Résumé de Cours
Algorithmique
2001-2002

Xavier Perséguers - Faculté IC - EPFL
xavier.perseguers@epfl.ch

16 septembre 2002

Table des matières

1 Comptages **1**
1.1 Principes . . . 1
1.2 Comptages élémentaires . . . 1
1.2.1 Nombre de mots de longueur k . . . 1
1.2.2 Nombre de permutations de n éléments . . . 1
1.2.3 Nombre de mots de longueur k sans répétition . . . 1
1.2.4 Nombre de mots croissants de longueur k, sans répétition . . . 1
1.2.5 Nombre de mots croissants de longueur k, répétitions possibles 2
1.2.6 Calcul de $S(n,k)$ . . . 2
1.2.7 Calcul de $\sum_{i=1}^{n} i^k$ . . . 2
1.3 Techniques d'énumérations . . . 3
1.3.1 Vecteurs binaires de longueur n . . . 3
1.3.2 $n!$ permutations de n objets . . . 4
1.4 Dénombrement . . . 4
1.4.1 Méthode de Polya . . . 4
1.4.2 Exemples d'application . . . 5

2 Relations de récurrence **7**
2.1 Récurrences linéaires homogènes à coefficients constants . . . 7
2.2 Récurrences linéaires non homogènes à coefficients constants . . . 8
2.2.1 Transformation en une équation homogène . . . 9

3 Complexité **11**
3.1 La notation *de l'ordre de* ou notation *grand O* . . . 11
3.2 La notation Omega (Ω) . . . 11
3.3 La notation Thêta (Θ) . . . 12

4 Algorithmes **13**
4.1 Algorithmes de tri . . . 13
4.1.1 Tri par sélection (*Selection Sort*) . . . 13
4.1.2 Tri par insertion (*Insertion Sort*) . . . 14
4.1.3 Tri par bulles (*Bubble Sort*) . . . 15
4.1.4 Tri Shell (*Shell Sort*) . . . 16
4.1.5 Quicksort . . . 17
4.1.6 Tri par fusion (*Merge Sort*) . . . 18
4.1.7 Tri par tas (*Heap Sort*) . . . 19
4.2 Multiplication de grands entiers . . . 23

5 Graphes **25**
5.1 Introduction 25
5.1.1 Définitions 25
5.2 Les arbres 26
5.2.1 Arbres complets (ou parfaits) partiellement ordonnés 26
5.3 Représentation des graphes 26
5.3.1 Représentation matricielle 27
5.3.2 Représentation par une table de successeurs 27
5.4 Parcours des graphes 28
5.4.1 Parcours en profondeur 28
5.4.2 Tri topologique 28
5.4.3 Arbre maximal de poids minimal 29
5.4.4 Le plus court chemin (Algorithme de Dijkstra) 30
5.5 Composantes connexes 31
5.5.1 2-connexité 31
5.5.2 Composantes fortement connexes 32

6 Exercices corrigés **35**
6.1 Propédeutique II - 12 juillet 2002 35
6.2 Test d'algorithmique - 7 juin 2002 38

Chapitre 1

Comptages

1.1 Principes

Principe d'égalité : S'il existe une bijection de A sur B, alors $|A| = |B|$.

Principe d'addition : Si A et B sont disjoints, alors $|A \cup B| = |A| + |B|$.

Principe de multiplication : On a $|A \times B| = |A| \cdot |B|$.

1.2 Comptages élémentaires

Soit $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ un alphabet ordonné de n lettres. Un mot est une suite finie $x = \sigma_{i1}\sigma_{i2}\ldots\sigma_{ik}$ de lettres de Σ. $|x| = \lg(x) = k$ est la longueur du mot.

$$\Sigma^k = \{x \mid \lg(x) = k\}$$

ie. l'ensemble de mots de longueur k.

1.2.1 Nombre de mots de longueur k

$$|\Sigma^k| = |\Sigma|^k = n^k$$

1.2.2 Nombre de permutations de n éléments

$$P(n) = n! = n(n-1)(n-2)\ldots(2)(1)$$

1.2.3 Nombre de mots de longueur k sans répétition

$$A_k^n = \frac{P(n)}{(n-k)!} = \frac{n!}{(n-k)!} = n(n-1)(n-2)\ldots(n-k+1)$$

1.2.4 Nombre de mots croissants de longueur k, sans répétition

Nombre de façons de choisir k lettres parmi n :

$$C_k^n = \binom{n}{k} = \frac{A_k^n}{k!} = \frac{n!}{(n-k)!\,k!} = \frac{n(n-1)\ldots(n-k+1)}{1 \cdot 2 \cdot \ldots \cdot k}$$

Notation : Les manuels français écrivent C_n^k notre définition de C_k^n. Par contre tout le monde s'accorde à utiliser la notation simplifiée $\binom{n}{k}$.

1.2.5 Nombre de mots croissants de longueur k, répétitions possibles

On représente de tels mots à l'aide de vecteurs binaires. Un "1" indique un changement de lettre. Les "0" entre deux "1" indiquent le nombre de fois que l'on prend une lettre.

Dans de tels vecteurs, il y a $(n-1)$ fois "1" et k fois "0". Ces vecteurs ayant $n-1+k$ composantes, le nombre de mots cherchés est donc :

$$C_k^{n-1+k} = \binom{n-1+k}{k} = \frac{(n-1+k)!}{(n-1)!\,k!}$$

Exemple

$\Sigma = \{a,b\}$, $k = 3$

On a donc 4 mots croissants : *aaa*, *aab*, *abb* et *bbb*.

$$\begin{array}{lcl} aaa & \rightarrow & 0001 \\ aab & \rightarrow & 0010 \\ abb & \rightarrow & 0100 \\ bbb & \rightarrow & 1000 \end{array}$$

1.2.6 Calcul de $S(n,k)$

Soit E un ensemble de n éléments. S_n^k est le nombre de partitions de E en k sous-ensembles non vides.

Soit x un élément quelconque de E, et S, un sous-ensemble d'une partition de E. Deux cas de figure peuvent se présenter :

1. x est le seul élément de S
   Il y a $S(n-1,k-1)$ partitions de ce type ;
2. S contient d'autres éléments que x
   Il y a $k \cdot S(n-1,k)$ partitions de ce type.

On a donc la formule de récursion suivante :

$$\left\{ \begin{array}{lcl} S(n,k) & = & S(n-1,k-1) + k \cdot S(n-1,k) \\ S(n,n) & = & S(n,1) = 1 \end{array} \right.$$

1.2.7 Calcul de $\sum_{i=1}^{n} i^k$

Cette expression est également notée S_n^k.

$$\begin{array}{lcl} S_n^0 & = & n \\ S_n^1 & = & \dfrac{n(n+1)}{2} \\ S_n^2 & = & \dfrac{n(n+1)(2n+1)}{6} \end{array}$$

Rappel : $(a+b)^k = \sum_{i=0}^{k} \binom{k}{i} a^{k-i} b^i$

$$
\begin{array}{rcl}
(0+1)^{k+1} & = & \binom{k+1}{0}0^{k+1} + \binom{k+1}{1}0^k + \binom{k+1}{2}0^{k-1} + \ldots + \binom{k+1}{k+1}0^0 \\
(1+1)^{k+1} & = & \binom{k+1}{0}1^{k+1} + \binom{k+1}{1}1^k + \binom{k+1}{2}1^{k-1} + \ldots + \binom{k+1}{k+1}1^0 \\
\vdots & & \\
(n+1)^{k+1} & = & \binom{k+1}{0}n^{k+1} + \binom{k+1}{1}n^k + \binom{k+1}{2}n^{k-1} + \ldots + \binom{k+1}{k+1}n^0 \\
\hline
S_{n+1}^{k+1} & = & S_n^{k+1} + (k+1)\,S_n^k + \binom{k+1}{n}S_n^{k-1} + \ldots + (n+1)
\end{array}
$$

$$\Rightarrow S_{n+1}^{k+1} - S_n^{k+1} = (n+1)^{k+1} = (k+1)S_n^k + \sum_{i=2}^{k}\binom{k+1}{i}S_n^{k-i+1} + (n+1)$$

On a donc :

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

1.3 Techniques d'énumérations

1.3.1 Vecteurs binaires de longueur n

Considérons les vecteurs binaires de longueur n ayant k fois "1" et $(n-k)$ fois "0". Application possible : énumération des C_k^n mots croissants de longueur k, sans répétition.

Algorithme

1. Partir de $(\underbrace{11\ldots1}_{k}\underbrace{00\ldots0}_{n-k})$
2. Répéter
   (a) Choisir le "1" le plus à droite qui a un "0" à sa droite.
   S'il n'en existe pas : STOP ;
   (b) Déplacer ce "1" d'une case à droite en le permutant avec le "0" qui le suit.
   Coller les "1" plus à droite du "1" déplacé contre le "1" déplacé.

Exemple

$n = 5$, $k = 3$

$$
\begin{array}{ll}
C_1^3 \text{ mots avec } \sigma_1, \text{ avec } \sigma_2 & \left\{ \begin{array}{ccccc} 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{array} \right\} C_2^4 \text{ mots avec } \sigma_1 \\
C_2^3 \text{ mots avec } \sigma_1, \text{ sans } \sigma_2 & \left\{ \begin{array}{ccccc} 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \end{array} \right\} \\
C_2^3 \text{ mots sans } \sigma_1, \text{ avec } \sigma_2 & \left\{ \begin{array}{ccccc} 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{array} \right\} C_3^4 \text{ mots sans } \sigma_1 \\
C_3^3 \text{ mots sans } \sigma_1, \text{ sans } \sigma_2 & \left\{ \begin{array}{ccccc} 0 & 0 & 1 & 1 & 1 \end{array} \right\}
\end{array}
$$

1.3.2 $n!$ permutations de n objets

Soit un vecteur à n composantes entières. A chaque composante on associe une flèche qui pointe à gauche ou à droite.

Définition

Une composante est dite **mobile** si sa flèche pointe vers un entier de plus petite valeur.

Algorithme

1. Débuter avec $\overleftarrow{1}\,\overleftarrow{2}\,\ldots\,\overleftarrow{n}$;
2. S'il n'y a pas de composante mobile : STOP ;
3. Soit m la plus grande composante mobile ;
4. Permuter m avec la composante vers laquelle sa flèche pointe ;
5. Changer la direction des flèches sur toutes les composantes de valeur $> m$ et aller à 2.

