tab` correspondant à son numéro (*cf.* FIGURE 5.8) : la racine dans `tab[0]`, le fils gauche de la racine dans `tab[1]`... Avec cette numérotation le père du nœud i est le nœud $\lfloor \frac{i-1}{2} \rfloor$, et les fils du nœud i sont les nœuds $2i+1$ (fils gauche) et $2i+2$ (fils droit). La propriété de maximier (ordre sur les clefs) se traduit sur le tableau par :

$\forall 1 \leq i \leq n-1$, `tab[`$i$`]` $\leq$ `tab[`$\lfloor \frac{i-1}{2} \rfloor$`]`.

Détaillons à présent les opérations de base (celles listées ci-dessus) sur les tas représentés par des tableaux.

Initialisation. Pour coder un tas avec un tableau il faut au moins trois variables : le tableau `tab` où ranger les nœuds, le nombre maximum `max` de nœuds que l'on peut mettre dans ce tas, et le nombre de nœuds déjà présents `n`. Le plus simple est de coder cela au moyen de la structure suivante :

```
1 struct heap {
2   int max;
3   int n;
4   int* tab;
5 };
```

Pour créer un tas (vide) de taille maximale m donnée en paramètre, on crée un objet de type tas pour lequel `tab` est un tableau fraîchement alloué :

```
1 heap* init_heap(int m) {
2   heap* h = (heap*) malloc(sizeof(heap));
3   h->max = m;
4   h->n = 0;
5   h->tab =  (int) malloc(m*sizeof(int));
6   return h;
7 }
```

Nous voyons alors apparaître l'inconvénient de l'utilisation d'un tableau : le nombre maximum d'éléments à mettre dans le tas doit être défini à l'avance, dès l'initialisation. Cette limitation peut toutefois être contournée en utilisant une réallocation dynamique du tableau : on peut dans ce cas ajouter un niveau au tableau à chaque fois que le niveau précédent est plein. On réalloue alors un nouveau tableau (qui sera de taille double) et on peut recopier le tableau précédent au début de celui là. Comme vu dans la section sur les tableau, la taille étant doublée à chaque réallocation, cela n'affecte pas la complexité moyenne d'une insertion d'élément.

Insertion. L'opération d'insertion comme celle de suppression comporte deux phases : la première vise à insérer (resp. supprimer) la clef considérée (resp. la clef de la racine), tout en préservant la propriété de complétude de l'arbre, la deuxième est destinée à rétablir la propriété de maximier, *i.e.* l'ordre sur les clefs des nœuds de l'arbre.

Pour insérer un nœud de clef v, on crée un nouveau nœud dans le niveau de profondeur le plus élevé de l'arbre, et le plus à gauche possible [5]. Ensuite, on opère une « ascension » dans l'arbre (plus précisément dans la branche reliant le nouveau nœud à la racine) de façon à

5. Si l'arbre est complètement rempli sur le dernier niveau, on crée un niveau supplémentaire.

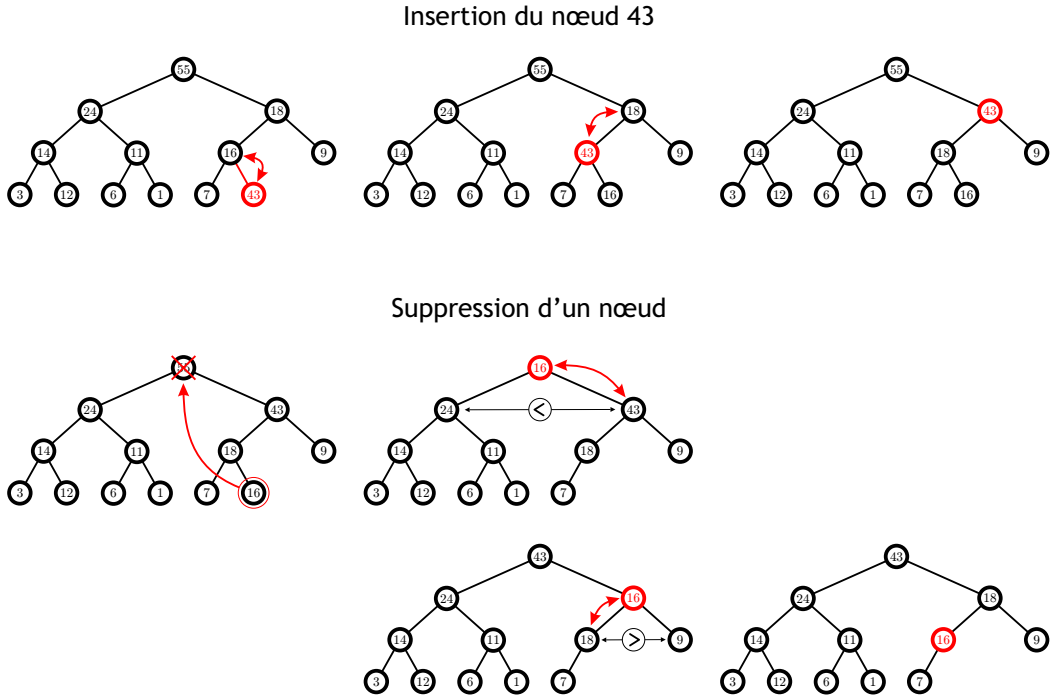


FIGURE 5.9 – *Insertion et suppression d'un nœud d'un tas.*

placer ce nouveau nœud au bon endroit : pour cela, on compare v avec la clef de son nœud père. Si v est supérieure à celle-ci, on permute ces deux clefs. On recommence jusqu'à ce que v soit inférieure ou égale à la clef de son nœud père ou que v soit à la racine de l'arbre (en cas 0 de `tab`). Les étapes d'une insertion sont représentées sur la FIGURE 5.9.

```
1  void heap_insert(int v, heap* h) {
2    if (h->n == h->max) {
3      printf("Tas plein.");
4      return;
5    }
6    int tmp;
7    int i = h->n;
8    h->tab[i] = v; /* on insère v à la fin de tab */
9    while ((i>0) && (h->tab[i] > h->tab[(i-1)/2])) {
10     /* tant que l'ordre n'est pas bon et
11        que la racine n'est pas atteinte */
12     tmp = h->tab[i];
13     h->tab[i] = h->tab[(i-1)/2];
14     h->tab[(i-1)/2] = tmp;
15     i=(i-1)/2;
16   }
17   h->n++;
18 }
```

Suppression. Pour supprimer la clef contenue dans la racine, on remplace celle-ci par la clef (notée v) du nœud situé au niveau de profondeur le plus élevé et le plus à droite possible (le dernier nœud du tableau), nœud que l'on supprime alors aisément étant donné qu'il n'a pas de fils. Ensuite, pour rétablir l'ordre sur les clefs de l'arbre, *i.e.* mettre v à la bonne place, on effectue une « descente » dans l'arbre : on compare v aux clefs des fils gauche et droit de la racine et si v est inférieure ou égale à au moins une de ces deux clefs, on permute v avec la plus grande des deux clefs. On recommence cette opération jusqu'à ce que v soit supérieure ou égale à la clef de chacun de ses nœuds fils, ou jusqu'à arriver à une feuille. Les étapes d'une suppression sont représentées sur la FIGURE 5.9.

Voici comment peut se programmer l'opération de suppression qui renvoie au passage l'élément le plus grand du tas que l'on vient de supprimer.

```
1  int heap_del(heap* h) {
2    if (h->n == 0) {
3      printf("Erreur : le tas est vide.");
4      return -1;
5    }
6    /* on sauvegarde le max pour le retourner à la fin */
7    int max = h->tab[0];
8    int i = 0;
9    bool cont = true;
10   h->n--;
11   /* on met la clef du dernier noeud à la racine */
12   h->tab[0] = h->tab[h->n];
13   while (cont) {
14     if (2*i+2 > h->n) {
15       /* si le noeud i n'a pas de fils */
16       cont = false;
17     } else if (2*i+2 == h->n) {
18       /* si le noeud i a un seul fils (gauche)
19          on inverse les deux si nécessaire */
20       if (h->tab[i] < h->tab[2*i+1]) {
21         swap(&(h->tab[i]),&(h->tab[2*i+1]));
22       }
23       cont = false;
24     } else {
25       /* si le noeud i a deux fils
26          on regarde si l'un des deux est plus grand */
27       if ((h->tab[i] < h->tab[2*i+1]) ||
28                          (h->tab[i] < h->tab[2*i+2])) {
29         /* on cherche le fils le plus grand */
30         int greatest;
31         if (h->tab[2*i+1] > h->tab[2*i+2]) {
32           greatest = 2*i+1;
33         } else {
34           greatest = 2*i+2;
35         }
36         /* on inverse et on continue la boucle */
37         swap(&(h->tab[i]),&(h->tab[greatest]));
```

```
38          i = greatest;
39        } else {
40          cont = false;
41        }
42      }
43    }
44    return max;
45  }
46
47  void swap(int* a, int* b) {
48    int tmp = (*a);
49    (*a)=(*b);
50    (*b)=tmp;
51  }
```

La fonction `swap` permet d'échanger deux cases du tableau (ou deux entiers situés n'importe où dans la mémoire). On est obligé de passer en arguments des pointeurs vers les entiers à échanger car si l'on passe directement les entiers, ils seront recopiés dans des variables locales de la fonction, et ce sont ces variable locale qui seront échangées, les entiers d'origine restant bien tranquillement à leur place.

Complexité des opérations sur les tas. Comme souvent dans les arbres, la complexité des différentes opération dépend essentiellement de la hauteur totale de l'arbre. Ici, avec les tas, nous avons de plus des arbres complets et donc la hauteur d'un tas et toujours logarithmique en son nombre de nœuds : $h = \Theta(\log(n))$.

Les boucles `while` des opérations d'insertion ou de suppression sont exécutées au maximum un nombre de fois égal à la hauteur du tas. Chacune de ces exécution contenant un nombre constant d'opération la complexité total des opérations d'insertion et de suppression est donc $\Theta(h) = \Theta(\log(n))$.

5.2.4 Tri par tas

La représentation précédente d'un ensemble d'entiers (clefs) donne lieu de manière naturelle à un algorithme de tri appelé tri pas tas (*heapsort* en anglais). Soient n entiers à trier, on les insère dans un tas, puis on répète n fois l'opération de suppression. La complexité de ce tri est $n \times \Theta(\log(n))$ pour les n insertions de clefs plus $n \times \Theta(\log(n))$ pour les n suppressions, soit $\Theta(n \log(n))$ au total. Il est à noter que, de par la définition des opérations d'insertion/suppression, cette complexité ne dépend pas de la distribution initiale des entiers à trier, *i.e.* elle est la même en moyenne ou dans le pire cas. Voici une implémentation d'un tel tri : on part d'un tableau `tab` que l'on re-remplit avec les entiers triés (par ordre croissant).

```
1  void heapsort(int* tab, int n) {
2    int i;
3    heap* h = init_heap(n);
```

```
4   for (i=0; i<n; i++) {
5     heap_insert(tab[i],h);
6   }
7   for (i=n-1; i>=0; i--) {
8     tab[i] = heap_del(h);
9   }
10  free(h->tab);
11  free(h);
12 }
```

5.3 Arbres équilibrés

Les algorithmes sur les arbres binaires de recherche donnent de bons résultats dans le cas moyen, mais ils ont de mauvaises performances dans le pire cas. Par exemple, lorsque les données sont déjà triées, ou en ordre inverse, ou contiennent alternativement des grandes et des petites clefs, le tri par ABR a un très mauvais comportement.

Avec le tri rapide, le seul remède envisageable était le choix aléatoire d'un élément pivot, dont on espérait qu'il éliminerait le pire cas. Heureusement pour la recherche par ABR, il est possible de faire beaucoup mieux, car il existe des techniques générales d'*équilibrage* d'arbres permettant de *garantir que le pire cas n'arrive jamais*.

Ces opérations de transformation d'arbres sont plus ou moins simples, mais sont peu coûteuses en temps. Elles permettent de rendre l'arbre le plus régulier possible, dans un sens qui est mesuré par un paramètre dépendant en général de sa hauteur. Une famille d'arbres satisfaisant une telle condition de régularité est appelée une famille d'arbres *équilibrés*. Il existe plusieurs familles d'arbres équilibrés : les arbres AVL, les arbres rouge-noir, les arbres *a-b*... Nous verrons ici essentiellement les arbres AVL.

5.3.1 Rééquilibrage d'arbres

Nous présentons ici une opération d'équilibrage appelée *rotation*, qui s'applique à tous les arbres binaires. Soit donc A, un arbre binaire non vide. On écrira $A = (x, Z, T)$ pour exprimer que x est la racine de A, et Z et T ses sous-arbres gauche et droit respectivement.

Soit $A = (x, X, B)$ avec B non vide, et posons $B = (y, Y, Z)$. L'opération de *rotation gauche* de A est l'opération :

$$A = (x, X, (y, Y, Z)) \longrightarrow \mathcal{G}(A) = (y, (x, X, Y), Z).$$

Autrement dit, on remplace le nœud racine x par le nœud y, le fils gauche du nœud y pointe sur le nœud x, son fils droit (inchangé) pointe sur le sous-arbre Z, et le fils droit du nœud x est mis à pointer sur le sous-arbre Y. Cette opération est représentée sur la Figure 5.10.

La rotation droite est l'opération inverse :

$$A = (y, (x, X, Y), Z) \longrightarrow \mathcal{D}(A) = (x, X, (y, Y, Z)).$$

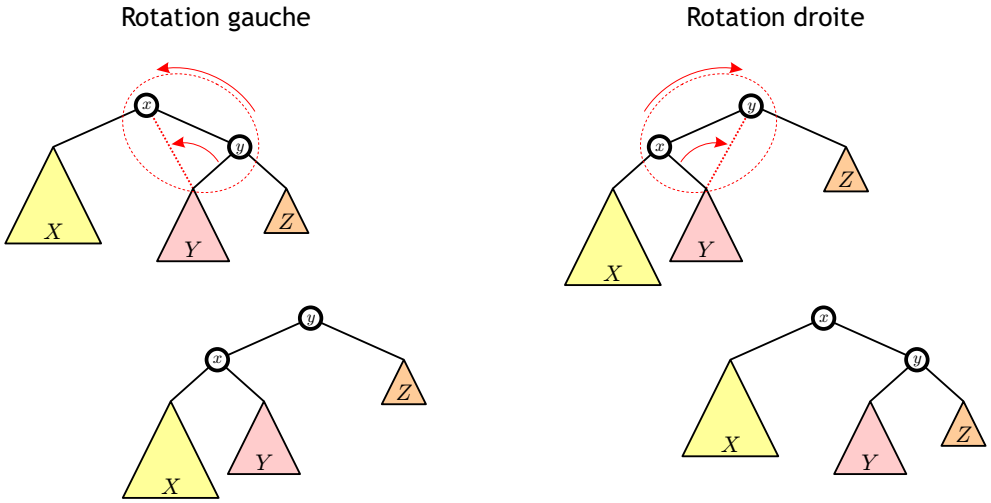


FIGURE 5.10 – *Opération de rotation pour le rééquilibrage d'arbre binaire.*

On remplace le nœud racine y par le nœud x, le fils gauche du nœud x (inchangé) pointe sur le sous-arbre X, son fils droit pointe sur le nœud y, et le fils gauche du nœud y est mis à pointer sur le sous-arbre Y.

