

Enseignement de l'informatique

<https://github.com/ylebbah/teaching>

POLYCOPIE

Algorithmique Avancée et Complexité

LEBBAH Yahia

Professeur, Université d'Oran 1

Département Informatique, Faculté des Sciences Exactes et Appliquées, Université d'Oran 1
B.P. 1524, El-M'Naouar Oran, Algérie

Version du 5 janvier 2025

Table des matières

1	Introduction	5
1.1	Calculer les nombres de Fibonacci [Papadimitriou et al., 2006]	5
1.1.1	Une première version	6
1.1.2	Une version polynomiale	7
1.2	Algorithme, correction et complexité - Illustration sur le tri par insertion [Cormen et al., 2001]	8
1.2.1	Correction	10
1.2.2	Complexité	12
1.3	Récursivité et l'approche diviser pour régner [Gaudel et al., 1990, Cormen et al., 1994, Cormen et al., 2001]	13
1.3.1	Tri par fusion	14
1.3.2	Exponentiation rapide	17
2	Principes de l'analyse des algorithmes	19
2.1	Mesure du coût [Beauquier et al., 1992, Cormen et al., 1994, Cormen et al., 2001, Gaudel et al., 1990]	19
2.1.1	La complexité dans le meilleur des cas	22
2.1.2	La complexité dans le pire des cas	22
2.1.3	La complexité en moyenne	22
2.2	Mesure asymptotique, grandeurs des fonctions et notations de Landau : O , ω , ... [Gaudel et al., 1990]	23
2.2.1	Exemple : Multiplication de matrices carrées	31
2.2.2	Exemple : Recherche linéaire	32
2.3	Optimalité d'un algorithme [Gaudel et al., 1990]	33
2.3.1	Exemple : recherche du plus grand élément d'une liste [Gaudel et al., 1990]	34
2.4	Relations de récurrence [Sedgewick and Flajolet, 1996]	35
2.4.1	Réurrences du premier ordre [Sedgewick and Flajolet, 1996] .	36
2.4.2	Réurrences d'ordre supérieur [Sedgewick and Flajolet, 1996] .	37
2.4.3	Réurrence diviser pour régner	38
2.4.4	Autres méthodes de résolution	40
2.5	Algorithmes de tri	41
2.5.1	Tri par tas [Cormen et al., 1994, Cormen et al., 2001]	41
2.5.1.1	Les tas	41
2.5.2	Application des tas : Files de priorité	48
2.5.3	Tri rapide	49
2.5.3.1	Introduction à la méthode de substitution [Cormen et al., 1994]	54

2.6	Travaux dirigés I	56
2.6.1	Algorithme de tri	56
2.6.2	Algorithme de recherche dichotomique	56
2.6.3	*** Algorithme de recherche	56
2.7	Travaux dirigés II	57
2.7.1	Réurrence	57
2.7.2	Grandeurs des fonctions et notations asymptotiques	57
2.7.3	Réurrences d'ordre supérieur et diviser pour régner	57
2.7.4	Tris	57
2.8	Mini projets	58
2.8.1	Recherche des deux points les plus rapprochés	59
3	Structures de données avancées	65
3.1	Piles, Files, et Listes [Cormen et al., 1994, Cormen et al., 2001]	65
3.2	Table de hachage [Cormen et al., 1994, Cormen et al., 2001]	65
3.2.1	Table à adressage direct	66
3.2.2	Tables de hachage	68
3.2.3	Fonctions de hachage	71
3.2.4	Adressage ouvert	72
3.3	Arbres binaires [Cormen et al., 1994, Cormen et al., 2001]	75
3.4	Arbres AVL	80
3.5	Arbres rouge et noir [Cormen et al., 1994, Cormen et al., 2001]	85
3.6	B-arbres [Cormen et al., 1994, Cormen et al., 2001]	86
3.7	Tas binomiaux [Cormen et al., 1994, Cormen et al., 2001]	88
3.7.1	Arbres binomiaux	89
3.7.2	Définition d'un tas binomial	90
3.8	Analyse amortie [Robert et al., 2005]	92
3.9	Travaux dirigés III	93
3.9.1	Tables de hachage	93
3.9.2	Arbres binaires de recherche	93
3.9.3	Opérations sur les arbres binaires de recherche	93
3.9.4	Arbres AVL	93
4	Conception avancée	95
4.1	Programmation dynamique [Cormen et al., 1994, Cormen et al., 2001, Papadimitriou et al., 2006]	95
4.1.1	Multiplication d'une suite de matrices	96
4.1.2	Plus longue sous-séquence commune [Cormen et al., 2001]	103
4.1.3	Triangulation optimale de polygones	109
4.2	Algorithmes gloutons [Cormen et al., 1994, Cormen et al., 2001, Papadimitriou et al., 2006]	110
4.2.1	Le problème du choix d'activités	110
4.2.2	Eléments de la stratégie gloutonne	113
4.2.3	Codages de Huffman	113
4.3	Travaux dirigés IV	116
4.3.1	Algorithme glouton vs prog. dyn. : pièces de monnaies	116
4.3.2	Algorithme glouton 2 : ordonnancement	116

4.3.3	Programmation dynamique 1 : course	116
4.3.4	Programmation dynamique 2 : arbres binaires	117
4.3.5	Programmation dynamique 3 : politique de vente	117
4.3.6	Programmation dynamique 4 : sac-à-dos et PVC	118
5	Eléments sur la théorie de la complexité	119
5.1	Problèmes faciles et problèmes difficiles [Prins, 1994]	119
5.1.1	Problèmes faciles	119
5.1.2	Problèmes difficiles	120
5.2	NP-complétude et les classes P et NP [Prins, 1994]	121
5.3	La classe P et NP [Prins, 1994, Cori et al., 2001]	122
5.4	Les problèmes NP-complets [Prins, 1994, Cori et al., 2001]	123
5.5	Comment démontrer qu'un problème est NP-complet ? [Prins, 1994, Cori et al., 2001]	124
5.5.1	3-SAT [Cori et al., 2001]	124
5.5.2	Autres instances de SAT dans NPC [Cori et al., 2001]	126
5.5.3	Autres instances de SAT dans P [Cori et al., 2001]	126
5.5.4	Stable [Cori et al., 2001]	126
5.5.5	Clique [Cori et al., 2001]	127
5.5.6	Recouvrement par sommets [Cori et al., 2001]	127
5.5.7	Circuit hamiltonien [Cori et al., 2001]	128
5.5.8	Voyageur de commerce [Cori et al., 2001]	129
5.5.9	Autres instances des graphes dans NPC [Cori et al., 2001]	130
5.5.10	Autres instances des graphes dans P [Cori et al., 2001]	130
5.5.11	Colorabilité dans NPC [Cori et al., 2001]	130
5.5.12	Colorabilité dans C [Cori et al., 2001]	130
5.5.13	Somme de sous-ensemble [Cori et al., 2001]	130
5.5.14	Programmation entière [Cori et al., 2001]	131
5.6	La conjecture $P \neq NP$ [Prins, 1994]	131
5.7	Impact sur l'approche pratique d'un problème réel [Prins, 1994]	132
5.8	Problèmes de décision et langages [Alliot and Schiex, 1993]	133
5.9	Compléments sur la classification des problèmes	134
5.10	Travaux dirigés V	135
6	Algorithmes d'approximation [Robert et al., 2005]	137
6.1	Vertex cover	137
6.2	Voyageur de commerce : TSP	137
6.3	Bin Packing : BP	137
6.4	2-Partition	137
7	Définitions et types d'algorithmes	139
8	Ressources d'auto-formation	141
8.1	LeetCode	141
9	Annexe : écriture des algorithmes	143

10 Annexe : compléments sur les récurrences	145
10.1 Récurrences non linéaire du premier ordre [Sedgewick and Flajolet, 1996]	145
10.2 Séries génératrices [Sedgewick and Flajolet, 1996]	147
10.2.1 Exercices	151
11 Annexe : préliminaires mathématiques	153
11.1 Calcul ensembliste	153
11.1.1 Langage ensembliste	153
11.1.2 Fonctions	153
11.1.3 Cardinaux	154
11.1.4 Opérateurs et relations	154
11.2 Ensembles ordonnés, récursion et induction	155
11.2.1 Les relations d'ordre	155
11.2.2 Ensembles bien fondés et induction	156
11.2.3 Induction sur les entiers	156
11.2.4 Définitions inductives, et preuves par induction structurelle .	156
11.2.5 Treillis et points fixes	157
11.3 Logique	158
11.3.1 Logique propositionnelle	158
11.3.2 Inférence et déduction	159
11.3.3 La logique du premier ordre	160

Chapitre 1

Introduction

Une des premières références historiques sur les algorithmes date du neuvième siècle, à Bagdad, dont l'auteur est Al Khwarizmi qui a proposé les méthodes de base pour additionner, multiplier, et diviser des nombres entiers - et aussi le calcul des racines d'une équation, et les décimales du nombre π . Ces procédures étaient précises, non-ambigües, mécaniques, efficaces, et correctes : elles étaient simplement des *algorithmes*, un terme qui a honoré son concepteur El Khwarizmi.

Depuis cet avènement, et celui de l'adoption du système décimal, les scientifiques ont développé, et développent des algorithmes complexes pour résoudre des problèmes variés.

1.1 Calculer les nombres de Fibonacci [Papadimitriou et al.,

Soit la suite de Fibonacci (F_n)

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{si } n > 1, \\ 1 & \text{si } n = 1, \\ 0 & \text{si } n = 0. \end{cases}$$

La suite génère les nombres suivants

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \dots$$

Cette suite a aussi la caractéristique d'être de nature exponentielle, d'ailleurs il existe l'approximation $F_n \approx 2^{0.694n}$. Mais, comment pourrons nous calculer des nombres aussi grands que F_{100} ou même F_{200} ?

1.1.1 Une première version

Algorithm 1 FIB1(n)

```

1: if  $n = 0$  then
2:   return 0
3: else if  $n = 1$  then
4:   return 1
5: else
6:   return fib1( $n - 1$ ) + fib1( $n - 2$ )
7: end if
```

Trois questions essentielles sont posées :

1. **L'algorithme est-il correct ?**
2. **Combien de temps exige-t-il pour terminer, en fonction de n ?**
3. **Peut-on l'améliorer ?**

La réponse à la première question est plutôt évidente : l'algorithme est bien correct, car il met en oeuvre exactement la définition récurrente de la suite de Fibonacci. La deuxième question est délicate. Soit $T(n)$ le nombre de pas dont a besoin l'algorithme pour calculer F_n . Une première évidence :

$$T(n) \leq 2 \text{ pour } n \leq 1.$$

Pour des nombres n plus grands, on a la relation suivante :

$$T(n) = T(n - 1) + T(n - 2) + 3 \text{ pour } n > 1.$$

Nous disposons d'une solution approchée $T_n \approx 2^{0.694n}$. Ceci veut dire que le nombre de pas croît aussi vite que la suite de Fibonacci. $T(n)$ a évidemment une **croissance exponentielle**.

Soit par exemple le calcul de F_{200} , le calcul de FIB1(n) exécute $T(200) \geq F_{200} \geq 2^{138}$ pas élémentaires. On se pose la question sur le temps nécessaire pour ce calcul sur un ordinateur ? La réponse est immédiate : prenons un ordinateur puissant à l'heure actuelle, exécutant 40 mille milliards 40×10^{12} instructions par seconde (un ordinateur personnel, un PC, avec un processeur i7-3770 peut exécuter 20×10^{10} opérations flottantes par seconde - on dit 200 gigaFLOPS). Nous aurons besoin sur cet ordinateur de **2⁹² secondes** ! Ce qui veut dire que l'on doit attendre jusqu'à l'extinction théorique du soleil !

D'après l'hypothèse de Moore qui suppose que les ordinateurs doublent leur puissance tous les 18 mois, alors, on pourrait arriver au fait que : si on arrivait à calculer raisonnablement F_{100} , alors avec l'hypothèse de Moore, on pourrait vraisemblablement, calculer avec la même puissance $F(101), \dots$. Voulant dire qu'on gagnera chaque année un chiffre. C'est ainsi qu'on se voit totalement découragé face à la réalité terrible de la croissance exponentielle qui **ne peut être cassée par aucune machine de nos jours !**

La question reste posée : **peut on mieux faire en terme de coût de calcul ?** Et sans passer par l'amélioration des machines, dont on vient de voir l'impuissance !

1.1.2 Une version polynomiale

On montre dans la Figure 1.1, le comportement de l'algorithme FIB1.

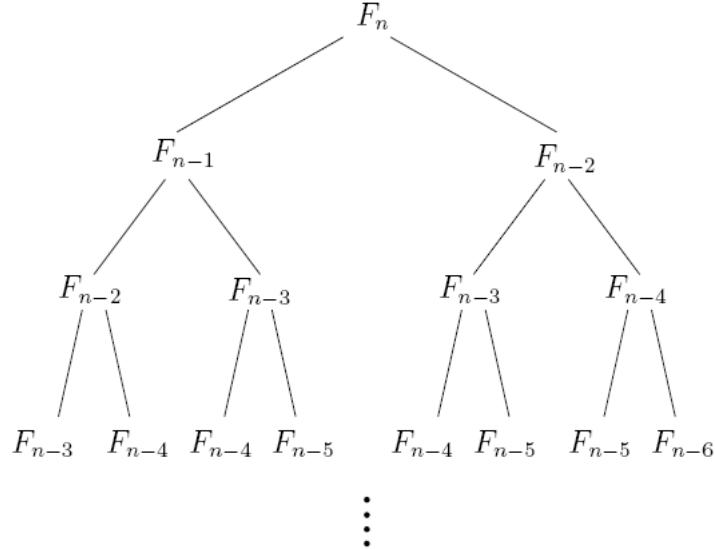


FIGURE 1.1 – Trace de $\text{FIB1}(n)$ [Papadimitriou et al., 2006]

En analysant cette trace, on voit bien que plein d'appels sont répétés. On peut donc adopter l'astuce qui consiste à stocker les calculs déjà faits. On obtient la version FIB2.

Algorithm 2 $\text{FIB2}(n)$

```

1: if  $n = 0$  then
2:   return 0
3: end if
4: créer un tableau  $F[0..n]$ 
5:  $F[0] \leftarrow 0$ 
6:  $F[1] \leftarrow 1$ 
7: for  $i \leftarrow 2..n$  do
8:    $F[i] \leftarrow F[i - 1] + F[i - 2]$ 
9: end for
10: return  $F[n]$ 
  
```

L'algorithme est évidemment correct, car, encore une fois, il est similaire à la définition récurrente de Fibonacci. La question : combien requiert cet algorithme pour calculer F_n en fonction de n ? La boucle de l'algorithme nécessite $n - 1$ pas de calculs. De ce fait, le temps d'exécution de **cet algorithme est de nature linéaire en n** . Il devient maintenant possible de calculer F_{200000} avec peu d'efforts !

Bien concevoir un algorithme fait gagner des exponentielles !

1.2 Algorithme, correction et complexité - Illustration sur le tri par insertion [Cormen et al., 2001]

Un algorithme est un ensemble d'opérations de calcul élémentaires, organisées selon des règles précises dans le but de résoudre un problème donné. Pour chaque donnée du problème, l'algorithme retourne une réponse après un nombre fini d'opérations. Les opérations élémentaires sont par exemple les opérations arithmétiques usuelles, les transferts de données, les comparaisons entre données, etc.

Il apparaît utile de ne **considérer comme véritablement élémentaires que les opérations dont le temps de calcul est constant**, c'est-à-dire ne dépend pas de la taille des opérandes. Par exemple :

- l'addition d'entiers de taille bornée à priori (les int en C) est une opération élémentaire ;
- l'addition d'entiers de taille quelconque ne l'est pas.
- De même, le test d'appartenance d'un élément à un ensemble n'est pas une opération élémentaire en ce sens, parce que son temps d'exécution dépend de la taille de l'ensemble, et ceci même si dans certains langages de programmation, il existe des instructions de base qui permettent de réaliser cette opération.

Nous avons déjà pris goût aux algorithmes avec l'algorithme de calcul des termes de la suite de Fibonacci pour lequel nous avons introduit une première analyse simple. Pour illustrer notre démarche, nous aborderons un problème posé fréquemment en pratique, celui du tri d'une suite de nombres en ordre croissant.

Nous pouvons définir formellement ce problème comme suit :

Entrée Une séquence de n nombres $\langle a_1, a_2, \dots, a_n \rangle$.

Sortie Une permutation $\langle a'_1, a'_2, \dots, a'_n \rangle$ de la permutation de la suite d'entrée telle que $a_1 \leq a_2 \leq \dots \leq a_n$.

Etant donnée une suite d'entrée telle que $\langle 5, 90, 45, 89 \rangle$, un algorithme de tri fournira en sortie $\langle 5, 45, 89, 90 \rangle$. On dira d'une telle suite d'entrée que c'est une instance du problème.

Nous introduirons une première solution au problème du tri avec la version du TRI PAR INSERTION.

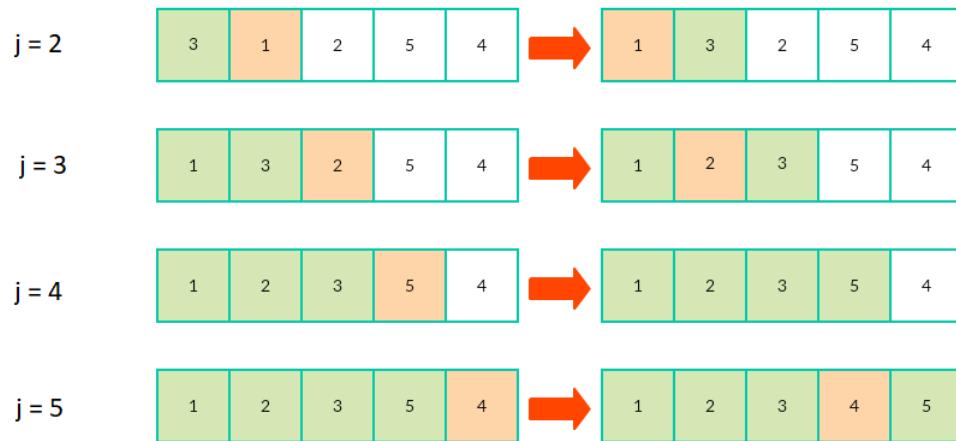
Algorithm 3 TRI-INSERTION(A)

Input: Une séquence de n nombres $A = \langle a_1, a_2, \dots, a_n \rangle$.

Output: Une permutation $\langle a'_1, a'_2, \dots, a'_n \rangle$ de la permutation de la suite d'entrée telle que $a_1 \leq a_2 \leq \dots \leq a_n$.

```

1: for  $j \leftarrow 2$  to  $\text{longueur}(A)$  do
2:    $pivot \leftarrow A[j]$ 
3:   {insertion de  $A[j]$  dans la suite triée  $A[1..j - 1]$ }
4:    $i \leftarrow j - 1$ 
5:   while ( $i > 0$ )  $\wedge (A[i] > pivot)$  do
6:      $A[i + 1] \leftarrow A[i]$ 
7:      $i \leftarrow i - 1$ 
8:   end while
9:    $A[i + 1] \leftarrow pivot$ 
10: end for
```

FIGURE 1.2 – Illustration de TRI-INSERTION(A)

1.2.1 Correction

On dit qu'un algorithme est correct si, pour chaque instance d'entrée, il se termine avec la sortie désirée correcte. Dans ce document, nous décrirons généralement les algorithmes au moyen de programmes écrits en pseudo-code, très proche de C, Java, ou aussi Pascal.

Nous rappelons que l'invariant d'une boucle, dans un algorithme, est une propriété qui est tout le temps vraie à :

1. l'entrée du corps de la boucle,
2. la sortie du corps de la boucle,
3. à la fin de la boucle.

Pour prouver l'invariant, il est nécessaire de procéder comme suit :

Initialisation La propriété invariante est vraie à l'entrée de la première itération.

Maintenance Si la propriété est vraie dans une itération, l'invariant reste vraie dans l'itération suivante.

Terminaison A la fin de la boucle, l'invariant est suffisant pour démontrer la correction.

L'établissement de cette preuve permet systématiquement d'avoir la preuve de correction de l'algorithme.

Exemple

L'invariant de la boucle de l'algorithme est :

Invariant 1 (Invariant de la boucle TRI-INSERTION). *Le sous-tableau $A[1..j - 1]$ contient tous les éléments originaux de $A[1..j - 1]$, et dans un ordre croissant.*

On exploitera cette propriété pour démontrer que l'algorithme est correct.

La preuve se présente comme suit :

Initialisation Il est évident que $A[1..1]$ est bien trié.

Maintenance Supposons que l'algorithme est arrivé à la j ième itération, d'où $A[1..j - 1]$ est trié. A la fin de la j ième itération, l'algorithme aurait placé $A[j]$ à la première case $i + 1$, $i < j$ où $A[i] < A[j]$. Ceci va faire en sorte à ce que $A[1..j]$ soit aussi trié.