1.4 Dénombrement

1.4.1 Méthode de Polya

Soit $F = \{f : X \to Y\}$, et G un sous-ensemble du groupe symétrique $S_{|x|}$.
Définissons un poids $\omega(f)$ associé à toute fonction $f \in F$ de la manière suivante :

$$\omega(f) = \prod_{x \in X} f(x)$$

Exemple 1

$$f = \boxed{\begin{array}{cc} B & N \\ B & B \end{array}} \Rightarrow \omega(f) = B^3 N$$

Exemple 2

$$f = (0,0,1) = (\text{faux,faux,vrai}) \Rightarrow \omega(f) = \text{faux}^2\text{vrai}$$

Notation

Soit σ une permutation. On note $n_i(\sigma)$ le nombre de cycles de longueur i dans σ.

Définition

On appelle **Polynôme de Polya**, le polynôme suivant :

$$P_G(a_1, a_2, \ldots, a_{|x|}) = \frac{1}{|G|}\left(\sum_{\sigma \in G} a_1^{n_1(\sigma)} a_2^{n_2(\sigma)} a_3^{n_3(\sigma)} \ldots a_{|x|}^{n_{|x|}(\sigma)}\right)$$

1.4.2 Exemples d'application

Exemple 1

Considérons 3 boules B, N_1 et N_2. On désire ranger ces boules dans 3 tirois a, b et c. Les boules N_1 et N_2 sont indistinguables. Combien y a-t-il de rangements de boules dans les tiroirs qui soient distinguables ?

$F = \{\text{rangements possibles}\} = \{f : \{B,N_1,N_2\} \to \{a,b,c\}\}$
$G = \{(1)(2)(3), (1)(23)\}$ car 2 rangements sont équivalents s'ils sont identiques ou si l'on peut obtenir un rangement à partir de l'autre en permutant les rangements des boules N_1 et N_2.

$$P_G(a_1, a_2, a_3) = \frac{a_1^3 + a_1^1 a_2^2}{2} \Rightarrow |F/\sim| = \frac{1}{2}(3^3 + 3^2) = 18$$

Exemple 2

Combien y a-t-il de mots croissants de longueur k dans un alphabet de n lettres ?

Rappel : la réponse est C_k^{n-1+k} ou $\binom{n-1+k}{k}$

Considérons le cas où $n = 4$ et $k = 3$. On a donc $\binom{n-1+k}{k} = \binom{6}{3} = \frac{6!}{3!\,3!} = 20$ mots. On peut faire le même calcul par Polya. En effet, un mot peut être vu comme une affectation des lettres a, b, c ou d aux trois positions du mot. L'ensemble des mots possibles est donc un ensemble F de fonctions avec :

$$F = \{f : \{1,2,3\} \to \{a,b,c,d\}\}$$

Deux mots sont équivalents si l'on peut obtenir l'un à partir de l'autre en permutant les lettres (seul le mot croissant nous intéresse ; c'est le représentant de la classe d'équivalence). On a donc :

$$G = S_3 = \{(1)(2)(3), (12)(3), (13)(2), (1)(23), (123), (132)\}$$

On en déduit que

$$P_G(a_1, a_2, a_3) = \frac{a_1^3 + 2a_3^1 + 3a_1^1 a_2^1}{6}$$

Donc

$$|F/\sim| = \frac{1}{6}(4^3 + 2 \cdot 4 + 3 \cdot 4 \cdot 4) = 20$$

Exemple 3

Dans les statistiques d'une entreprise de 800 personnes, on relève 424 personnes mariées, 300 hommes dont 166 sont mariés, 552 personnes syndiquées parmi lesquelles on trouve 188 hommes, dont 144 mariés, et 208 personnes mariées. En étudiant le nombre de femmes célibataires et non syndiquées, que concluez-vous sur les statistiques de l'entreprise ?

$$\begin{array}{lcll} E & = & 800 & - \text{ Nombre de personnes dans l'entreprise} \\ M & = & 424 & - \text{ Nombre de personnes mariées} \\ H & = & 300 & - \text{ Hommes dans l'entreprise} \\ H_M & = & 166 & - \text{ Hommes mariés} \\ S & = & 552 & - \text{ Personnes syndiquées} \\ H_S & = & 188 & - \text{ Hommes syndiqués} \\ H_{SM} & = & 144 & - \text{ Hommes syndiqués et mariés} \\ S_M & = & 208 & - \text{ Personnes syndiquées et mariées} \end{array}$$

Calcul du nombre de femmes non syndiquées et célibataires :

$$\begin{aligned} |F_{\overline{SM}}| &= |F_{\overline{M}}| - |F_{S\overline{M}}| \\ |F_{\overline{M}}| &= |\overline{M}| - |H_{\overline{M}}| \\ |F_{S\overline{M}}| &= |F_S| - |F_{SM}| \end{aligned}$$

$$\begin{aligned} |\overline{M}| - |H_{\overline{M}}| &= (|E| - |M|) - (|H| - |H_M|) \\ &= (800 - 424) - (300 - 166) \\ &= 376 - 134 \\ |F_{\overline{M}}| &= 242 \end{aligned}$$

$$\begin{aligned} |F_S| - |F_{SM}| &= (|S| - |H_S|) - (|S_M| - |H_{SM}|) \\ &= (552 - 188) - (208 - 144) \\ &= 364 - 64 \\ |F_{S\overline{M}}| &= 300 \end{aligned}$$

$$\begin{aligned} |F_{\overline{M}}| - |F_{S\overline{M}}| &= 242 - 300 \\ |F_{\overline{SM}}| &= \mathbf{-58} \end{aligned}$$

On peut donc en déduire que les statistiques mériteraient d'être refaites.

Chapitre 2

Relations de récurrence

2.1 Récurrences linéaires homogènes à coefficients constants

Une formule de récurrence est dite *linéaire homogène à coefficients constants* si elle est de la forme :

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_i t_{n-i} + \ldots + a_k t_{n-k} = 0.$$

Exemple 1

La suite de Fibonnacci se définit comme suit :

$$\begin{cases} f_0 = 0.\ f_1 = 1 \\ f_n = f_{n-1} + f_{n-2} \quad \text{si } n \geq 2 \end{cases}$$

Celle-ci est une récurrence linéaire homogène à coefficients constants. Pour le voir, écrire $f_n - f_{n-1} - f_{n-2} = 0$. Dans ce cas, on a $k = 2$, $a_0 = 1$ et $a_1 = a_2 = -1$.

Le polynôme caractéristique est :

$$x^2 - x - 1$$

dont les racines sont $r_1 = \frac{1+\sqrt{5}}{2}$ et $r_2 = \frac{1-\sqrt{5}}{2}$.

La forme générale des solutions est donc : $f_n = c_1 r_1^n + c_2 r_2^n$. Si $n = 0$, on sait que $f_0 = 0$ et donc $c_1 + c_2 = 0$. Pour $n = 1$, on obtient $r_1 c_1 + r_2 c_2 = 1$ et donc $c_1 = \frac{1}{\sqrt{5}}$ et $c_1 = -\frac{1}{\sqrt{5}}$, ce qui donne :

$$f_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right].$$

Formule connue sous le nom de *formule de de Moivre*.

Exemple 2

$$t_n = \begin{cases} n & \text{si } n \in \{0,1,2\} \\ 5t_{n-1} - 8t_{n-2} + 4t_{n-3} & \text{si } n \geq 3 \end{cases}$$

Réécrivons la récurrence sous la forme :

$$t_n - 5t_{n-1} + 8t_{n-2} - 4t_{n-3} = 0$$

Son polynôme caractéristique est :

$$x^3 - 5x^2 + 8x - 4 = (x-1)(x-2)^2$$

Ses racines sont donc $r_1 = 1$ de multiplicité $m_1 = 1$ et $r_2 = 2$ de multiplicité $m_2 = 2$. La solution générale est donc :

$$t_n = c_1 1^n + c_2 2^n + c_3 n 2^n$$

Les conditions initiales donnent :

$$\begin{array}{lcll} c_1 + c_2 & = & 0 & \text{pour } n = 0 \\ c_1 + 2c_2 + 2c_3 & = & 1 & \text{pour } n = 1 \\ c_1 + 4c_2 + 8c_3 & = & 2 & \text{pour } n = 1 \end{array}$$

Ce qui donne $c_1 = -2$, $c_2 = 2$ et $c_3 = -1/2$. Donc

$$t_n = 2^{n+1} - n2^{n-1} - 2.$$

2.2 Récurrences linéaires non homogènes à coefficients constants

Considérons l'équation suivante :

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_i t_{n-i} + \ldots + a_k t_{n-k} = b^n p(n). \tag{2.1}$$

où

- b est une constante ;
- $p(n)$ est un polynôme en n de degré d.

Théorème

L'équation (2.1) a pour polynôme caractéristique :

$$(a_0 x^k + a_1 x^{k-1} + \ldots + a_k)(x-b)^{d+1}$$

Généralisation

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = b_1^n\, p_1(n) + b_2^n\, p_2(n) + \ldots + b_l^n\, p_l(n) \tag{2.2}$$

Les $p_i(n)$ sont des polynômes en n de degré i.
L'équation (2.2) a pour polynôme caractéristique :

$$(a_0 x^k + a_1 x^{k-1} + \ldots + a_k)(x-b_1)^{d_1+1}(x-b_2)^{d_2+1}\ldots(x-b_l)^{d_l+1}$$

2.2.1 Transformation en une équation homogène

Comme on sait, d'après la section précédente, résoudre une équation de récurrence homogène, transformons l'équation non homogène.

Exemple 1

$$t_n - 2t_{n-1} = 3^n \tag{2.3}$$

Ici, $b = 3$ et $p(n) = 1$.

- Multiplions (2.3) par 3, ce qui donne $3t_n - 6t_{n-1} = 3^{n+1}$;
- Remplaçons n par $n-1$, ce qui donne

$$3t_{n-1} - 6t_{n-2} = 3^n ; \tag{2.4}$$

- Soustrayons (2.4) à (2.3) pour obtenir

$$t_n - 5t_{n-1} + 6t_{n-2} = 0. \tag{2.5}$$

On sait résoudre (2.5) par la méthode de la section précédente. Sa fonction caractéristique est $x^2 - 5x + 6 = (x-2)(x-3)$. Donc deux racines distinctes et donc toutes les solutions sont de la forme :

$$t_n = c_1 2^n + c_2 3^n.$$

Exemple 2

$$\begin{cases} t_0 = 0 \\ t_n = 2t_{n-1} + n + 2^n \quad \text{si } n \geq 1 \end{cases}$$

s'écrit $t_n - 2t_{n-1} = n + 2^n$. Donc $b_1 = 1$, $p_1(n) = n$, $b_2 = 2$, $p_2(n) = 1$, $d_1 = 1$ et $d_2 = 0$.