Utilisation des rotations. Le but des rotations est de pouvoir rééquilibrer un ABR. On opère donc une rotation gauche lorsque l'arbre est « déséquilibré à droite », *i.e.* son sous-arbre droit est plus haut que son sous-arbre gauche. On opère une rotation droite dans le cas contraire. Il est aisé de vérifier que les rotations préservent la condition sur l'ordre des clefs d'un ABR. On a alors :

Proposition 5.3.1. *Si A est un ABR, et si la rotation gauche (resp. droite) est définie sur A, alors $\mathcal{G}(A)$ (resp. $\mathcal{D}(A)$) est encore un ABR.*

On peut également définir des doubles rotations (illustrées sur la FIGURE 5.11) comme suit : la rotation *gauche-droite* associe à l'arbre $A = (x, A_g, A_d)$, l'arbre $\mathcal{D}(x, \mathcal{G}(A_g), A_d)$. De manière analogue, la rotation *droite-gauche* associe à l'arbre $A = (x, A_g, A_d)$, l'arbre $\mathcal{G}(x, A_g, \mathcal{D}(A_d))$. Ces opérations préservent également l'ordre des ABR. Une propriété importante des rotations et doubles rotations est qu'elles s'implémentent *en temps constant* : en effet, lorsqu'un ABR est représenté par une structure de données du type `binary_tree` définie plus haut dans ce chapitre, une rotation consiste essentiellement en la mise à jour d'un nombre fixé (*i.e.* indépendant du nombre de nœuds de l'arbre ou de sa hauteur) de pointeurs.

5.3.2 Arbres AVL

Les arbres AVL ont été introduits par Adelson, Velskii et Landis en 1962. Ils constituent une famille d'ABR équilibrés en hauteur.

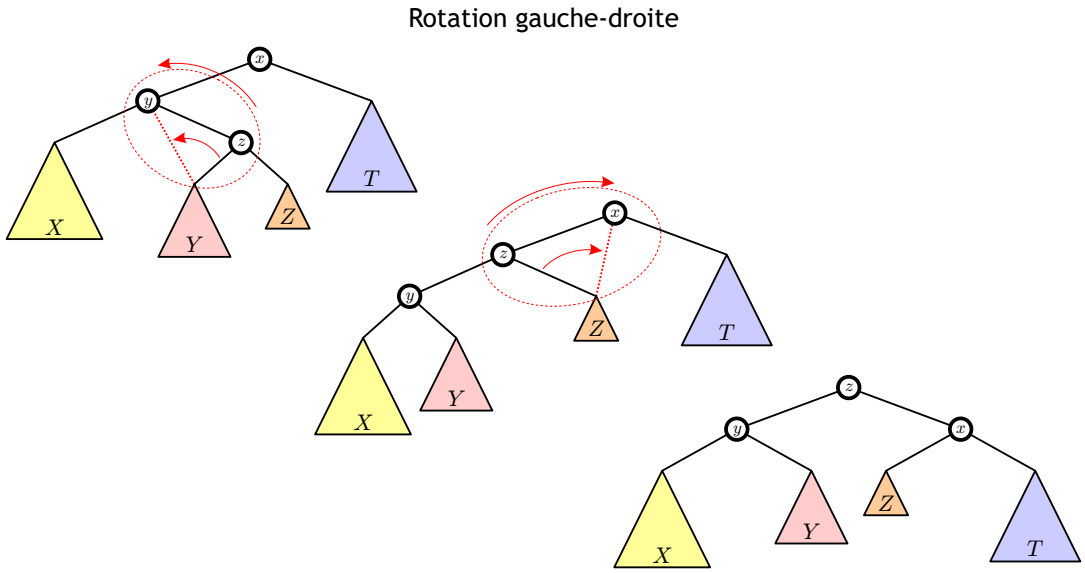


FIGURE 5.11 – *Exemple de double rotation sur un ABR.*

De manière informelle, un ABR est un arbre AVL si, pour tout nœud de l'arbre, les hauteurs de ses sous-arbres gauche et droit diffèrent d'au plus 1. Plus précisément, posons $\delta(A) = 0$ si A est l'arbre vide, et dans le cas général :

$$\delta(A) = h(A_g) - h(A_d)$$

où A_g et A_d sont les sous-arbres gauche et droit de A, et $h(A_g)$ la hauteur de l'arbre A_g (par convention, la hauteur de l'arbre vide est -1 et la hauteur d'une feuille est 0 et tous deux sont des AVL).

$\delta(A)$ est appelé l'équilibre de A. Pour plus de simplicité, la notation $\delta(x)$ où x est un nœud désignera l'équilibre de l'arbre dont x est la racine. Une définition plus rigoureuse d'un arbre AVL est alors :

Définition 5.1. *Un ABR est un arbre AVL si, pour tout nœud x de l'arbre, $\delta(x) \in \{-1, 0, 1\}$.*

La propriété fondamentale des AVL est que l'on peut borner leur hauteur en fonction du log du nombre de nœuds dans l'arbre. Plus précisément :

Proposition 5.3.2. *Soit A un arbre AVL possédant n sommets et de hauteur h. Alors*

$$\log_2(1+n) \leq 1 + h \leq c \log_2(2+n),$$

avec $c = 1/\log_2((1+\sqrt{5})/2) \leq 1,44$.

Preuve. Pour une hauteur h donnée, l'arbre possédant le plus de nœuds est l'arbre complet, à $2^{h+1} - 1$ nœuds. Donc $n \leq 2^{h+1} - 1$ et $\log_2(1+n) \leq 1 + h$.

Soit maintenant $N(h)$, le nombre minimum de nœuds d'un arbre AVL de hauteur h. On a $N(-1) = 0$, $N(0) = 1$, $N(1) = 2$, et, pour $h \geq 2$,

$$N(h) = 1 + N(h-1) + N(h-2).$$

En effet, si l'arbre est de hauteur h, l'un (au moins) de ses sous arbre est de hauteur $h-1$. Plus le deuxième sous arbre est petit, moins il aura de nœud, mais la propriété d'AVL fait qu'il ne peut pas être de hauteur plus petite que $h-2$. Donc l'arbre AVL de hauteur h contenant le moins de nœuds est constitué d'un nœud racine et de deux sous-arbres AVL de nombre de nœuds minimum, l'un de hauteur $h-1$, l'autre de hauteur $h-2$.

Posons alors $F(h) = N(h) + 1$. On a $F(0) = 2$, $F(1) = 3$, et, pour $h \geq 2$,

$$F(h) = F(h-1) + F(h-2),$$

donc $F(h) = \mathcal{F}_{h+3}$, où $\mathcal{F}_k$ est le k-ième nombre de Fibonacci. Pour tout arbre AVL à n sommets et de hauteur h, on a par conséquent :

$$n + 1 \geq F(h) = \frac{1}{\sqrt{5}}(\phi^{h+3} - (1-\phi)^{h+3}) > \frac{1}{\sqrt{5}}\phi^{h+3} - 1,$$

avec $\phi = (1+\sqrt{5})/2$. D'où

$$h + 3 < \frac{\log_2(n+2)}{\log_2(\phi)} + \log_\phi(\sqrt{5}) \leq \frac{1}{\log_2(\phi)}\log_2(n+2) + 2.$$

□

Par exemple, un arbre AVL à 100 000 nœuds a une hauteur comprise entre 17 et 25. Le nombre de comparaisons nécessaires à une recherche, insertion ou suppression dans un tel arbre sera alors de cet ordre.

ARBRES DE FIBONACCI

La borne supérieure de la proposition précédente est essentiellement atteinte pour les arbres de Fibonacci *définis comme suit :* $\Phi(0)$ *est l'arbre vide,* $\Phi(1)$ *est réduit à une feuille, et, pour* $k \geq 2$*, l'arbre* Φ_{k+2} *a un sous-arbre gauche égal à* Φ_{k+1}*, et un sous-arbre doit égal à* Φ_k*. La hauteur de* Φ_k *est* $k-1$*, et* Φ_k *possède* $\mathcal{F}_{k+2}$ *nœuds.*

L'implémentation des AVL est analogue à celle des ABR, à ceci près que l'on rajoute à la structure `binary_tree` un champ qui contient la hauteur de l'arbre dont la racine est le nœud courant. Cette modification rend cependant le opération d'insertion et de suppression un peu plus compliquées : il est nécessaire de remettre à jour ces hauteurs chaque fois que cela est nécessaire.

Insertion dans un AVL. L'opération d'insertion dans un AVL se fait de la même manière que dans un ABR : on descend dans l'arbre à partir de la racine pour rechercher la feuille où mettre le nouveau nœud. Ensuite il suffit de remonter dans l'arbre pour remettre

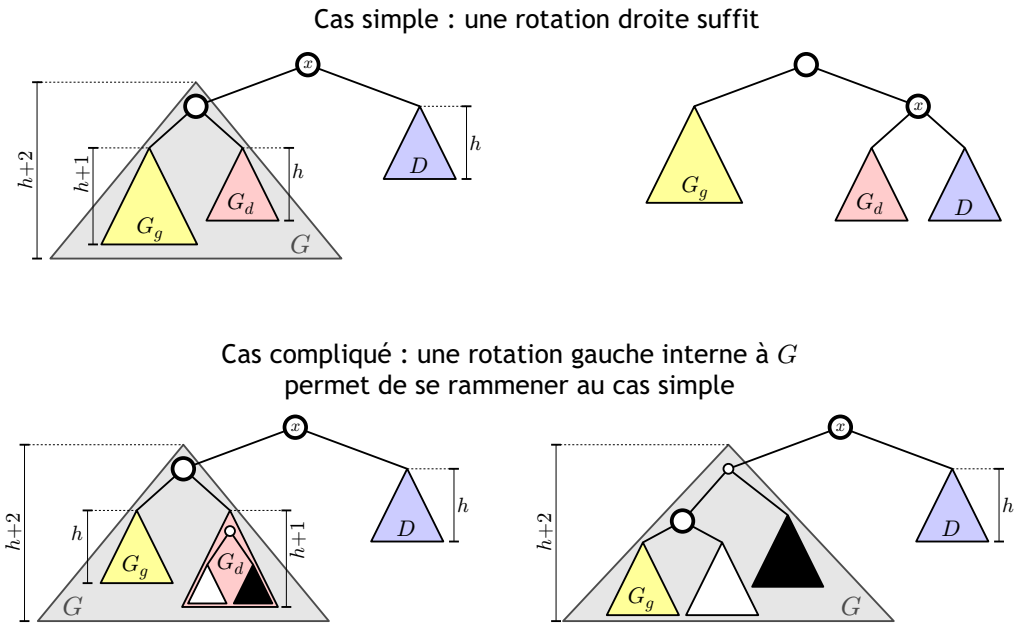


FIGURE 5.12 – *Rééquilibrage après insertion dans un AVL.*

à jour les hauteurs de tous les sous-arbres (dans une version récursive de l'insertion, cela se fait aisément en abandonnant la récursion terminale et en remettant à jour les hauteur juste après l'appel récursif pour l'insertion).

Toutefois, cette opération peut déséquilibrer l'arbre, *i.e.* l'arbre résultant n'est plus AVL. Pour rétablir la propriété sur les hauteurs, il suffit de rééquilibrer l'arbre par des rotations (ou doubles rotations) le long du chemin qui mène de la racine à la feuille où s'est fait l'insertion. Dans la pratique, cela se fait juste après avoir remis à jour les hauteurs des sous-arbres : si on constate un déséquilibre entre les deux fils de -2 ou 2 on effectue une rotation, ou une double rotation. Supposons que le nœud x ait deux fils G et D et qu'après insertion $\delta(x) = h(G) - h(D) = 2$. Le sous-arbre G est donc plus haut que D. Pour savoir si l'on doit faire un double rotation ou une rotation simple en x il faut regarder les hauteurs des deux sous-arbres de G (notés G_g et G_d). Si $\delta(G) > -1$ alors on fait une rotation droite sur x qui suffit à rééquilibrer l'arbre. Si $\delta(G) = h(G_g) - h(G_d) = -1$ alors il faut faire une double rotation : on commence par une rotation gauche sur G afin que $\delta(G) > -1$, puis on est ramené au cas précédent et une rotation droite sur x suffit. Cette opération est illustrée sur la FIGURE 5.12.

Dans le cas où le sous-arbre D est le plus haut, il suffit de procéder de façon symétrique. Il est important de noter qu'après une telle rotation (ou double rotation), l'arbre qui était déséquilibré après l'insertion retrouve sa hauteur initiale. Donc la propriété d'AVL est nécessairement préserver pour les nœuds qui se trouvent plus haut dans l'arbre.

Proposition 5.3.3. *Après une insertion dans un arbre AVL, il suffit d'une seule rotation ou double rotation pour rééquilibrer l'arbre. L'opération d'insertion/rééquilibrage dans un AVL à n nœuds se réalise donc en $O(\log_2(n))$.*

Suppression dans un AVL. L'opération de suppression dans un AVL se fait de la même manière que dans un ABR : on descend dans l'arbre à partir de la racine pour rechercher le nœud contenant la clef à supprimer. S'il s'agit d'une feuille, on supprime celle-ci ; sinon, on remplace le nœud par son nœud successeur, et on supprime le successeur. Comme dans le cas de l'insertion, cette opération peut déséquilibrer l'arbre. Pour le rééquilibrer, on opère également des rotations ou doubles rotations le long du chemin qui mène de la racine à la feuille où s'est fait la suppression ; mais ici, le rééquilibrage peut nécessiter plusieurs rotations ou doubles rotations mais le nombre de rotations (simples ou doubles) nécessaires est au plus égal à la hauteur de l'arbre (on en fait au maximum une par niveau), et donc :

Proposition 5.3.4. *L'opération de suppression/rééquilibrage dans un AVL à n nœuds se réalise en $O(\log_2(n))$.*

Conclusion sur les AVL. Les AVL sont donc des arbres binaires de recherches vérifiant juste une propriété d'équilibrage supplémentaire. Le maintien de cette propriété n'augmente pas le coût des opérations de recherche, d'insertion ou de suppression dans l'arbre, mais permet en revanche de garantir que la hauteur de l'arbre AVL reste toujours logarithmique en son nombre de nœuds. Cette structure est donc un peu plus complexe à implémenter qu'un ABR classique mais n'offre que des avantages. D'autres structures d'arbres équilibrés permettent d'obtenir des résultats similaires, mais à chaque fois, le maintien de la propriété d'équilibrage rend les opération d'insertion/suppression un peu plus lourdes à implémenter.