Terminaison A la fin de la boucle, on aurait $A[1..n]$ trié. D'où la correction de l'algorithme.

1.2.2 Complexité

Le temps pris par la procédure TRI-INSERTION dépend de la **taille de son entrée** : le tri d'un millier de nombres prend plus de temps que le tri de trois nombres. La notion de taille d'entrée dépend du problème étudié, parfois il est plus approprié de décrire la taille de l'entrée avec deux nombres. Par exemple, la taille de l'entrée dans un problème sur les graphes, peut être le nombre de sommets et le nombre d'arêtes. **Pour chaque problème étudié, nous préciserons la taille (mesure) utilisée.**

Le temps d'exécution d'un algorithme sur une entrée particulière est le nombre d'opérations élémentaires exécutées.

Le temps d'exécution de l'algorithme est la somme des temps d'exécution de chaque instruction exécutée ; une instruction qui s'exécute en un temps c_i et qui est exécutée n fois interviendra pour $c_i n$. Pour calculer $T(n)$, le temps d'exécution de TRI-INSERTION, on additionne les produits des coûts par le nombre de fois, et on obtient :

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2..n} t_j + c_6 \sum_{j=2..n} (t_j - 1) + c_7 \sum_{j=2..n} (t_j - 1) + c_9(n-1),$$

où t_j est le nombre d'itérations de la deuxième boucle à l'itération j de la première boucle. Si la suite est déjà ordonnée, on a

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_9(n-1) = (c_1 + c_2 + c_4 + c_5 + c_9)n + (c_2 + c_4 + c_5 + c_9).$$

On peut exprimer cette dernière formule sous la forme $an + b$, où a et b sont des constantes. On dit que l'algorithme est en $\mathcal{O}(n)$.

Si la suite est dans un ordre inverse, on se trouve dans le pire des cas. On doit comparer chaque élément $A[j]$ avec chaque élément de sous-tableau trié $A[1..j - 1]$, et donc $t_j = j$ pour $j = 2, 3, \dots, n$. Sachant que

$$\sum_{j=2..n} j = n(n+1)/2 - 1$$

et

$$\sum_{j=2..n} (j - 1) = n(n - 1)/2.$$

On obtient ainsi

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n(n+1)/2 - 1) + c_6(n(n-1)/2) + c_7(n(n-1)/2) + c_9(n-1)$$

$$T(n) = (c_5/2 + c_6/2 + c_7/2)n^2 + (c_1 + c_2 + c_4 + c_5/2 + c_6/2 + c_7/2 + c_9)n.$$

Ce dernier coût est de la forme $an^2 + bn + c$ où a , b et c sont des constantes. On dit que l'algorithme est en $\mathcal{O}(n^2)$. Nous venons donc de calculer le coût de l'algorithme du TRI-INSERTION dans le pire des cas, c'est-à-dire le temps d'exécution le plus long pour une entrée quelconque de taille n . C'est la mesure de coût la plus utilisée en pratique. Nous verrons dans la suite un ensemble de procédés permettant de calculer ce coût.

1.3 Récursivité et l'approche diviser pour régner [Gaudel et al., 1990, Cormen et al., 1994, Cormen et al., 1990]

L'expression d'algorithmes sous forme récursive permet des descriptions concises qui se prêtent bien à des démonstrations par récurrence. **Le principe est d'utiliser, pour décrire l'algorithme sur une donnée d , l'algorithme lui-même appliqué à un sous-ensemble de d ou à une donnée d' plus petite.** Le programme de calcul des nombres de FIBONACCI donné dans le chapitre précédent est une bonne illustration du procédé.

Nombre d'algorithmes utiles sont de structure récursive : pour résoudre un problème donné, ils s'appellent eux-mêmes récursivement une ou plusieurs fois sur des sous-problèmes très similaires. Ces algorithmes choisissent l'approche "diviser pour régner" : ils séparent le problème en plusieurs sous-problèmes similaires au problème initial, mais de taille moindre, résolvent les sous-problèmes de façon récursive, puis combinent ces solutions pour retrouver une solution au problème initial.

Ce paradigme de résolution donne lieu à trois étapes à chaque niveau de récursivité :

Diviser le problème en un certain nombre de sous-problèmes.

Régner sur les sous-problèmes en les résolvant récursivement. Par ailleurs, si la taille d'un sous-problème est assez réduite, on peut le résoudre directement.

Combiner les solutions aux sous-problèmes en une solution complète pour le problème initial.

1.3.1 Tri par fusion

L'algorithme de tri par fusion suit très fidèlement la règle du "diviser pour régner". Il agit de la manière suivante :

Diviser Diviser la séquence de n éléments à trier en deux sous-séquences de $n/2$ éléments.

Régner Trier les deux séquences récursivement à l'aide du tri par fusion.

Combiner Fusionner les deux sous-séquences triées pour produire la réponse triée.

Algorithm 4 TRI-FUSION(A, p, r)

Input: Une séquence de n nombres $A = \langle a_1, a_2, \dots, a_n \rangle$.

Output: Une permutation $\langle a'_1, a'_2, \dots, a'_n \rangle$ de la permutation de la suite d'entrée telle que $a_1 \leq a_2 \leq \dots \leq a_n$.

```

1: if  $p < r$  then
2:    $q \leftarrow \lfloor (p + r)/2 \rfloor$ 
3:   TRI-FUSION( $A, p, q$ )
4:   TRI-FUSION( $A, q + 1, r$ )
5:   FUSIONNER( $A, p, q, r$ )
6: end if

```

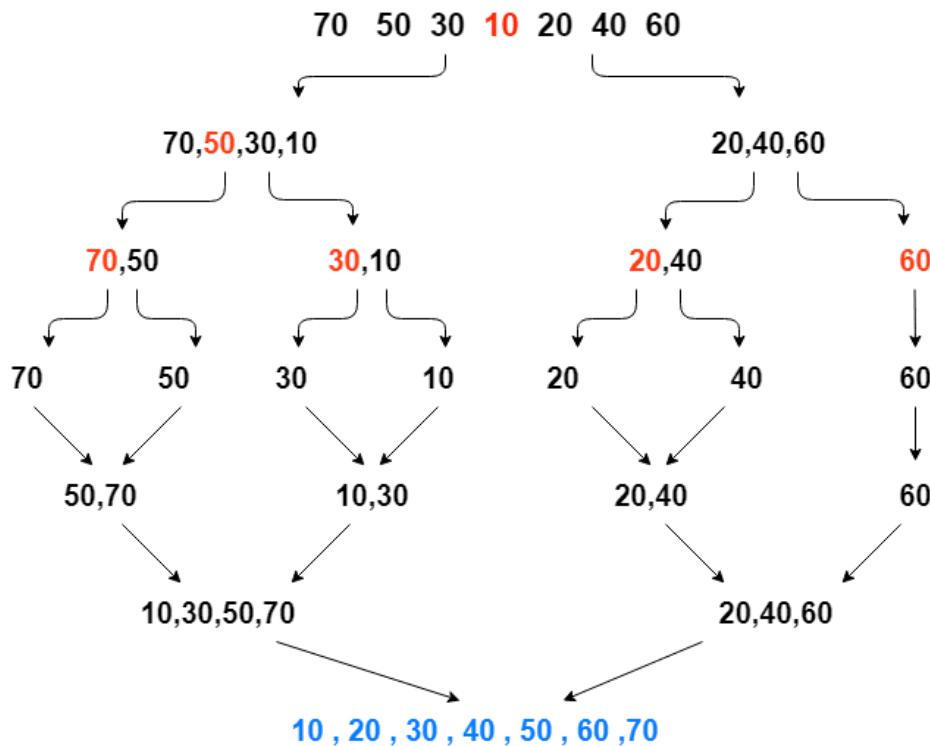


FIGURE 1.3 – Illustration de TRI-FUSION <https://dev.to/rajatm544/implement-5-sorting-algorithms-using-javascript-1fc3>

Lorsqu'un algorithme contient un appel récursif à lui même, son temps d'exécution peut souvent être décrit par une équation de récurrence ou récurrence, qui décrit le temps d'exécution global pour un problème de taille n en fonction du temps d'exécution pour des entrées de taille moindre. On peut se servir d'outils mathématiques pour résoudre la récurrence et trouver des bornes pour les performances de l'algorithme.

Ici, on appelle $T(n)$ le temps d'exécution d'un problème de taille n . Si la taille du problème est assez réduite, disons $n \leq c$ pour une certaine constante c , la solution directe consomme un temps constant, qu'on écrit $\Theta(1)$. Supposons qu'on divise le problème en a sous-problèmes, la taille de chacun étant $1/b$ de la taille du problème initial. Si l'on prend un temps $D(n)$ pour diviser le problème en sous-problèmes et un temps $C(n)$ pour construire la solution finale à partir des solutions aux sous-problèmes, on obtient la récurrence

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq c, \\ aT(n/b) + D(n) + C(n) & \text{sinon.} \end{cases}$$

Notre analyse fondée sur la récurrence est simplifiée si nous supposons que la taille du problème initial est une puissance de deux. Chaque étape "diviser" génère alors deux sous-suites de taille $n/2$ exactement. Nous verrons plus loin que cette hypothèse n'affecte pas l'ordre de grandeur de la solution de la récurrence.

Le tri par fusion sur un seul élément prend un temps constant. Avec $n > 1$ éléments, on segmente le temps d'exécution comme suit.

Diviser Elle se contente de calculer le milieu du sous-tableau, ce qui consomme un temps constant, $D(n) = \Theta(1)$.

Régner On résoud récursivement deux sous problèmes, d'où $2T(n/2)$.

Combiner On peut démontrer que la procédure FUSIONNER sur un sous-problème à n éléments est de la forme $C(n) = an + b$, où a et b sont des constantes, on notera cette forme comme suit $C(n) = \Theta(n)$.

On a donc

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq c, \\ 2T(n/2) + \Theta(n) & \text{si } n > 1. \end{cases}$$

On démontrera plus loin que $T(n)$ est de la forme $an\log(n) + bn + c$, où a , b , et c sont des constantes. On notera cette forme $\Theta(n \log(n))$. On peut aussi remarquer que ce coût est nettement inférieur à celui du tri par insertion, car $\exists n_0, \forall n, n > n_0, n^2 > n\log(n)$.

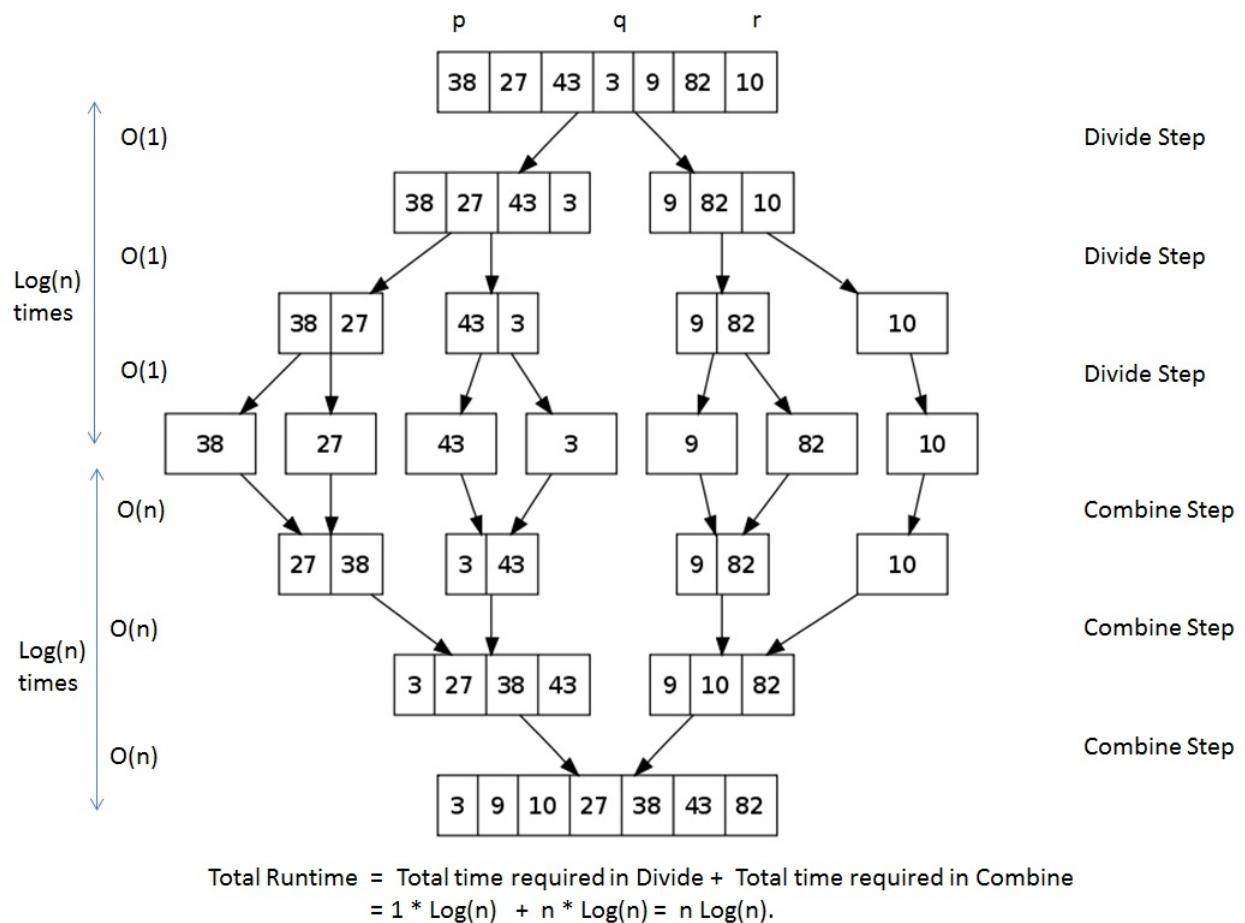


FIGURE 1.4 – Complexité de TRI-FUSION <http://miftyisbored.com/quicksort-implementation-java/>

1.3.2 Exponentiation rapide

Il s'agit de calculer x^n pour x et n données, en calculant la complexité par rapport à n . La méthode naïve (multiplier n fois 1 par x) donne une complexité linéaire. En utilisant le fait que

$$\begin{cases} x^0 = 1 \\ x^n = (x^2)^{\frac{n}{2}} \\ x^n = x(x^2)^{\frac{n-1}{2}} \end{cases}$$

qui se traduit d'un point de vue algorithmique en :

- Si n est pair, x^n revient à calculer la puissance de x^2 à $n/2$.
 - Si n est impair, x^n revient à multiplier x par la puissance de x^2 à $(n-1)/2$.
- D'où l'algorithme :

Algorithm 5 PUISSDYN(x, n)

```

1: if ( $n = 0$ ) then
2:   return 1
3: else if  $n$  est pair then
4:   return PUISSDYN( $x \times x, n/2$ )
5: else
6:   return  $x \times$ PUISSDYN( $x \times x, (n - 1)/2$ )
7: end if
```

La complexité de l'exponentiation rapide est donc en $O(\log n)$.

Chapitre 2

Principes de l'analyse des algorithmes

2.1 Mesure du coût [Beauquier et al., 1992, Cormen et al., 1990, Cormen et al., 2001, Gaudel et al., 1990]

Analyser un algorithme consiste à prévoir **les ressources à cet algorithme**. Parfois, les ressources pertinentes sont la **mémoire** utilisée, la **largeur de bande d'une communication**, ou les **portes logiques**, mais le plus souvent, on souhaite mesurer le **temps de calcul**. Dans ce cours, on prendra comme modèle de calcul, **une machine à accès aléatoire, à processeur unique**. Dans ce modèle, les instructions sont exécutées l'une après l'autre, sans opérations simultanées.

Considérons un problème donné, et un algorithme pour le résoudre. Sur une donnée x de taille n , l'algorithme requiert un certain temps, mesuré en nombre d'opérations élémentaires, soit $c(x)$. Le coût en temps varie évidemment avec la taille de la donnée, mais peut aussi varier sur les différentes données de même taille n .

Par exemple, considérons l'algorithme de tri qui, partant d'une suite (a_1, \dots, a_n) de nombres réels distincts à trier en ordre croissant, cherche la première descente, c'est-à-dire le plus petit entier i tel que $a_i > a_{i+1}$, échange ces deux éléments, et recommence sur la suite obtenue. Si l'on compte le nombre d'inversions ainsi réalisées, il varie de 0 pour une suite triée à $n(n - 1)/2$ pour une suite décroissante. Notre but est d'**évaluer le coût d'un algorithme, selon certains critères, et en fonction de la taille n des données**.

Pour certains problèmes, on peut **mettre en évidence une ou plusieurs opérations qui sont fondamentales** au sens où le temps d'exécution d'un algorithme résolvant ce problème est toujours proportionnel au nombre de ces opérations. Il est alors possible de comparer des algorithmes traitant ce problème selon cette mesure simplifiée.

Donnons quelques exemples d'**opérations fondamentales** :

1. pour la recherche d'un élément dans une liste en mémoire centrale : le nombre de comparaisons entre cet élément et les entrées de la liste ;
2. pour la recherche d'un élément sur un disque : le nombre d'accès à la mémoire secondaire ;
3. pour trier une liste d'éléments : on peut considérer deux opérations fondamentales : le nombre de comparaisons entre des éléments et le nombre de déplacements d'éléments ;
4. pour multiplier deux matrices de nombres : le nombre de multiplications et le nombre d'additions.

Remarquons que si l'on choisit plusieurs opérations fondamentales, on **peut les décompter séparément** puis, si besoin est, on les affecte chacune d'un poids qui tient compte des temps d'exécution différents.

1. En faisant **varier le nombre d'opérations fondamentales**, on fait varier le **degré de précision de l'analyse**, et aussi son degré d'abstraction, i.e. d'indépendance par rapport à l'implémentation. A la limite, si l'on veut **faire une microanalyse très précise du temps d'exécution du programme, il suffit de décider que toutes les opérations du programme sont fondamentales**.
2. On a fait l'hypothèse que le temps d'exécution est proportionnel à la mesure choisie. On ne peut pas comparer deux algorithmes utilisant des mesures différentes.

Après avoir déterminé les opérations fondamentales, il s'agit de compter le nombre d'opérations de chaque type. Il n'existe pas de systèmes complet de règles permettant de compter le nombre d'opérations en fonction de la syntaxe des algorithmes mais l'on peut faire **quelques remarques** :

Séquence Lorsque les opérations sont dans une séquence d'instructions, leurs nombres s'ajoutent.

if-then-else Pour les branchements conditionnels, il est en général difficile de déterminer quelle branche de la condition est exécutée, et donc quelles sont les opérations à compter. Cependant, on peut majorer ce nombre d'opérations.

Boucles Pour les boucles, le nombre d'opérations dans la boucle est $\sum P(i)$, où i est la variable de contrôle de la boucle, et $P(i)$ le nombre d'opérations fondamentales lors de l'exécution de la i ème itération. Si le nombre d'itérations est difficile à calculer, on peut se contenter d'une bonne majoration.

Appels procédures Pour les appels de procédures et fonctions non récursives, on peut s'arranger à calculer la complexité de ces appels, et les prendre en compte suivant l'imbrication de l'appel dans l'algorithme.

Appels récursifs Pour les appels de procédures et fonctions récursives, compter le nombre d'opérations fondamentales donne en général lieu à la **Résolution de relations de récurrence**. En effet le nombre $T(n)$ d'opérations dans l'appel de la procédure avec un argument de taille n s'écrit, selon la récursion, en **fonction de divers $T(k)$, pour $k < n$** . L'exemple de Fibonacci donné dans le chapitre d'introduction illustre bien ce cas.

Il est évident que **le calcul du coût d'un algorithme dépend de la donnée sur laquelle il opère**.

1. Il faut d'abord **définir une mesure de taille sur les données** qui reflète la quantité d'information contenue. **Par exemple, si l'on additionne ou on multiplie des entiers, une mesure significative est le nombre de chiffres des nombres.**
2. Pour certains algorithmes, le temps d'exécution ne dépend que de la taille des données ; mais la plupart du temps la complexité varie aussi, pour une taille fixée des données, en fonction de la donnée elle-même. **Ainsi le nombre d'opérations variera suivant la nature de la donnée : d'où le recours aux complexités dans le pire des cas, dans le meilleur des cas, et en moyenne.**

Etant un problème $P(n)$ dont la taille de donnée est n . Notons **x une instance du problème P** . Exemple : soit $P(n)$ le problème de mise en ordre croissant d'une liste de longueur n . Une instance x est, par exemple, ordonner le tableau $[5, 7, 12, 6, 8]$ dont $n = 5$. Une autre instance $[154, 45, 89, 12, 1547, 4], \dots$

Le problème est défini d'une façon formelle pour avoir une compréhension **unique**, par contre son nombre d'instances est en général **infini**.