Le polynôme caractéristique est :

$$(x-2)(x-1)^2(x-2) = (x-2)^2(x-1)^2$$

et par conséquent toutes les solutions de notre équation sont de la forme :

$$t_n = c_1 1^n + c_2 n 1^n + c_3 2^n + c_4 n 2^n. \tag{2.6}$$

Si l'on ne s'intéresse qu'au comportement asymptotique, ce qui est en général le cas en algorithmique, il suffit de savoir si c_4 est bien strictement positif. Pour cela, substituons l'équation (2.6) dans l'équation qui définit la récurrence. On obtient :

$$n + 2^n = (2c_2 - c_1) - c_2 n + c_4 2^n.$$

Et donc en égalant les coefficients de 2^n, on obtient que $c_4 = 1$. Si l'on veut vraiment les autres coefficients, on peut les obtenir en écrivant les équations données par les trois ou quatre premières valeurs de la suite, selon que l'on utilise le fait que l'on connaît c_4 ou non. On obtient :

$$t_n = n2^n + 2^{n+1} - n - 2.$$

Chapitre 3

Complexité

3.1 La notation *de l'ordre de* ou notation *grand O*

Soit $f : \mathbb{N} \to \mathbb{R}_+$ une fonction quelconque. On note $O(f(n))$ l'ensemble de toutes les fonctions $t : \mathbb{N} \to \mathbb{R}_+$ telles que :

$$\exists\, n_0 \in \mathbb{N}, C \in \mathbb{R}_+^*(\forall n \geq n_0(t(n) \leq C\ f(n)))$$

La règle du maximum

Avec $f, g : \mathbb{N} \to \mathbb{R}_+$, $O(f(n) + g(n)) = O(\max(f(n), g(n)))$.

Théorème 1

Si

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R}_+ \text{ alors } f(n) \in O(g(n)) \text{ et } g(n) \in O(f(n)) \tag{3.1}$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \text{ alors } g(n) \in O(f(n)) \text{ mais } f(n) \notin O(g(n)) \tag{3.2}$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty \text{ alors } g(n) \in O(f(n)) \text{ mais } f(n) \notin O(g(n)) \tag{3.3}$$

3.2 La notation Omega (Ω)

Un algorithme de l'ordre de $n \log n$ est dans $O(n^2)$, dans $O(n^3)$, ... Ceci est dû au fait que la notation O a pour but de donner un majorant, pas nécessairement le plus petit majorant. Ceci crée le besoin d'un minorant.

Soient à nouveau deux fonctions $f, t : \mathbb{N} \to \mathbb{R}_+$. On dit que $t(n)$ est dans Omega de $f(n)$, noté $t(n) \in \Omega(f(n))$, si pour n assez grand, $t(n)$ est minoré par un multiple de $f(n)$. Mathématiquement cela donne :

$$\exists\, n_0 \in \mathbb{N}, C \in \mathbb{R}_+(\forall n \geq n_0(t(n) \geq C\ f(n)))$$

3.3 La notation Thêta (Θ)

Une fonction $t(n)$ est de l'ordre exacte de $f(n)$ si $t(n) \in \Theta(f(n))$, où

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n)).$$

Théorème 2

Si

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R}_+ \text{ alors } f(n) \in \Theta(g(n)) \tag{3.4}$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \text{ alors } f(n) \in \Theta(g(n)) \text{ mais } f(n) \notin \Theta(g(n)) \tag{3.5}$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty \text{ alors } f(n) \in \Theta(g(n)) \text{ mais } g(n) \notin \Theta(f(n)) \tag{3.6}$$

Chapitre 4

Algorithmes

4.1 Algorithmes de tri

Les algorithmes proposés dans cette section pourront être utilisés comme brique de base pour d'autres algorithmes spécifiques aux données à traiter. L'algorithme de tri Shell mis à part, ces méthodes ne sont pas très efficaces pour de très grandes populations d'éléments.

On distingue deux sortes de tris :

Tri interne. Tous les éléments sont stockés en mémoire et le tri s'opère donc exclusivement entre la mémoire de données, la mémoire d'instructions et le processeur.

Tri externe. Si les éléments à trier sont trop volumineux, la mémoire n'est pas en mesure de stocker tous les éléments. Il est aussi probable que seule une petite partie de la description complète d'un enregistrement soit utile pour le tri. Dans ce cas, seule cette information, ou une partie de cette information, sera chargée en mémoire. Le tri s'opérera alors conjointement avec un support de stockage externe, comme une bande magnétique.

4.1.1 Tri par sélection (*Selection Sort*)

Le tri par sélection parcourt les éléments à la recherche du plus petit. Une fois trouvé, il déplace en début de liste, à la fin de la partie triée. Il recommence à chercher le plus petit élément dans la partie non encore triée.

FIG. 4.1 – *Tri par sélection*

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

Comparaisons : $O(n^2/2)$.

Échanges : $O(n)$.

Preuve : On cherche le plus petit de n puis de $n-1$ ...de j, de $j-1$ ...de 2 éléments d'où $(n-1)+(n-2)+\ldots+j-1+\ldots+1$ donc $n(n-1)/2$ comparaisons. Sauf si le plus petit élément parmi ceux qui restent est le premier de ceux-ci, on fait un échange chaque fois, donc en tout au maximum $n-1$ échanges.

Stabilité : Le tri par sélection n'est pas stable. En triant le tableau suivant selon les lettres $[B_1, B_2, A_3]$, on obtient $[A_3, B_2, B_1]$ donc l'ordre des B a été inversé.

Listing 4.1: *Tri par sélection*

```
TriSelection(int t[], int N)
{
   int i, j, min, q;
   for (i = 1; i < N; i++)
   {
      min = i;
      for (j = i + 1; j <= N; j++)
         if (t[j] < t[min]) min = j;
      q = t[min]; t[min] = t[i]; t[i] = q;
   }
}
```

4.1.2 Tri par insertion (*Insertion Sort*)

Le tri par insertion prend le premier élément de la partie non triée et l'insère à la bonne place dans la partie triée. Ensuite il recommence jusqu'à n'avoir plus aucun élément dans la partie non triée.

FIG. 4.2 – *Tri par insertion*

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

Comparaisons : $O(n^2/4)$. Dans le pire des cas $O(n^2/2)$.

Échanges : $O(n^2/4)$. Dans le pire des cas $O(n^2/2)$.

Remarque : Linéaire, $O(n)$, dans le cas de fichiers triés ou presque triés, *ie.* dans le meilleur des cas.

Preuve : Quand il y a déjà i éléments classés, par un raisonnement analogue à celui fait dans la recherche séquentielle, on fera en moyenne $i/2$ comparaisons pour trouver sa place. On a donc au total $\sum_{i=2}^{n}(i/2) \approx n(n+1)/4$. Dans le pire des cas, on fait $\sum_{i=2}^{n}(i) \approx n(n+1)/2$. Le nombre d'échanges est égal au nombre de comparaisons.

Stabilité : Le tri par insertion est stable puisqu'un élément est toujours inséré *après* le "premier élément" (compté en arrière depuis l'élément à insérer) qui n'est pas plus grand. Donc l'ordre de deux éléments de la même valeur n'est pas changé.

Listing 4.2: *Tri par insertion*

```
TriInsertion(int t[], int N)
{
   int i, j, v;
   for (i = 2; j <= N; i++)
   {
      v = t[i];
      j = i;
      while (t[j - 1] > v)
         { t[j] = t[j - 1]; j--; }
      t[j] = v;
   }
}
```

4.1.3 Tri par bulles (*Bubble Sort*)

Le tri par bulles consiste à parcourir plusieurs fois la liste à chaque fois depuis le début, à prendre un élément et à le déplacer vers la fin de la liste en l'intervertissant avec son voisin[1] jusqu'à ce que celui-ci soit à sa place définitive. La partie triée se construit donc en fin de liste.

FIG. 4.3 – *Tri par bulles*

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

Comparaisons : $O(n^2/2)$, en moyenne et dans le pire des cas.

1. Certains appellent cette opération *Bubble*, mais le nom semble plutôt venir du fait que les gros éléments se déplacent petit à petit vers le haut tandis que les petits se déplacent vers le bas, à l'instar des bulles.

Échanges : $O(n^2/2)$, en moyenne et dans le pire des cas.

Preuve : Le fichier trié en sens inverse donne le pire cas pour le tri à bulles et le raisonnement est comme pour le tri par insertion. Pour un fichier trié, on ne fait qu'une passe. La démonstration de la complexité moyenne est très complexe.

Remarque : Suivant l'implémentation, on peut avoir $O(n^2)$ ou $O(n)$ dans le meilleur des cas, *ie.* avec des fichiers presque triés.

Stabilité : Le tri par bulles est stable. Deux éléments de la même valeur ne seront jamais échangés, donc l'ordre est préservé.

Listing 4.3: *Tri par bulles*

```
TriBulles(int t[], int N)
{
   int i, j, v;
   for (i = N; i >= 1; i--)
      for (j = 2; j <= i; j++)
         if (t[j - 1] > t[j])
            { v = t[j - 1]; t[j - 1] = t[j]; t[j] = v; }
}
```

4.1.4 Tri Shell (*Shell Sort*)

Le tri Shell améliore le tri par insertion en ne portant pas les échanges uniquement sur des éléments adjacents. L'idée est de réorganiser le fichier de façon à ce que les éléments qui sont loin de leur position finale puissent arriver à cette position sans trop de comparaisons et d'échanges. Pour cela, on trie les sous-suites d'éléments situés à une distance h les uns des autres. Puis on diminue h jusqu'à ce qu'il vaille 1. On a donc à chaque itération des sous-suites triées entrelacées.

FIG. 4.4 – *Tri Shell*

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

Comparaisons : En utilisant la suite (en ordre inversé) 1,23,77,281,1073,... $4^{j+1} + 3{,}2^j + 1$, $O(n^{4/3})$.

Listing 4.4: *Tri Shell*

```
TriShell(int t[], int N)
{
   int i, j, h, v;
   for (h = 1; h <= N / 9; h = 3 * h + 1);
```

```
    for (; h > 0; h /= 3);
    for (i = h + 1; i <= N; i++)
    {
        v = t[i]; j = i;
        while (j > h && t[j - h] > v)
            { t[j] = t[j - h]; j -= h; }
        t[j] = v;
    }
}
```

Ce programme utilise la suite ..., 1093, 364, 121, 40, 13, 4, 1.