Chapitre 6

Graphes

Nous nous intéressons ici essentiellement aux graphes orientés. Après un rappel de la terminologie de base associée aux graphes et des principales représentations de ceux-ci, nous présentons un algorithme testant l'existence de chemins, qui nous conduit à la notion de fermeture transitive. Nous nous intéressons ensuite aux parcours de graphes : le parcours en largeur est de nature itérative, et nous permettra d'introduire la notion d'arborescence des plus courts chemins ; le parcours en profondeur - essentiellement récursif - admet plusieurs applications, parmi lesquelles le tri topologique. Nous terminons par le calcul de chemins optimaux dans un graphe (algorithme de Aho-Hopcroft-Ullman).

6.1 Définitions et terminologie

Définition 6.1. *Un* graphe orienté $G = (S, A)$ *est la donnée d'un ensemble fini* S *de* sommets*, et d'un sous-ensemble* A *du produit* $S \times S$*, appelé ensemble des* arcs *de* G*.*

Un arc $a = (x, y)$ a pour *origine* le sommet x, et pour *extrémité* le sommet y. On note $org(a) = x$, $ext(a) = y$. Le sommet y est un *successeur* de x, x étant un *prédécesseur* de

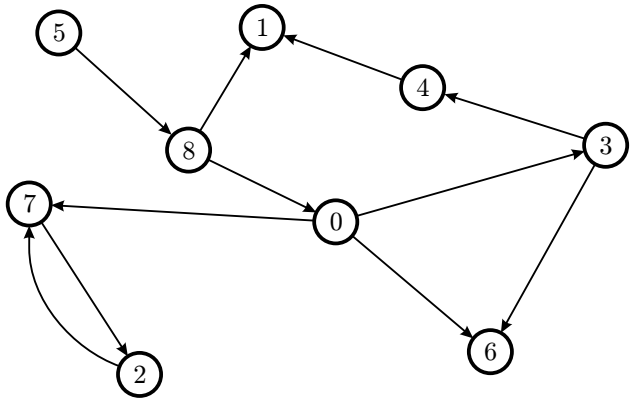


FIGURE 6.1 – *Exemple de graphe orienté.*

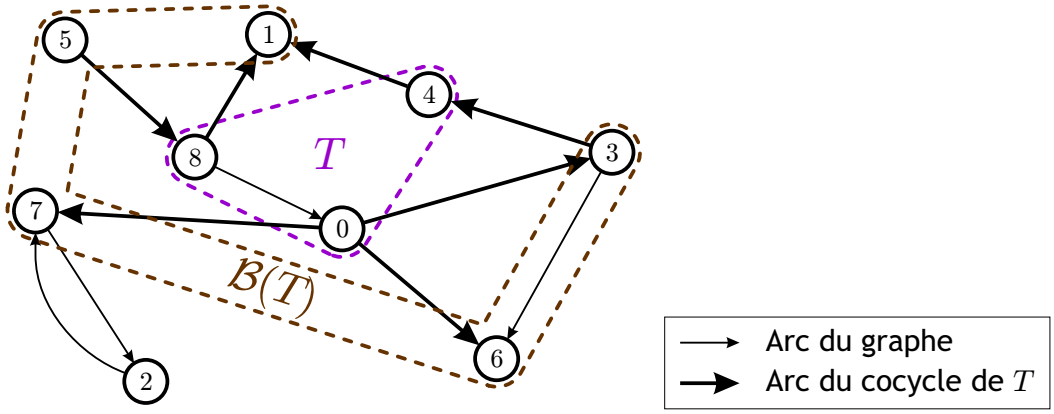


FIGURE 6.2 – *Bordure et cocycle d'un sous-ensemble T d'un graphe.*

y. Les sommets x et y sont dits *adjacents*. Si $x = y$, l'arc (x, x) est appelé *boucle*.

Un arc représente une liaison orientée entre son origine et son extrémité. Lorsque l'orientation des liaisons n'est pas une information pertinente, la notion de graphe *non orienté* permet de s'en affranchir. Un graphe non orienté $G = (S, A)$ est la donnée d'un ensemble fini S de sommets, et d'une famille de *paires* de S dont les éléments sont appelés *arêtes*. Étant donné un graphe orienté G, sa *version non orientée* est obtenue en supprimant les boucles, et en remplaçant chaque arc restant (x, y) par la paire $\{x, y\}$.

Soit $G = (S, A)$, un graphe orienté, et T, un sous-ensemble de S. L'ensemble

$$\omega(T) = \{a = (x, y) \in A,\ (x \in T, y \in S \setminus T) \text{ ou } (y \in T, x \in S \setminus T)\},$$

est appelé le *cocycle* associé à T. L'ensemble

$$\mathcal{B}(T) = \{x \in S \setminus T,\ \exists y \in T,\ (x, y) \text{ ou } (y, x) \in A\},$$

est appelé *bordure* de T (voir FIGURE 6.2). Si le sommet u appartient à $\mathcal{B}(T)$, on dit aussi que u est adjacent à T. Le graphe G est *biparti* s'il existe $T \subset S$, tel que $A = \omega(T)$.

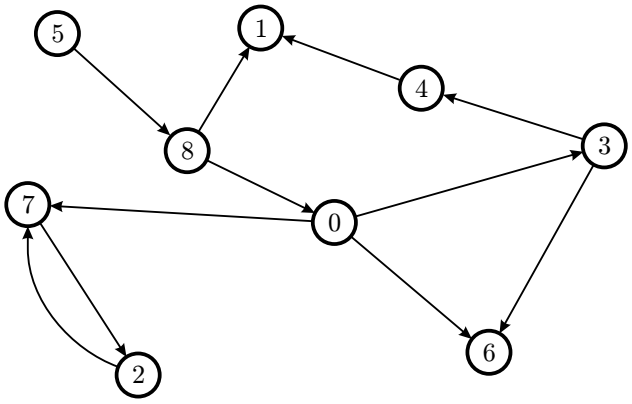
Définition 6.2. *Soit $G = (S, A)$, un graphe orienté. Un* chemin *f du graphe est une suite d'arcs $< a_1, \ldots, a_p >$, telle que*

$$org(a_{i+1}) = ext(a_i).$$

L'origine du chemin f, notée aussi $org(f)$, est celle de son premier arc a_1, et son extrémité, $ext(f)$, est celle de a_p. La longueur *du chemin est égale au nombre d'arcs qui le composent,* i.e. *p. Un chemin f tel que $org(f) = ext(f)$ est appelé un* circuit.

Dans un graphe non orienté la terminologie est parfois un peu différente : un chemin est appelé une *chaîne*, et un circuit un *cycle*. Un graphe sans cycle est un graphe *acyclique*.

Soit $G = (S, A)$, un graphe orienté, et soit x, un sommet de S. Un sommet y est un *ascendant* (resp. *descendant*) de x, s'il existe un chemin de y à x (resp. de x à y).



	0	1	2	3	4	5	6	7	8
0	0	0	0	1	0	0	1	1	0
1	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	1	0
3	0	0	0	0	1	0	1	0	0
4	0	1	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	1
6	0	0	0	0	0	0	0	0	0
7	0	0	1	0	0	0	0	0	0
8	1	1	0	0	0	0	0	0	0

FIGURE 6.3 – *Matrice d'adjacence associée à un graphe.*

Un graphe non orienté G est *connexe* si, pour tout couple de sommets, il existe une chaîne ayant ces deux sommets pour extrémités. Par extension, un graphe orienté est connexe si sa version non orientée est connexe. La relation définie sur l'ensemble des sommets d'un graphe non orienté par $x \simeq y$ s'il existe une chaîne reliant x à y, est une relation d'équivalence dont les classes sont appelées *composantes connexes* du graphe. Cette notion est très similaire à la notion de composantes connexe en topologie : on regarde les composantes connexe du dessin du graphe.

Soit $G = (S, A)$, un graphe orienté. Notons $\equiv_G$, la relation d'équivalence définie sur S par $x \equiv_G y$ si $x = y$ ou s'il existe un chemin joignant x à y et un chemin joignant y à x. Les classes d'équivalence pour cette relation sont appelées *composantes fortement connexes* de G.

6.2 Représentation des graphes

Il existe essentiellement deux façons de représenter un graphe. Dans la suite, $G = (S, A)$ désignera un graphe orienté.

6.2.1 Matrice d'adjacence

Dans cette représentation, on commence par numéroter de façon arbitraire les sommets du graphe : $S = \{x_1, \ldots, x_n\}$. On définit ensuite une matrice carrée M, d'ordre n par :

$$M_{i,j} = \begin{cases} 1 & \text{si } (x_i, x_j) \in A \\ 0 & \text{sinon.} \end{cases}$$

Autrement dit, $M_{i,j}$ vaut 1 si, et seulement si, il existe un arc d'origine x_i et d'extrémité x_j (voir FIGURE 6.3). M est la *matrice d'adjacence* de G. Bien entendu, on peut également représenter un graphe non orienté à l'aide de cette structure : la matrice M sera alors symétrique, avec des 0 sur la diagonale.

La structure de donnée correspondante - ainsi que son initialisation - est :

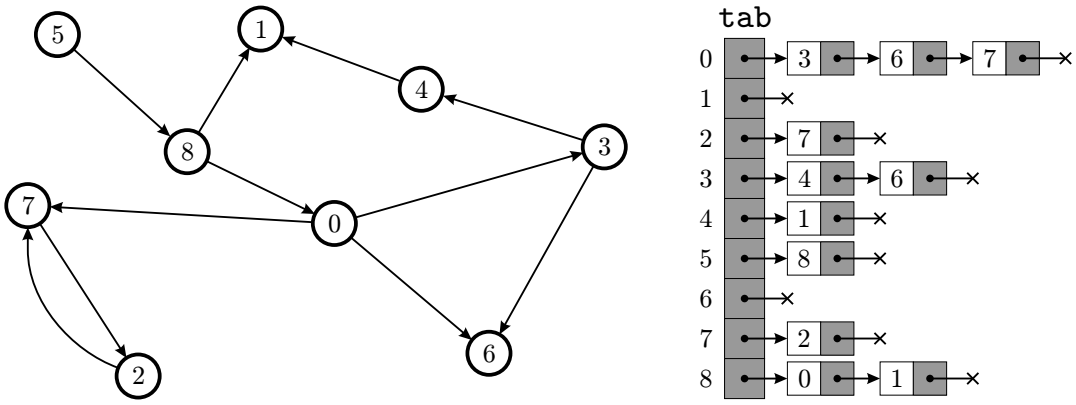


FIGURE 6.4 – *Représentation d'un graphe par liste de successeurs.*

```
1  struct graph_mat {
2    int n;
3    int** mat;
4  };
5
6  void graph_init(int n, graph_mat* G){
7    int i,j;
8    G->n = n;
9    G->mat = (int**) malloc(n*sizeof(int*));
10   for (i=0; i<n ; i++) {
11     G->mat[i] = (int*) malloc(n*sizeof(int));
12     for (j=0; j<n ; j++) {
13       G->mat[i][j] = 0;
14     }
15   }
16 }
```

Le coût en espace d'un tel codage de la matrice est clairement en $\Theta(n^2)$. Ce codage devient donc inutilisable dès que n dépasse quelques centaines de milliers. En outre, lorsque le graphe est peu dense (*i.e.* le rapport $|A|/|S|^2$ est petit et donc la matrice M est creuse) il est trop coûteux. Cependant, la matrice d'adjacence possède de bonnes propriétés algébriques qui nous serviront dans l'étude de l'existence de chemins (*cf.* section 6.3).

6.2.2 Liste de successeurs

Une autre façon de coder la matrice d'adjacence, particulièrement adaptée dans le cas d'un graphe peu dense, consiste à opter pour une représentation creuse de la matrice au moyen de listes chaînées : cette représentation est appelée *liste de successeurs* (*adjacency list* en anglais) : chaque ligne i de la matrice est codée par la liste chaînée dont chaque cellule est constituée de j et d'un pointeur vers la cellule suivante, pour tous les j tels que $M_{i,j}$ vaut 1. Autrement dit, on dispose d'un tableau `tab`$[n]$ de listes de sommets, tel

que `tab`$[i]$ contienne la liste des successeurs du sommet i, pour tout $1 \leq i \leq n$. Cette représentation est en $\Theta(n + |A|)$, donc possède une complexité en espace bien meilleure que la précédente (à noter que cette complexité est optimale d'un point de vue théorique). Les structures de données correspondantes sont :

```
1  int n;
2  /* un sommet contient une valeur et tous ses successeurs */
3  struct vertex {
4    int num;
5    vertices_list* successors;
6  };
7  /* les successeurs forment une liste */
8  struct vertices_list {
9    vertex* vert;
10   vertices_list* next;
11 };
12 /* on alloue ensuite le tableau de sommets */
13 vertex* adjacancy = (vertex*) malloc(n*sizeof(vertex));
```

Dans le cas d'un graphe non orienté, on parle de *liste de voisins*, mais le codage reste le même. Contrairement à la représentation par matrice d'adjacence, ici, les sommets du graphe ont une véritable représentation et peuvent contenir d'autres données qu'un simple entier. Avec la matrice d'adjacence, d'éventuelles données supplémentaires doivent être stockées dans une structure annexe.

6.3 Existence de chemins & fermeture transitive

Existence de chemins. Soit $G = (S, A)$, un graphe orienté. La matrice d'adjacence de G permet de connaître l'existence de chemins entre deux sommets de G, comme le montre le théorème suivant.