Le défi du calcul de la complexité est de pouvoir raisonner sur cette infinité d'instances, en établissant une fonction $c(n)$ qui caractérise le nombre d'opérations en fonction de la variable n sur la taille des données.

2.1.1 La complexité dans le meilleur des cas

Le coût $Min_A(n)$ d'un algorithme A dans le meilleur des cas sur un problème $P(n)$

$$Min_A(n) = \min_{|x|=n} c(x).$$

2.1.2 La complexité dans le pire des cas

Le coût $Max_A(n)$ d'un algorithme A , résolvant le problème $P(n)$, dans le cas le plus défavorable ou dans le cas le pire¹ est par définition le maximum des coûts, sur toutes les données de taille n :

$$Max_A(n) = \max_{|x|=n} c(x).$$

2.1.3 La complexité en moyenne

Dans des situations où l'on pense que **le cas le plus défavorable ne se présente que rarement, on est plutôt intéressé par le coût moyen de l'algorithme**. Une formulation correcte de ce coût moyen suppose que l'on connaisse une distribution de probabilités sur les données de taille n . **Si $p(x)$ est la probabilité de la donnée x** , le coût moyen $Moy_A(n)$ d'un algorithme A , résolvant le problème $P(n)$, sur les données de taille n est par définition

$$Moy_A(n) = \sum_{|x|=n} p(x)c(x).$$

Le plus souvent, on suppose que la distribution est uniforme, c'est-à-dire que $p(x) = 1/T(n)$, où **$T(n)$ est le nombre de types de données de taille n** . Alors, l'expression du coût moyen prend la forme

$$Moy_A(n) = \frac{1}{T(n)} \sum_{|x|=n} c(x).$$

En pratique, la complexité en moyenne est souvent beaucoup plus difficile à déterminer que la complexité dans le pire des cas , d'une part parceque l'analyse devient mathématiquement difficile, et d'autre part parcequ'il n'est pas toujours facile de déterminer un modèle de probabilités adéquat au problème.

En claire, il existe entre les complexités en moyenne et les complexités extrêmales la relation suivante :

$$Min_A(n) \leq Moy_A(n) \leq Max_A(n).$$

Si le comportement de l'algorithme dépend uniquement de la taille des données (comme dans l'exemple de la multiplication de matrices), alors ces trois quantités sont confondues. Mais en général, ce n'est pas le cas et l'on ne sait même pas si le coût moyen est plus proche du coût minimal ou du coût maximal.

1. “worst-case” en anglais.

2.2 Mesure asymptotique, grandeurs des fonctions et notations de Landau : O, ω, \dots [Gaudel et al., 1990]

On a déterminé la complexité d'un algorithme comme une fonction de la taille des données ; il est très important de connaître la rapidité de croissance de cette fonction lorsque la taille des données croît. En effet, pour traiter un problème de petite taille la méthode employée importe peu, alors que pour un problème de grande taille, les différences de performance entre algorithmes peuvent être énormes.

Souvent une simple approximation de la fonction de complexité suffit pour savoir si un algorithme est utilisable ou non, ou pour comparer entre différents algorithmes.

Par exemple, pour n grand, il est souvent secondaire de savoir si un algorithme fait $n+1$ ou $n+2$ opérations.

Parfois les constantes multiplicatives ont, elles aussi, peu d'importance.

- Supposons que l'on ait à comparer l'algorithme A_1 de complexité $M_1(n) = n^2$ et l'algorithme A_2 de complexité $M_2(n) = 2n$. A_2 est meilleur que A_1 pour presque tous les $n (n > 2)$;
- De même si $M_1(n) = 3n^2$ et $M_2(n) = 25n$; A_2 est meilleur que A_1 pour $n > 8$.
- [Négliger un terme dans une addition] Quelles que soient les constantes multiplicatives k_1 et k_2 telles que $M_1(n) = k_1n^2$ et $M_2(n) = k_2n$, l'algorithme A_2 est toujours meilleur que A_1 à partir d'un certain n , car la fonction $f(n) = n^2$ croît beaucoup plus vite que la fonction $g(n) = n$. En effet

$$\lim_{n \rightarrow \infty} g(n)/f(n) = 0.$$

On dit que l'ordre de grandeur asymptotique de $f(n)$ est strictement plus grand que celui de $g(n)$.

Dans ce cas, on peut écrire que $f(n)$ est du même ordre que $f(n) + g(n)$.

- [Négliger une constante] D'autre part, quelles que soient les constantes multiplicatives k_1 et k_2 telles que $M_1(n) = k_1f(n)$ et $M_2(n) = k_2f(n)$, l'algorithme A_2 est toujours du même ordre que A_1 , en effet

$$\lim_{n \rightarrow \infty} k_1f(n)/k_2f(n) = k_1/k_2.$$

Dans ce cas, on peut écrire que $M_1(n)$ est du même ordre que $M_2(n)$.

La Figure 2.1 met en évidence la différence de rapidité de croissance de certaines fonctions usuelles : les ordres de grandeur asymptotique des fonctions $1, \log_2(n), n\log_2(n), n^2, n^3, 2^n$ vont en croissant strictement ; ces fonctions forment une échelle de comparaison.

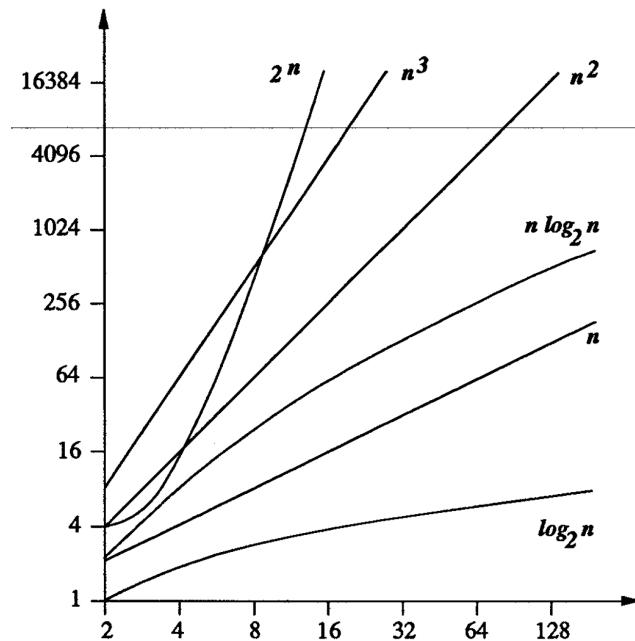


FIGURE 2.1 – Croissance de certaines fonctions usuelles [Gaudel et al., 1990]

Pour analyser la complexité $M_A(n)$ d'un algorithme A , on s'attache d'abord à déterminer l'ordre de grandeur asymptotique de $M_A(n)$: **on cherche dans une échelle de comparaison, éventuellement plus complète que celle qui est formée par les fonctions de la Figure 2.1, une fonction qui a une rapidité de croissance voisine de $M_A(n)$.**

Supposons que l'on ait à comparer deux algorithmes A_1 et A_2 de complexités $M_{A_1}(n)$ et $M_{A_2}(n)$. **Si l'ordre de grandeur de $M_{A_1}(n)$ est strictement plus grand que l'ordre de grandeur de $M_{A_2}(n)$, alors on peut conclure immédiatement que A_1 est meilleure que A_2 pour n grand.** Par contre, si $M_{A_1}(n)$ et $M_{A_2}(n)$ ont même ordre de grandeur asymptotique, il faut faire une analyse plus fine pour pouvoir comparer A_1 et A_2 .

Pour comparer l'ordre de grandeur asymptotique des fonctions, dans notre cas des fonctions de complexité, on a l'habitude d'utiliser la notion suivante :

Définition 1. Etant donné deux fonctions f et g de \mathbb{N} dans \mathbb{R}^+ ,

$$f = O(g)$$

si et seulement si $\exists c \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N}$ tel que

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

On dit que

$$f = \Omega(g)$$

si $\exists c \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N}$ tel que

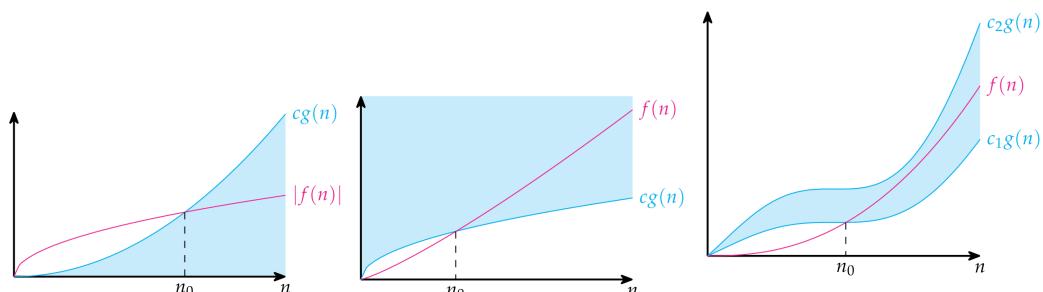
$$\forall n > n_0, f(n) \geq c.g(n) \geq 0.$$

En d'autres termes $f = \Omega(g)$ si et seulement si $g = O(f)$.

Ainsi $f = O(g)$ veut dire que l'ordre de grandeur asymptotique de f , est inférieur ou égal à celui de g , on dit que f est dominée asymptotiquement par g ; par exemple $2n = O(n^2)$, mais aussi $2n = O(n)$.

Cette notion qui donne un majorant de l'ordre de grandeur asymptotique de f , est très utile pour de nombreuses applications.

Interprétation de la définition 1 mathématique : $f(n)$ est la complexité inconnue de l'algorithme, $g(n)$ est une mesure asymptotique de $f(n)$. En effet, calculer le nombre exact d'opérations $f(n)$ est le plus souvent hors de portée, pour plusieurs raisons : dépendance du langage, options de compilation, système d'exploitation, processeur, complexité de la structure de l'algorithme. Cette difficulté est relevée en trouvant une fonction g qui majore, domine et borne le nombre d'opérations.



▲ Figure : $f(n) \in O(g(n))$: after some n_0 the function $|f(n)|$ is bounded above by $c g(n)$ for some constant c .

▲ Figure : $f(n) \in \Omega(g(n))$: after some n_0 the function $f(n)$ is bounded below by $c g(n)$ for some constant c .

▲ Figure : $f(n) \in \Theta(g(n))$: after some n_0 the function $f(n)$ is bounded by two copies of $g(n)$ with different scale factors.

FIGURE 2.2 – Illustration de $f = O(g)$, $f = \Omega(g)$ et $f = \Theta(g)$. <https://www.lrde.epita.fr/~adl/ens/algo/algo.pdf>

La majoration de la grandeur de la complexité, **n'est pas toujours suffisante** pour comparer les performances de différents algorithmes de même grandeur, car il faut connaître les ordres de grandeurs exacts, et non des majorants. Lorsque l'on dit que la complexité $M_A(n)$ d'un algorithme A est en $h(n)$, on veut dire que son **ordre de grandeur asymptotique est exactement** $h(n)$ (i.e. $h(n)$ est le plus petit majorant). Cette notion est formalisée comme suit :

Définition 2. *Etant donné deux fonctions f et g de \mathbb{N} dans \mathbb{R}^+ ,*

$$f = \Theta(g)$$

si et seulement si $\exists c, d \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N}$ *tel que*

$$\forall n > n_0, d.g(n) \leq f(n) \leq c.g(n).$$

Ou d'une façon équivalente,

$$f = \Theta(g)$$

si et seulement si $f = O(g)$ *et* $g = O(f)$.

On dit que ***f et g ont même ordre de grandeur asymptotique***. La notion ***Θ est plus précise que la notion O***. Par exemple $2n = \Theta(n)$, mais ***2n n'est pas en Θ(n²)***. Cependant, dans la plupart des ouvrages d'algorithmique, les résultats des analyses sont mis sous la forme $O(f(n))$, alors qu'un décompte précis des opérations fondamentales permet souvent de conclure que la complexité est exactement $\Theta(f(n))$. Par exemple, on dit souvent que le tri par tas est en $O(nlog(n))$, alors qu'en fait il est en $\Theta(nlog(n))$.

$\lambda = \Theta(1)$	$\lambda = O(1)$
$f(n) = \Theta(f(n))$	$f(n) = O(f(n))$
$\Theta(f(n)) + \Theta(g(n)) = \Theta(f(n) + g(n))$	$O(f(n)) + O(g(n)) = O(f(n) + g(n))$
$\Theta(f(n) + g(n)) = \Theta(\max(f(n), g(n)))$	$O(f(n) + g(n)) = O(\max(f(n), g(n)))$
$\Theta(f(n)) \cdot \Theta(g(n)) = \Theta(f(n) \cdot g(n))$	$O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$
$\Theta(\lambda f(n)) = \Theta(f(n))$	$O(\lambda f(n)) = O(f(n))$
if $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c > 0$	then $f(n) = \Theta(g(n))$
if $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$	then $f(n) = O(g(n))$ and $f(n) \neq \Theta(g(n))$
if $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$	then $f(n) = \Omega(g(n))$ and $f(n) \neq \Theta(g(n))$

FIGURE 2.3 – Propriétés de $f = O(g)$ et $f = \Theta(g)$. <https://www.lrde.epita.fr/~adl/ens/algo/algo.pdf>

SELECTIONSORT(A, n)	
1 for $i \leftarrow 0$ to $n - 2$ do	$\Theta(n)$
2 $min \leftarrow i$	$\Theta(n)$
3 for $j \leftarrow i + 1$ to $n - 1$ do	$\Theta(n^2)$
4 if $A[j] < A[min]$	$\Theta(n^2)$
5 $min \leftarrow j$	$O(n^2)$
6 $A[min] \leftrightarrow A[i]$	$\Theta(n)$
	$\Theta(n^2)$

INSERTIONSORT(A, n)	
1 for $i \leftarrow 1$ to $n - 1$ do	$\Theta(n)$
2 $key \leftarrow A[i]$	$\Theta(n)$
3 $j \leftarrow i - 1$	$\Theta(n)$
4 while $j \geq 0$ and $A[j] > key$ do	$O(n^2)$
5 $A[j + 1] \leftarrow A[j]$	$O(n^2)$
6 $j \leftarrow j - 1$	$O(n^2)$
7 $A[j + 1] \leftarrow key$	$\Theta(n)$
	$\Theta(n^2)$

$3n^2 - 100n + 6 = O(n^2)$, because I choose $c = 3$ and $3n^2 > 3n^2 - 100n + 6$;

$3n^2 - 100n + 6 = O(n^3)$, because I choose $c = 1$ and $n^3 > 3n^2 - 100n + 6$ when $n > 3$;

$3n^2 - 100n + 6 \neq O(n)$, because for any c I choose $c \times n < 3n^2$ when $n > c$;

$3n^2 - 100n + 6 = \Omega(n^2)$, because I choose $c = 2$ and $2n^2 < 3n^2 - 100n + 6$ when $n > 100$;

$3n^2 - 100n + 6 \neq \Omega(n^3)$, because I choose $c = 3$ and $3n^2 - 100n + 6 < n^3$ when $n > 3$;

$3n^2 - 100n + 6 = \Omega(n)$, because for any c I choose $cn < 3n^2 - 100n + 6$ when $n > 100c$;

$3n^2 - 100n + 6 = \Theta(n^2)$, because both O and Ω apply;

$3n^2 - 100n + 6 \neq \Theta(n^3)$, because only O applies;

$3n^2 - 100n + 6 \neq \Theta(n)$, because only Ω applies.

FIGURE 2.4 – Exemples d’application : Complexité du tri par sélection et du tri par insertion. <https://www.lrde.epita.fr/~adl/ens/algo/algo.pdf> et [Skiena, 2008]

Soulignons un point fondamental : les définitions de O et Θ reposent sur l'existence de certaines constantes finies, mais il n'est rien précisé sur la valeur de ces constantes. **Cela n'a pas d'importance pour obtenir des résultats asymptotiques lorsque les fonctions ont des ordres de grandeur différents.** Par exemple si $f(n) = 2n$ et $g(n) = n^2$, alors $f(n) < g(n)$ pour $n > 2$. Si $f(n) = 1000n$ et $g(n) = n^2$, alors **$f(n) < g(n)$ pour $n > 10^4$** .

Ainsi, si l'ordre de grandeur de f est plus petit que celui de g alors il existe un seuil à partir duquel la valeur de f est c fois plus petite que celle de g , mais on ne sait pas quel est ce seuil.

Par contre, si f et g ont même ordre de grandeur, il devient beaucoup plus difficile de les comparer : la détermination des constantes, et éventuellement des termes d'ordre inférieur nécessite en général des techniques mathématiques beaucoup plus complexes. Il faut bien être conscient de ce que l'obtention de résultats tels que : "l'algorithme A va deux fois plus vite que l'algorithme B sur un ordinateur standard", est en général très difficile.

La notion d'ordre de grandeur de la complexité des algorithmes a une grande importance pratique. Supposons que l'on dispose pour résoudre un problème donné de huit algorithmes dont les complexités dans le cas le pire ont respectivement pour ordre de grandeur 1 (c'est-à-dire une fonction constante, qui ne dépend pas de la taille des données), $\log_2(n)$, ..., 2^n (c'est-à-dire une fonction exponentielle).

▼ Table : An algorithm that requires $f(n)$ CPU cycles to process an input of size n will execute in $f(n)/(3 \times 10^9)$ seconds on a 3GHz CPU. This table shows run times for different f and n .

input size n	number $f(n)$ of operations to perform						
	$\log_2 n$	n	$n \log_2 n$	n^2	$n^{\log_2(7)}$	n^3	2^n
10^1	1.1 ns	3.3 ns	11.1 ns	33.3 ns	0.2 μ s	0.3 μ s	0.3 ms
10^2	2.2 ns	33.3 ns	0.2 μ s	3.3 μ s	0.1 ms	0.3 ms	1.3×10^{13} y
10^3	3.3 ns	0.3 μ s	3.3 μ s	0.3 ms	88.1 ms	0.3 s	1.1×10^{284} y
10^4	4.4 ns	3.3 μ s	44.2 μ s	33.3 ms	56.5 s	5.5 min	6.3×10^{3002} y
10^5	5.5 ns	33.3 μ s	0.5 ms	3.3 s	10.1 h	3.8 d	
10^6	6.6 ns	0.3 ms	6.6 ms	5.5 min	0.7 y	10.6 y	
10^7	7.8 ns	3.3 ms	77.5 ms	9.3 h	473.8 y	10 570.0 y	
10^8	8.9 ns	33.3 ms	0.9 s	28.6 d	30 402.1 y		
10^9	10.0 ns	0.3 s	10.0 s	10.6 y			
10^{10}	11.0 ns	3.3 s	1.8 min	1057.0 y			

FIGURE 2.5 – Temps d'exécution et taille des données <https://www.lrde.epita.fr/~adl/ens/algo/algo.pdf>

Le tableau de la Figure 2.5 donne une estimation du temps d'exécution de chacun de ces algorithmes pour différentes tailles n des données du problème sur un ordinateur pouvant effectuer 10^6 opérations par seconde. **Il montre bien que, plus la taille des données est grande, plus les écarts entre les différents temps d'exécution se creusent.**

Complexité Temps calcul	1	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
1s	∞	∞	10^6	63×10^3	10^3	100	19
1 mn	∞	∞	6×10^7	28×10^5	77×10^2	390	25
1 h	∞	∞	36×10^8	13×10^7	60×10^3	15×10^2	31
1 jour	∞	∞	86×10^9	27×10^8	29×10^4	44×10^2	36

FIGURE 2.6 – Temps d'exécution et taille des données [Gaudel et al., 1990]

Le tableau de la Figure 2.6 donne une estimation de la taille maximale des données que l'on peut traiter par chacun des algorithmes en un temps d'exécution fixé (et toujours sur un ordinateur effectuant 10^6 opérations par seconde).

D'après ces deux tableaux, il est clair que certains algorithmes sont utilisables pour résoudre des problèmes sur ordinateurs, et que d'autres ne sont pas, ou peu utilisables.

Les algorithmes utilisables pour des données de grande taille sont ceux qui s'exécutent en temps :

constant (c'est le cas de la complexité en moyenne de certaines méthodes de hachage) ;

logarithmique (par exemple la recherche dichotomique, ou les arbres binaires de recherche) ;

linéaire (par exemple la recherche séquentielle) ;

$n \cdot \log(n)$ (par exemple les bons algorithmes de tri).

Les algorithmes qui prennent un temps polynomial, c'est-à-dire en $\Theta(n^k)$ avec $k > 0$, ne sont vraiment utilisables que pour $k < 2$.

Lorsque $2 \leq k \leq 3$, on peut traiter que des problèmes de taille moyenne, et lorsque k dépasse 3 on ne peut traiter que des petits problèmes.