4.1.5 Quicksort

Algorithme de type *Divide-and-Conquer* (*ie.* Diviser pour régner), Quicksort opère selon le principe qu'il est plus simple de trier une petite liste qu'une grande. La première étape consiste donc à partager récursivement la liste à trier en deux sous-listes jusqu'à obtenir un ensemble d'éléments à trier contenus dans une liste de taille raisonnable. A ce moment, toutes ces listes sont triées puis combinées deux à deux pour retrouver la liste initiale, mais triée. Quicksort porte le nom de *tri par segmentation* dans certains ouvrages.

Dans l'exemple, les parties en gras donnent l'état de la mémoire à la fin de chaque passage dans la fonction de tri avec partitionnement, juste avant l'appel récursif sur les deux sous-listes, à gauche et à droite du pivot. Le pivot, par souci de simplicité, a été choisi comme étant le dernier élément de la liste à trier.

FIG. 4.5 – *Quicksort*

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

Comparaisons : En moyenne et dans le meilleur des cas $O(n \log n)$. Dans le pire des cas (liste triée dans un sens ou dans l'autre), $O(n^2)$.

Partitionnement par la médiane : La cause de mauvaises performances est due à un mauvais choix de pivot, ce qui arrive s'il y a trop de structure dans le fichier. Pour cela, une première façon d'y remédier est de tirer ce pivot au hasard. L'utilisation d'un générateur de nombres aléatoires est un peu trop coûteuse dans le cadre d'un algorithme dont on veut optimiser au maximum les performances. Une solution qui donne de très bons résultats est la suivante : on compare les éléments situés dans la première case, la dernière case et la case du milieu. On les trie à l'aide de quelques lignes de code spécifiques. Finalement, on place la médiane dans l'avant-dernière case. Elle servira de pivot et on ne partitionne que sur $T[g+1] \dots T[d-2]$.

Cette méthode s'appelle méthode de partitionnement par la *médiane de trois*. On peut prendre plus de trois, par exemple 5 éléments, mais les gains sont négligables.

Listing 4.5: *Structure récursive du tri rapide*

```
TriRapide(int t[], int g, int d)
{
   int i;
   if (d > g)
   {
      i = Partionnement(t, g, d);
      TriRapide(t, g, i - 1);
      TriRapide(t, i + 1, d);
   }
}
```

Listing 4.6: *Tri rapide et Partitionnement*

```
TriRapide(int t[], int g, int d)
{
   int i, j, u, v;
   if (d > g)
   {
      v = t[d]; i = g - 1; j = d;
      for (;;)
      {
         while (t[++i] < v);
         while (t[--j] > v);
         if (i >= j) break;
         u = t[i]; t[i] = t[j]; t[j] = u;
      }
      u = t[i]; t[i] = t[d]; t[d] = u;
      TriRapide(t, g, i - 1);
      TriRapide(t, i + 1, d);
   }
}
```

Listing 4.7: *Sélection du k-ième élément*

```
SelectionK(int t[], int g, int d, int k)
{
   int i;
   if (d > g)
   {
      i = Partionnement(t, g, d);
      if (i > g + k - 1)
         SelectionK(t, g, i - 1, k);
      else if (i < g + k - 1)
         SelectionK(t, i + 1, d, k - i);
   }
}
```

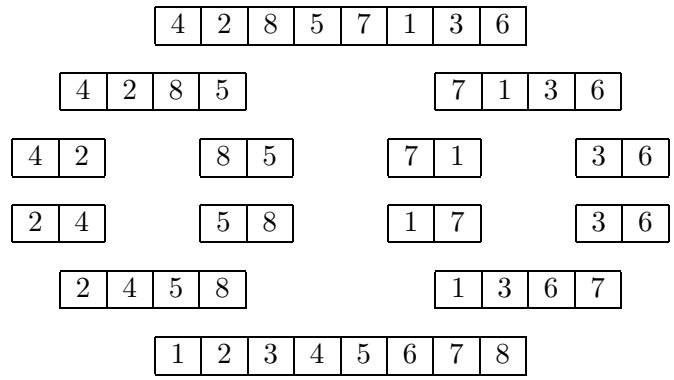
Par exemple, l'appel `SelectionK(t, 1, N, (N+1)/2)` partitionne le tableau `t` sur sa valeur médiane.

4.1.6 Tri par fusion (*Merge Sort*)

Très souvent les gros fichiers sont fabriqués à partir de fichiers plus petits qui sont déjà triés. On se place ici dans le cas de deux fichiers triés dans l'ordre croissant contenus dans deux tableaux S et T. On veut l'ensemble des éléments triés dans un troisième tableau U. Malheureusement il est très difficile de se passer de ce troisième tableau, et c'est le point faible de cette méthode. A noter que les limitations dues à cet inconvénient sont atténuées aujourd'hui avec la croissance des tailles de mémoire.

Comparaisons : $O(n \log n)$. La version itérative fait au plus $\log n$ passes et chaque passe nécessite au plus n comparaisons.

FIG. 4.6 – *Tri par fusion*



Preuve : Pour la version récursive, on a la relation $K(n) = 2K(n/2) + cn$ avec $K(1) = 0$. On sait que la solution est en $O(n\log(n))$. Pour la version itérative, on fait au plus $\log(n)$ passes et chaque passe nécessite au plus n comparaisons, d'où le résultat.

Place mémoire supplémentaire : $O(n)$.

Stabilité : Le tri par fusion est stable et c'est facile à voir.

Remarque : Le tri par fusion n'est pas sensible à l'ordre des éléments. Ce tri fait le même nombre de comparaisons quelle que soit la liste à trier, celles-ci étant faites dans la fusion.

Listing 4.8: *Fusion de deux listes chaînées*

```
struct noeud
{
   int cle;
   struct noeud *suivant;
}
struct noeud *z;
struct noeud *Fusion (struct noeud *s, struct noeud *t)
{
   struct noeud *u;
   u = z;
   do
      if (s->cle <= t->cle)
         { u->suivant = s; u = s; s = s->suivant; }
      else
         { u->suivant = t; u = t; t = t->suivant; }
   while (u != z);
   u = z->suivant; z->suivant = z;
   return u;
}
```

4.1.7 Tri par tas (*Heap Sort*)

L'implémentation d'une file de priorité à l'aide d'un tas conduit à un algorithme de tri efficace nommé le tri par tas.

L'idée de base consiste à insérer les éléments du tableau à trier dans un tas, la valeur de priorité étant donnée par le champs selon lequel s'effectue le tri. Il reste alors à ressortir les éléments du plus grand au plus petit en vidant le tas progressivement.

Principes de base

Les différentes opérations possibles sur un arbre sont les suivantes :

Ajout dans un arbre parfait partiellement ordonné

- On ajoute l'élément sur la première feuille libre, de manière à ce que l'arbre reste parfait.
- On rétablit la propriété *partiellement ordonné* en échangeant l'élément avec son père si la valeur de ce dernier est inférieure, et en propageant ce type d'échange aussi loin qu'il le faut vers la racine.

Le coût de cet algorithme (en nombre de comparaisons ou d'échanges) est égal au pire à la hauteur de l'arbre, soit $\log_2(n)$.

Suppression du maximum

- On remplace l'élément racine, qui contient le maximum, par le dernier élément (dernière feuille) de manière à ce que l'arbre reste parfait.
- On rétablit la propriété *partiellement ordonné* en effectuant si nécessaire des échanges de pères avec le plus grand de leurs fils, en partant de la racine de l'arbre et en descendant.

Comme pour l'ajout, le coût de cet algorithme est au pire égal à la hauteur de l'arbre, soit $\log_2(n)$.

Toutes les opérations : Au maximum en $O(\log(n))$.

Listing 4.9: *File de priorité (liste non ordonnée)*

```
static int t[maxN+1], N;
Construction(int s[], int M)
{
   for (N = 1; N <= M; N++)
      t[N] = s[N];
}

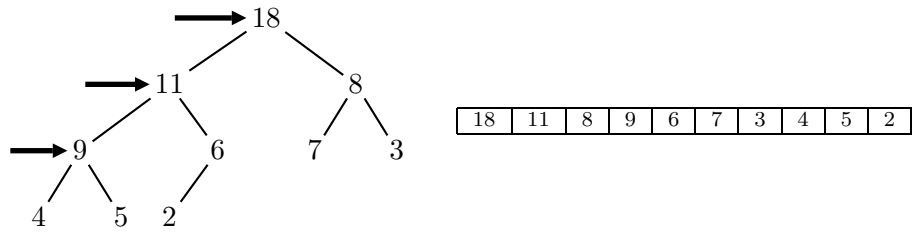
Insertion(int v)
{
   t[++N] = v;
}

int SupprimeMax()
{
   int j, max, v;
   max = 1;
   for (j = 2; j <= N; j++)
      if (t[j] > t[max])
         max = j;
      v = t[max];
      t[max] = t[N--];
      return v;
}
```

Propriétés d'un tas T :

- $T[1]$ désigne la racine de l'arbre parfait correspondant ;
- $T[i \ div \ 2]$ désigne le père de $T[i]$;
- $T[2i]$ et $T[2i+1]$ sont les fils de $T[i]$ dans l'arbre parfait, s'ils existent ;
- Si l'arbre a p nœuds avec $p = 2i$, alors $T[i]$ n'a qu'un fils, $T[p]$;
- Si $i > p \ div \ 2$, alors $T[i]$ est une feuille.

FIG. 4.7 – *Représentation d'un arbre parfait à l'aide d'un tas*



Application au tri par tas

La construction du tas par insertions successives se fait en

$$\log(1) + \log(2) + \ldots + \log(n) = \log(n!) \leq \log(n^n) = n\log(n).$$

Un calcul analogue montre que les suppressions successives pour passer du tas au tableau trié se font également en $n\log(n)$ dans le pire des cas.

Algorithme

1. Insertions successives dans le tas (partie gauche du tableau) ;
2. Suppressions successives des éléments les plus grands.

Construction ascendante : Une amélioration consiste à remplacer la première phase (insertions successives dans le tas) par une réorganisation directe du tableau de départ. Pour cela, considérons l'arbre parfait associé au tableau (et qui n'est pas, en général, partiellement ordonné). On appelle sur chaque nœud interne de cet arbre, du dernier jusqu'à la racine, une procédure `réorganiser` qui effectue des échanges père-fils en chaîne, jusqu'au bas de l'arbre si nécessaire. Ce procédé permet de donner une structure de tas au tableau en un temps $O(n)$ au lieu de $n\log(n)$.