Théorème 6.3.1. *Soit M^p, la puissance p-ième de la matrice d'adjacence M de G. Alors le coefficient $M^p_{i,j}$ est égal au nombre de chemins de longueur p de G, dont l'origine est le sommet x_i et l'extrémité x_j.*

Preuve. On procède par récurrence sur p. Pour $p = 1$, le résultat est vrai puisqu'un chemin de longueur 1 est un arc du graphe. Fixons un entier $p \geq 2$, et supposons le théorème vrai pour tout $j \leq p - 1$. On a :

$$M^p_{i,j} = \sum_{k=1}^{n} M^{p-1}_{i,k} M_{k,j}.$$

Or, tout chemin de longueur p entre x_i et x_j se décompose en un chemin de longueur $p-1$ entre x_i et un certain x_k, suivi d'un arc entre x_k et x_j. Par hypothèse, $M^{p-1}_{i,k}$ est le nombre de chemins de longueur $p - 1$ entre x_i et x_k, donc le nombre de chemins de longueur p

entre x_i et x_j est la somme, sur tout sommet « intermédiaire » x_k, $1 \leq k \leq n$, de $M_{i,k}^{p-1}$ multiplié par le nombre d'arcs (0 ou 1) reliant x_k à x_j. □

La longueur d'un chemin entre deux sommets étant au plus n, on en déduit :

Corollaire 6.3.1. *Soit $N = M + \ldots + M^n$. Alors il existe un chemin entre les sommets x_i et x_j si et seulement si, $N_{i,j} \neq 0$.*

Un algorithme de recherche de l'existence d'un chemin entre deux sommets x et y de G est le suivant :

```
1  int path_exists(graph_mat* G, int x, int y) {
2    int i;
3    int** R = (int**) malloc(G->n*sizeof(int*));
4    for (i=0; i<G->n; i++) {
5      R[i] = (int*) malloc(G->n*sizeof(int));
6    }
7    if (G->mat[x][y] == 1) {
8      return 1;
9    }
10   copy_mat(G->mat,R);
11   for (i=1; i<G->n; i++) {
12     mult_mat(&R,R,G->mat);
13     if (R[x][y] == 1) {
14       return i+1;
15     }
16   }
17 }
18 return -1;
```

Où les fonction `copy_mat(A,B)` et `mult_mat(&C,A,B)` permettent respectivement de copier la matrice `A` dans `B` et de sauver le produit des matrices `A` et `B` dans `C`. Cet algorithme retourne ensuite -1 si aucun chemin n'existe entre les deux sommets et renvoies sinon la longueur du plus court chemin entre les deux sommets.

Dans l'algorithme `path_exists`, on peut avoir à effectuer n produits de matrices pour connaître l'existence d'un chemin entre deux sommets donnés (c'est le cas pour trouver un chemin de longueur n ou pour être certain qu'aucun chemin n'existe). Le produit de deux matrices carrées d'ordre n requérant n^3 opérations [1], la complexité de recherche de l'existence d'un chemin entre deux sommets par cet algorithme est en $\Theta(n^4)$. C'est aussi bien entendu la complexité du calcul de N.

Fermeture transitive. La fermeture transitive d'un graphe $G = (S, A)$ est la relation binaire *transitive* minimale contenant la relation A sur S. Il s'agit d'un graphe $G^* = (S, A^*)$,

1. Les meilleurs algorithmes requièrent une complexité en $\Theta(n^{2.5})$, mais les constantes dans le Θ sont telles que pour des tailles inférieurs à quelques milliers l'algorithme cubique de base est souvent le plus rapide.

tel que $(x, y) \in A^*$ si et seulement s'il existe un chemin dans G d'origine x et d'extrémité y.

La matrice N définie précédemment calcule la fermeture transitive du graphe G. En effet, la matrice d'adjacence M^* de G^* est obtenue à partir de N en posant $M^*_{i,j} = 0$ si $N_{i,j} = 0$, $M^*_{i,j} = 1$ si $N_{i,j} \neq 0$. Une fois calculée G^*, on peut répondre en temps constant à la question de l'existence de chemins entre deux sommets x et y de G.

EXEMPLE D'APPLICATION DU CALCUL DE LA FERMETURE TRANSITIVE

Lors de la phase de compilation d'un programme, un graphe est associé à chaque fonction : c'est le graphe des dépendances *(entre les variables) de la fonction et les sommets de ce graphe représentent les variables, un arc entre deux sommets x et y indique que le calcul de la valeur de x fait appel au calcul de la valeur de y. Le calcul de la fermeture transitive de ce graphe permet alors d'obtenir toutes les variables intervenant dans le calcul d'une variable donnée.*

Dans la suite, nous présentons un algorithme de calcul de la fermeture transitive A^* d'un graphe $G = (S, A)$, qui admet une complexité en $\Theta(n^3)$. Soit x un sommet du graphe G et notons $\Phi_x(A)$, l'opération qui ajoute à A tous les arcs (y, z) tels que y est un prédécesseur de x, et z un successeur :

$$\Phi_x(A) = A \cup \{(y, z),\ (y, x) \in A \text{ et } (x, z) \in A\}.$$

Cette opération vérifie les propriétés suivantes, que nous admettrons :

Propriété 6.3.1.

$$\Phi_x(\Phi_x(A)) = \Phi_x(A),$$

et, pour tout couple de sommets (x, y) :

$$\Phi_x(\Phi_y(A)) = \Phi_y(\Phi_x(A)).$$

Si l'on considère l'itérée de l'action des Φ_{x_i} sur A, on voit aisément que

$$\Phi_{x_1}(\Phi_{x_2}(\ldots(\Phi_{x_n}(A))\ldots)) \subseteq A^*.$$

Mieux :

Proposition 6.3.1. *La fermeture transitive A^* de G est donnée par*

$$A^* = \Phi_{x_1}(\Phi_{x_2}(\ldots(\Phi_{x_n}(A))\ldots)).$$

Preuve. Il reste à montrer $A^* \subseteq \Phi_{x_1}(\Phi_{x_2}(\ldots(\Phi_{x_n}(A))\ldots))$. Soit f, un chemin joignant deux sommets x et y dans G. Il existe donc des sommets de G $y_1, \ldots y_p$ tels que :

$$f = (x, y_1)(y_1, y_2)\ldots(y_p, y),$$

donc $(x, y) \in \Phi_{y_1}(\Phi_{y_2}(\ldots(\Phi_{y_p}(A))\ldots))$. D'après la propriété ci-dessus, on peut permuter l'ordre des y_i dans cette écriture, donc on peut considérer par exemple que les y_i sont

ordonnés suivant leurs numéros croissants. Pour $(i_1, \ldots, i_k) \subseteq \{1, \ldots, n\}$, avec, si $1 \leq \ell < j \leq n$, $i_\ell < i_j$, notons $A_{(i_1,\ldots,i_k)} = \Phi_{x_{i_1}}(\Phi_{x_{i_2}}(\ldots(\Phi_{x_{i_k}}(A))\ldots))$. On peut voir que la suite $A_{(i_1,\ldots,i_k)}$ est croissante (*i.e.* $A_{(i_1,\ldots,i_k)} \subseteq A_{(j_1,\ldots,j_\ell)}$ si $(i_1, \ldots, i_k)$ est une sous-suite de $(j_1, \ldots, j_\ell)$). De plus, on vient de voir que, pour $1 \leq p \leq n$, tout chemin de longueur p joignant deux sommets x et y dans G appartient à un $A_{(i_1,\ldots,i_p)}$, pour un p-uplet $(i_1, \ldots, i_p)$ d'éléments de $\{1, \ldots, n\}$. Or, pour tout tel p-uplet, on a :

$$A_{(i_1,\ldots,i_p)} \subseteq A_{(1,\ldots,n)} = \Phi_{x_1}(\Phi_{x_2}(\ldots(\Phi_{x_n}(A))\ldots)).$$

Donc tout chemin dans G (et tout arc dans A^*) appartient à $\Phi_{x_1}(\Phi_{x_2}(\ldots(\Phi_{x_n}(A))\ldots))$. □

Cette expression de la fermeture transitive d'un graphe donne lieu à l'algorithme de Roy-Warshall suivant :

```
1  int** roy_warshall(graph_mat* G) {
2    int i,j,k;
3    int** M = (int**) malloc(G->n*sizeof(int*));
4    for (i=0; i<G->n; i++) {
5      M[i] = (int*) malloc(G->n*sizeof(int));
6    }
7    copy(G->mat,M);
8    for (k=0; k<G->n; k++) {
9      for (i=0; i<G->n; i++) {
10       for (j=0; j<G->n; j++) {
11         M[i][j] = M[i][j] || (M[i][k] && M[k][j]);
12       }
13     }
14   }
15   return M;
16 }
```

Dans cet algorithme, les deux boucles `for` internes implémentent exactement l'opération $\Phi_{x_k}(A)$; en effet, la matrice d'adjacence M^* de G^* est construite à partir de celle de G par : (x_i, x_j) est un arc de G^* si c'est un arc de G ou si (x_i, x_k) et (x_k, x_j) sont deux arcs de G. C'est exactement ce qui est calculé au milieu de l'algorithme. Clairement, la complexité de l'algorithme de Roy-Warshall est en $\Theta(n^3)$ opérations booléennes.

Remarque : on obtient la *fermeture réflexive-transitive* de G en ajoutant à la matrice d'adjacence de G^* la matrice Identité d'ordre n.

6.4 Parcours de graphes

Nous étudions ici les algorithmes permettant de *parcourir* un graphe quelconque G, c'est-à-dire de visiter tous les sommets de G une seule fois. Il existe essentiellement deux méthodes de parcours de graphes que nous exposons ci-après. Chacune d'elles utilise la notion d'*arborescence* : pour parcourir un graphe, on va en effet produire un *recouvrement* du graphe par une arborescence, ou plusieurs si le graphe n'est pas connexe.

6.4.1 Arborescences

Une arborescence (S, A, r) de racine $r \in S$ est un graphe tel que, pour tout sommet x de S, il existe un unique chemin d'origine r et d'extrémité x. La longueur d'un tel chemin est appelé la *profondeur* de x dans l'arborescence. Dans une arborescence, tout sommet, sauf la racine, admet un unique prédécesseur. On a donc $|A| = |S| - 1$. Par analogie avec la terminologie employée pour les arbres, le prédécesseur d'un sommet est appelé son *père*, les successeurs étant alors appelés ses *fils*. La différence entre une arborescence et un arbre tient seulement au fait que, dans un arbre, les fils d'un sommet sont ordonnés.

On peut montrer qu'un graphe connexe sans cycle est une arborescence. Donc si G est un graphe sans cycle, G est aussi la forêt constituée par ses composantes connexes.

Une arborescence est représentée par son *vecteur père*, $\pi[n]$, où n est le nombre de sommets de l'arborescence, tel que $\pi[i]$ est le père du sommet i. Par convention, $\pi[r] =$ `NULL`.

6.4.2 Parcours en largeur

Une arborescence souvent associée à un graphe quelconque est l'*arborescence des plus courts chemins*. C'est le graphe dans lequel ne sont conservées que les arêtes appartenant à un plus court chemin entre la racine et un autre sommet. On peut la construire grâce à un parcours en largeur d'abord du graphe (*breadth-first traversal* en anglais). Le principe est le suivant : on parcourt le graphe à partir du sommet choisi comme racine en visitant tous les sommets situés à distance (*i.e* profondeur) k de ce sommet, avant tous les sommets situés à distance $k + 1$.

Définition 6.3. *Dans un graphe $G = (S, A)$, pour chaque sommet x, une arborescence des plus courts chemins de racine x est une arborescence (Y, B, x) telle que*

- *un sommet y appartient à Y si, et seulement si, il existe un chemin d'origine x et d'extrémité y.*
- *la longueur du plus court chemin de x à y dans G est égale à la profondeur de y dans l'arborescence (Y, B, x).*

Remarque : cette arborescence existe bien puisque, si $(a_1, a_2, \ldots, a_p)$ est un plus court chemin entre $org(a_1)$ et $ext(a_p)$, alors le chemin $(a_1, a_2, \ldots, a_i)$ est un plus court chemin entre $org(a_1)$ et $ext(a_i)$, pour tout i, $1 \leq i \leq p$. En revanche, elle n'est pas toujours unique.

Théorème 6.4.1. *Pour tout graphe $G = (S, A)$, et tout sommet x de G, il existe une arborescence des plus courts chemins de racine x.*

Preuve. Soit x, un sommet de S. Nous allons construire une arborescence des plus courts chemins de racine x. On considère la suite $\{Y_i\}_i$ d'ensembles de sommets suivante :

- $Y_0 = \{x\}$.
- $Y_1 = Succ(x)$, l'ensemble des successeurs [2] de x.

2. Si $(x, x) \in A$, alors on enlève x de cette liste de successeurs.

– Pour $i \geq 1$, Y_{i+1} est l'ensemble obtenu en considérant tous les successeurs des sommets de Y_i qui n'appartiennent pas à $\bigcup_{j=1}^{i} Y_j$.

Pour chaque Y_i, $i > 0$, on définit de plus l'ensemble B_i des arcs dont l'extrémité est dans Y_i et l'origine dans Y_{i-1}. Attention B_i ne contient pas tous les arcs possibles, mais un seul arc par élément de Y_i : on veut que chaque élément n'ait qu'un seul père. On pose ensuite $Y = \bigcup_i Y_i$, $B = \bigcup_i B_i$. Alors le graphe (Y, B) est par construction une arborescence. C'est l'arborescence des plus courts chemins de racine x, d'après la remarque ci-dessus. □

Cette preuve donne un algorithme de construction de l'arborescence des plus courts chemins d'un sommet donné : comme pour le parcours en largeur d'un arbre on utilise une file qui gère les ensembles Y_i, *i.e.* les sommets à traiter (ce sont les sommets qui, à un moment donné de l'algorithme, ont été identifiés comme successeurs, mais qui n'ont pas encore été parcourus ; autrement dit, ce sont les sommets « en attente »). Par rapport à un parcours d'arbre, la présence éventuelle de cycles fait que l'on utilise en plus un coloriage des sommets avec trois couleurs : les sommets en blanc sont ceux qui n'ont pas encore été traités (au départ, tous les sommets, sauf le sommet x choisi comme racine, sont blancs). Les sommets en gris sont ceux de la file, *i.e.* en attente de traitement, et les sommets en noir sont ceux déjà traités (ils ont donc, à un moment, été défilés). On définit aussi un tableau d qui indique la distance du sommet considéré à x ; en d'autres termes, $d[v]$ est la profondeur du sommet v dans l'arborescence en cours de construction (on initialise chaque composante de d à ∞).