Les algorithmes en temps exponentiel, c'est-à-dire en $\Theta(2^n)$ par exemple, sont à peu près inutilisables, sauf pour des problèmes de très petit taille. Ce sont de tels algorithmes que l'on a qualifiés d'inefficaces.

Le tableau de la Figure 2.6 montre comment la taille des données et le temps d'exécution varient en fonction l'un de l'autre. On voit en particulier que si l'on multiplie par 10 la vitesse de calcul de l'ordinateur, on ne modifie quasiment pas la taille maximale des données que l'on peut traiter avec un algorithme exponentiel, alors que l'on multiplie évidemment par 10 la taille des données traitables par un algorithme linéaire. **Il est donc toujours d'actualité de rechercher des algorithmes efficaces, même si les projets technologiques accroissent les performances du matériel.**

Complexité	1	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
Evolution du temps quand la taille est multipliée par 10	t	$t+3,32$	$10 \times t$	$(10+\varepsilon) \times t$	$100 \times t$	$1000 \times t$	t^{10}
Evolution de la taille quand le temps est multiplié par 10	∞	n^{10}	$10 \times n$	$(10-\varepsilon) \times n$	$3,16 \times n$	$2,15 \times n$	$n+3,32$

FIGURE 2.7 – Temps d'exécution et taille des données [Gaudel et al., 1990]

2.2.1 Exemple : Multiplication de matrices carrées

Soit l'algorithme MATMULT de multiplication de deux matrices.

Algorithm 6 MATMULT(a, b, c)

Input: $A = a_{ij}$, $B = b_{ij}$, $C = c_{ij}$ trois matrices $n \times n$ à coefficients dans \mathbb{R}

Output: $C = A \times B$

```

1: for  $i \leftarrow 1$  to  $n$  do
2:   for  $j \leftarrow 1$  to  $n$  do
3:      $c[i, j] \leftarrow 0$ 
4:     for  $k \leftarrow 1$  to  $n$  do
5:        $c[i, j] \leftarrow c[i, j] + a[i, j] * b[k, j]$ 
6:     end for
7:   end for
8: end for

```

La complexité de l'algorithme MATMULT, comptée en nombre de multiplications de réels ne dépend que de la taille des matrices.

$$\text{Min}(n) = \text{Moy}(n) = \text{Max}(n) = \sum_{i=1..n} \sum_{j=1..n} \sum_{k=1..n} 1 = n^3.$$

2.2.2 Exemple : Recherche linéaire

On cherche à déterminer la complexité, en nombre de comparaisons, de l'algorithme RECHSEQ de recherche séquentielle d'un élément dans une liste.

Algorithm 7 RECHSEQ(L, j)

Input: L une liste à n éléments

Output: Recherche la place j d'un élément X dans une liste L . Si X n'est pas dans la liste, j est mis à zéro.

```

1:  $j \leftarrow 1$ 
2: while ( $j \leq n$ )  $\wedge (L[j] \neq X)$  do
3:    $j \leftarrow j + 1$ 
4: end while
5: if ( $j > n$ ) then
6:    $j \leftarrow 0$ 
7: end if
```

Dans le meilleur des cas, l'élément X se trouve dans la première case de la liste, d'où $Min(n) = 1$. Dans le pire des cas, X se trouverait dans la dernière case de la liste, d'où $Max(n) = n$.

Pour calculer $Moy(n)$, on doit se donner des probabilités sur L , et X ; on suppose tous les éléments distincts :

- soit q la probabilité que X soit dans L ;
- on suppose que X est dans L , toutes les places sont équiprobables.

On note $D_{n,i}$ pour $1 \leq i \leq n$, l'ensemble des données où X apparaît à la i ème place et $D_{n,0}$ l'ensemble des données où X est absent. D'après les conventions ci-dessus, on a :

$$p(D_{n,i}) = q/n$$

et

$$p(D_{n,0}) = 1 - q.$$

D'après l'analyse de l'algorithme, on a :

$$cout(D_{n,i}) = i$$

et

$$cout(D_{n,0}) = n.$$

On a donc :

$$Moy(n) = \sum_{i=0..n} p(D_{n,i}.cout(D_{n,i})) = (1 - q).n + \sum_{i=1..n} i.q/n$$

$$Moy(n) = (1 - q).n + (n + 1).q/2.$$

Si on sait que X est dans la liste, on a $q = 1$, on a :

$$Moy(n) = (n + 1)/2.$$

Si X a une chance sur deux d'être dans la liste, on a $q = 1/2$, et :

$$Moy(n) = n/2 + (n + 1)/4 = (3n + 1)/4.$$

2.3 Optimalité d'un algorithme [Gaudel et al., 1990]

Supposons que l'on dispose d'un algorithme A pour résoudre un problème donné, il est alors naturel de se demander si l'on peut trouver un algorithme "meilleur" que A ; il est donc intéressant de connaître la complexité du meilleur algorithme possible pour traiter un problème.

Soit un problème P , on considère la classe C de tous les algorithmes résolvant P , qui utilisent des opérations d'un certain type, et dont les données sont organisées d'une certaine manière (notons que l'on ne connaît pas nécessairement tous les algorithmes de la classe). On se donne également une mesure de complexité, le nombre d'opérations fondamentales (évalué soit dans le pire des cas, soit en moyenne).

On définit la complexité optimale de la classe C , comme la borne inférieure des complexités des algorithmes de la classe. Un algorithme A de C est dit optimal si sa complexité est égale à la complexité optimale de C , que l'on note $M_{opt}(n)$. Par conséquent, un algorithme A de classe C est optimal, s'il n'existe pas d'algorithme B dans C qui résolve le programme P en moins d'opérations que A .

Parfois, on ne peut pas déterminer $M_{opt}(n)$ avec précision, mais on peut connaître l'ordre de grandeur de la complexité optimale de la classe. **Dans ce cas, un algorithme A de la classe C est dit optimal si sa complexité est d'ordre de grandeur inférieur ou égal à la complexité de tout algorithme de classe C :**

$$\forall B \in C, M_A = O(M_B).$$

L'ordre de grandeur de $M_{opt}(n)$ est alors $\Theta(M_A)$. Il est clair que, dans ce cas, plusieurs algorithmes de même ordre de complexité peuvent être optimaux.

Les techniques de démonstration d'optimalité dépendent énormément du problème et de la classe d'algorithmes étudiés et il n'existe pas de méthode générale pour établir des résultats d'optimalité. Cependant, on en établit plusieurs, pour les problèmes suivants : recherche des deux plus grands éléments d'un tableau, recherche dichotomique, tri par comparaisons.

Il faut savoir que beaucoup de problèmes d'optimalité sont difficiles et qu'un grand nombre n'est pas encore résolu. Prenons l'exemple du produit de deux matrices, réalisé par des algorithmes utilisant les opérations arithmétiques classiques ($+, -, *, /$). On mesure la complexité en prenant la multiplication comme opération fondamentale. **La multiplication de matrices utilisent n^3 multiplications ; d'autre part, il est établi que la résolution du problème de la multiplication de deux matrices $n \times n$ nécessite au moins n^2 multiplications. Cependant, on ne connaît pas d'algorithme résolvant le problème avec seulement n^2 multiplications, et on sait pas s'il en existe.**

Déterminer la complexité du meilleur algorithme possible pour ce problème est actuellement un problème ouvert.

2.3.1 Exemple : recherche du plus grand élément d'une liste [Gaudel et al., 1990]

Soit l'algorithme RECHMAX.

Algorithm 8 RECHMAX(L, M)

Input: L une liste non vide à n éléments entiers

Output: Recherche le plus grand élément M de la liste non-vide L .

```

1:  $M \leftarrow L[1]$ 
2: for  $j \leftarrow 2$  to  $n$  do
3:   if  $L[j] > M$  then
4:      $M \leftarrow L[j]$ 
5:   end if
6: end for
```

On voit bien que le nombre de comparaisons est égal au nombre d'itérations dans la boucle FOR. Quelque soit la mesure d'évaluation considérée, la complexité est donc la même

$$\text{Min}_A(n) = \text{Moy}_A(n) = \text{Max}_A(n) = n - 1.$$

Maintenant, on peut se demander si ce nombre $n - 1$ peut être amélioré ? On va en fait démontrer que cet algorithme est optimal.

Considérons la classe C des algorithmes qui résolvent le problème de la recherche du maximum de n éléments en utilisant comme critère de décision, les comparaisons entre éléments. On montre, par une analyse directe du problème, que tout algorithme de C effectue au moins $n - 1$ comparaisons, quelle que soit la donnée sur laquelle il travaille. Prouvons cela.

Proposition 1. *Etant donné un algorithme E de C et un tableau quelconque, tout élément autre que le maximum est comparé au moins une fois avec un élément qui lui est plus grand.*

Preuve :

Soit i_0 l'indice du tableau L où se trouve le maximum M , résultat de l'algorithme E . Raisonnons par l'absurde en supposant qu'il existe $j_0 \neq i_0$ tel que $L[j_0]$ n'a pas été comparé avec le maximum $L[i_0]$. Construisons alors le tableau L' identique à L , sauf pour l'indice j_0 , où il vaut $M + 1$.

L'algorithme E effectuera les mêmes comparaisons sur L' que sur L , et par suite ne comparera pas $L'[j_0]$ et $L'[i_0]$. Il donnera donc comme maximum $L'[i_0]$ et sera incorrect. D'où la contradiction.

□

Il résulte de cette dernière proposition qu'il est impossible de déterminer l'élément maximum d'un tableau quelconque de n éléments en moins de $n - 1$ comparaisons. L'algorithme est donc optimal en nombre de comparaisons.

2.4 Relations de récurrence [Sedgewick and Flajolet, 1996]

Les algorithmes que l'on souhaite analyser sont en général formulés en termes de procédures récursives ou itératives, ce qui signifie que le coût de résolution d'un problème particulier s'exprime en fonction du coût de résolution de problèmes plus petits. L'approche mathématique la plus élémentaire consiste à former une relation de récurrence. Elle matérialise le lien direct entre la structure récursive d'un programme et la représentation récursive d'une fonction en décrivant les propriétés.

L'étude de méthodes plus complexes révèlera que la récurrence n'est pas forcément l'outil mathématique le plus adapté à l'analyse d'algorithmes. L'utilisation de méthodes symboliques permet de déduire des relations entre séries génératrices que l'on analyse ensuite directement. Ce principe, on le verra dans la section consacrée aux séries génératrices.

On verra dans le chapitre suivant consacrés aux algorithmes de tri, trois récurrences :

$$C_N = \left(1 + \frac{1}{N}C_{N-1} + 2\right) \quad \text{pour } N > 1 \text{ avec } C_1 = 2.$$

$$C_N = C_{\lfloor N/2 \rfloor} + C_{\lceil N/2 \rceil} + N \quad \text{pour } N > 1 \text{ et } C_1 = 0.$$

$$C_N = N + 1 + \frac{1}{N} \sum_{1 \leq j \leq N} (C_{j-1} + C_{N-j}) \quad \text{pour } N > 0 \text{ avec } C_0 = 0$$

Chaque équation présente un problème particulier. Soit aussi la fameuse récurrence linéaire de Fibonacci

$$F_n = F_{n-1} + F_{n-2} \quad \text{pour } n > 1 \text{ avec } F_0 = 0 \text{ et } F_1 = 1$$

Les nombres de Fibonacci interviennent explicitement dans la conception et l'analyse d'importants algorithmes. La résolution de ce type de récurrence fait appel à un certain nombre de techniques.

Les récurrences sont classées selon la combinaison des termes, la nature des coefficients ainsi que le nombre et la nature des termes précédents qui y figurent. Le tableau suivant présente quelques-unes des récurrences qui seront examinées, ainsi que des exemples représentatifs.

premier ordre	
linéaire	$a_n = na_{n-1} - 1$
non linéaire	$a_n = 1/(1 + a_{n-1})$
deuxième ordre	
linéaire	$a_n = a_{n-1} + 2a_{n-2}$
non linéaire	$a_n = a_{n-1}2a_{n-2} + \sqrt{a_n - 2}$
coefficients variables	$a_n = na_{n-1} + (n-1)a_{n-2} + 1$
ordre t	$a_n = f(a_{n-1}, a_{n-2}, \dots, a_{n-t})$
complète	$a_n = n + a_{n-1} + a_{n-2} \dots a_1$
diviser pour régner	$a_n = a_{\lfloor n/1 \rfloor} + a_{\lceil n/2 \rceil} + n$

TABLE 2.1 – Classification des récurrences

2.4.1 Récurrences du premier ordre [Sedgewick and Flajolet, 1996]

Les récurrences les plus simples sont sans doute celles qui se ramènent à un produit. La récurrence

$$a_n = x_n a_{n-1}$$

pour $n > 0$ avec $a_0 = 1$, est équivalente à

$$a_n = \prod_{1 \leq k \leq n} x_k.$$

Cette transformation est un exemple simple d'itération : on applique la récurrence à elle-même jusqu'à ce que l'on aboutisse aux constantes et aux valeurs initiales, puis on simplifie. La récurrence suivante, très fréquemment rencontrée, est également un exemple simple d'itération

$$a_n = a_{n-1} + y_n$$

pour $n > 0$ avec $a_0 = 0$, est équivalente à

$$\sum_{1 \leq k \leq n} y_k.$$

Le tableau de la Figure 2.8 montre quelques sommes discrètes fréquemment rencontrées. Notez que $C_n^k = \binom{n}{k} = \frac{A_n^k}{k!} = \frac{n!}{(n-k)!k!}$

<i>séries géométriques</i>	$\sum_{0 \leq k < n} x^k = \frac{1 - x^n}{1 - x}$
<i>séries arithmétiques</i>	$\sum_{0 \leq k < n} k = \frac{n(n-1)}{2} = \binom{n}{2}$
<i>coefficients binomiaux</i>	$\sum_{0 \leq k \leq n} \binom{k}{m} = \binom{n+1}{m+1}$
<i>théorème du binôme</i>	$\sum_{0 \leq k \leq n} \binom{n}{k} x^k y^{n-k} = (x+y)^n$
<i> nombres harmoniques</i>	$\sum_{1 \leq k \leq n} \frac{1}{k} = H_n$
<i>somme des harmoniques</i>	$\sum_{1 \leq k < n} H_k = nH_n - n$
<i>convolution de Vandermonde</i>	$\sum_{0 \leq k \leq n} \binom{n}{k} \binom{m}{t-k} = \binom{n+m}{t}$

FIGURE 2.8 – Sommes discrètes élémentaires [Sedgewick and Flajolet, 1996]

2.4.2 Récurrences d'ordre supérieur [Sedgewick and Flajolet, 1996]

On considère maintenant les récurrences telles que le membre droit de l'équation définissant a_n est une combinaison linéaire de $a_{n-1}, a_{n-2}, a_{n-3}, \dots, a_0$. Examinons par exemple la récurrence

$$a_n = 3a_{n-1} - 2a_{n-2}$$

pour $n > 1$, avec $a_0 = 0$, et $a_1 = 1$.

Il suffit d'observer que $a_n - a_{n-1} = 2(a_{n-1} - a_{n-2})$; en itérant ce produit, on obtient $a_n - a_{n-1} = 2^{n-1}$. On a ainsi $a_n = 2^n - 1$.

Théorème 1 (Récurrences linéaires à coefficients constants). *Les solutions de la récurrence*

$$a_n = x_1 a_{n-1} + x_2 a_{n-2} + \dots + x_t a_{n-t}$$

pour $n \geq t$, sont des combinaisons linéaires de la forme

$$a_n = \sum_{i=1..r} \left[\sum_{0 \leq j < v_i} c_{i,j} n^j \beta_i^n \right]$$

dont les coefficients dépendent des conditions initiales a_0, a_1, \dots, a_{t-1} de termes de la forme $n^j \beta_i^n$ où β_i est la racine i ème parmi les r racines du polynôme caractéristique

$$q(z) = z^t - x_1 z^{t-1} - x_2 z^{t-2} - \dots - x_t$$

et j est tel que $0 \leq j < v_i$ si β_i a pour multiplicité v_i .

Preuve Voir [Sedgewick and Flajolet, 1996]. \square

Considérons la récurrence

$$a_n = 5a_{n-1} - 6a_{n-2}$$

pour $n \geq 2$ avec $a_0 = 0$ et $a_1 = 1$. Son équation caractéristique est $z^2 - 5z + 6 = (z - 3)(z - 2)$, donc

$$a_n = c_0 3^n + c_1 2^n.$$

Cette formule engendre les équations suivantes pour $n = 0$ et $n = 1$

$$a_0 = 0 = c_0 + c_1$$

$$a_1 = 1 = 3c_0 + 2c_1$$

La solution à ce système linéaire est $c_0 = 1$ et $c_1 = -1$, donc $a_n = 3^n - 2^n$. Lorsque les coefficients sont nuls et/ou les solutions sont multiples, le résultat peut alors être très subtile. En fait, il faut prendre garde aux conditions initiales en étudiant de telles récurrences.

Exercice : étudier la suite de Fibonacci !

Dans le cas des coefficients non constants, on utilise des techniques plus sophistiquées. Le théorème 1 n'est plus applicable. Ici, on pourrait faire appel aux séries génératrices introduites dans la section 10.2.

Certaines récurrences peuvent être résolues grâce à la méthode des facteurs sommants. Considérons la récurrence :

$$a_n = na_{n-1} + n(n - 1)a_{n-2}$$

pour $n > 1$ avec $a_1 = 1$ et $a_0 = 0$. On la résout en divisant les deux membres par $n!$, retrouvant ainsi la suite de Fibonacci en $a_n/n!$, d'où $a_n = n!F_n$.

2.4.3 Récurrence diviser pour régner

On rappelle la récurrence du nombre de comparaisons effectuées par le tri fusion

$$C_N = C_{\lfloor N/2 \rfloor} + C_{\lfloor N/2 \rfloor} + N$$

pour $N > 1$ avec $C_1 = 0$. Ce genre de récurrence proviennent des algorithmes de type "Diviser pour régner". On les appelle aussi récurrence de partition. La Figure 2.9 donne des solutions à quelques récurrences de ce type.

$a_N = a_{N/2} + 1$	$\lg N + O(1)$
$a_N = a_{N/2} + N$	$2N + O(\lg N)$
$a_N = a_{N/2} + N \lg N$	$\Theta(N \lg N)$
$a_N = 2a_{N/2} + 1$	$\Theta(N)$
$a_N = 2a_{N/2} + \lg N$	$\Theta(N)$
$a_N = 2a_{N/2} + N$	$N \log N + O(N)$
$a_N = 2a_{N/2} + N \lg N$	$\frac{1}{2}N \lg N^2 + O(N \log N)$
$a_N = 2a_{N/2} + N \lg^{\delta-1} N$	$\delta^{-1} N \lg^\delta N + O(N \lg^{\delta-1} N)$
$a_N = 2a_{N/2} + N^2$	$2N^2 + O(N)$
$a_N = 3a_{N/2} + N$	$\Theta(N^{\lg 3})$
$a_N = 4a_{N/2} + N$	$\Theta(N^2)$

FIGURE 2.9 – [Sedgewick and Flajolet, 1996]

Pour trouver une solution générale, on considère la formule récursive

$$a(x) = \alpha a(x/\beta) + f(x)$$

pour $x > 1$ avec $a(x) = 0$ pour $x \leq 1$. Cela correspond à un algorithme "diviser pour régner" décomposant un problème de taille x en α sous-problèmes de taille x/β , et dont le coût de reconstitution est $f(x)$.

Théorème 2 (Théorème général). *Soient $a \geq 1$, et $b > 1$ deux constantes, soit $f(n)$ une fonction, et soit $T(n)$ définie pour les entiers positifs par la récurrence*

$$T(n) = aT(n/b) + f(n),$$

où l'on interprète n/b soit comme $\lfloor n/b \rfloor$, soit comme $\lceil n/b \rceil$. $T(n)$ peut alors être bornée asymptotiquement comme suit :

1. *Si $f(n) = O(n^{\log_b a - \epsilon})$ pour une certaine constante $\epsilon > 0$, alors $T(n) = \Theta(n^{\log_b a})$.*
2. *Si $f(n) = \Theta(n^{\log_b a})$, alors $T(n) = \Theta(n^{\log_b a} \log n)$.*
3. *Si $f(n) = \Omega(n^{\log_b a + \epsilon})$ pour une constante $\epsilon > 0$, et si $af(n/b) \leq cf(n)$ pour une constante $c < 1$ et tous les n suffisamment grands, alors $T(n) = \Theta(f(n))$.*

Illustrations : prendre des cas de la Figure 2.9.

2.4.4 Autres méthodes de résolution

En plus des méthodes que vient de décrire, une panoplie d'autres méthodes mathématiques existent dans la littérature pour résonner sur les récurrences, on peut en citer : résolution par séries génératrices (voir la section 10.2), résolution par répertoire, résolution par Rebouclage (bootstrapping), résolution par perturbation, etc. Elles sont pour la plupart décrites dans la référence [Sedgewick and Flajolet, 1996].