Preuve : Essayons de calculer un majorant du nombre de fois qu'un élément descend. Soit k tel que $2^k \leq N < 2^{k+1}$. L'élément dans la case 1 descend au plus de k, ses deux fils de $k-1$, ses quatre petits fils de $k-2$, etc. On a donc au plus :

$$k + 2(k-1) + 4(k-2) + \ldots + 2^i(k-i) + \ldots + 2^{k-1} \times 1$$

descentes. On s'arrête à $k-1$ car après, ce sont des feuilles.

Multiplions et divisons par 2^{k-1}, on a :

$$2^{k-1}\left(\frac{k}{2^{k-1}} + \frac{k-1}{2^{k-2}} + \ldots + \frac{k-i}{2^{k-i-1}} + \ldots + 1 \times 1\right).$$

Posons $x = 1/2$; on trouve :

$$\sum_{i=1}^{k-1} i \times x^{i-1} < \sum_{i=1}^{\infty} i \times x^{i-1}.$$

Or puisque $x < 1$, la série $\sum_{i=1}^{\infty} x^{i-1}$ converge et vaut $\frac{1}{1-x}$, et la limite de la série qui nous intéresse est sa dérivée, et donc

$$\sum_{i=1}^{\infty} i \times x^{i-1} = \left(\frac{1}{1-x}\right)^2.$$

Ce qui nous montre que le nombre total de descente est borné par $2N$. Notons que la complexité total du tri par tas reste $N \log_2 N$, du fait de la seconde phase.

FIG. 4.8 – *Tri par tas - Phase de construction*

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

FIG. 4.9 – *Tri par tas - Phase de suppressions*

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

Listing 4.10: *Insertion dans un tas (maximier)*

```
Taslacmite(int k)
{
   int v;
   v = t[k]; t[0] = ENTIER_MAX;
   while (t[k/2] <= v)
      { t[k] = t[k/2]; k = k / 2; }
   t[k] = v;
}

Insertion(int v)
{
   t[++N] = v;
   Taslacmite(N);
}
```

Listing 4.11: *Tri par tas ou tri maximier*

```
TriMaximier(int t[], int N)
{
   int k;
   Construction(t, 0);
   for (k = 1; k <= N; k++) Insertion(t[k]);
   for (k = N; k >= 1; k--) t[k] = SupprimeMax();
}
```

4.2 Multiplication de grands entiers

Nous voulons multiplier deux entiers X et Y composés de n chiffres en base 2. La méthode habituelle consiste à fabriquer n produits partiels de taille n. On a donc une complexité en $O(n^2)$.

Une approche de type *Divide-and-Conquer* consiste à scinder X et Y en deux entiers de $n/2$ chiffres :

$$\begin{aligned} X &= A \cdot 2^{\frac{n}{2}} + B\,; \\ Y &= C \cdot 2^{\frac{n}{2}} + D. \end{aligned}$$

On trouve alors

$$X \cdot Y = A \cdot C \cdot 2^n + (A \cdot D + B \cdot C) \cdot 2^{\frac{n}{2}} + B \cdot D. \tag{4.1}$$

Si on évalue $X \cdot Y$ de cette manière, on a :

- quatre multiplications à $\frac{n}{2}$ chiffres ;
- trois additions avec au maximum $2n$ chiffres ;
- deux décalages (multiplications par 2^n et $2^{\frac{n}{2}}$). Ces deux derniers types d'opérations sont en $O(n)$.

On a donc :

$$\begin{aligned} T(1) &= 1\,; \\ T(n) &= 4T(\frac{n}{2}) + cn. \end{aligned}$$

Théorème. Si $T(n) = a \cdot T(n/b) + c \cdot n$, avec $T(1) = C$, $a > 1$ et $b > 1$, alors :

$$\begin{aligned} a < b &\Rightarrow T(n) = O(n) \\ a = b &\Rightarrow T(n) = O(n \log n) \\ a > b &\Rightarrow T(n) = O(n^{\log_b(a)}) \end{aligned}$$

En posant $a = 4$ et $b = 2$, nous avons pour notre exemple :

$$T(n) = O(n^{\log_2(4)}) = O(n^2).$$

L'amélioration n'est donc pas encore visible. Par contre, on peut reformuler l'équation (4.1) de la façon suivante :

$$X \cdot Y = A \cdot C \cdot 2^n + [(A - B)(D - C) + A \cdot C + B \cdot D] \cdot 2^{\frac{n}{2}} + B \cdot D.$$

Il n'y a plus que trois sous-produits à effectuer. On a donc :

$$\begin{aligned} T(1) &= 1\,; \\ T(n) &= 3T(\frac{n}{2}) + c'n\,; \end{aligned}$$

d'où $T(n) = O(n^{\log_2(3)}) = O(n^{1,59})$.

Chapitre 5

Graphes

5.1 Introduction

Un graphe $G = (V, E)$ est donné par un ensemble de **sommets** V — cet ensemble est aussi parfois appelé **points** ou encore **nœuds**, *vertices* en anglais, d'où le V — et d'un ensemble d'arcs $E \subseteq V \times V$ (*edges* en anglais). L'ensemble E définit une **relation binaire** sur V. Par la suite, sauf notation contraire, nous supposerons que $n = |V|$ et $m = |E|$. On a bien sûr $m \leq n^2$.

5.1.1 Définitions

Graphe complet. Si $\mathcal{E}$ est la relation pleine : $E = V \times V$. On accepte aussi ce terme de complet si les boucles sont absentes.

Graphe non orienté. Si la relation $\mathcal{E}$ est symétrique :

$$(x, y) \in E \Rightarrow (y, x) \in E$$

sinon, le graphe est dit **orienté**.

Arc. Tout élément de $\mathcal{E}$. Si le graphe n'est pas orienté, on parle alors d'**arête** pour désigner indifféremment (x, y) ou (y, x). x est l'**origine** ou **extrémité initiale** de l'arce (x, y), et y son **extrémité terminale**. Dans le cas non orienté, on ne parlera que d'extrémités.

Boucle. Un arc $(u, u) \in E$.

Demi-degré intérieur d'un sommet. (resp. *extérieur*) Nombre d'arcs ayant ce sommet pour extrémité finale (resp. *origine*). Le **degré** est la somme de ces deux valeurs. Noter que l'arc (x, x) sera compté deux fois dans le degré.

Chemin. de longueur k, d'origine x_0 et d'extrémité x_k : suite $x_0, \ldots, x_i, \ldots, x_k$ telle que les (x_i, x_{i+1}) sont des arcs. Une **chaîne** est l'équivalent non orienté du chemin. Un chemin vide est un chemin de longueur 0.

Circuit. Chemin de longueur $\neq 0$ tel que $x_0 = x_n$. Un **cycle** est la version non orientée du chemin.

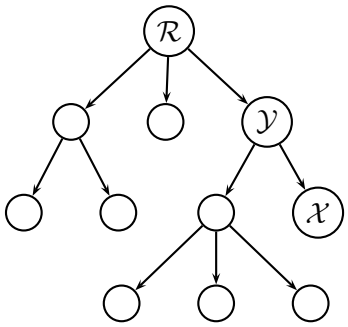
Chemin élémentaire. Chemin ne contenant pas deux fois le même sommet. Il sera dit simple s'il ne contient pas deux fois le même arc. Cette définition s'étent naturellement aux chaînes, circuits et cycles. De fait, dans la littérature française, l'habitude est de considérer que chaînes, circuits et cycles sont toujours simples.

Graphe connexe. Tout couple de sommets est relié par une chaîne du graphe (sans tenir compte du sens des arcs). On appelle **composante connexe** tout sous-ensemble maximal (au sens de l'inclusion) de sommets qui est connexe.

Chemin hamiltonien. Tout sommet de $\mathcal{G}$ n'est visité qu'une et une seule fois. Cette notion s'étend aux chaînes, circuits et cycles. Un chemin est dit **eulérien** si tout arc apparaît une et une seule fois dans le chemin.

5.2 Les arbres

Un **arbre** est un graphe connexe sans cycle. Une **arborescence** est un arbre orienté possédant une racine r, *ie.* il existe un unique chemin allant de r à n'importe quel autre sommet de l'arborescence.



Etant donnée une arborescence,

- Une **feuille** est un sommet sans successeur ;
- La **profondeur** d'un sommet $\mathcal{X}$ est le nombre d'arcs sur le chemin allant de $\mathcal{R}$ à $\mathcal{X}$;
- La **hauteur** de l'arborescence est la profondeur maximale d'un de ses sommets ;
- Un **fils** est un successeur immédiat, alors qu'un **père** est un prédécesseur immédiat. Dans l'exemple ci-dessus, $\mathcal{Y}$ est le père de $\mathcal{X}$ et $\mathcal{X}$ est le fils de $\mathcal{Y}$. De plus, $\mathcal{X}$ est une feuille de profondeur 2 alors que la hauteur de l'arborescence est égale à 3.

5.2.1 Arbres complets (ou parfaits) partiellement ordonnés

Arbre partiellement ordonné : Arbre dont la valeur en tout nœud est supérieure ou égale à celles de ses fils.

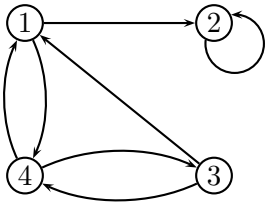
Arbre complet ou parfait : Arbre binaire dont les feuilles sont sur deux niveaux seulement, dont l'avant-dernier niveau est complet, et dont les feuilles du dernier niveau sont groupées le plus à gauche possible.

Remarque

Les algorithmes de tri par insertion et par sélection sont basés sur des comparaisons. On peut représenter ces algorithmes par des arborescences binaires.

5.3 Représentation des graphes

Les graphes, à la différence des structures de données classiques, n'ont pas de représentation standard mais, au gré des problèmes qui sont traités, des représentations classiques. Chacune a des avantages et des inconvénients.



5.3.1 Représentation matricielle

On représente le graphe par une matrice G de taille $n \times n$: $G(i, j)$ sera un booléen pour indiquer l'existence d'un arc de i vers j. Cette représentation est appelée *adjacence sommets-sommets* en recherche opérationnelle.