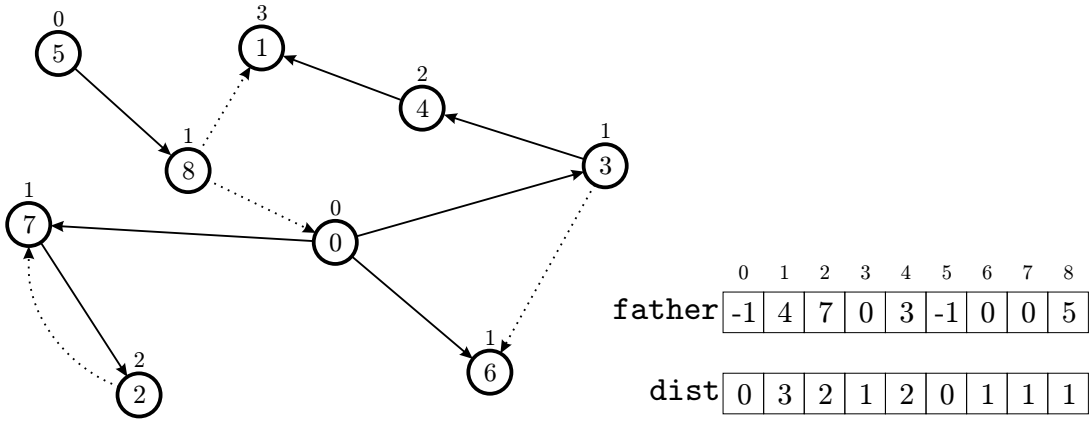
Selon la structure de donnée utilisée pour représenter le graphe, la façon de programmer cet algorithme peut varier et la complexité de l'algorithme aussi ! On se place ici dans le cas le plus favorable : on a une structure de donnée similaire à un arbre, où chaque sommet du graphe contient un pointeur vers la liste de ses successeurs, et en plus, les sommets du graphe sont tous indexés par des entiers compris entre 0 et n.

```
1  struct vertex {
2    int num;
3    vertices_list* successors;
4  };
5  struct vertices_list {
6    vertex* vert;
7    vertices_list* next;
8  };
9  int n;
10 int* color = NULL;
11 int* dist = NULL;
12 int* father = NULL;
13
14 void init(vertex* root, int nb) {
15   int i;
16   n = nb;
17   if (color == NULL) {
18     /* initialiser uniquement si cela n'a jamais été initialisé */
19     color = (int*) malloc(n*sizeof(int));
```

```
20      dist = (int*) malloc(n*sizeof(int));
21      father = (int*) malloc(n*sizeof(int));
22      for (i=0; i<n; i++) {
23        color[i] = 0;   /* 0 = blanc, 1 = gris, 2 = noir */
24        dist[i] = -1;   /* -1 = infini */
25        father[i] = -1; /* -1 = pas de père */
26      }
27    }
28    color[root->num] = 1;
29    dist[root->num] = 0;
30  }
31
32  void minimum_spanning_tree(vertex* root, int num) {
33    vertex* cur;
34    vertices_list* tmp;
35    init(root,num);
36    push(root);
37    while (!queue_is_empty()) {
38      while((cur=pop()) != NULL) {
39        tmp = cur->successors;
40        while (tmp != NULL) {
41          if (color[tmp->vert->num] == 0) {
42            /* si le noeud n'avait jamais été atteint, on fixe son
43               père et on le met dans la file de noeuds à traiter. */
44            father[tmp->vert->num] = cur->num;
45            dist[tmp->vert->num] = dist[cur->num]+1;
46            color[tmp->vert->num] = 1;
47            push(tmp->vert);
48          }
49          tmp = tmp->next;
50        }
51        color[cur->num] = 2;
52      }
53      swap_queues();
54    }
55  }
```

Pour simplifier, la gestion des files à été réduite à son minimum : on a en fait deux files, l'une dans laquelle on ne fait que retirer des éléments avec le fonction `pop` et l'autre dans laquelle on ajoute des éléments avec la fonction `push`. La fonction `swap_queues` permet d'échanger ces deux files et la fonction `queue_is_empty` retourne vrai quand la première file est vide (on n'a plus de sommets à retirer).

Chaque parcours en largeur à partir d'un sommet fournissant une seule composante connexe du graphe, si le graphe en possède plusieurs il faudra nécessairement appeler la fonction `minimum_spanning_tree` plusieurs fois avec comme argument à chaque fois un sommet x non encore visité. Si on se donne un tableau `vertices` contenant des pointeurs vers tous les sommets du graphe, un algorithme de parcours en largeur de toutes les composantes connexes est alors :



	0	1	2	3	4	5	6	7	8
father	-1	4	7	0	3	-1	0	0	5
dist	0	3	2	1	2	0	1	1	1

FIGURE 6.5 – *Application de* `all_spanning_trees` *à un graphe. Les sommets 0 et 5 sont des racines. Les arcs en pointillés ne font pas partie d'une arborescence.*

```
1  void all_spanning_trees(vertex** vertices, int nb) {
2    int i;
3    /* un premier parcours en partant du noeud 0 */
4    minimum_spanning_tree(vertices[0], nb);
5    for (i=1; i<nb; i++) {
6      if (color[i] != 2) {
7        /* si le noeud i n'a jamais été colorié en noir */
8        minimum_spanning_tree(vertices[i], nb);
9      }
10   }
11 }
```

Complexité. Dans l'algorithme `minimum_spanning_tree`, la phase d'initialisation prend un temps en $\Theta(n)$ la première fois (on initialise des tableaux de taille n) et un temps constant les suivantes, puis, pour chaque sommet dans la file, on visite tous ses successeurs une seule fois. Chaque sommet étant visité (traité) une seule fois, la complexité de l'algorithme est en $\Theta(n + |A|)$.

Maintenant, si le graphe possède plusieurs composantes connexes, on peut le décomposer en sous-graphes $G_i = (S_i, A_i)$. La complexité de l'algorithme `all_spanning_trees` sera alors n pour la première initialisation, plus n pour la boucle sur les sommets, plus $\Theta(|S_i|+|A_i|)$ pour chaque sous-graphe G_i. Au total on obtient une complexité en $\Theta(n+|A|)$.

⚠ ATTENTION

Cette complexité est valable pour une représentation du graphe par listes de successeurs, mais dans le cas d'une représentation par matrice d'adjacence, l'accès à tous les successeurs d'un sommet nécessite de parcourir une ligne complète de la matrice. La complexité de l'algorithme devient alors $\Theta(n^2)$.

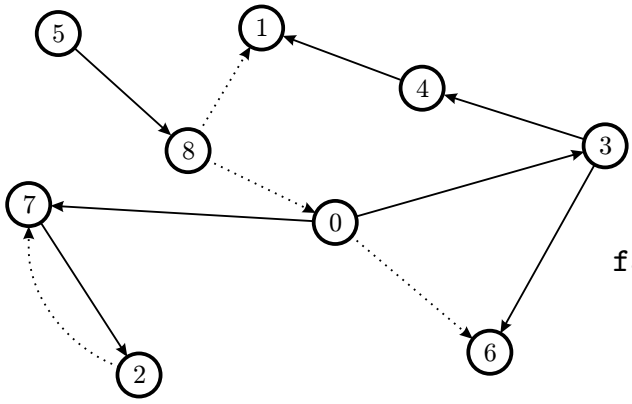
6.4.3 Parcours en profondeur

Alors que le parcours en largeur est essentiellement itératif (implémentation par file), le parcours en profondeur est de nature récursive [3]. Le principe du parcours en profondeur est de visiter tous les sommets en allant d'abord le plus profondément possible dans le graphe. Un système de datation (utilisant deux tableaux `beg[]` et `end[]`) permet de mémoriser les dates de début et de fin de traitement d'un sommet. L'algorithme de parcours crée une forêt [4] par un procédé récursif. Chaque arborescence de la forêt est créée à partir d'un sommet x par l'algorithme suivant (le code couleur utilisé pour colorier les sommets a la même signification que dans le cas du parcours en largeur - les sommets gris étant ceux de la pile, et correspondant ici aux sommets « en cours de visite », dans le sens où le traitement d'un tel sommet se termine lorsque l'on a parcouru tous ses descendants). Voici une implémentation du parcours en profondeur, reprenant les mêmes structures que l'implémentation du parcours en largeur.

```
1  int n;
2  int date = 0;
3  int* color = NULL;
4  int* father = NULL;
5  int* beg = NULL;
6  int* end = NULL;
7
8  void init(int nb) {
9    int i;
10   n = nb;
11   if (color == NULL) {
12     /* initialise uniquement si cela n'a jamais été initialisé */
13     color = (int*) malloc(n*sizeof(int));
14     father = (int*) malloc(n*sizeof(int));
15     beg = (int*) malloc(n*sizeof(int));
16     end = (int*) malloc(n*sizeof(int));
17     for (i=0; i<n; i++) {
18       color[i] = 0;   /* 0 = blanc, 1 = gris, 2 = noir */
19       father[i] = -1; /* -1 = pas de père */
20       beg[i] = -1;    /* -1 = pas de date */
21       end[i] = -1;    /* -1 = pas de date */
22     }
23   }
24 }
25
26 void depth_first_spanning_tree(vertex* x) {
27   vertices_list* tmp;
28   color[x->num] = 1;
```

3. Une version itérative de l'algorithme peut être obtenue en utilisant une pile.

4. C'est la forêt correspondant aux arcs effectivement utilisés pour explorer les sommets, donc, même si le graphe est connexe, il se peut qu'un parcours en profondeur de celui-ci produise une forêt (*cf.* FIGURE 6.6). En revanche, si le graphe est fortement connexe, on est certain d'avoir un seul arbre dans cette forêt.



	0	1	2	3	4	5	6	7	8
father	-1	4	7	0	3	-1	3	0	5
beg	0	3	10	1	2	14	6	9	15
end	13	4	11	8	5	17	7	12	16

FIGURE 6.6 – *Forêt engendrée lors du parcours en profondeur d'un graphe. Le graphe de départ est connexe, mais deux arbres sont nécessaires pour le représenter entièrement. Les arcs en pointillés sont ceux qui ne font partie d'aucune arborescence. Ici, les sommets ont été parcourus dans l'ordre : 0, 3, 4, 1, 6, 7, 2 puis 5, 8.*

```
29   beg[x->num] = date;
30   date++;
31   tmp = x->successors;
32   while (tmp != NULL) {
33     if (color[tmp->vert->num] == 0) {
34       /* un successeur non encore visité a été trouvé */
35       father[tmp->vert->num] = x->num;
36       depth_first_spanning_tree(tmp->vert);
37     }
38     tmp = tmp->next;
39   }
40   /* une fois tous les successeurs traités
41      le traitement du sommet x est fini */
42   color[x->num] = 2;
43   end[x->num] = date;
44   date++;
45 }
```

Comme pour le parcours en largeur, il faut maintenant appeler cet algorithme pour chaque arbre de la forêt. On utilise donc la fonction suivante :

```
1 void all_depth_first_spanning_trees(vertex** vertices, int nb) {
2   int i;
3   init(nb);
4   for (i=0; i<nb; i++) {
5     if (color[i] != 2) {
6       /* si le noeud i n'a jamais été colorié en noir */
7       depth_first_spanning_tree(vertices[i]);
8     }
```

```
9    }
10 }
```

Le parcours en profondeur passe une seule fois par chaque sommet et fait un nombre d'opérations proportionnel au nombre d'arcs. La complexité d'un tel parcours est donc en $\Theta(n + |A|)$.

Application au labyrinthe. Il est possible de représenter un labyrinthe par un graphe, dont les sommets sont les embranchements, ceux-ci étant reliés selon les chemins autorisés. Le parcours d'un graphe en profondeur d'abord correspond au cheminement d'une personne dans un labyrinthe : elle parcourt un chemin le plus en profondeur possible, et reviens sur ses pas pour en emprunter un autre, tant qu'elle n'a pas trouvé la sortie. Ce « retour sur ses pas » est pris en charge par la récursivité.

Le parcours en largeur quand à lui modéliserait plutôt un groupe de personnes dans un labyrinthe : au départ, le groupe est au point d'entrée du labyrinthe, puis il se répartit de façon à ce que chaque personne explore un embranchement adjacent à l'embranchement où il se trouve. Il modélise également un parcours de labyrinthe effectué par un ordinateur, dans lequel le sommet destination (la sortie) est connu, et pour lequel il s'agit de trouver *le plus court chemin* de l'origine (entrée du labyrinthe) à la destination, *i.e.* un chemin particulier dans l'arborescence des plus courts chemins de racine l'origine.

6.5 Applications des parcours de graphes

6.5.1 Tri topologique

Les graphes orientés sans circuit sont utilisés dans de nombreuses applications pour représenter des précédences entre évènements (on parle alors de *graphe des dépendances*). Le tri topologique d'un graphe consiste à ordonnancer les tâches représentées par les sommets du graphe selon les dépendances modélisées par ses arcs. Par convention, une tache doit être accomplie avant une autre, si un arc pointe du sommet correspondant à cette tache vers le sommet correspondant à l'autre. Autrement dit, étant donné un graphe orienté acyclique $G = (S, A)$, le tri topologique de G ordonne les sommets de G en une suite telle que l'origine d'un arc apparaisse avant son extrémité. Il peut être vu comme un alignement des sommets de G le long d'une ligne horizontale, de manière que tous les arcs soient orientés de gauche à droite.

Le parcours en profondeur d'un graphe permet de résoudre le problème du tri topologique. L'idée est d'effectuer un parcours en profondeur du graphe, puis de retourner la liste des sommets dans l'ordre *décroissant* de leur date de fin de traitement. En effet, si (u, v) est un arc (et donc si la tâche représentée par u doit être effectuée avant celle représentée par v) alors, dans l'algorithme de parcours en profondeur, le sommet v aura nécessairement fini d'être traité avant le sommet u : un sommet n'a jamais fini d'être traité avant que tous ses successeurs n'aient été traités. L'algorithme de tri topologique est donc le suivant :

```
1  void topological_sort(vertex** vertices, int nb) {
2    all_depth_first_spanning_trees(vertices, nb);
3    /* on doit faire un tri en temps linéaire : on utilise
4       le fait que les valeurs de end sont entre 0 et 2*nb */
5    int rev_end[2*nb];
6    int i;
7    for (i=0; i<2*nb; i++) {
8      rev_end[i] = -1;
9    }
10   for (i=0; i<nb; i++) {
11     rev_end[end[i]] = i;
12   }
13   for (i=2*nb-1; i>=0; i--) {
14     if (rev_end[i] != -1) {
15       printf("%d", rev_end[i]);
16     }
17   }
18 }
```

On obtient ainsi un algorithme de complexité $\Theta(|S| + |A|)$, *i.e.* linéaire en la taille du graphe.

Test de Cyclicité d'un Graphe

De façon similaire, on peut tester si un graphe est cyclique par un parcours en profondeur : en effet, si un graphe possède un cycle, cela veut dire que, dans une arborescence de recouvrement, un sommet x est l'origine d'un arc dont l'extrémité est un ancêtre de x dans cette arborescence. Dans le parcours en profondeur, cela veut dire que à un moment, le successeur d'un sommet gris sera gris. Il suffit donc de modifier légèrement l'algorithme de parcours en profondeur pour ne pas seulement tester si la couleur d'un sommet est blanche, mais aussi tester si elle est grise et afficher un message en conséquence.

6.5.2 ⋆ Calcul des composantes fortement connexes

Une composante fortement connexe d'un graphe est un ensemble de sommets tel qu'un chemin existe de n'importe quel sommet de cette composante vers n'importe quel autre. Typiquement, un cycle dans un graphe forme une composante fortement connexe. Tout graphe peut se décomposer en composantes fortement connexe disjointes : un ensemble de composantes fortement connexes tel que s'il existe un chemin allant d'un sommet d'une composante vers un sommet d'une autre, le chemin inverse n'existe pas. Par exemple, un graphe acyclique n'aura que des composantes fortement connexes composées d'un seul sommet.