2.5 Algorithmes de tri

2.5.1 Tri par tas [Cormen et al., 1994, Cormen et al., 2001]

Le tri par tas introduit une technique originale de conception des algorithmes : on utilise une structure de données, ici appelée "tas", pour gérer les informations pendant l'exécution de l'algorithme.

2.5.1.1 Les tas

La structure de tas (binaire) est un object tabulé qui peut être vu comme un arbre binaire presque complet (voir la Figure 2.10).

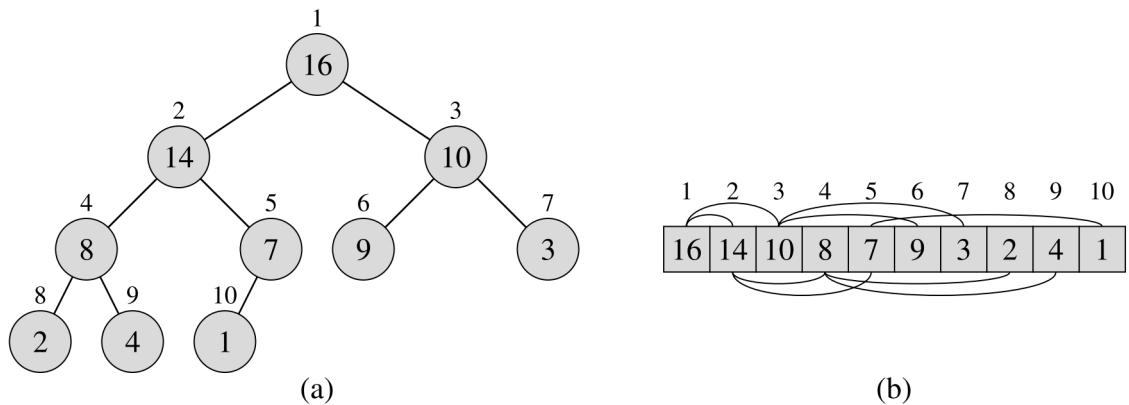


FIGURE 2.10 – [Cormen et al., 1994, Cormen et al., 2001]

Les tas nécessitent trois opérations de base :

$\text{PERE}(i)$
retourner $\lfloor i/2 \rfloor$

$\text{GAUCHE}(i)$
retourner $2i$

$\text{DROIT}(i)$
retourner $2i + 1$

Etant donné un tableau A , nous avons la propriété basique des tas :

$$A[\text{Pere}[i]] \geq A[i] \quad (2.1)$$

La valeur d'un noeud est au plus égale à la valeur de son père.

La hauteur d'un noeud dans un arbre est défini comme le nombre d'arcs sur le chemin le plus long allant du noeud à une feuille. La hauteur d'un tas est la hauteur de sa racine.

Proposition 2. La hauteur h d'un tas de n éléments est en $O(\log n)$

Preuve :

Le nombre de noeuds au niveau zéro d'un tas est 2^0 , puis 2^1 au niveau 1, ..., 2^{h-1} au niveau $h - 1$. Un tas de n noeuds de hauteur h , a tous ses niveaux pleins, sauf le dernier niveau h qui contient au moins un élément, et au plus 2^h éléments. Nous avons donc les inégalités :

$$\begin{aligned} n &\geq 2^0 + 2^1 + \dots + 2^{h-1} + 1 \\ &\geq \sum_{0 \leq k < h} 2^k + 1 = \frac{1-2^h}{1-2} + 1 = 2^h - 1 + 1 = 2^h \\ n &\leq 2^0 + 2^1 + \dots + 2^h \\ &\leq \sum_{0 \leq k \leq h} 2^k = \frac{1-2^{h+1}}{1-2} = 2^{h+1} - 1 \end{aligned}$$

En passant au log :

$$\begin{aligned} n &\geq 2^h \\ h &\leq \log_2(n) \end{aligned}$$

La complexité de ENTASSER dans le pire des cas est au plus le parcours de la hauteur de l'arbre, et est donc en $O(\log(n))$. \square

Algorithm 9 ENTASSER(A , i)

Input: Un tableau A et un indice i dans le tableau

Output: Faire descendre la valeur de $A[i]$ dans le tas en respectant la propriété (2.1).

```

1:  $l \leftarrow \text{GAUCHE}(i)$ 
2:  $r \leftarrow \text{DROIT}(i)$ 
3: if ( $l \leq \text{taille}[A]$ )  $\wedge (A[l] > A[i])$  then
4:    $max \leftarrow l$ 
5: else
6:    $max \leftarrow i$ 
7: end if
8: if ( $r \leq \text{taille}[A]$ )  $\wedge (A[r] > A[max])$  then
9:    $max \leftarrow r$ 
10: end if
11: if ( $max \neq i$ ) then
12:   Echanger  $A[i] \leftrightarrow A[max]$ 
13:   ENTASSER( $A, max$ )
14: end if

```

A chaque étape, on détermine le plus grand des éléments $A[i]$, $A[\text{Gauche}(i)]$, et $A[\text{Droit}(i)]$, et son indice est rangé dans max . Si $A[i]$ est le plus grand, alors le sous-arbre enraciné au noeud i est un tas. Sinon, l'un des deux fils contient l'élément le plus grand, et $A[i]$ est échangé avec $A[max]$, ce qui permet au noeud i et à ses fils de satisfaire la propriété des tas. Toutefois le noeud max contient la valeur initial de $A[i]$, et donc le sous-arbre enraciné en max viole peut-être la propriété de tas. ENTASSER doit être appelée récursivement sur ce sous-arbre.

Le temps d'exécution de ENTASSER est dominé par la hauteur du tas, et donc en $O(\log(n))$. La Figure 2.11 illustre le procédé.

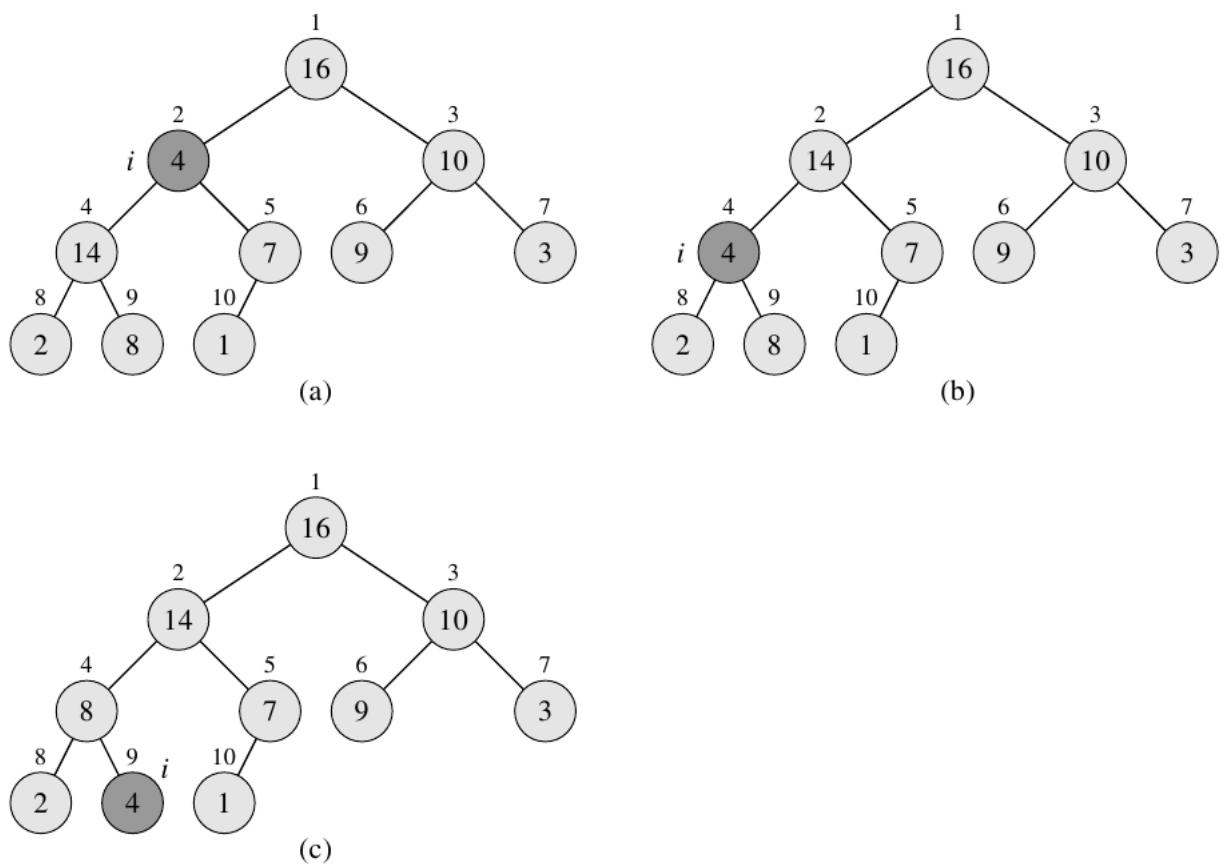


FIGURE 2.11 – Illustration de ENTASSER (scan [Cormen et al., 1994])

Algorithm 10 CONSTRUIRETAS(A)**Input:** Un tableau A **Output:** Construire le tas de A .

```

1: for  $i \leftarrow \lfloor \text{longueur}[A]/2 \rfloor$  à 1 do
2:   ENTASSER( $A, i$ )
3: end for

```

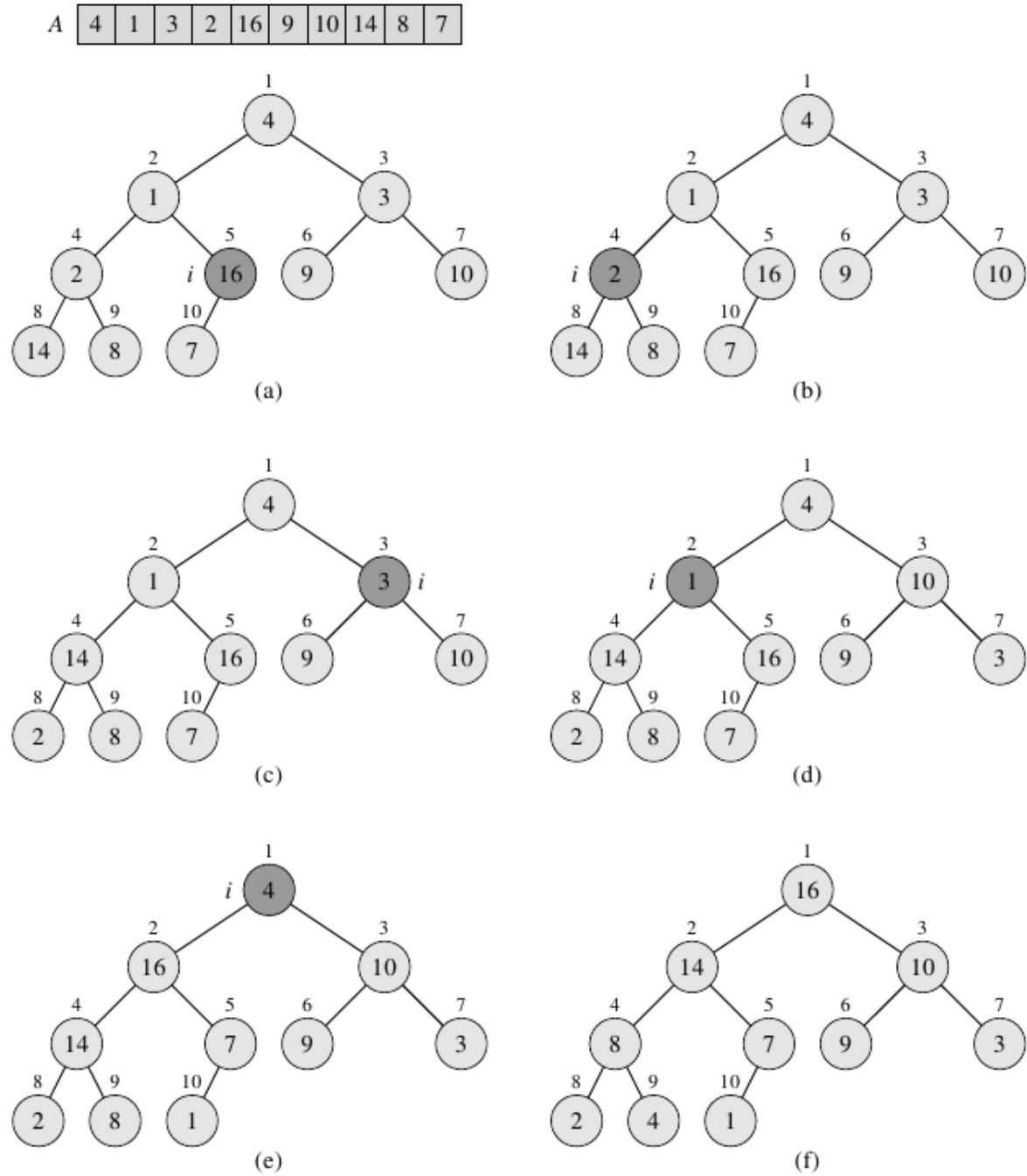


FIGURE 2.12 – Illustration de CONSTRUIRETAS (scan [Cormen et al., 1994])

La Figure 2.12 illustre le procédé de CONSTRUIRETAS. Comme la procédure ENTASSER est en $O(\log n)$, et il existe $O(n)$ appels, alors **le temps d'exécution de CONSTRUIRETAS est au plus $O(n \log n)$** . Cette borne supérieure, quoique correcte, n'est pas approchée asymptotiquement. Plus justement, CONSTRUIRETAS est en $\Theta(n)$ comme démontré dans la proposition suivante.

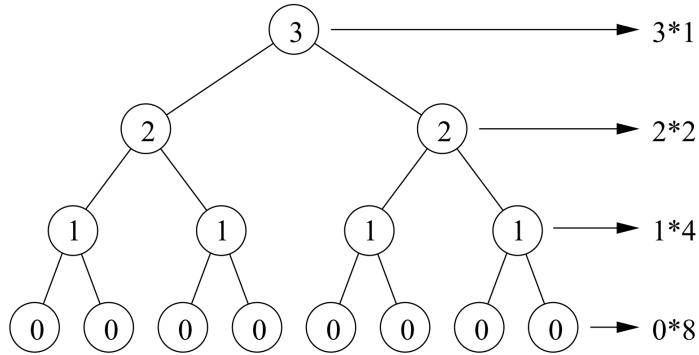


FIGURE 2.13 – Nombre de parcours par hauteur CONSTRUIRETAS

Proposition 3. Soit $T(n)$ la complexité de CONSTRUIRETAS. $T(n) = \Theta(n)$.

Preuve :

La figure 2.13 illustre le coût des entassements par hauteurs dans le pire des cas.

$$\begin{aligned} T(n) &= \sum_{j=0..h} j2^{h-j} = \sum_{j=0..h} j \frac{2^h}{2^j} \\ T(n) &= 2^h \sum_{j=0..h} \frac{j}{2^j} \end{aligned}$$

Nous exploitons une somme issue des développements limités :

$$\sum_{j=0..\infty} x^j = \frac{1}{1-x}.$$

En dérivant par rapport à x :

$$\sum_{j=0..\infty} jx^{j-1} = \frac{1}{(1-x)^2}.$$

$$\sum_{j=0..\infty} jx^j = \frac{x}{(1-x)^2}.$$

D'où

$$\sum_{j=0..h} \frac{j}{2^j} = \frac{1/2}{(1 - 1/2)^2} = \frac{1/2}{1/4} = 2.$$

En somme :

$$T(n) = 2^h \sum_{j=0..h} \frac{j}{2^j} \leq 2^h \sum_{j=0..\infty} \frac{j}{2^j} \leq 2^h 2 = 2^{h+1}.$$

Rappelons que $n = 2^{h+1} - 1$, ainsi $T(n) \leq n + 1$, d'où $T(n) = O(n)$. Il est évident que $T(n) = \Omega(n)$ dès lors qu'il parcourt le tableau $n/2$ fois. D'où $T(n) = \Theta(n)$.

□

L'algorithme 16 du tri par tas commence par se servir de CONSTRUIRETAS pour construire un tas sur le tableau d'entrée $A[1..n]$, où $n = \text{longueur}[A]$. Comme l'élément maximum du tableau est stocké à la racine $A[1]$, on peut le placer dans sa position finale correcte en l'échangeant avec $A[n]$. Si on sort le noeud n du tas (en décrémentant $\text{taille}[A]$), on observe que $A[1..(n-1)]$ peut facilement être transformé en tas, en relançant l'entassement de $A[1]$. La procédure continue ainsi jusqu'à ce que le tas se ramène à une seule case.

Une illustration est donnée dans la Figure 2.14.

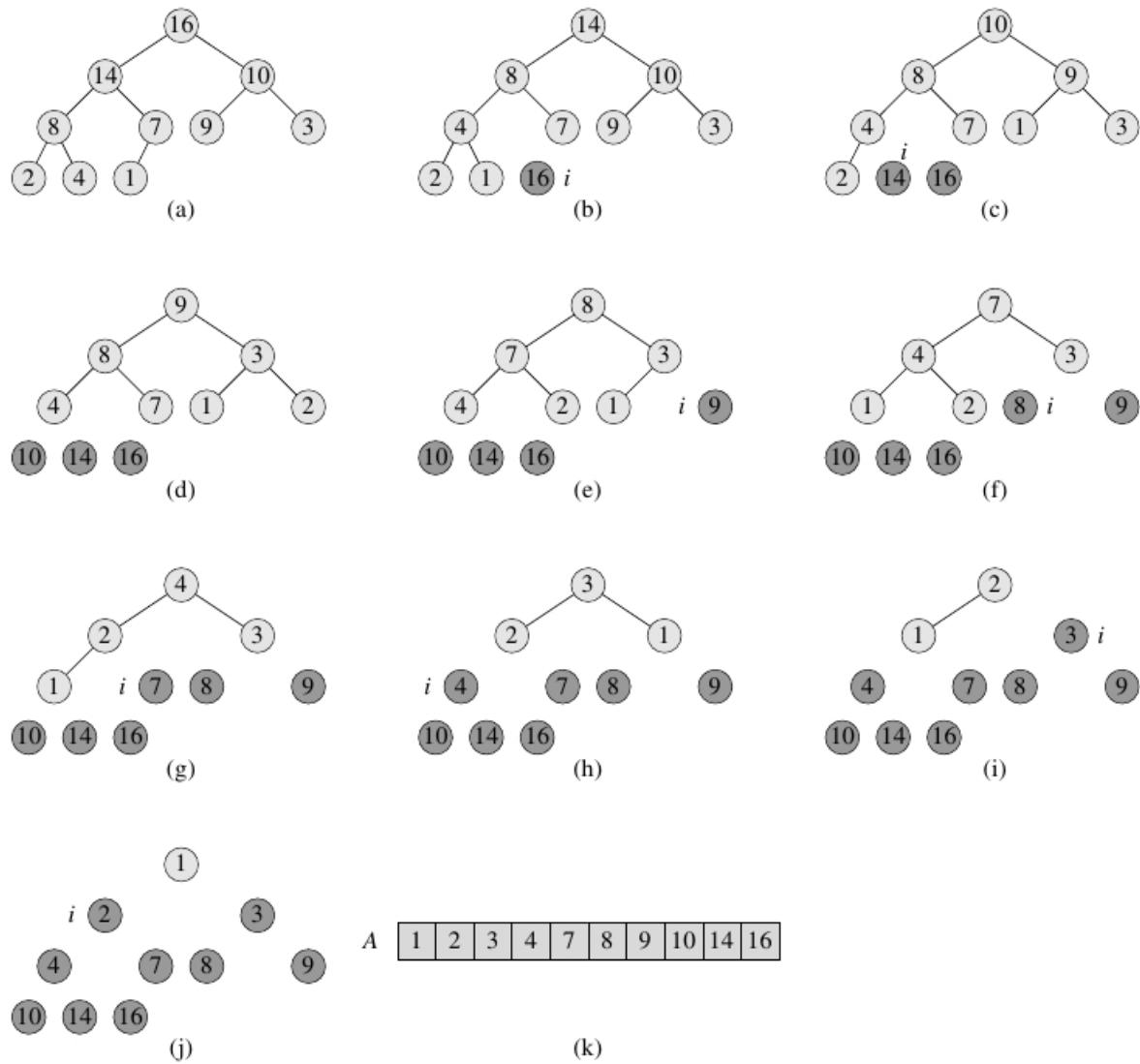


FIGURE 2.14 – Illustration de TRIESTAS (scan [Cormen et al., 1994])

Algorithm 11 TRI TAS(A)

Input: Un tableau A **Output:** Trier le tableau A .

```
1: CONSTRUIRETAS( $A$ )
2: for  $i \leftarrow longueur[A]$  à 2 do
3:   Echanger  $A[1] \leftrightarrow A[i]$ 
4:    $taille[A] \leftarrow taille[A] - 1$ 
5:   ENTASSER( $A, 1$ )
6: end for
```

Il est évident que le temps d'exécution du tri par tas est $O(n \log n)$.