$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \text{ avec } m_{ij} = \begin{cases} 1 & \text{si } v_i \text{ adjacent à } v_j \\ 0 & \text{sinon} \end{cases}$$

Rappel : L'autre représentation est la représentation *incidence sommets-arêtes* pour les graphes non-orientés et *incidence sommets-arcs* pour les graphes orientés. Dans notre exemple, nous aurions la matrice

$$\begin{pmatrix} -1 & -1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & -1 & 1 \\ 0 & 1 & 0 & -1 & 1 & -1 \end{pmatrix} \text{ avec } m_{ij} = \begin{cases} -1 & \text{si } v_i \text{ extrémité initiale de } e_k \\ 1 & \text{si } v_i \text{ extrémité finale de } e_k \\ 0 & \text{sinon} \end{cases}$$

Il faut noter qu'il est impossible d'inclure la boucle sur le nœud 2 avec cette notation.

Avantages

Accès direct aux arcs depuis les sommets. Tous les '1' d'une ligne donnent les arcs sortants du sommet correspondant. Tous les '1' d'une colonne donnent les arcs entrant sur le sommet correspondant.

Inconvénients

Occupation mémoire de l'ordre de $O(n^2)$: beaucoup de place perdue si n est grand et si le graphe n'est pas dense. Néanmoins, c'est une représentation compacte si le graphe est presque complet.

5.3.2 Représentation par une table de successeurs

A chaque sommet est associé l'ensemble des successeurs. Le plus simple est de mettre ces successeurs dans une liste. On gère donc un tableau de liste de successeurs.

Avantages

Représentation compacte pour les graphes creux (*les plus fréquents*), accès itératif facile lorsqu'il faut traiter les successeurs dans une boucle.

Remarque

On aimerait parfois avoir les prédécesseurs. Dans ce cas, il suffit de faire un tableau de listes de prédécesseurs.

5.4 Parcours des graphes

5.4.1 Parcours en profondeur

Algorithme

```
Tous les sommets non marqués

Pour tous les sommets ν ∈ V faire
  Si ν pas marqué
    explorer(ν)
    imprimerListe
    viderListe

explorer(sommet ν)
début
  marquer ν, metDansUneListe

  Pour tous les sommets ω ∈ V t.q. (ν,ω) ∈ E (ou (ω,ν) si non-orienté)
    si pas marqué(ω)
      explorer(ω)
fin
```

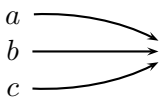
Explications

Les arcs (ou arêtes) qui servent dans le parcours définissent des arbres qui ont comme racine les sommets qui donnent lieu à une exploration dans le pgoramme principal.

5.4.2 Tri topologique

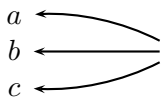
1. Poser $k = 1$;
2. Inclure tous les sommets dans une liste ;
3. Trouver un sommet v_i sans prédécesseur et définir $v_i = k$;
4. Incrémenter k : $k = k + 1$;
5. Supprimer le sommet v_i de la liste ;
6. Retourner au point 3 jusqu'à ce que la liste soit vide.

En recherche opérationnelle



d ne commence qu'après la fin de a, b et c

En informatique

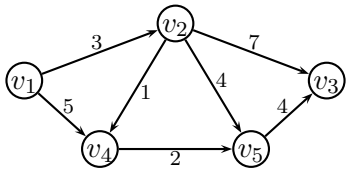


d dépend de a, b et c

La raison de la différence vient de la façon de représenter un ensemble d'opérations à effectuer. Si les sommets représentent des opérations à effectuer (pour un calcul, un programme, ...), et qu'une opération ne peut se faire que si certaines autres sont terminées, alors les informaticiens font un graphe de dépendance et disent que l'opération v dépend de u et mettent une flèche (arc) de v vers u, tandis que les chercheurs opérationnels disent que l'opération v ne peut se faire qu'après u et la flèche est dans le même sens que le temps. Dans tous les cas, l'opération qui recevra le numéro d'ordre le plus petit sera celle qui commencera.

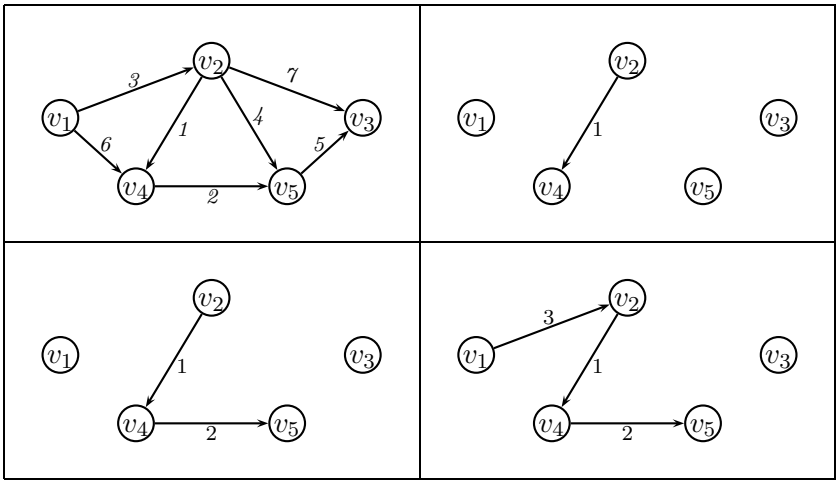
5.4.3 Arbre maximal de poids minimal

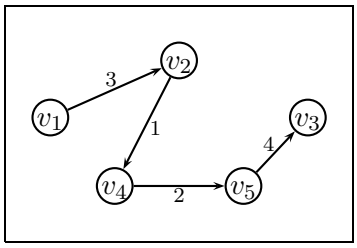
Le but est de relier d'une manière ou d'une autre tous les sommets d'un arbre et ce, de la manière la plus économique possible (les arêtes étant pondérées).



Algorithme de Kruskal

1. Numéroter les arêtes dans l'ordre croissant de leur pondération ;
2. Ajout des arêtes selon leur numéro d'ordre si et seulement si l'arc que l'on veut ajouter ne forme pas de cycle avec la solution existante.





Coût : $3 + 1 + 2 + 4 = 10$.

Algorithme de Prim

1. Choix arbitraire d'un sommet de départ ;
2. On fait croître la solution de la façon la plus économique à chaque étape.

5.4.4 Le plus court chemin (Algorithme de Dijkstra)

Les variables suivantes sont utilisées :

- Vecteur longueur λ ;
- Candidats L ;
- Sommets disponibles T ;
- Prédécesseur $p(i)$.

Algorithme

1. $\lambda_s = 0$, $\lambda_i = \infty$, $p(i) = \varnothing$;
2. Tant que $T \neq \varnothing$: $(T = L \cup \{j \mid \lambda_j = \infty\})$
   (a) soit i le sommet de plus petite étiquette λ_i ;
   (b) si $i = \varnothing \Rightarrow$ STOP (pas atteignable) ;
   (c) sinon retirer i de T et tester les successeurs j. Si $\lambda_j > \lambda_i + c_{ij}$, alors $\lambda_i = \lambda_i + c_{ij}$ et $p(j) = i$.

Exemple



Itération	i_{min}	$\lambda_1/p(1)$	$\lambda_2/p(2)$	$\lambda_3/p(3)$	$\lambda_4/p(4)$	T
0	—	$0/\varnothing$	$\infty/\varnothing$	$\infty/\varnothing$	$\infty/\varnothing$	$\{v_1,v_2,v_3,v_4\}$
1	v_1	$\mathbf{0}/\varnothing$	$3/v_1$	$1/v_1$	$\infty/\varnothing$	$\{v_2,v_3,v_4\}$
2	v_3		$2/v_3$	$\mathbf{1/v_1}$	$4/v_3$	$\{v_2,v_4\}$
3	v_2		$\mathbf{2/v_3}$		$\mathbf{4/v_3}$	$\{v_4\}$
4	v_4					$\{\varnothing\}$

Ce qui se lit par exemple : le chemin le plus court de v_1 à v_4 passe par v_3 et a un coût de 4 unités.

5.5 Composantes connexes

N'importe quelle méthode de parcours de graphes peut servir à trouver les composantes connexes d'un graphe donné puisqu'elles sont toutes fondées sur la stratégie générale d'exploration de tous les nœuds d'une même composante connexe avant de passer à la suivante. Une manière simple de lister les composantes connexes est de modifier l'un des programmes récursifs de parcours en profondeur pour que la procédure `Explorer` (p. 28) énumère le sommet couramment visité (en imprimant `Nom(`ν`)` juste en fin d'exécution par exemple) et que soit donnée une indication relative au commencement d'une nouvelle composante connexe juste avant l'appel (non récursif) d'`Explorer` (par exemple en imprimant deux lignes vides).

5.5.1 2-connexité

Il est parfois utile de s'assurer de l'existence de plusieurs itinéraires entre deux points d'un graphe, ne serait-ce que pour permettre de pallier des défections éventuelles aux points de connexion (sommets). Ainsi, on peut aller de Saint-Martin-d'Hères [1] à Bruxelles si Paris est enseveli sous la neige en passant par Zurich et Cologne. Les principaux réseaux de connexion d'un circuit imprimé sont souvent 2-connexes, de sorte que le reste du circuit puisse quand même fonctionner si l'un des composants tombe en panne.

Un *point d'articulation* dans un graphe connexe est un sommet dont la suppression décompose le graphe en plusieurs composantes. Un graphe sans point d'articulation est dit *2-connexe*. Dans un tel graphe, deux chemins distincts relient chaque couple de sommets. Un graphe qui n'est pas 2-connexe se divise en *composantes 2-connexes*, qui sont des ensembles de nœuds mutuellement accessibles par (au moins) deux chaînes distinctes.