On a vu que le parcours en profondeur permet de tester l'existence de cycles dans un graphe. Comme nous allons le voir, une modification de l'algorithme de parcours en pro-

fondeur permet de retrouver toutes les composantes fortement connexes. C'est l'algorithme de Tarjan, inventé en 1972. Le principe de l'algorithme est le suivant :

- on effectue un parcours en profondeur du graphe,
- à chaque fois que l'on traite un nouveau sommet on lui attribue un `index` (attribué par ordre croissant) et un `lowlink` initialisé à la valeur de l'`index` et on ajoute le sommet à une pile,
- le `lowlink` correspond au plus petit `index` accessible par un chemin partant du sommet en question, donc à la fin du traitement d'un sommet on met à jour son `lowlink` pour être le minimum entre les `lowlink` de tous ses successeurs et l'`index` du sommet courant,
- chaque fois qu'un successeur d'un sommet est déjà colorié en gris on met à jour le `lowlink` de ce sommet en conséquence,
- à chaque fois que l'on a fini de traiter un sommet dont le `lowlink` est égal à l'`index` on a trouvé une composante fortement connexe. On peut retrouver l'ensemble des éléments de cette composante en retirant des éléments de la pile jusqu'à atteindre le sommet courant.

Cet algorithme est donc un peu plus compliqué que les précédents, mais reste relativement simple à comprendre. Si on a un graphe acyclique, à aucun moment un successeur d'un sommet n'aura un `index` plus petit que le sommet courant. Du coup, les `lowlink` restent toujours égaux à l'`index` du sommet et à la fin du traitement de chaque sommet une nouvelle composant connexe a été trouvée : cette composante contient juste le sommet courant et on retrouve bien le résultat attendu : un graphe acyclique contient n composantes fortement connexes composées d'un seul sommet.

Maintenant, si le graphe contient un cycle, tous les éléments de ce cycle vont avoir leur `lowlink` égal à l'`index` du premier sommet visité. Donc, un seul sommet du cycle aura un son `index` égal à son `lowlink` : le premier visité, qui est donc aussi le plus profond dans la pile. Tous les éléments qui sortiront de la pile avant lui sont alors les autres sommets de la composante fortement connexe formée par le cycle. Voici le code correspondant à l'algorithme de Tarjan (un exemple d'exécution est visible sur la FIGURE 6.7) :

```
1  int cur_index = 0;
2  int color[n];   /* initialisés à 0  */
3  int index[n];   /* initialisés à -1 */
4  int lowlink[n]; /* initialisés à -1 */
5
6  void Tarjan(vertex* x) {
7    vertices_list* tmp;
8    vertex* tmp2;
9    index[x->num] = cur_index;
10   lowlink[x->num] = cur_index;
11   color[x->num] = 1;
12   cur_index++;
13   push(x);
14   tmp = x->successors;
15   while (tmp != NULL) {
16     if (index[tmp->vert->num] == -1) {
```

```
17        /* si le successeur n'a jamais été visité,
18           on le visite */
19        Tarjan(tmp->vert);
20        /* et on met à jour le lowlink de x */
21        lowlink[x->num] = min(lowlink[x->num], lowlink[tmp->vert->num]);
22      } else if (color[tmp->vert->num] == 1) {
23        /* si le successeur est en cours de visite on a trouvé un cycle */
24        lowlink[x->num] = min(lowlink[x->num], lowlink[tmp->vert->num]);
25      }
26      tmp = tmp->next;
27    }
28    if (lowlink[x->num] == index[x->num]) {
29      /* on a fini une composante fortement connexe
30         et on dépile jusqu'à retrouver x. */
31      printf("CFC : ");
32      do {
33        tmp2 = pop();
34        color[tmp2->num] = 2;
35        printf("%d,", tmp2->num);
36      while (tmp2 != x)
37      printf("\n");
38    }
39  }
```

Notons que `index` joue exactement le même rôle que la variable `beg` dans le parcours en profondeur, en revanche le coloriage change un peu : on ne colorie un sommet en noir que lorsqu'il sort de la pile, pas directement quand on a fini de traiter ses successeurs. Comme pour le parcours en profondeur, cette algorithme ne va pas forcément atteindre tous les sommets du graphe, il faut donc l'appeler plusieurs fois, exactement comme avec l'algorithme `all_depth_first_spanning_trees`.

6.5.3 Calcul de chemins optimaux

On considère ici un graphe $G = (S, A)$ pondéré, *i.e.* à chaque arc est associée une valeur appelée *poids*. On appellera poids d'un chemin la somme des poids des arcs qui le composent. Le problème est alors, étant donné deux sommets, de trouver un chemin de poids minimal reliant ces deux sommets (s'il en existe un).

La résolution de ce problème à beaucoup d'applications, par exemple pour le calcul du plus court chemin d'une ville à une autre (en passant par des routes dont on connaît la longueur) ou le calcul d'un chemin de capacité maximale dans un réseau de communication (dans lequel le taux de transmission d'un chemin est égal au minimum des taux de transmission de chaque liaison intermédiaire) sont des exemples de situations dans lesquelles ce problème intervient.

Afin de traiter de manière uniforme les différents ensembles de définition pour les poids, induits par l'application (l'ensemble des nombres réels, l'ensemble $\{vrai, faux\}$...), on considère la situation générale où les poids appartiennent à un semi-anneau $\mathcal{S}(\sqcup, \odot, \emptyset, \epsilon)$, c'est-à-dire vérifiant les propriétés suivantes :

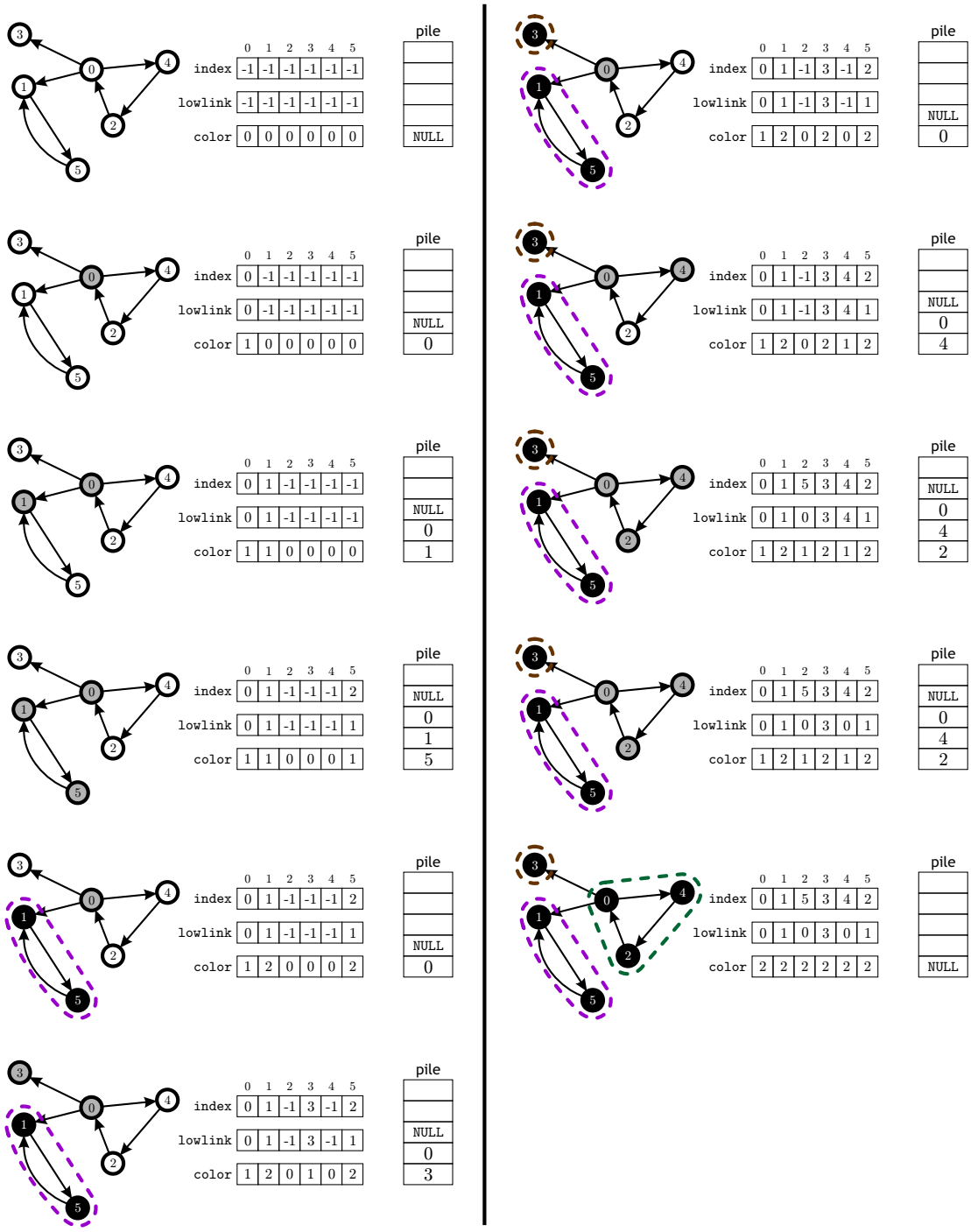


FIGURE 6.7 – *Exemple d'exécution de l'algorithme de Tarjan sur un petit graphe à trois composantes fortement connexes. On explore d'abord le sommet 0 et on voit ensuite l'évolution des différentes variables au fur et à mesure de l'exploration des autres sommets.*

- $\sqcup$ est une loi de composition interne sur $\mathcal{S}$, commutative, associative, idempotente ($x \sqcup x = x$), d'élément neutre $\emptyset$. On impose de plus qu'étant donnée une séquence infinie dénombrable $s_1, \ldots, s_i, \ldots$ d'éléments de $\mathcal{S}$, $s_1 \sqcup s_2 \sqcup \ldots \in \mathcal{S}$.
- $\odot$ est une loi de composition interne associative, d'élément neutre ϵ.
- $\emptyset$ est absorbant pour $\odot$.
- Pour tout $x \in \mathcal{S}$, on note x^*, l'élément de $\mathcal{S}$ défini par

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

Par exemple, dans le cas du problème du plus court chemin reliant une ville à une autre, $\mathcal{S}(\sqcup, \odot, \emptyset, \epsilon) = \{\mathbb{R} \cup \infty\}(\min, +, \infty, 0)$ (la longueur d'un chemin est la somme ($\odot$) des poids des arcs qui le composent, la longueur du plus court chemin étant le minimum ($\sqcup$) des longueurs des chemins existants ; à noter que, pour tout x, on a ici $x^* = 0$).

Soit donc $\mathcal{S}(\sqcup, \odot, \emptyset, \epsilon)$, un semi-anneau, et soit p, une fonction de A dans $\mathcal{S}$, qui associe un poids à chaque arc du graphe. On prolonge p à $S \times S$ par $p(i,j) = \emptyset$ si $(i,j) \notin A$. On étend également p aux chemins en définissant, pour tout chemin $(s_1, \ldots, s_k)$, $p((s_1, \ldots, s_k)) = p((s_1, s_2)) \odot p((s_2, s_3)) \odot \ldots \odot p((s_{k-1}, s_k))$. On cherche à déterminer le coût minimal (en fait optimal, selon l'application) des chemins reliant deux sommets quelconques du graphe. Autrement dit, si i et j sont deux sommets de G, on cherche à déterminer

$$\ell_{i,j} = \sqcup_{chemin \in \{chemin\ de\ i\ a\ j\}} p(chemin).$$

Il est clair que, pour un graphe quelconque, il n'est en général pas possible d'énumérer tous les chemins reliant deux sommets. Pour calculer la matrice $\ell = (\ell_{i,j})_{i,j}$, il existe un algorithme dû à Aho, Hopcroft et Ullman, qui admet une complexité en $\Theta(n^3)$, n étant le nombre de sommets de G. Si on appelle `Best(x,y)` la fonction qui calcule $x \sqcup y$, `Join(x,y)` la fonction $x \odot y$, `Star(x)` la fonction x^* et `elem` le type des éléments de $\mathcal{S}$, on peut alors écrire le code de l'algorithme AHU (qui ressemble beaucoup à l'algorithme de Roy-Warshall vu précédemment). Il prend en argument la matrice `M` telle que `M[i][j]` est le poids de l'arête reliant le sommet `i` au somment `j` :

```
1  void AHU(elem** M, int n) {
2    int i,j,k;
3    for (k=0; k<n; k++) {
4      for (i=0; i<n; i++) {
5        for (j=0; j<n; j++) {
6          M[i][j] = Optimal(M[i][j],
7                            Join(Join(M[i][k],Star(M[k][k])),M[k][j]));
8        }
9      }
10   }
11 }
```

Proposition 6.5.1. *À la fin de l'exécution de l'algorithme* `AHU`*, la matrice* `M` *a été modifiée et contient en* `M[i][j]` *le poids du chemin optimal reliant le sommet* `i` *au sommet* `j`*. La matrice ℓ que l'on voulait calculer est donc définie par $\ell_{i,j}$ =* `M[i][j]`*.*

Preuve. Notons $\ell_{i,j}^{(0)}$, la valeur de la matrice `M` passée en argument : $\ell_{i,j}^{(0)} = p(i,j)$ ou $\emptyset$. Considérons la suite $(\ell^{(k)})_k$ de matrices définie par récurrence par

$$\ell_{i,j}^{(k)} = \ell_{i,j}^{(k-1)} \sqcup (\ell_{i,k}^{(k-1)} \odot (\ell_{k,k}^{(k-1)})^* \odot \ell_{k,j}^{(k-1)}).$$

Alors nous allons prouver par récurrence que le coefficient $\ell_{i,j}^{(k)}$ est égal au coût minimal d'un chemin reliant le sommet i au sommet j, en ne passant que par des sommets intermédiaires de numéro inférieur ou égal à k.

En effet, cette propriété des $\ell_{i,j}^{(k)}$ est vraie pour $k = 0$. Soit $k \geq 1$, et supposons cette propriété vraie pour $k-1$. Chaque chemin reliant le sommet i au sommet j en ne passant que par des sommets intermédiaires de numéro inférieur ou égal à k peut soit ne pas passer par k - auquel cas, par hypothèse de récurrence, son coût minimal est égal à $\ell_{i,j}^{(k-1)}$ - soit être décomposé en un chemin de i à k, d'éventuels circuits partant et arrivant en k, et un chemin de k à j. Par hypothèse de récurrence, les coût minimaux de ces chemins intermédiaires sont resp. $\ell_{i,k}^{(k-1)}$, $(\ell_{k,k}^{(k-1)})^*$ et $\ell_{k,j}^{(k-1)}$. Le coût minimal d'un chemin reliant i à j en ne passant que par des sommets intermédiaires de numéro inférieur ou égal à k est donc donné par la formule ci-dessus.