2.5.2 Application des tas : Files de priorité

Nous présentons l'une des applications les plus célèbres des tas : son utilisation comme file de priorité efficace. Une file de priorité est une structure de données permettant de gérer un ensemble S d'éléments, chacun ayant une valeur associée appelée $clé$. Une file de priorité supporte les opérations suivantes : INSÉRER, MAXIMUM, EXTRAIREMAX. Parmi les applications des files de priorité, nous citons la planification de tâches dans un système d'exploitation.

L'opération MAXIMUM est en $O(1)$. L'opération INSÉRER est en $O(\log n)$ provenant de la hauteur de l'arbre du tas. L'opération EXTRAIRE-MAX est en $O(\log n)$ dû à l'entassement.

Algorithm 12 INSÉRER(S , $clé$)

Input: ensemble S

Output: Insérer l'élément x dans l'ensemble S .

```

1:  $taille[A] \leftarrow taille[A] + 1$ 
2:  $i \leftarrow taille[A]$ 
3: while ( $i > 1$ )  $\wedge (A[Pere(i)] < clé)$  do
4:    $A[i] \leftarrow A[Pere(i)]$ 
5:    $i \leftarrow Pere(i)$ 
6: end while
7:  $A[i] \leftarrow clé$ 
```

Algorithm 13 MAXIMUM(S)

Input: ensemble S

Output: Retourne l'élément de S ayant la plus grande clé.

```
1: return  $A[1]$ 
```

Algorithm 14 EXTRAIREMAX(S)

Input: ensemble S

Output: supprime et retourne l'élément S ayant la plus grande clé.

```

1: if  $taille[A] < 1$  then
2:   erreur "débordement négatif"
3: end if
4:  $max \leftarrow A[1]$ 
5:  $A[1] \leftarrow A[taille[A]]$ 
6:  $taille[A] \leftarrow taille[A] - 1$ 
7: ENTASSER( $A$ , 1)
8: return  $max$ 
```

2.5.3 Tri rapide

Le tri rapide est un algorithme de tri dont le temps d'exécution dans le pire des cas est en $\Theta(n^2)$ sur un tableau de dimension n . Cependant, le tri rapide est souvent meilleur en pratique à cause de son efficacité remarquable en moyenne : son temps d'exécution attendu est $\Theta(n \log n)$, et les facteurs constants cachés dans la notion sont très réduits. Il a aussi l'avantage de trier sur place. Le tri rapide est fondé sur le paradigme "diviser pour régner". Voici ses trois étapes :

Diviser Le tableau $A[p..r]$ est partitionné en deux sous-tableaux non vides $A[p..q]$ et $A[q + 1..r]$ tels que chaque élément de $A[p..q]$ soit inférieur ou égal à chaque élément de $A[q + 1..r]$.

Régner Les deux sous-tableaux $A[p..q]$ et $A[q + 1..r]$ sont triés par des appels récursifs à la procédure principale du tri rapide.

Combiner Comme les sous-tableaux sont triés sur place, aucun travail n'est nécessaire pour les recombiner : le tableau final est maintenant trié.

Algorithm 15 TRIRAPIDE(A, p, r)

Input: Tableau A ; deux indices p et r du sous-tableau ;

Output: Le tableau A est trié.

```

1: if  $p < r$  then
2:    $q \leftarrow \text{PARTITIONNER}(A, p, r)$ 
3:   TRIRAPIDE( $A, p, q - 1$ )
4:   TRIRAPIDE( $A, q + 1, r$ )
5: end if

```

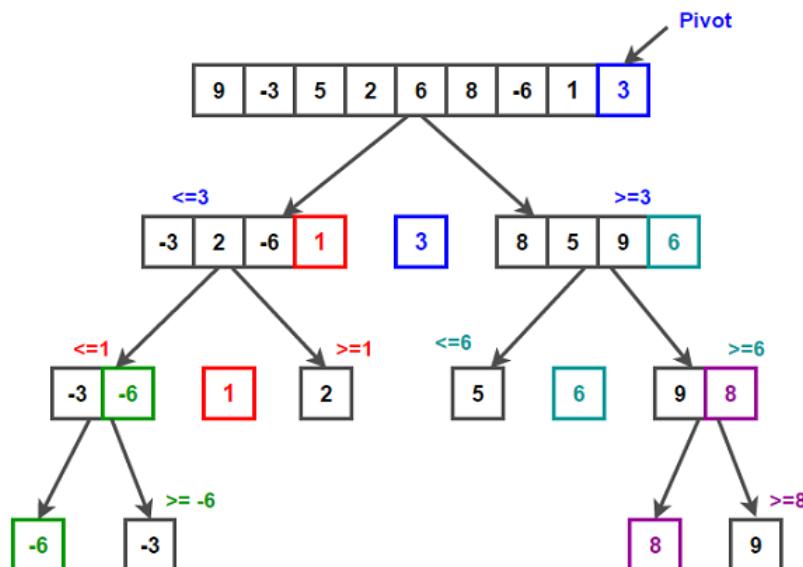


FIGURE 2.15 – Illustration de TRISRAPIDE (scan <https://afteracademy.com/blog/quick-sort>)

Algorithm 16 PARTITIONNER(A, p, r)

Input: Tableau A ; deux indices p et r du sous-tableau ;

Output: Retourner un indice q de partitionnement entre p et r , tel que $A[p..q]$ soit \leq au pivot $A[r]$, et $A[q+1..r] \geq$ au pivot $A[r]$.

```

1:  $x \leftarrow A[r]$ 
2:  $i \leftarrow p - 1$ 
3: for  $j \leftarrow p..r - 1$  do
4:   if ( $A[j] \leq x$ ) then
5:      $i \leftarrow i + 1$ 
6:     permuter  $A[i] \leftrightarrow A[j]$ 
7:   end if
8: end for
9: permuter  $A[i + 1] \leftrightarrow A[r]$ 
10: return  $i + 1$ 
```

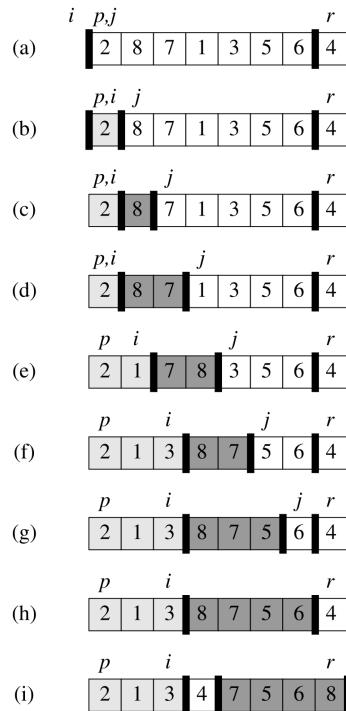


Figure 7.1 Fonctionnement de PARTITION sur un exemple. Les éléments en gris clair sont tous dans la première partition, avec des valeurs pas plus grandes que x . les éléments en gris foncé sont dans la seconde partition, avec des valeurs supérieures à x . Les éléments non en gris n'ont pas encore été placés dans l'une des deux premières partitions et l'élément blanc final est le pivot. (a) Le tableau initial et les configurations de variable initiales. Aucun des éléments n'a été placé dans l'une quelconque des deux premières partitions. (b) La valeur 2 est « permutee avec elle-même » et placée dans la partition des petites valeurs. (c)-(d) Les valeurs 8 et 7 sont ajoutées à la partition des grandes valeurs. (e) Les valeurs 1 et 8 sont permutees et la petite partition grossit. (f) Les valeurs 3 et 7 sont échangées et la petite partition grossit. (g)-(h) La grande partition grossit pour accueillir 5 et 6 et la boucle se termine. (i) Sur les lignes 7–8, l'élément pivot est permute de façon à aller entre les deux partitions.

FIGURE 2.16 – Illustration de TRISRAPIDE (scan [Cormen et al., 1994])

Initialisation : Avant la première itération, $i = p - 1$ et $j = p$. Il n'y a pas de valeurs entre p et i , ni de valeurs entre $i + 1$ et $j - 1$, de sorte que les deux premières conditions de l'invariant de boucle sont satisfaites de manière triviale. L'affectation en ligne 1 satisfait à la troisième condition.

Conservation : Comme le montre la figure 7.3, il y a deux cas à considérer, selon le résultat du test en ligne 4. La figure 7.3(a) montre ce qui se passe quand $A[j] > x$; l'unique action faite dans la boucle est d'incrémenter j . Une fois j incrémenté, la condition 2 est vraie pour $A[j - 1]$ et toutes les autres éléments restent inchangés. La figure 7.3(b) montre ce qui se passe quand $A[j] \leq x$; i est incrémenté, $A[i]$ et $A[j]$ sont échangés, puis j est incrémenté. Compte tenu de la permutation, on a maintenant $A[i] \leq x$ et la condition 1 est respectée. De même, on a aussi $A[j - 1] > x$, car l'élément qui a été permué avec $A[j - 1]$ est, d'après l'invariant de boucle, plus grand que x .

Terminaison : À la fin, $j = r$. Par conséquent, chaque élément du tableau est dans l'un des trois ensembles décrits par l'invariant et l'on a partitionné les valeurs du tableau en trois ensembles : les valeurs inférieures ou égales à x , les valeurs supérieures à x et un singleton contenant x .

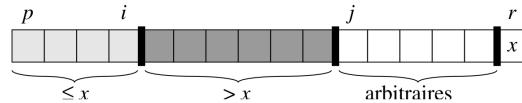


Figure 7.2 Les quatre régions gérées par la procédure PARTITION sur un sous-tableau $A[p..r]$. Les valeurs de $A[p..i]$ sont toutes inférieures ou égales à x , les valeurs de $A[i+1..j-1]$ sont toutes supérieures à x et $A[r] = x$. Les éléments de $A[j..r-1]$ peuvent prendre n'importe quelles valeurs.

FIGURE 2.17 – Preuve de correction de TRISRAPIDE (scan [Cormen et al., 1994])

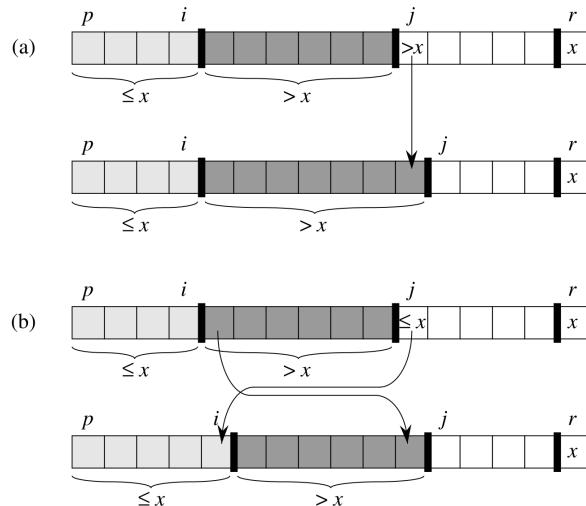


Figure 7.3 Les deux cas pour une itération de la procédure PARTITION. (a) Si $A[j] > x$, l'unique action est d'incrémenter j , ce qui conserve l'invariant de boucle. (b) Si $A[j] \leq x$, l'indice i est incrémenté, $A[i]$ et $A[j]$ sont échangés, puis j est incrémenté. Ici aussi, l'invariant de boucle est conservé.

FIGURE 2.18 – Preuve de correction de TRISRAPIDE (scan [Cormen et al., 1994])

Le temps d'exécution de PARTITIONNER sur un tableau $A[p..r]$ est $\Theta(n)$, avec $n = r - p + 1$.

Dans le pire des cas, le partitionnement du tri rapide sur n se limite à un seul appel récursif sur $n - 1$. On obtient ainsi la récurrence

$$T(n) = T(n - 1) + \Theta(n).$$

Pour résoudre cette récurrence, on procède comme suit :

$$\begin{aligned} T(n) &= T(n - 1) + \Theta(n) \\ &= \sum_{k=1..n} \Theta(k) \\ &= \Theta(\sum_{k=1..n} k) \\ &= \Theta(n^2) \end{aligned}$$

Si le partitionnement est équilibré (en prenant toujours le milieu du tableau), le tri rapide aura la complexité du tri par fusion. **D'où le cas de la complexité dans le meilleur des cas qui serait donc $\Theta(n \log n)$.**

Proposition 4. Le tri rapide a une complexité en moyenne $O(n \log n)$.

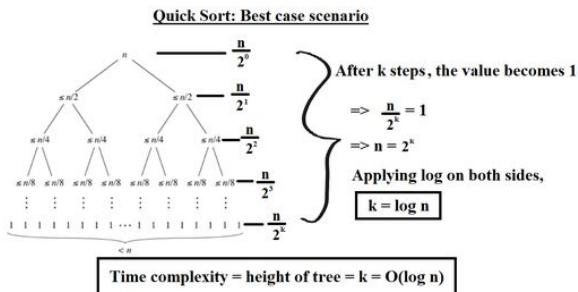


FIGURE 2.19 – Complexité TRISRAPIDE <https://www.interviewbit.com/tutorial/quicksort-algorithm/>

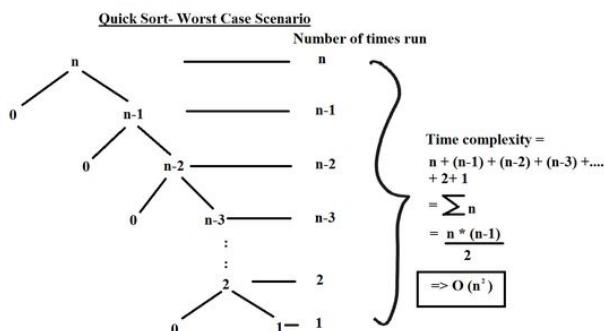


FIGURE 2.20 – Complexité TRISRAPIDE <https://www.interviewbit.com/tutorial/quicksort-algorithm/>

Preuve :

Nous supposons que les n partitionnements possibles sont équiprobables. Nous avons donc :

$$\begin{aligned} T(n) &= \frac{1}{n-1} [\sum_{q=1..n-1} (T(q) + T(n-q))] + \theta(n) \\ T(n) &= \frac{2}{n-1} \sum_{q=1..n-1} T(q) + \theta(n) \end{aligned}$$

Nous faisons appel à la méthode de substitution pour résoudre cette récurrence (voir la sous-section suivante).

On suppose que $T(n) \leq an \log(n) + b$.

D'où :

$$\begin{aligned} T(n) &= \frac{2}{n-1} \sum_{q=1..n-1} T(q) + \theta(n) \\ T(n) &\leq \frac{2}{n-1} \sum_{q=1..n-1} (ak \log(k) + b) + \theta(n) \\ T(n) &\leq \frac{2a}{n-1} \sum_{q=1..n-1} k \log(k) + \frac{2b}{n-1}(n-1) + \theta(n) \\ &\quad \text{On peut démontrer que } \sum_{q=1..n-1} k \log(k) \leq 1/2n^2 \log(n) - 1/8n^2 \\ T(n) &\leq \frac{2a}{n-1} [1/2n^2 \log(n) - 1/8n^2] + \frac{2b}{n-1}(n-1) + \theta(n) \\ &\leq an \log(n) + b \end{aligned}$$

□

2.5.3.1 Introduction à la méthode de substitution [Cormen et al., 1994]

La méthode de substitution recouvre deux phases :

- Conjecturer la forme de la solution.
- Employer une récurrence mathématique pour trouver les constantes et prouver que la solution est correcte.

Le nom vient du fait que l'on substitue la réponse supposée à la fonction quand on applique l'hypothèse de récurrence aux valeurs plus petites. Cette méthode ne peut s'utiliser que lorsque la forme de la réponse est facile à deviner. Par exemple, calculons une borne supérieure pour la récurrence

$$T(n) = 2T(n/2) + n.$$

On conjecture que la solution est $T(n) = O(n \log(n))$. La méthode consiste à démontrer que $T(n) \leq cn \log(n)$ pour un choix approprié de la constante $c < 0$. On commence par supposer que cette borne est valable pour $n/2$, autrement dit que $T(n/2) \leq cn/2 \log(n/2)$. La substitution donne :

$$\begin{aligned} T(n) &\leq 2(cn/2 \log(n/2)) + n \\ &\leq cn \log(n/2) + n \\ &= cn \log(n) - cn \log(2) + n \\ &= cn \log(n) - cn + n \\ &\leq cn \log(n), \end{aligned}$$

la dernière étape étant vraie si $c \geq 1$.

Cette méthode peut aussi se justifier en faisant appel au deuxième principe d'induction (voir [Arnold and Guessarian, 1997]) :

Proposition 5 (deuxième principe d'induction). *Soit $P(n)$ un prédictat (une propriété) dépendant de l'entier n . Si la proposition suivante est vérifiée :*

$$(I') \forall n \in \mathbb{N}, (\forall k < n, P(k) \Rightarrow P(n))$$

alors $\forall n \in \mathbb{N}, P(n)$ est vraie.

2.6 Travaux dirigés I

2.6.1 Algorithme de tri

Soit un algorithme TRISEL pour trier n nombres stockés dans un tableau A . TRISEL commence par trouver le plus petit élément de A et l'échange avec la première case de A . Puis, l'algorithme trouve le second plus petit élément et l'échange avec la deuxième case. Et ainsi de suite avec les autres éléments.

1. Ecrire l'algorithme en pseudo-code.
2. Exprimer l'invariant de la boucle de l'algorithme.
3. Trouver le coût dans le meilleur des cas, et le pire des cas avec la notation Θ .

2.6.2 Algorithme de recherche dichotomique

Algorithm 17 RECHDICO(A, x)

Input: Tableau $A[1..n]$; x élément à chercher dans A ;

Output: Retourner l'indice si l'élément existe, sinon -1 (n'existe pas).

```

1:  $L \leftarrow 0$ 
2:  $R \leftarrow n - 1$ 
3: while  $L \leq R$  do
4:    $m \leftarrow \lfloor ((L + R)/2) \rfloor$ 
5:   if  $A[m] < x$  then
6:      $L \leftarrow m + 1$ 
7:   else if  $A[m] > x$  then
8:      $R \leftarrow m - 1$ 
9:   else
10:    return  $m$ 
11:   end if
12: end while
13: return -1

```

Démontrer que l'algorithme 17 est de complexité $O(\log(n))$.

2.6.3 *** Algorithme de recherche

Soit le problème de la recherche de deux éléments entiers dans un tableau à n éléments entiers, tels que la somme de ces deux éléments soit égale à un nombre entier donné x . Décrire un algorithme en $\Theta(n\log(n))$ résolvant ce problème (N.B./Exploiter le tri par fusion).

2.7 Travaux dirigés II

2.7.1 Récurrence

Soit n une puissance exacte de 2, démontrer par induction que $T(n) = n \log(n)$ est une solution de l'équation récurrente (en illustrant un algorithme qui a ce comportement)

$$T(n) = \begin{cases} 2 & \text{si } n = 2, \\ 2T(n/2) + n & \text{si } n = 2^k, \text{ pour } k > 1. \end{cases}$$

2.7.2 Grandeurs des fonctions et notations asymptotiques

Exprimer $n^3/1000 - 100n^2 - 100n + 3$ avec la notation Θ .

Les deux identités $2^{n+1} = O(2^n)$ et $2^{2n} = O(2^n)$, sont elles correctes ?

Démontrer que pour tout a, b réels, où $b > 0$ que $(n + a)^b = \theta(n^b)$.

2.7.3 Récurrences d'ordre supérieur et diviser pour régner

1. Résoudre la récurrence $a_n = 2a_{n-1} - a_{n-2}$ pour $n \geq 2$ avec $a_0 = 1$ et $a_1 = 2$.
2. Démontrer la complexité de l'algorithme récursif de Fibonacci en utilisant le théorème "Récurrences linéaires à coefficients constants".
3. Démontrer la complexité de l'algorithme de la recherche dichotomique en utilisant le théorème approprié.
4. Démontrer la complexité de l'algorithme du tri par fusion en utilisant le théorème approprié.