Listing 5.1: *Recherche de composantes 2-connexes*

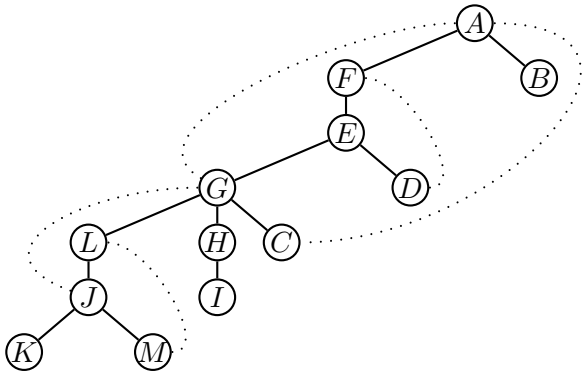
```
int Explorer(int k)
{
   struct noeud *t;
   int m, min;
   val[k] = ++id; min = id;
   for (t = Adj[k]; t != z; t = t->suivant)
      if (val[t->s] == 0)
      {
         m = Explorer(t->s);
         if (m < min) min = m;
         if (m >= val[k]) printf("%c", Nom(k));
      }
      else if (val[t->s] < min) min = val[t->s];
   return min;
}
```

Cette procédure détermine récursivement le nœud de l'arborescence le plus haut accessible (par un arc en pointillé) depuis tout descendant du sommet `k`, et utilise cette information pour déterminer si `k` est un point d'articulation. Normalement, cette opération ne nécessite que de tester si la valeur minimum accessible depuis un fils est située plus haut dans l'arborescence ou non. Malgré tout, il faut effectuer un test supplémentaire pour déterminer si `k` est la racine d'une arborescence de parcours en profondeur (ou, de manière équivalente, s'il s'agit du premier appel d'`Explorer` pour la composante connexe contenant `k`), puisque l'on utilise le programme récursif

1. Cette ville de 926 hectares dont 96 réservés au Domaine Universitaire comptait 35 777 habitants en 1999. Elle est située à l'est de Grenoble, en France. *Ndlr.*

dans les deux cas. Il est préférable d'effectuer ce test à l'extérieur de l'exploration récursive et celui-ci n'apparaît donc pas dans le programme précédent.

Propriété. Les composantes 2-connexes d'un graphe peuvent être déterminées en temps linéaire. Il faut noter qu'un programme semblable, fondé sur une représentation par matrice d'incidence (p. 27) aurait une complexité en $O(S^2)$.



En plus de servir à améliorer la fiabilité, la 2-connexité peut permettre de décomposer les grands graphes en parties plus maniables. Si l'utilité du traitement d'un grand graphe composante par composante ne fait pas de doute, il est parfois pratique, bien que de manière moins évidente, de pouvoir traiter un graphe composante 2-connexe par composante 2-connexe.

5.5.2 Composantes fortement connexes

Si un graphe contient un *circuit* (si l'on peut revenir à un nœud initial en suivant une série d'arcs dans la direction qu'ils indiquent), la structure n'est pas un GOSC (graphe orienté sans circuit, *dag* en anglais, pour *directed acyclic graph*) et ne peut donc donner lieu à un tri topologique : tout sommet d'un circuit listé en premier admettra un autre sommet non encore listé pointant vers lui. Les nœuds d'un même circuit sont mutuellement accessibles dans le sens où il existe (au moins) un moyen de se rendre d'un nœud à un autre et d'en revenir. D'un autre côté, même si un graphe donné est connexe, il y a peu de chances que chacun de ses nœuds soit accessible depuis n'importe quel autre par un chemin (orienté). En fait, les nœuds se divisent en sous-ensembles, appelés *composantes fortement connexes*, tels que tous les nœuds à l'intérieur de la même composante sont mutuellement accessibles mais qu'il n'existe aucun moyen d'aller d'un nœud d'une composante à un nœud d'une autre et d'en revenir.

Les composantes fortement connexes d'un graphe orienté peuvent être trouvées à l'aide d'une variante de l'exploration en profondeur. La méthode que nous examinerons a été découverte par R.E. Tarjan, en 1972. Comme elle est fondée sur l'exploration en profondeur, sa complexité théorique est en $O(S + A)$ (Sommmets, Arrêtes).

La version récursive `Explorer` donnée ci-dessous utilise le même calcul de minimum pour trouver le sommet accessible (par un lien ascendant) le plus haut depuis un quelconque descendant du sommet k que le listing 5.1, mais utilise la valeur de `min` d'une manière légèrement différente dans le but de lister les sommets des composantes fortement connexes.

Listing 5.2: *Recherche de composantes fortement connexes*

```
int Explorer(int k)
{
   struct noeud *t;
   int m, min;
   val[k] = ++id; min = id;
   pile[p++] = k;
   for (t = Adj[k]; t != z; t = t->suivant)
   {
      m = (!val[t->s]) ? Explorer(t->s) : val[t->s];
      if (m < min) min = m;
   }
   if (min == val[k])
   {
      while (pile[p] != k)
      {
         printf("%c", Nom(pile[--p]));
         val[pile[p]] = S + 1;
      }
      printf("\n");
   }
   return min;
}
```

Le programme empile les noms de sommet à l'entrée d'`Explorer`, puis les dépile et les affiche au moment où vient d'être exploré le dernier élément de chaque composante fortement connexe. L'idée forte de cette procédure est de tester l'égalité de `min` et `val[k]` : si les deux quantités sont égales, tous les sommets rencontrés depuis l'entrée dans la fonction (sauf ceux déjà listés) appartiennent à la même composante fortement connexe que `k`.

Propriété. Les composantes fortement connexes d'un graphe peuvent être déterminées en temps linéaire.

Chapitre 6

Exercices corrigés

6.1 Propédeutique II - 12 juillet 2002

Exercice 1 Montrer que $\log(n!) \in O(n \log n)$, où $n! = (n \times (n-1) \ldots \times 2 \times 1)$.

Solution On a :

$$\begin{aligned}
\log(n!) &= \log\left[n \times (n-1) \ldots \times 2 \times 1\right] \\
&= \log(n) + \log(n-1) + \ldots + \log(2) + \log(1) \\
&< \underbrace{\log(n) + \log(n) + \ldots + \log(n) + \log(n)}_{n \text{ termes}} \\
&= n \times \log(n) \\
&\in O(n \log(n))
\end{aligned}$$

Exercice 2 Montrer par récurrence qu'un arbre binaire de hauteur (ou profondeur) h possède au plus (un maximum de) 2^h feuilles. En déduire qu'un arbre binaire avec n feuilles a une hauteur (ou profondeur) d'au moins $\lceil \log n \rceil$, où $\lceil x \rceil$ est le plus petit entier supérieur ou égal à x (arrondi de x vers le haut). Quelle est la profondeur maximum d'un arbre binaire qui possède n feuilles ?

Solution 2.1 Par définition, un arbre binaire a, pour chaque branche, un maximum de deux branches conduisant au niveau suivant. On double donc le nombre de branches maximum à chaque niveau. Par hypothèse, nous avons $n_{maxF}(h) = 2^h$. Nous vérifions que $n_{maxF}(1) = 2$. Supposons vrai pour $h = k$. Nous avons donc $n_{maxF}(k) = 2^k$ et $n_{maxF}(k+1) = 2^{k+1}$ par hypothèse de récurrence. Par définition, nous avons aussi que $n_{maxF}(k+1) = 2 \times n_{maxF}(k) = 2 \times 2^k = 2^{k+1}$.

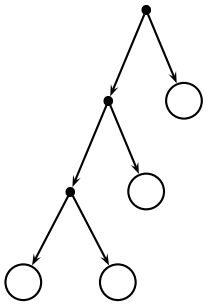
Solution 2.2 Nous avons la déduction suivante :

$$\begin{aligned}
N = n_{maxF}(h) &= 2^h \\
\log_2(N) &= \log_2(2^h) \\
h &= \log_2(N)
\end{aligned}$$

et donc $h_{min} \geq \log_2(n)$. Or h_{min} étant un entier, on a $h_{min} = \lceil \log_2 n \rceil$.

Il est intéressant de noter que la donnée n'était pas correcte car elle ne spécifiait pas que le log était en base 2.

Solution 2.3 Un arbre binaire complètement dégénéré sera de la forme :



On trouve de façon graphique que $h_{max}(n) = n - 1$.

Exercice 3 Soit $T(n, m)$, le nombre de solutions entières de l'équation :

$$x_1 + x_2 + \ldots + x_m = n \tag{6.1}$$

1. Montrer que $T(n, m) = T(n, m-1)+T(n-1, m)$. En d'autres termes, interpréter la formule et montrer son exactitude ;
2. Donner les conditions aux bords ;
3. Montrer qu'une programmation récursive directe de la formule précédente amène des calculs redondants ;
4. Ecrire un algorithme récursif qui calcule efficacement, *ie.* sans calcul redondant, $T(n, m)$ où m et n sont des paramètres.

Par solution entière, on entend une solution qui prend ses valeurs dans l'ensemble $\{0, 1, 2 \ldots, n\}$.

Solution 3.1 Supposons pour clarifier les idées que l'on a n boules à placer dans m boîtes. On désire savoir le nombre de façons différentes de placer nos boules dans les boîtes. Etant donné que l'on sait comment les mettre dans les $m - 1$ premières boîtes, $T(n, m - 1)$, il suffit de chercher combien de solutions utilisent la dernière boîte.

Pour trouver de combien de façons on peut mettre n boules dans m boîte en occupant la dernière, on peut commencer à en mettre une dans la dernière boîte, et ensuite il restera $n - 1$ boules à placer où l'on veut dans m boîtes.

Mais justement, on sait qu'il y a $T(n - 1, m)$ façons différentes de faire cela. En résumé, il y a donc

$$\begin{cases} T(n, m-1) & \text{solutions qui n'utilisent pas du tout la dernière boîte ;} \\ T(n-1, m) & \text{solutions qui utilisent la dernière boîte.} \end{cases}$$

Ce qui donne un total de $T(n,m) = T(n, m - 1) + T(n - 1, m)$.

Solution 3.2 Il est clair d'après l'équation (6.1) que si le premier paramètre, n, vaut 0, il n'existe qu'une et une seule solution qui consiste à donner la valeur nulle à toutes les variables puisque l'on part du principe que les solutions sont toutes positives.

De plus, nous pouvons affirmer que n'importe quelle équation linéaire à un seul terme possède exactement 1 solution. Ce qui amène à poser les conditions aux bords :

$$\begin{cases} T(0,m) = 1 & \forall m \in \mathbb{N} \\ T(n,1) = 1 & \forall n \in \mathbb{N} \end{cases}$$

Solution 3.3 Nous allons afficher le début de la liste d'appels de $T(6,6)$:

$$\begin{aligned} T(6,6) &= T(6,5) + T(5,6) \\ &= [T(6,4) + T(5,5)] + [T(5,5) + T(4,6)] \\ &= \dots \end{aligned}$$

Il est donc clair que, dès la deuxième étape, les calculs sont redondants puisque $T(5,5)$ sera calculé deux fois.