A la fin de l'algorithme, la matrice $(\ell_{i,j}^{(n)})$ a été calculée, qui correspond à la matrice des coûts minimaux des chemins reliant deux sommets quelconques du graphe, en ne passant que par des sommets intermédiaires de numéro inférieur ou égal à n, *i.e.* par n'importe quel sommet. □

- Dans l'implémentation de l'algorithme donnée ici on n'utilise qu'une seule matrice, donc en calculant les coefficients de la fin de la matrice à l'étape k ceux du début ont déjà été modifiés depuis l'étape $k-1$. La matrice `M` ne suit donc pas exactement la formule de récurrence de la preuve, mais le résultat final reste quand même identique.
- Lorsque $\mathcal{S}(\sqcup, \odot, \emptyset, \epsilon) = \{\text{vrai,faux}\}$(ou, et, faux, vrai), l'algorithme ci-dessus est exactement l'algorithme de Roy-Warshall de calcul de la fermeture transitive de G (on a ici x^*=vrai pour tout $x \in \{\text{vrai,faux}\}$).
- Dans le cas spécifique du calcul de plus courts chemins, *i.e.* lorsque $\mathcal{S}(\sqcup, \odot, \emptyset, \epsilon) = \{\mathbb{R} \cup \infty\}(\min, +, \infty, 0)$, l'algorithme de Aho-Hopcroft-Ullman est plus connu sous le nom d'algorithme de Floyd-Warshall.

L'algorithme `AHU` *calcule* le coût minimal d'un chemin entre deux sommets quelconques du graphe, sans *fournir* un chemin qui le réalise. Nous présentons ci-après un algorithme permettant de trouver effectivement un tel chemin.

Calcul d'un chemin de coût minimal. Soit $G = (S, A)$, un graphe sans circuit représenté par sa matrice d'adjacence, et soit p, une fonction (poids) définie sur $S \times S$ comme ci-dessus. On suppose ici que cette fonction vérifie $p(i,i) = 0$.

On suppose également que l'on a calculé les coûts minimaux des chemins entre tous les couples de sommets, *i.e.* que l'on dispose de la matrice $\ell = (\ell_{i,j})_{i,j}$. Pour calculer ces chemins, on définit une *matrice de liaison* $\Pi = (\pi_{i,j})_{ij}$, de la manière suivante : $\pi_{i,j} = \infty$

si $i = j$ ou s'il n'existe aucun chemin entre i et j. Sinon, $\pi_{i,j}$ est le prédécesseur de j sur un chemin de coût minimal issu de i.

La connaissance de Π, permet d'imprimer un chemin de coût minimal entre deux sommets i et j de G avec l'algorithme récursif suivant (∞ est remplacé par -1) :

```
1  void print_shortest_path(int** Pi, int i, int j) {
2    if (i==j) {
3      printf("%d\n", i)
4    } else {
5      if (Pi[i][j] == -1) {
6        printf("Pas de chemin de %d à %d\n", i, j);
7      } else {
8        print_shortest_path(Pi, i, Pi[i][j]);
9        printf(",%d", j);
10     }
11   }
12 }
```

Sous-Graphe de Liaison

On définit le sous-graphe de liaison *de G pour i par $G_{\pi,i} = (S_{\pi,i}, A_{\pi,i})$, où :*

$$S_{\pi,i} = \{j \in S,\ \pi_{i,j} \neq \infty\} \cup \{i\},$$
$$A_{\pi,i} = \{(\pi_{i,j}, j),\ j \in S_{\pi,i} \setminus \{i\}\}.$$

Dans le cas où le poids est donné par la longueur, on peut montrer que $G_{\pi,i}$ est une arborescence des plus courts chemins de racine i.

Il est possible de modifier l'algorithme `AHU` pour calculer les matrices $(\ell_{i,j})_{i,j}$ et Π en même temps. Le principe est le même que dans l'algorithme initial, *i.e.* on calcule des suites de matrices en considérant des sommets intermédiaires d'un chemin de coût minimal. Plus précisément, on définit la séquence $\Pi^{(0)}, \Pi^{(1)}, \ldots, \Pi^{(n)}$, où $\pi^{(k)}_{i,j}$ est le prédécesseur du sommet j dans un chemin de coût minimal issu de i et ne passant que par des sommets intermédiaires de numéro inférieur ou égal à k. On a bien entendu $\Pi^{(n)} = \Pi$, et on peut exprimer $\pi^{(k)}_{i,j}$ récursivement par :

$$\pi^{(0)}_{i,j} = \begin{cases} \infty & \text{si } i = j \text{ ou } p(i,j) = \emptyset, \\ i & \text{sinon,} \end{cases}$$

et, pour $k \geq 1$,

$$\pi^{(k)}_{i,j} = \begin{cases} \pi^{(k-1)}_{i,j} & \text{si } \ell^{(k)}_{i,j} = \ell^{(k-1)}_{i,j}, \\ \pi^{(k-1)}_{k,j} & \text{si } \ell^{(k)}_{i,j} = \ell^{(k-1)}_{i,k} \odot \ell^{(k-1)}_{k,j}. \end{cases}$$

Avec les notations utilisées dans la preuve de la proposition 6.5.1 : on a ici $\ell^{(0)}_{i,j} = p(i,j)$.

Au final, on obtient `AHU_link`, la version modifiée de `AHU` calculant aussi la matrice Π. Comme l'algorithme `AHU` de départ cet algorithme prend en argument des matrices déjà initialisées : $\pi_{i,j} = -1$ si $(i,j) \notin A$ et $\pi_{i,j} = i$ si $(i,j) \in A$.

```
1  void AHU_link(elem** M, int** Pi, int n) {
2    int i,j,k;
3    elem tmp;
4    for (k=0; k<n; k++) {
5      for (i=0; i<n; i++) {
6        for (j=0; j<n; j++) {
7          tmp = Optimal(M[i][j],Join(M[i][k],M[k][j]));
8          if (M[i][j] != tmp) {
9            M[i][j] = tmp;
10           Pi[i][j] = Pi[k][j];
11         }
12       }
13     }
14   }
15 }
```

Chapitre 7

Recherche de motifs

La recherche de motifs dans un texte est une autre opération utile dans plusieurs domaines de l'informatique. Une application directe est la recherche (efficace) de séquences d'acides aminés dans de très longues séquences d'ADN, mais d'autres applications sont moins évidentes : par exemple, lors de la compilation d'un programme on doit identifier certains mots clef (`for`*, *`if`*, *`sizeof`*...), identifier les noms de variables et de fonctions... Tout cela revient en fait à une recherche d'une multitude de motifs en parallèle.*

7.1 Définitions

Un *alphabet* est un ensemble fini de *symboles*. Un *mot* sur un alphabet Σ est une suite finie de symboles de Σ. Par exemple, pour l'alphabet latin $\Sigma = \{a, b, c, ..., z\}$, les symboles sont des lettres et un mot est une suite finie de lettres. On considère également le *mot vide* noté ε, qui ne contient aucun symbole. La *longueur* d'un mot est le nombre de symboles qui le compose : le mot vide ε est donc de longueur 0. Un mot est dit *préfixe* d'un autre si ce mot apparaît au début de l'autre (*mot* est préfixe de $motif$). De même il est *suffixe* d'un autre s'il apparaît à la fin (tif est suffixe de $motif$).

Les mots étant de longueur finie mais quelconque il semble naturelle de les coder avec une structure de liste, cependant, par soucis d'efficacité, il seront en général codés avec des tableaux. Dans ce contexte, le problème de la recherche de motif (*pattern-matching* en anglais) consiste simplement à trouver toutes les occurrences d'un mot P dans un texte T.

Soient donc T de longueur n codé par `T[0]`...`T[n-1]` et P de longueur m codé par `P[0]`...`P[m-1]`. On considère que les éléments de l'alphabet Σ sont codés par des `int`. La recherche de motifs va consister à trouver tous les décalages $s \in [0, n-m]$ tels que :

$$\forall j \in [0, m-1], P[j] = T[s+j].$$

L'énoncé de ce problème est donc très simple, en revanche, les algorithmes efficaces pour y répondre le sont un peu moins.

Un premier algorithme naïf. L'algorithme le plus naïf pour la recherche de motif est déduit directement de l'énoncé : on essaye tous les décalages possibles, et pour chaque décalage on regarde si le texte correspond au motif. Voici le code d'un tel algorithme :

```
1  void basic_pattern_lookup(int* T, int n, int* P, int m) {
2    int i,j;
3    for (i=0; i<=n-m; i++) {
4      for (j=0; j<m; j++) {
5        if (T[i+j] != P[j]) {
6          break;
7        }
8      }
9      if (j == m) {
10       printf("Motif trouvé à la position %d.", i);
11     }
12   }
13 }
```

Cet algorithme a une complexité dans le pire cas en $\Theta(nm)$ qui n'est clairement pas optimale : l'information trouvée à l'étape s est totalement ignorée à l'étape $s+1$.

7.2 L'algorithme de Rabin-Karp

L'idée de l'algorithme de Rabin-Karp est de reprendre l'algorithme naïf, mais de remplacer la comparaison de mots par une comparaison d'entiers. Pour cela, l'idée et de considérer le motif recherché comme un entier codé en base d, où d est le nombre d'éléments de Σ. Ainsi, à un motif de longueur m est associé un entier compris entre 0 et $d^m - 1$. De même, pour chaque décalage, on va définir un entier correspondant au m symboles de T partant de ce décalage. On définit donc :

$$p = \sum_{i=0}^{m-1} P[i]d^i,$$

$$\forall s \in [0, n-m], \quad t_s = \sum_{i=0}^{m-1} T[s+i]d^i.$$

Une fois ces entiers définis, la recherche de motif consiste simplement à comparer p à chacun des t_s. Cependant, même si on considère que les opérations sur les entiers se font toujours en temps constant, cette méthode ne permet pas encore de gagner en complexité car le calcul de chacun des t_s a une complexité en $\Theta(m)$ et donc calculer tous les t_s coûte $\Theta(nm)$. Afin d'améliorer cela on va calculer les t_s de façon récursive : entre t_s et t_{s+1} un symbole est ajouté et un autre enlevé. On a donc :

$$t_{s+1} = \left\lfloor \frac{t_s}{d} \right\rfloor + d^{m-1}T[s+m].$$

Si les calculs sur les entiers se font en temps constant, le calcul de tous les t_s a alors une complexité en $\Theta(n+m)$ et le coût total de la recherche de motif est aussi $\Theta(n+m)$. Voici le code de cet algorithme :

```
1  void Rabin_Karp_Partic(int* T, int n, int* P, int m, int d) {
2    int i,h,p,t;
3    /* on calcule d^(m-1) une fois pour toute */
4    h = pow(d,m-1);
5    /* on calcule p */
6    p=0;
7    for (i=m-1; i>=0; i--) {
8      p = P[i] + d*p;
9    }
10   /* on calcule t_0 */
11   t=0;
12   for (i=m-1; i>=0; i--) {
13     t = T[i] + d*t;
14   }
15   /* on teste tous les décalages */
16   for (i=0; i<n-m; i++) {
17     if (t == p) {
18       printf("Motif trouvé à la position %d.", i);
19     }
20     t = t/d + h*T[i+m];
21   }
22   if (t == p) {
23     printf("Motif trouvé à la position %d.", n-m);
24   }
25 }
```

En pratique, cette méthode est très efficace quand les entiers t et p peuvent être représentés par un `int`, mais dès que m et d grandissent cela n'est plus possible. L'utilisation de grands entiers fait alors perdre l'avantage gagné car un opération sur les grands entiers aura un coût en $\Theta(m\log(d))$ et la complexité totale de l'algorithme devient alors $\Theta(nm\log(d))$ (notons que cette complexité est identique à celle de l'algorithme naïf : le facteur $\log(d)$ correspond au coût de la comparaison de deux symboles de Σ qui est négligé dans la complexité de l'algorithme naïf).

Toutefois, une solution existe pour améliorer cela : il suffit de faire tous les calculs modulo un entier q. Chaque fois que le calcul modulo q tombe juste (quand on obtient $p = t_s \mod q$), cela veut dire qu'il est possible que le bon motif soit présent au décalage s, chaque fois que le calcul tombe faux on est certain que le motif n'est pas présent avec un décalage s. Quand le calcul tombe juste on peut alors vérifier que le motif est bien présent avec l'algorithme naïf. On obtient alors l'algorithme suivant :

```
1  void Rabin_Karp(int* T, int n, int* P, int m, int d, int q) {
2    int i,j,h,p,t;
```

```
3   /* on calcule d^(m-1) mod q une fois pour toute */
4   h = ((int) pow(d,m-1)) % q;
5   /* on calcule p mod q*/
6   p=0;
7   for (i=m-1; i>=0; i--) {
8     p = (P[i] + d*p) % q;
9   }
10  /* on calcule t_0 mod q */
11  t=0;
12  for (i=m-1; i>=0; i--) {
13    t = (T[i] + d*t) % q;
14  }
15  /* on teste tous les décalages */
16  for (i=0; i<n-m; i++) {
17    if (t == p) {
18      /* on vérifie avec l'algorithme naïf */
19      for (j=0; j<m; j++) {
20        if (T[j+i] != P[j]) {
21          break;
22        }
23      }
24      if (j == m) {
25        printf("Motif trouvé à la position %d.", i);
26      }
27    }
28    t = (t/d + h*T[i+m]) % q;
29  }
30  for (j=0; j<m; j++) {
31    if (T[j+n-m] != P[j]) {
32      break;
33    }
34  }
35  if (j == m) {
36    printf("Motif trouvé à la position %d.", n-m);
37  }
38 }
```

Si q est bien choisi (typiquement une puissance de 2 plus petite que 2^{32}), la réduction modulo q peut se faire en temps constant, et tous les calcul d'entiers se font aussi en temps constant. La complexité de l'algorithme dépend alors du nombre de « fausse alertes » (les décalages pour lesquels le calcul modulo q est bon, mais le motif n'est pas présent) et du nombre de fois que le motif est réellement présent. En pratique, la complexité sera souvent très proche de l'optimal $\Theta(n+m)$.