2.7.4 Tris

1. Montrer qu'un tas de n éléments a une hauteur $\log n$.
2. Montrer si la séquence $<23, 17, 14, 6, 13, 10, 1, 5, 7, 12>$ est un tas ou non.
3. Illustrer le déroulement de l'algorithme de TRITAS sur le tableau $<5, 13, 2, 25, 7, 17, 20, 8, 4>$.
4. Démontrer la correction de l'algorithme TRITAS avec l'invariant suivant : *Au début de chaque itération de la boucle for, le sous-tableau $A[1..i]$ est un tas, contenant les i plus petits éléments du tableau $A[1..n]$. Le sous-tableau $A[i + 1..n]$ contient les $n - i$ plus grands éléments de $A[1..n]$ triés.*
5. Quelle est la complexité du TRITAS sur un tableau croissant, et sur un tableau décroissant ?
6. Démontrer que la complexité du TRITAS dans le pire des cas est $\Omega(n \log n)$ (mini-projet ...).
7. Illustrer le déroulement de l'algorithme de tri rapide sur le tableau $<5, 13, 2, 25, 7, 17, 20, 8, 4>$.
8. Quelle est la complexité du tri rapide si tous ses éléments sont égaux ?
9. Détailler la preuve de la complexité du tri rapide en moyenne (mini-projet ...).

2.8 Mini projets

2.8.1 Recherche des deux points les plus rapprochés

Références : [Cormen et al., 1994, Cormen et al., 2001], <http://perso.eleves.ens-rennes.fr/~mruffini/Files/Other/ppr.pdf>, <https://www.geeksforgeeks.org/closest-pair-of-points-using-divide-and-conquer-algorithm>

Entrée : Un tableau Q de $n \geq 2$ points du plan

Sortie : Les deux points de Q les plus rapprochés (pour la distance euclidienne)

Un algorithme naïf consisterait à tester toutes les paires de points. Il y en a $\binom{n}{2} = \Theta(n^2)$.

Étape 1

On va utiliser le paradigme diviser pour régner.

Soit P un sous-ensemble de Q . Soit X un tableau contenant les points de P triés par ordre croissant des abscisses ; Soit Y un tableau contenant les points de P triés par ordre croissant des ordonnées. Si $|P| \leq 3$, on utilise l'algorithme naïf. Comme ça, on appelle jamais l'algorithme sur un ensemble de cardinal ≤ 1 . Sinon :

Diviser : On trouve une droite Δ qui sépare les points de P en P_G et P_D qui sont les points à droite et à gauche de Δ , avec $|P_G| = \lfloor \frac{|P|}{2} \rfloor$ et $|P_D| = \lfloor \frac{|P|}{2} \rfloor + 1$.

Régner : On appelle l'algorithme récursivement sur P_G et P_D . Soient δ_G et δ_D les plus petites distances pour P_G et P_D . Soit $\delta = \min(\delta_G, \delta_D)$

Combiner : La paire de points recherchée est soit celle espacée de δ , soit une telle qu'un point soit dans P_G et l'autre dans P_D . On cherche s'il existe une telle paire espacée de moins de δ . Si ces deux points existent, ils doivent être espacés de moins de δ de la droite Δ .

Étape 2

Implémentation de l'algorithme

Algorithme 1 : PPR(P)

$X \leftarrow$ trier Q par abscisses croissantes;
 $Y \leftarrow$ trier Q par ordonnées croissantes;
PPR-rec(X, Y)

Algorithme 2 : PPR-rec(X, Y)

```

si  $X \leq 3$  alors
|   retourner PPR-naif( $X$ )
|    $(X_G, X_D, Y_G, Y_D) \leftarrow$  PPR-diviser( $X, Y$ );
|    $\delta_G \leftarrow$  PPR-rec( $X_G, Y_G$ );
|    $\delta_D \leftarrow$  PPR-rec( $X_D, Y_D$ );
|    $\delta \leftarrow \min(\delta_G, \delta_D)$ ;
|    $Y' \leftarrow$  extraire de  $Y$  les points espacés de moins de  $\delta$  de  $\Delta$ ;
|   /* Comme dans PPR-diviser
|    $\delta' \leftarrow$  PPR-bande( $Y'$ );
|   retourner  $\min(\delta, \delta')$ 

```

FIGURE 2.21 – Closest Pair of Points <http://perso.eleves.ens-rennes.fr/~mruffini/Files/Other/ppr.pdf>

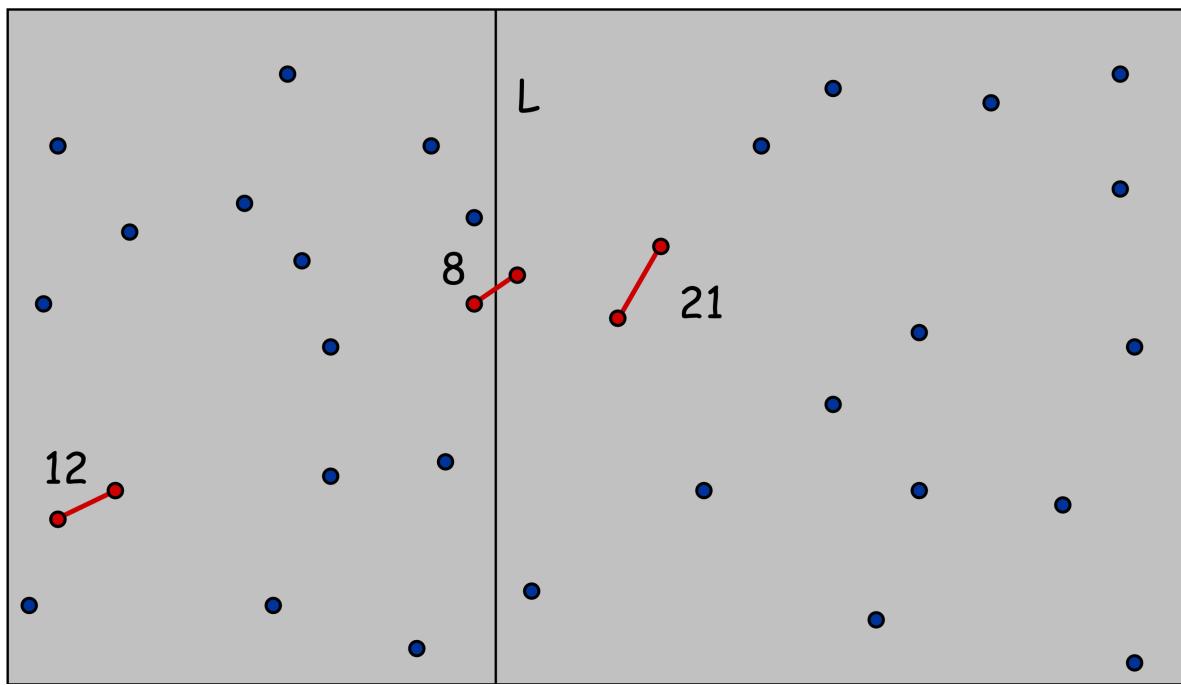


FIGURE 2.22 – Closest Pair of Points https://ocw.tudelft.nl/wp-content/uploads/Algoritmiek_Closest_Pair_of_Points.pdf

Étape 3

Description de PPR-diviser

On veut divise X et Y en X_G, X_D, Y_G, Y_D qui sont les points de P_G et P_D triés par ordre croissant des abscisses ou des ordonnées.

Algorithm 3 : PPR-diviser(X, Y)

```

 $X_G \leftarrow X[1, \lfloor |X|/2 \rfloor];$ 
 $X_D \leftarrow X[\lfloor |X|/2 \rfloor + 1, |X|];$ 
 $x \leftarrow X[\lfloor |X|/2 \rfloor].abs;$ 
 $Y_G \leftarrow \text{tableau de taille } |X_G|;$ 
 $Y_D \leftarrow \text{tableau de taille } |X_D|;$ 
 $t_G \leftarrow 1;$ 
 $t_D \leftarrow 1;$ 
pour  $1 \leq i \leq |Y|$  faire
  si  $Y[i].abs \leq x$  alors
     $Y[t_G] \leftarrow T[i];$ 
     $t_G \leftarrow t_G + 1$ 
  sinon
     $Y[t_D] \leftarrow T[i];$ 
     $t_D \leftarrow t_D + 1$ 
  fin
fin
retourner  $(X_G, X_D, Y_G, Y_D)$ 

```

Cet algorithme renvoie bien les tableaux X_G, X_D, Y_G, Y_D des points à droite et à gauche de Δ triés par abscisses et ordonnées croissantes. La complexité est en $O(n)$, où $n = |X| = |Y|$.

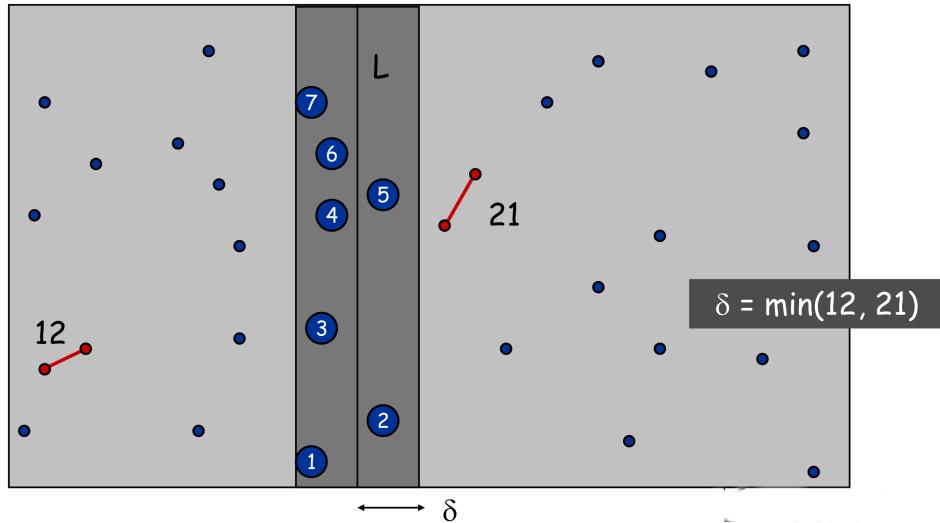


FIGURE 2.23 – Closest Pair of Points <http://perso.eleves.ens-rennes.fr/~mruffini/Files/0ther/ppr.pdf> https://ocw.tudelft.nl/wp-content/uploads/Algoritmiek_Closest_Pair_of_Points.pdf

Étape 4**Description de PPR-bande**

On note $b_1 \dots b_r$ les points de Y' triés par ordonnées croissantes. On va utiliser le résultat suivant :

Proposition 1

Il existe $b_i \neq b_j \in Y'$, avec $d(b_i, b_j) < \delta$ ssi il existe $i, j \in [|1, r|]$ tels que : $i < j \leq i + 7$ et $d(b_i, b_j) < \delta$

Démonstration : — “ \Leftarrow ” : immédiat.

— “ \Rightarrow ” : On suppose qu'il existe $b_i \neq b_j \in Y'$, avec $d(b_i, b_j) < \delta$. (OPS $i < j$)

Alors b_i et b_j sont dans un rectangle R de dimension $\delta \times 2\delta$ centré en Δ . Quitte à translater, on peut supposer que b_i est sur le côté inférieur de R . Montrons qu'au plus 8 points de P se trouvent dans R .

Pour cela, considérons le carré $\delta \times \delta$ formant la moitié gauche de ce rectangle. Puisque tous les points de P_G sont distants d'au moins δ , il y en a au plus 4 qui se trouvent dans ce carré (1 par sous-carré $\delta/2 \times \delta/2$ de diagonale $\sqrt{2}\delta/2 < \delta$).

De même, il y a au plus 4 points qui se trouvent dans la moitié droite de R . Donc, R contient au plus 8 points et on a bien (les b_i sont classés par ordonnées croissantes) :

$$i < j \leq i + 7 \text{ et } d(b_i, b_j) < \delta$$

■

Ainsi, on a l'algorithme suivant :

Cet algorithme a une complexité en $O(|Y'|)$ donc en $O(n)$.

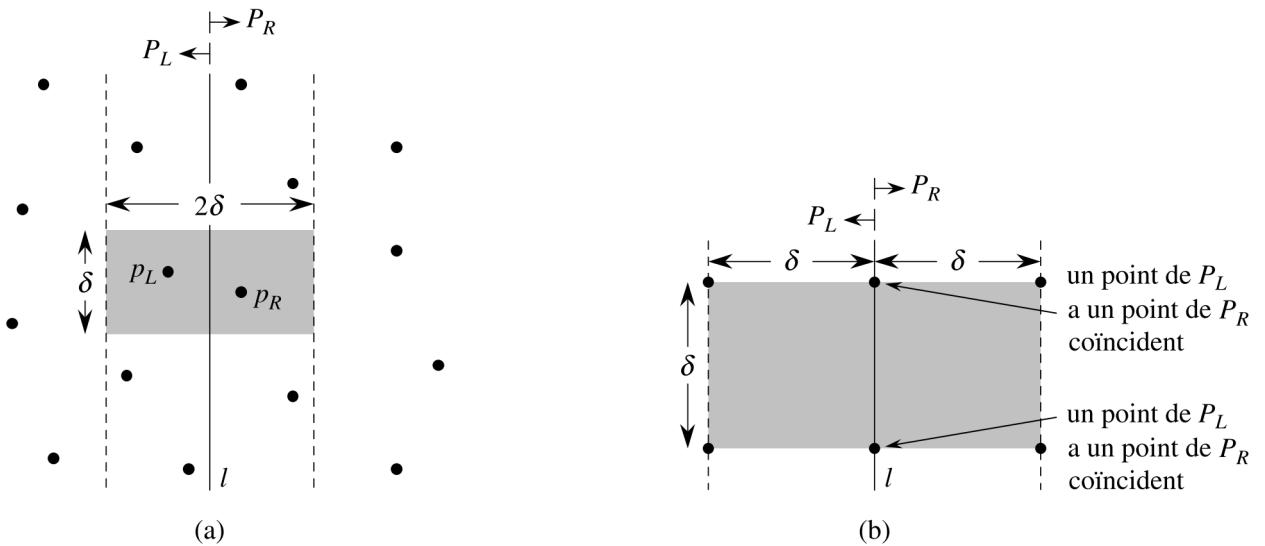
Algorithme 4 : PPR-bande(Y')

```

 $\delta' \leftarrow \delta;$ 
pour  $1 \leq i \leq |Y'| - 1$  faire
  pour  $i + 1 \leq j \leq \max(i + 7, |Y'|)$  faire
    si  $d(Y'[i], Y'[j]) < \delta'$  alors
       $\delta' \leftarrow d(Y'[i], Y'[j])$ 
    fin
  fin
retourner ( $\delta'$ )

```

FIGURE 2.24 – Closest Pair of Points <http://perso.eleves.ens-rennes.fr/~mruffini/Files/Other/ppr.pdf>



Remarques fondamentales pour démontrer que l'algorithme de recherche des deux points les plus rapprochés n'a besoin de tester que les 7 points qui suivent chaque point du tableau Y' . (a) Si $p_G \in P_G$ et $p_D \in P_D$ sont éloignés d'une distance inférieure à δ , ils doivent se trouver à l'intérieur d'un rectangle $\delta \times 2\delta$ centré sur la droite l . (b) Comment 4 points qui sont distants deux à deux d'au moins δ peuvent tous se trouver dans un carré $\delta \times \delta$. Les 4 points de gauche appartiennent à P_G et les 4 points de droite appartiennent à P_D . On peut placer 8 points dans le rectangle $\delta \times 2\delta$ si les points représentés sur la droite l sont en réalité des paires de points coïncidents, l'un appartenant à P_G et l'autre à P_D .

FIGURE 2.25 – Closest Pair of Points [Cormen et al., 1994, Cormen et al., 2001]

Étape 5

Complexité de PPR.

Soit $T(n)$ la complexité de PPR-rec pour n points. On a :

$$T(n) = \begin{cases} 2T(n/2) + O(n) & \text{si } n > 3 \\ O(1) & \text{si } n \leq 3 \end{cases}$$

D'où

$$T(n) = O(n \log n)$$

Finalement, l'algorithme PPR est en $O(n \log n)$

FIGURE 2.26 – Closest Pair of Points <http://perso.eleves.ens-rennes.fr/~mruffini/Files/Other/ppr.pdf>

Chapitre 3

Structures de données avancées

3.1 Piles, Files, et Listes [Cormen et al., 1994, Cormen et al., 2001]

Piles

PILEVIDE(P)

EMPILER(P, x)

DÉPILER(P)

Files

ENFILER(F, x)

DEFILER(F)

Listes

RECHERCHELISTE(L, k)

LISTEINSÉRER(L, x)

LISTESUPPRIMER(L, x)

3.2 Table de hachage [Cormen et al., 1994, Cormen et al., 2001]

De nombreuses applications font appel à des ensembles dynamiques qui ne supportent que les opérations de dictionnaire **INSÉRER**, **RECHERCHER** et **SUPPRIMER**. Une table de hachage est une structure de données permettant d'implémenter des dictionnaires. Bien que la recherche d'un élément dans une table de hachage soit aussi longue que la recherche d'un élément dans une liste chaînée - $\theta(n)$ dans le pire des cas -, en pratique, le hachage est très efficace. Avec des hypothèses raisonnables, le temps moyen de recherche pour un élément dans une table de hachage est $O(1)$.

3.2.1 Table à adressage direct

L'adressage direct est une technique simple qui fonctionne bien lorsque l'univers U des clés est raisonnablement petit. Supposons qu'une application ait besoin d'un ensemble dynamique dans lequel chaque élément possède une clé prise dans l'univers $U = \{0, 1, \dots, m - 1\}$, où m n'est pas trop grand. On supposera que deux éléments ne peuvent pas partager la même clé.

On utilise un tableau $T[0, 1, \dots, m - 1]$, dans lequel chaque position, ou alvéole, correspond à une clé dans l'univers U . Chaque position de $T[0, 1, \dots, m - 1]$ (dite alvéole) correspond à une clé dans l'univers U . L'alvéole k pointe sur un élément de l'ensemble ayant pour clé k . Si l'ensemble ne contient aucun élément de clé k , alors $T[k] = \text{nil}$. Le problème de cet adressage est évident : si l'univers U est grand !

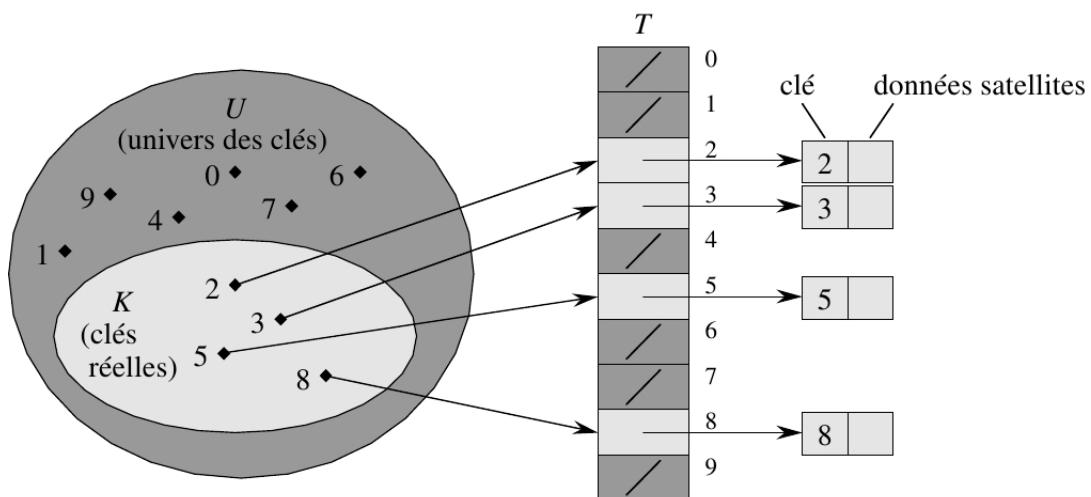


Figure Implémentation d'un ensemble dynamique à l'aide d'une table à adressage direct T . Chaque clé de l'univers $U = \{0, 1, \dots, 9\}$ correspond à un indice de la table. L'ensemble $K = \{2, 3, 5, 8\}$ des clés réelles détermine les alvéoles de la table qui contiennent des pointeurs vers des éléments. Les autres alvéoles, en gris foncé, contiennent NIL.

FIGURE 3.1 – Illustration de la table à adressage direct (scan de [Cormen et al., 2001])

L'ensemble des opérations $\text{RECHERCHEADDRESSAGEDIRECT}(T, k)$, $\text{INSERERADDRESSAGEDIRECT}(T, x)$, $\text{SUPPRIMERADDRESSAGEDIRECT}(T, x)$, est rapide : le temps est en $O(1)$.

RECHERCHER-ADRESSAGE-DIRECT(T, k)
retourner $T[k]$

INSÉRER-ADRESSAGE-DIRECT(T, x)
 $T[\text{clé}[x]] \leftarrow x$

SUPPRIMER-ADRESSAGE-DIRECT(T, x)
 $T[\text{clé}[x]] \leftarrow \text{NIL}$

FIGURE 3.2 – Algorithmes pour l'adressage direct (scan de [Cormen et al., 2001])

3.2.2 Tables de hachage

Le problème de l'adressage direct est évident : si l'univers U est grand, conserver une table T de taille $|U|$ peut se révéler impossible. L'ensemble K des clés réellement conservées peut être tellement petit comparé à U que la majeure partie de T est gaspillé.