Solution 3.4

```
#include <stdio.h>
#define MAXN 10
#define MAXM 10

int calcT[MAXN][MAXM];

int T(int n, int m) {
   if (m > 0) if (!calcT[n][m - 1]) calcT[n][m - 1] = T(n, m - 1);
   if (n > 0) if (!calcT[n - 1][m]) calcT[n - 1][m] = T(n - 1, m);

   return calcT[n][m - 1] + calcT[n - 1][m];
}

void main() {
   for (int i = 0; i < MAXM; i++)
      for (int j = 0; j < MAXN; j++)
         calcT[j][i] = 0;  /* initialisation du tableau */

   for (int i = 0; i < MAXM; i++)
      calcT[0][i] = 1;     /* condition au bord #1 */

   for (int i = 0; i < MAXN; i++)
      calcT[i][1] = 1;     /* condition au bord #2 */

   printf("%d", T(3, 2));  /* calcul de T(3, 2) */
}
```

Exercice 4 Préciser très clairement où se trouve l'erreur de raisonnement dans la démonstration suivante :

Théorème : Tous les nombres $n \geq 2$ sont pairs (multiples de 2).

Démonstration par récurrence. C'est vrai si $n = 2$. Supposons que ce soit vrai jusqu'à l'ordre $k \geq 2$. Alors $k+1 = a+b$ avec $a \leq k$ et $b \leq k$, donc par hypothèse de récurrence, $a = 2a'$ et $b = 2b'$ et donc $k+1 = a+b = 2a' + 2b' = 2(a'+b')$ et donc $k+1$ est pair. Par conséquent, tous les nombres $n \geq 2$ sont pairs.

Solution On pose $k+1 = a+b$ avec $a = 2a'$ et $b = 2b'$ et on en déduit que $k+1 = 2(a'+b')$. La démonstration semble donc bonne, mais si k est pair, on peut poser $k = c+d$ avec c et d pairs par hypothèse de récurrence ($c = 2c'$ et $d = 2d'$). On a donc $k+1 = c+d+1$. Mais comme c et d sont pairs, nous obtenons $k+1 = 2c' + 2d' + 1 = 2(c'+d') + 1$ qui est impair par définition. Nous avons donc une contradiction qui conduit à infirmer le théorème selon le principe d'une démonstration par l'absurde.

6.2 Test d'algorithmique - 7 juin 2002

Exercice 2 Dans le graphe orienté de la figure suivante, donner l'ordre de visite des sommets dans un parcours en profondeur à partir du sommet a. Quand un choix se présente, on prend le sommet de plus petit nom dans l'ordre alphabétique. Même question pour un parcours en largeur avec toujours les voisins d'un même sommet parcourus dans l'ordre alphabétique.

On dit qu'un sommet est terminé dans un parcours en profondeur quand tous les appels récursifs qu'il a engendré son terminés. Donner l'ordre dans lequel les sommets sont terminés dans le parcours en profondeur précédent. Pour le parcours en profondeur, mettre l'ordre de visite dans le carré et l'ordre de terminaison dans le cercle.

Solution pour le parcours en profondeur

Voir la figure 6.1.

Solution pour le parcours en largeur

L'ordre de visite des sommets est le suivant :

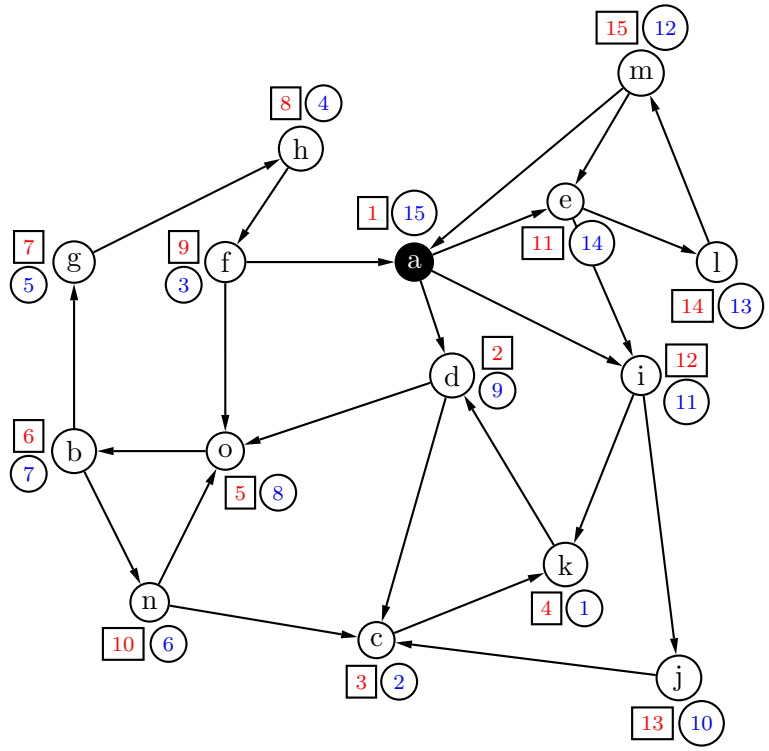
$$a \to d \to e \to i \to c \to o \to l \to j \to k \to b \to m \to g \to n \to h \to f$$

Voir aussi la figure 6.2 pour la solution graphique.

Exercice 7 Vérifier que le tableau suivant est bien un tas. Dessiner l'arbre binaire correspondant. Y ajouter la valeur 7. Donner le résultat sous forme de tableau.

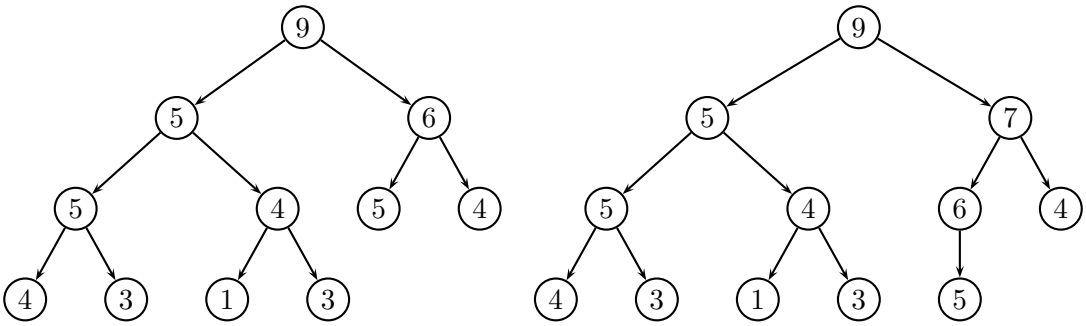
9	5	6	5	4	5	4	4	3	1	3	

FIG. 6.1 – *Exercice 2 - Parcours en profondeur*



Solution

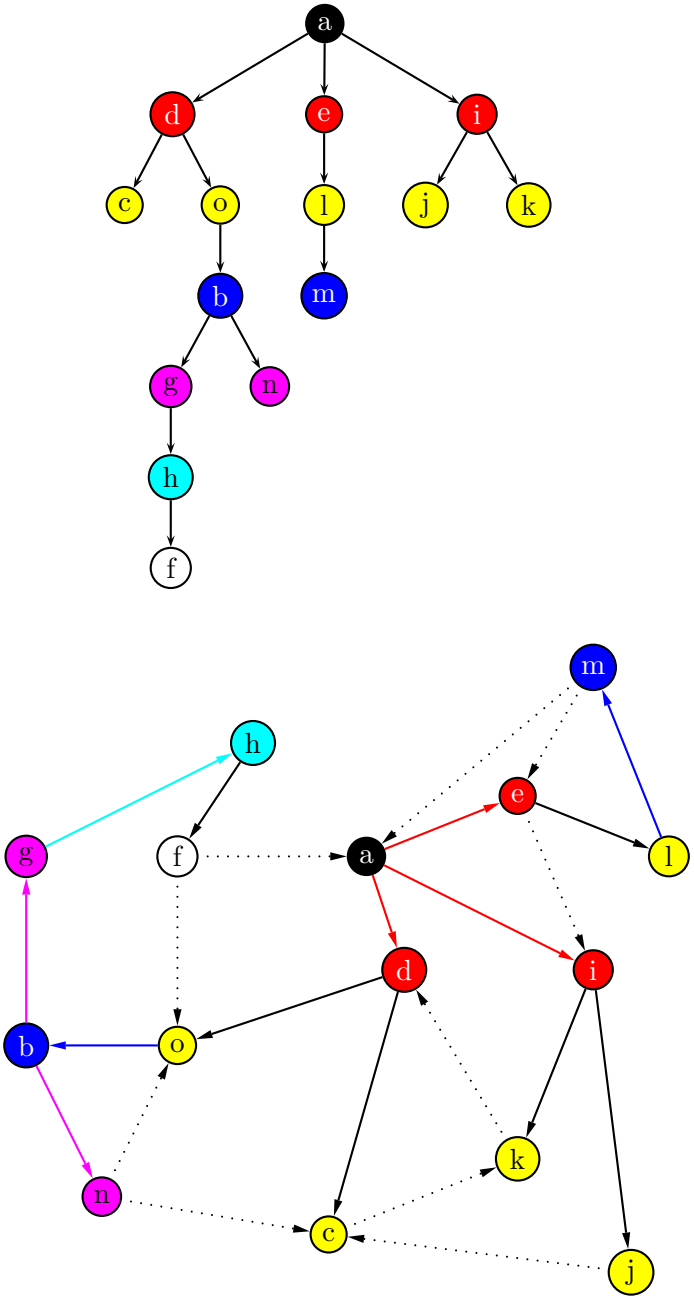
Le plus simple pour vérifier que le tableau proposé représente bien un tas est de le construire et de vérifier que l'arbre satisfait bien les propriétés d'un tas.



Nous pouvons donc constater que c'est bien le cas. A présent, l'insertion de l'élément 7 se fait comme fils du nœud 5 de la branche droite. Il faut ensuite faire des pivotages d'éléments pour retrouver un tas valable. Le 7 remonte donc à la place du 5 puis remonte encore d'un niveau pour prendre la place du 6. Le tas final après insertion est donc :

9	5	7	5	4	6	4	4	3	1	3	5

FIG. 6.2 – *Exercice 2 - Parcours en largeur*



Références

[1] Denis Naddef. *Notes de cours d'algorithmique*, EPFL / Département de mathématiques, 2001–2002.

[2] Alain Hertz. *Bases de l'algorithmique*, EPFL / Département de mathématiques, 1999.

[3] Robert Sedgevick. *Algorithmes en langage C – Cours et exercices*, Addison-Wesley, 2001.

[4] Jean-François Hêche. *Notes de cours de recherche opérationnelle*, EPFL / Département de mathématiques, 2001–2002.