7.3 Automates pour la recherche de motifs

Les automates finis sont des objets issus de l'informatique théorique particulièrement adaptés à la résolution de certains problèmes : typiquement, la recherche de motif. Après

Fonction de transition

Q	Σ	Q
S_0	0	S_1
S_0	1	S_0
S_1	0	S_0
S_1	1	S_1

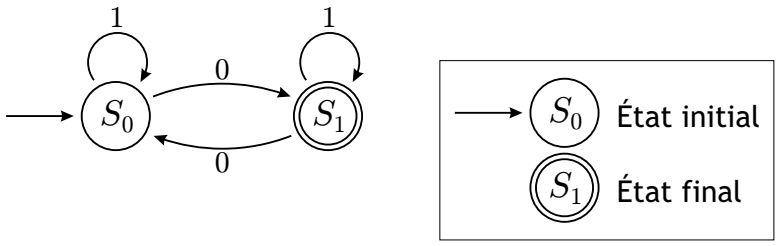


FIGURE 7.1 – *Exemple d'automate et la table de sa fonction de transition.*

avoir défini ce qu'est un automate fini, nous verrons comment obtenir un algorithme de recherche de motif qui aura toujours (pas uniquement dans les cas favorables) une complexité en $\Theta(n + m|\Sigma|)$, donc très proche de l'optimal quand n est grand par rapport à m.

7.3.1 Automates finis

Un *automate fini* (*finite state machine* en anglais) est une machine pouvant se trouver dans un nombre fini de configurations internes ou *états*. L'automate reçoit une suite discrète de signaux, chaque signal provoquant un changement d'état ou *transition*. En ce sens, un automate est un graphe dont les sommets sont les états possibles et les arcs les transitions. Il existe aussi des automates ayant un nombre infini d'états, mais nous ne considérons ici que des automates finis. Dans la suite, le terme automate désignera donc toujours un automate fini.

Plus formellement, un automate M est la donnée de :

- un alphabet fini Σ,
- un ensemble fini non vide d'états Q,
- une fonction de transition $\delta : Q \times \Sigma \to Q$,
- un état de départ noté q_0,
- un ensemble F d'*états finaux*, $F \subseteq Q$.

On notera $M = (\Sigma, Q, \delta, q_0, F)$. Un exemple de tel automate est dessiné en FIGURE 7.1.

Fonctionnement. L'automate fonctionne de la manière suivante : étant dans l'état $q \in Q$ et recevant le signal $\sigma \in \Sigma$, l'automate va passer à l'état $\delta(q, \sigma)$. Notons ε, le mot vide de Σ. On étend δ en une fonction $\bar{\delta} : Q \times \Sigma^* \to Q$ prenant en argument un état et un mot (vide ou non) et retournant un état :

$$\bar{\delta}(q, \varepsilon) = q, \text{ et } \bar{\delta}(q, wa) = \delta(\bar{\delta}(q, w), a),\ q \in Q,\ w \in \Sigma^*,\ a \in \Sigma.$$

On définit alors le langage reconnu par M comme étant le sous-ensemble de Σ^* des mots dont la lecture par M conduit à un état final ; autrement dit

$$L(M) = \{w \in \Sigma^*,\ \bar{\delta}(q_0, w) \in F\}.$$

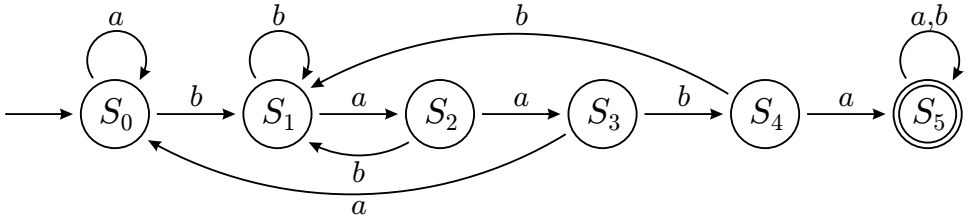


FIGURE 7.2 – *Exemple d'automate reconnaissant le motif "baaba".*

Si $w \in L(M)$, on dit que M *accepte* le mot w. Par exemple, l'automate de la FIGURE 7.1 accepte tous les mots binaires qui contiennent un nombre impaire de 0.

7.3.2 Construction d'un automate pour la recherche de motifs

Nous allons ici montrer que les automates permettent de réaliser efficacement la recherche de motifs dans un texte. Soit donc Σ un alphabet fini, et $w \in \Sigma^*$ avec $|w| = m$ le mot que l'on va chercher. Nous cherchons à déterminer toutes les occurrences de w dans un texte t de longueur n. Pour cela, nous allons construire un automate M reconnaissant le langage $\Sigma^* w \Sigma^*$.

L'ensemble des états de l'automate est $Q = \{S_0, S_1, \ldots, S_m\}$, l'état initial est S_0, et il possède un unique état final $F = \{S_m\}$. Avant de définir la fonction de transition, on introduit la fonction $\lambda : \Sigma^* \to Q$, définie par $\lambda(u) = \max\{0 \leq i \leq m,\ u = \Sigma^* w_i\}$, où w_i est le préfixe de longueur i de w. Ainsi, $\lambda(u)$ est la longueur du plus long préfixe de w qui soit suffixe de u. On cherche à définir δ (et donc $\bar{\delta}$) de façon à réaliser les équivalences suivantes (où $u \in \Sigma^*$) :

1. Pour $0 \leq i < m$, $\bar{\delta}(S_0, u) = S_i$ si et seulement si w *n'apparaît pas* dans u et $\lambda(u) = i$.
2. $\delta(S_0, u) = S_m$ si et seulement si w apparaît dans u.

L'équivalence 2. correspond exactement à la fonction de l'automate M : on doit atteindre l'état final si et seulement si le motif est apparu dans le texte. L'équivalence 1. nous sert à construire l'automate : chaque fois l'on rajoute un symbole x à u, on passe de la case $S_{\lambda(u)}$ à la case $S_{\lambda(ux)}$, et dès que l'on atteint la case S_m on y reste. On obtient alors un automate tel que celui de la FIGURE 7.2.

En pratique, pour construire efficacement cet automate, le plus simple est de procéder par récurrence. Pour construire un automate qui reconnaît le mot wx, on part de l'automate qui reconnaît le mot w de longueur m et on l'étend pour reconnaître wx de longueur $m+1$. Supposons que l'automate M_w reconnaissant w soit construit. Pour construire l'automate M_{wx} on commence par ajouter le nœud final S_{m+1} à M_w. Ne reste plus ensuite qu'à calculer tous les liens partant de la S_m (la dernière case de l'automate M_w, donc l'avant dernière de M_{wx}). Un lien est simple à calculer : la transition en ajoutant x pointe vers S_{m+1}. Les autre liens peuvent être calculés simplement en utilisant l'automate M_w précédemment construit. En effet, si $t \neq x$, alors $\lambda(wt) \leq m$ (en ajoutant une mauvaise transition on ne peut jamais avancer dans l'automate, au mieux on reste sur place). Si on appelle y le

premier symbole de w de telle sorte que $w = yv$, alors on aura aussi $\lambda(wt) = \lambda(vt)$. Or le motif vt est de longueur m et donc $\lambda(vt)$ peut être calculé en utilisant M_w. L'automate M_w termine dans l'état $S_{\lambda(vt)}$ quand on lui donne le motif vt en entrée : il suffit alors de faire pointer le lien indexé par t de S_m au $S_{\lambda(vt)}$ obtenu.

Il y a $|\Sigma| - 1$ tels liens à trouver et le coût pour trouver son extrémité est un parcours d'automate qui a une complexité $\Theta(m)$. Cependant, les $m - 1$ premiers caractères sont communs pour les $|\Sigma| - 1$ parcours à effectuer. Il suffit donc de faire un parcours de $m - 1$ caractères puis de recopier les $|\Sigma| - 1$ transitions issues de l'état que l'on a atteint. De plus, le parcours des $m - 1$ premiers caractères correspond en fait à l'ajout d'un caractère à la suite des $m - 2$ caractères parcourus pendant la construction de M_w. Si l'on a mémorisé l'état auquel aboutit le parcours de ces $m - 2$ caractères, il faut donc $\Theta(|\Sigma|)$ opérations pour construire l'automate M_{wx} à partir de l'automate M_w. Le coût total de la construction d'un automate M_w pour un motif de longueur m est donc $\Theta(m|\Sigma|)$, mais cela demande de programmer comme il faut ! Voici un exemple de code C pour la construction d'un tel automate : `P` est le motif recherché, `m` sa longueur et `sigma` la taille de l'alphabet Σ ; le tableau d qui est retourné correspond à la fonction de transition δ avec `d[i][j]` $= \delta(S_i, j)$.

```
1  int** automata_construction(int* P, int m, int sigma) {
2    int i,j;
3    int etat_mem;
4    /* on initialise le tableau de transitions */
5    int** d;
6    d = (int**) malloc((m+1) * sizeof(int*));
7    for (i=0; i<m+1; i++) {
8      d[i] = (int*) malloc(sigma * sizeof(int));
9    }
10   /* on définit les transitions de l'état 0 */
11   for (i=0; i<sigma; i++) {
12     d[0][i] = 0;
13   }
14   d[0][P[0]] = 1;
15   etat_mem = 0;
16
17   /* on traite les états suivants */
18   for (i=1; i<m+1; i++) {
19     for (j=0; j<sigma; j++) {
20       d[i][j] = d[etat_mem][j];
21     }
22     if (i < m) {
23       d[i][P[i]] = i+1;
24       etat_mem = d[etat_mem][P[i]];
25     }
26   }
27   return d;
28 }
```

Ensuite, pour rechercher un motif w dans un texte t de longueur n il suffit de construire

l'automate M reconnaissant $\Sigma^* w \Sigma^*$, puis de faire entrer les n symboles de t dans M. Le coût total de la recherche est donc $\Theta(n + m|\Sigma|)$.

7.3.3 ⋆ Reconnaissance d'expression régulières

Les automates sont un outil très puissant pour la recherche de motif dans un texte, mais leur utilisation ne se limite pas à la recherche d'un motif fixé à l'intérieur d'un texte. Une *expression régulière* est un motif correspondant à un ensemble de chaînes de caractères. Elle est elle même décrite par une chaîne de caractère suivant une syntaxe bien précise (qui change selon les langages, sinon les choses seraient trop simples !). Reconnaître une expression régulière revient à savoir si le texte d'entrée fait partie des chaînes de caractère qui correspondent à cette expression régulière.

La description complète d'une syntaxe d'expression régulière est un peu complexe donc ne sont donnés ici que quelques exemples d'expressions (et les chaînes de caractères qu'elles représentent), pour se faire une idée de la signification des différents symboles.

- `a` = la chaîne `a` uniquement
- `abc` = le mot `abc` uniquement
- `.` = n'importe quel symbole = les chaînes de longueur 1
- `a*` = toutes les chaînes composées de 0 ou plusieurs `a` (et rien d'autre)
- `a?` = 0 ou une occurrence de `a`
- `a+` = toutes les chaînes composées de au moins un `a` (et rien d'autre)
- `.*a` = toutes les chaînes se terminant par `a`
- `[ac]` = le caractère `a` ou le caractère `c`
- `.*coucou.*` = toutes les chaînes contenant le mot coucou (l'objet de ce chapitre)
- `(bob|love)` = le mot `bob` ou le mot `love`
- `[^ab]` = n'importe quel caractère autre que `a` ou `b`

Les combinaisons de ces différentes expressions permettent de décrire des ensembles de chaînes très complexes et décider si un texte correspond ou pas à une expression peut être difficile. En revanche, il est toujours possible de construire un automate fini (mais des fois très grand !) qui permet de décider en temps linéaire (en la taille du texte) si un texte est accepté par l'expression ou non. La complexité de la construction de cet automate peut en revanche être plus élevée. La FIGURE 7.3 donne deux exemples d'automates et les expressions régulières qu'ils acceptent.

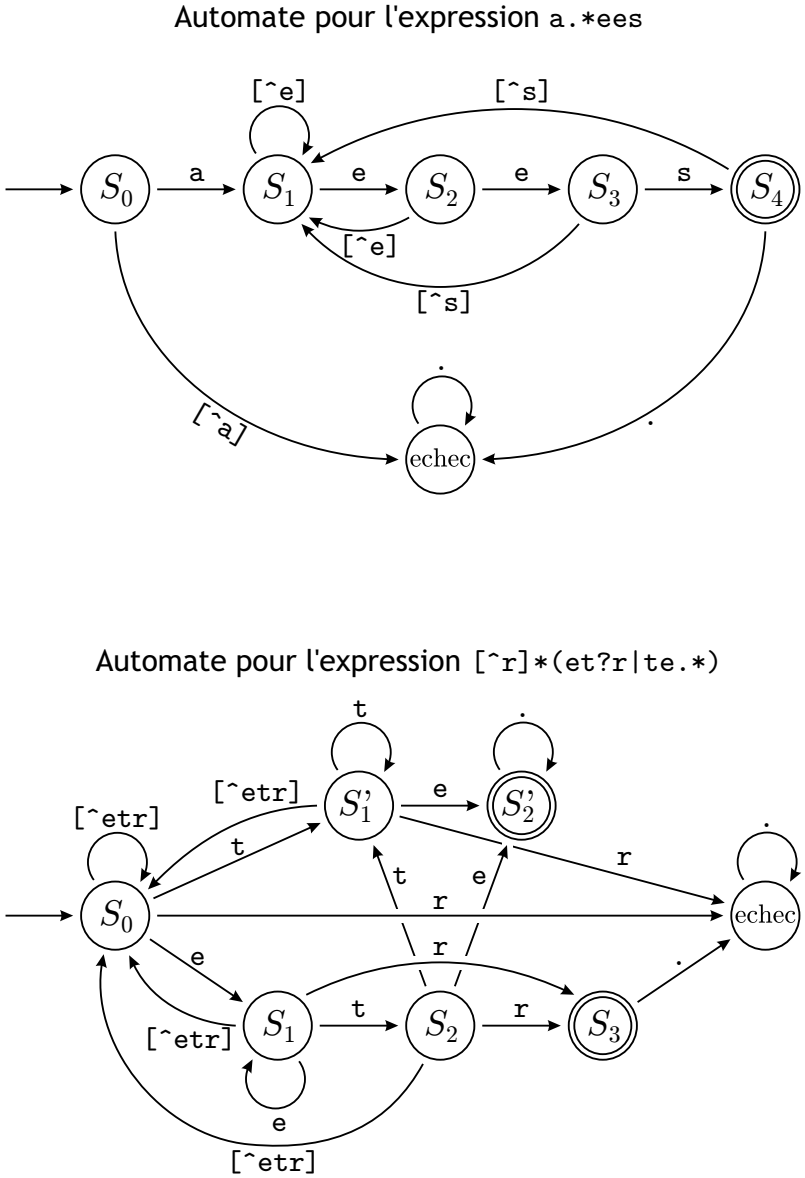


FIGURE 7.3 – *Exemples d'automates reconnaissant des expressions régulières.*