Avec une table de hachage, le besoin en stockage se réduit à $\Theta(k)$, bien que la recherche d'un élément dans la table de hachage est en moyenne $O(1)$.

Avec le hachage, un élément de clé k est stocké dans l'alvéole $h(k)$; on utilise une fonction de hachage h pour calculer l'alvéole à partir de la clé k . h établit une correspondance entre l'univers U des clés et les alvéoles d'une table de hachage $T[0..m - 1]$.

$$h : U \rightarrow \{0, 1, \dots, m - 1\}.$$

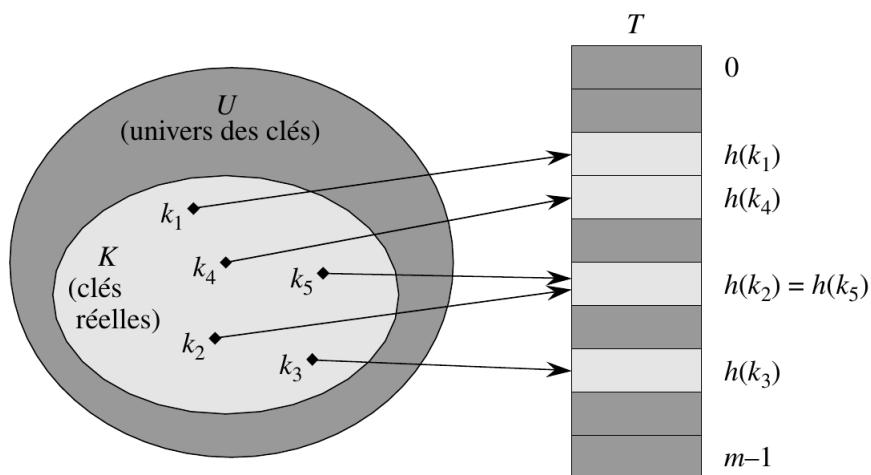


Figure Utilisation d'une fonction de hachage h pour faire correspondre les clés à des alvéoles d'une table de hachage. Les clés k_2 et k_5 correspondent à la même alvéole et entrent donc en collision.

FIGURE 3.3 – Illustration d'une table de hachage (scan de [Cormen et al., 2001])

On dit qu'un élément de clé k est haché dans l'alvéole $h(k)$; $h(k)$ est aussi dite valeur de hachage.

L'inconvénient de cette idée est que deux clés peuvent être hachées dans la même alvéole - ce qu'on appelle collision. Eviter complètement les collisions est impossible, par contre, la résolution est indispensable. La résolution la plus simple, dite par chainage. Elle consiste à stocker les éléments ayant la même valeur de hachage dans une liste chaînée.

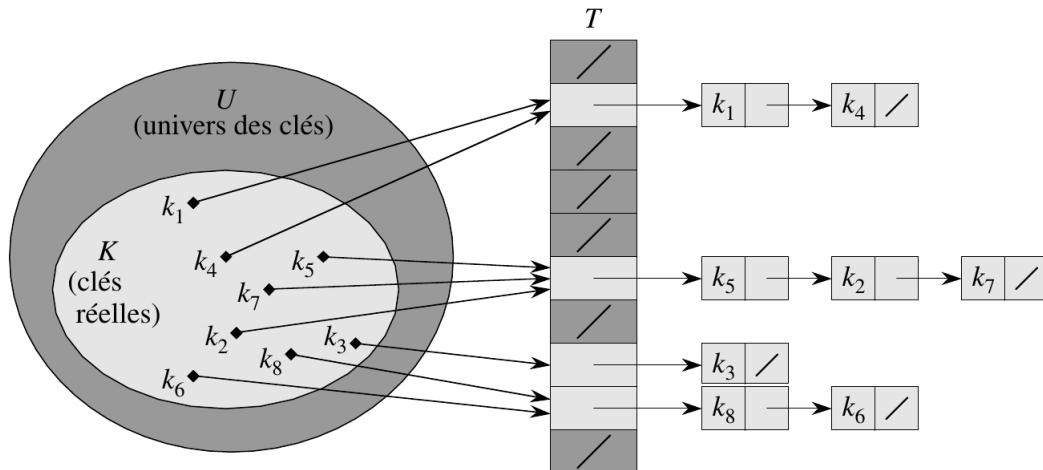


Figure Résolution des collisions par chaînage. Chaque alvéole de la table de hachage $T[j]$ contient une liste chaînée de toutes les clés dont la valeur de hachage est j . Par exemple $h(k_1) = h(k_4)$ et $h(k_5) = h(k_2) = h(k_7)$.

FIGURE 3.4 – Illustration d'une table de hachage par chainage (scan de [Cormen et al., 2001])

Les opérations de dictionnaire sur une table de hachage T sont :

INSÉRERHACHAGECHAINÉE insère x en tête de la liste $T[h(cl[x])]$. L'algorithme est en $O(1)$.

RECHERCHERHACHAGECHAINÉE recherche un élément dans la liste chainée $T[h(k)]$. Dans le pire elle recherche dans toute la liste.

SUPPRIMERHACHAGECHAINÉE supprime x de la liste $T[h(cl[x])]$. Il a la même complexité que la recherche.

INSÉRER-HACHAGE-CHAÎNÉE(T, x)

insère x en tête de la liste $T[h(clé[x])]$

RECHERCHER-HACHAGE-CHAÎNÉE(T, k)

recherche un élément de clé k dans la liste $T[h(k)]$

SUPPRIMER-HACHAGE-CHAÎNÉE(T, x)

supprime x de la liste $T[h(clé[x])]$

FIGURE 3.5 – Algorithmes pour les tables de hachage (scan de [Cormen et al., 2001])

Etant donnée une table de hachage T avec m alvéoles et conservant n éléments, on définit pour T le **facteur de remplissage α par n/m , c'est-à-dire le nombre moyen d'éléments stockés dans la chaîne**. L'analyse se fera en fonction de α . On suppose que α reste fixe lorsque n et m tendent vers l'infini.

Le comportement dans le pire des cas du hachage par chainage est très mauvais : si les n clés se retrouvent dans la même alvéole, elles y forment une liste de longueur n . Le temps d'exécution de la recherche dans ce cas est en $\Theta(n)$ qui est celui d'une structure de chainage.

La performance en moyenne du hachage dépend de la manière dont la fonction de hachage h répartit en moyenne l'ensemble des clés sur les m alvéoles. On se donne une hypothèse raisonnable : **Chaque clé a la même chance d'être haché dans toute alvéole. Les alvéoles sont équiprobables dans leur remplissage. C'est l'hypothèse de hachage uniforme simple.** On suppose que le calcul du hachage $h(k)$ est en $O(1)$, de même l'accès à l'alvéole.

Proposition 6. *Dans une table de hachage pour laquelle les collisions sont résolues par chainage, une recherche prend en moyenne un temps en $\Theta(1 + \alpha)$, sous l'hypothèse d'un hachage uniforme simple.*

La preuve de la proposition est basée sur le fait que la longueur des listes chainées est α .

3.2.3 Fonctions de hachage

Une bonne fonction de hachage vérifie l'hypothèse de hachage uniforme simple. Soit $P(k)$ la probabilité pour que k soit tirée de l'univers U des clés. L'hypothèse de hachage uniforme simple se traduit par

$$\sum_{k:h(k)=j} P(k) = 1/m, \text{ pour } j = 0, 1, \dots, m-1. \quad (3.1)$$

Par exemple, supposons que les clés soient des nombres réels k aléatoires, répartis indépendamment et uniformément dans l'intervalle $0 \leq k < 1$. Dans ce cas, on peut montrer que la fonction de hachage

$$h(k) = \lfloor km \rfloor$$

vérifie (3.1).

Les fonctions de hachage supposent que l'univers des clés est l'ensemble des entiers naturels $\{0, 1, 2, \dots\}$. **Par exemple, une clé sous forme de chaîne peut être interprétée comme un entier exprimé dans une base adaptée.** L'identificateur pt pourra être interprété comme la paire d'entiers décimaux $(112, 116)$, où $p = 112$, $t = 116$ dans le code ASCII. Ensuite, en l'exprimant en base 128, devient $112*128+116 = 14452$. Le plus souvent, il est facile de trouver une méthode aussi simple pour une application donnée quelconque, permettant d'interpréter chaque clé comme un entier naturel potentiellement grand.

La méthode de la division fait correspondre, pour créer des fonctions de hachage, une clé k avec l'une des m méthodes en prenant le reste de la division de k par m . En d'autres termes, la fonction de hachage est

$$h(k) = k \mod m.$$

Par exemple, si $m = 12$, $k = 100$, alors $h(k) = 4$. Cette méthode de hachage est très rapide. Pour que cette méthode ait de bonnes propriétés, il vaut mieux la faire dépendre de tous les bits de k . Ce qui n'est pas le cas si m est un multiple de 2, 2^p , donnant un hachage prenant en compte seulement les p bits inférieurs. Prendre m premier semble le meilleur choix. Soit $n = 2000$ chaînes, si on se donne $\alpha = 3$, alors on devrait prendre $m = 2000/3$ dont le premier le plus proche est 701.

La méthode de la multiplication agit en deux étapes

$$h(k) = \lfloor m(kA \mod 1) \rfloor$$

où $(kA \mod 1)$ représente la partie fractionnaire de k , c'est-à-dire $kA - \lfloor kA \rfloor$, et $A \in]0, 1[$. Knuth suggère que

$$A \approx (\sqrt{5} - 1)/2 = 0.6180339887\dots$$

a de bonnes chances de bien hacher.

Le hachage universel consiste à tirer au hasard la fonction de hachage pendant l'exécution. Cela permettra d'améliorer la sécurité du stockage des données.

3.2.4 Adressage ouvert

La résolution du problème des collisions par un chainage pose le problème de la lenteur de l'allocation mémoire des éléments stockés. **L'adressage ouvert propose une solution d'allocation statique : on alloue statiquement un vecteur des alvéoles ayant la taille des éléments à stocker. Puis, la collision est résolue via une fonction de sondage.**



FIGURE 3.6 – Adressage par chainage vs. adressage ouvert fr.wikipedia.org/wiki/Table_de_hachage

La fonction de hachage est de la forme

$$h : U \times \{0, 1, \dots, m - 1\} \rightarrow \{0, 1, \dots, m - 1\}.$$

Avec l'adressage ouvert, il faut que pour chaque clé k , la séquence de sondage

$$< h(k, 0), h(k, 1), \dots, h(k, m - 1) >$$

soit une permutation de $< 0, 1, \dots, m - 1 >$, de façon que chaque position de la table de hachage finisse par être considérée comme une alvéole pour une nouvelle clé lors du remplissage de la table. **En d'autres termes :**

- $h(k, 0)$ est la première case considérée pour stocker la clé k ;
- $h(k, 1)$ est la deuxième case considérée pour stocker la clé k ;
- ...
- $h(k, m - 1)$ est la dernière case considérée pour stocker la clé k ;

Par exemple : soit $k \in 1..7\,000\,000$, mais les données à stocker sont de taille $m = 7$. On peut avoir le comportement suivant de l'adressage ouvert :

- la séquence de la clé $k = 1\,000$ est $\langle 0, 1, 2, 3, 4, 5, 6 \rangle$
- la séquence de la clé $k = 2\,000$ est $\langle 1, 2, 0, 3, 4, 5, 6 \rangle$
- la séquence de la clé $k = 333\,000$ est $\langle 2, 0, 1, 3, 4, 5, 6 \rangle$
- la séquence de la clé $k = 500\,000$ est $\langle 0, 2, 1, 3, 4, 5, 6 \rangle$
- la séquence de la clé $k = 55$ est $\langle 6, 2, 0, 3, 4, 5, 1 \rangle$
- la séquence de la clé $k = 700$ est $\langle 5, 0, 1, 3, 4, 2, 6 \rangle$
- la séquence de la clé $k = 999\,000$ est $\langle 1, 0, 2, 3, 4, 5, 6 \rangle$

L'adressage ouvert stocke les valeurs de hachage dans des cases contiguës. Les positions de ces cases sont fixées avec une méthode de sondage. Pour chercher un élément, si la case obtenue par hachage direct ne permet pas d'obtenir la bonne clé, une recherche sur les cases suivantes obtenues par une méthode de sondage est effectuée jusqu'à trouver la clé, ou non, ce qui indique qu'aucune clé de ce type n'appartient à la table. Les Figures 3.8 et 3.7 détaillent les deux algorithmes de recherche et d'insertion dans l'adressage ouvert.

```

HASH-INSERT( $T, k$ )
1  $i \leftarrow 0$ 
2 repeat  $j \leftarrow h(k, i)$ 
3   if  $T[j] = \text{NIL}$ 
4     then  $T[j] \leftarrow k$ 
5     return  $j$ 
6   else  $i \leftarrow i + 1$ 
7 until  $i = m$ 
8 error "hash table overflow"

```

FIGURE 3.7 – Algorithme d'insertion pour l'adressage ouvert (scan de [Cormen et al., 2001])

```

HASH-SEARCH( $T, k$ )
1  $i \leftarrow 0$ 
2 repeat  $j \leftarrow h(k, i)$ 
3   if  $T[j] = k$ 
4     then return  $j$ 
5    $i \leftarrow i + 1$ 
6 until  $T[j] = \text{NIL}$  or  $i = m$ 
7 return NIL

```

FIGURE 3.8 – Algorithme de recherche pour l'adressage ouvert (scan de [Cormen et al., 2001])

La suppression d'un élément n'est pas gérable facilement. Une solution simple pour la suppression consiste à marquer l'élément à supprimer avec OTÉ, facilitant les nouvelles insertions dans cette case.

Le sondage linéaire fait appel à

$$h(k, i) = (h'(k) + i) \mod m,$$

où $i = 0, 1, \dots, m - 1$. Ce sondage souffre du phénomène, dit de la grappe forte, d'occupations d'une longue suite d'alvéoles, augmentant ainsi le temps de recherche.

Le sondage quadratique

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \mod m,$$

où h' est une fonction de hachage auxiliaire, c_1 et c_2 sont des constantes auxiliaires, et $i = 0, 1, \dots, m - 1$.

Le double hachage

$$h(k, i) = (h_1(k) + i h_2(k)) \mod m,$$

où h_1 et h_2 sont des fonctions de hachage auxiliaires. C'est l'une des meilleures fonctions pour l'adressage ouvert.

Proposition 7. Soit une table de hachage en adressage ouvert avec un facteur de remplissage $\alpha = n/m < 1$ (n est le nombre de clés à stocker, et non la valeur de la plus grande clé ! on a toujours $0 \leq \alpha \leq 1$). Pour donner une estimation asymptotique de la complexité des opérations, on fait tendre n et m vers l'infini, à α constant. Le nombre attendu de sondage lors de la :

recherche infructueuse vaut au plus $1/(1 - \alpha)$,
 recherche fructueuse vaut au plus $1/\alpha + \ln(1/(1 - \alpha))$,
 si l'on suppose que le hachage est uniforme.

Preuve : voir [Cormen et al., 1994, Cormen et al., 2001].

Par exemple pour une table à moitié pleine, on doit s'attendre à faire 2 accès pour la recherche d'un objet ne se trouvant pas dans la table, et à faire 3,387 accès s'il s'y trouve. Il s'agit bien d'un algorithme en $\theta(1)$.

3.3 Arbres binaires [Cormen et al., 1994, Cormen et al., 200

Les arbres de recherche sont des structures de données pouvant supporter nombre d'opérations sur les ensembles dynamiques : **RECHERCHER, MINIMUM, MAXIMUM, PREDECESSEUR, SUCCESEUR, INSÉRER, et SUPPRIMER.**

Définition 3. Soit x un noeud d'un arbre binaire de recherche. Si y est un noeud du sous-arbre gauche de x , alors $cl[y] \leq cl[x]$. Si y est un noeud du sous-arbre droit de x , alors $cl[x] \leq cl[y]$.

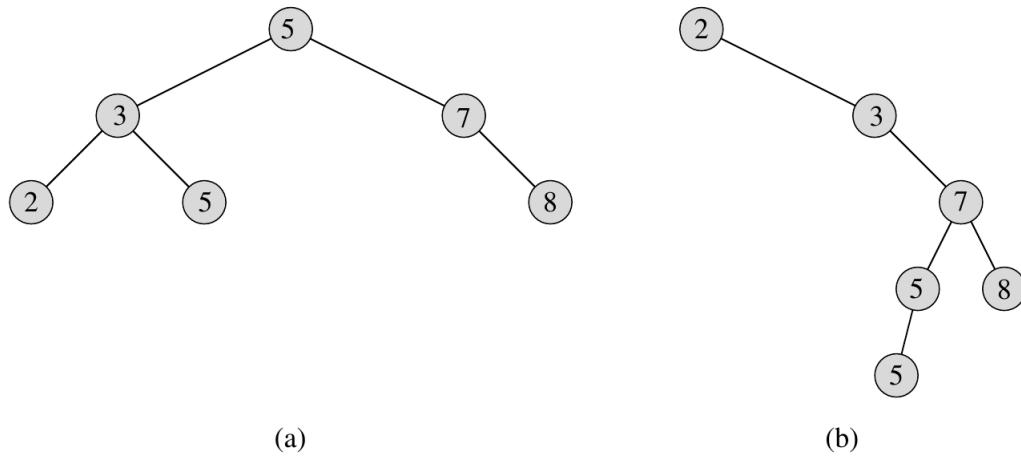


Figure Arbres binaires de recherche. Pour un nœud x , les clés du sous-arbre de gauche de x valent au plus $cl[x]$ et celles du sous-arbre de droite valent au moins $cl[x]$. Des arbres binaires de recherche différents peuvent représenter le même ensemble de valeurs. Le temps d'exécution, dans le cas le plus défavorable, pour la plupart des opérations d'arbre de recherche est proportionnel à la hauteur de l'arbre. (a) Un arbre binaire de recherche de 6 nœuds et de hauteur 2. (b) Un arbre binaire de recherche moins efficace de hauteur 4, contenant les mêmes clés.

FIGURE 3.9 – Illustration des arbres binaires (scan de [Cormen et al., 1994])

Les opérations sur un arbre binaire s'exécutent en $\Theta(h)$ dans le pire des cas, où h est la hauteur de l'arbre binaire. Pour un arbre binaire complet à n noeuds, ces opérations s'exécutent en $\Theta(\log n)$ dans le pire des cas. Dans le pire des cas, les opérations s'exécutent en $\Theta(n)$. La hauteur d'un arbre binaire de recherche construit aléatoirement est $O(\log n)$ qui est à l'origine de la complexité en $O(\log n)$.

ARBRERECHERCHER(x, k) Voir la Figure 3.11.

ARBRERECHERCHERITÉRATIVE(x, k) Voir la Figure 3.12.

ARBREMINIMUM(x) Voir la Figure 3.13.

ARBREMAXIMUM(x) Même principe que la recherche de l'élément minimale.

ARBREINSÉRER(T, z) Voir la Figure 3.15. Une illustration est donnée dans la Figure 3.16.

ARBRESUPPRIMER(T, z) Voir la Figure 3.17. Une illustration est donnée dans la Figure 3.18.

```

PARCOURS-INFIXE( $x$ )
1   si  $x \neq \text{NIL}$ 
2     alors PARCOURS-INFIXE( $\text{gauche}[x]$ )
3       afficher  $\text{clé}[x]$ 
4       PARCOURS-INFIXE( $\text{droite}[x]$ )

```

FIGURE 3.10 – Algorithme de parcours d'un arbres binaire (scan de [Cormen et al., 1994])

```

ARBRE-RECHERCHER( $x, k$ )
1   si  $x = \text{NIL}$  ou  $k = \text{clé}[x]$ 
2     alors retourner  $x$ 
3   si  $k < \text{clé}[x]$ 
4     alors retourner ARBRE-RECHERCHER( $\text{gauche}[x], k$ )
5     sinon retourner ARBRE-RECHERCHER( $\text{droite}[x], k$ )

```

FIGURE 3.11 – Algorithme de recherche dans un arbre binaire (scan de [Cormen et al., 1994])

```

ARBRE-RECHERCHER-ITÉRATIF( $x, k$ )
1   tant que  $x \neq \text{NIL}$  et  $k \neq \text{clé}[x]$ 
2     faire si  $k < \text{clé}[x]$ 
3       alors  $x \leftarrow \text{gauche}[x]$ 
4       sinon  $x \leftarrow \text{droite}[x]$ 
5   retourner  $x$ 

```

FIGURE 3.12 – Algorithme itératif de recherche dans un arbre binaire (scan de [Cormen et al., 1994])

```

ARBRE-MINIMUM( $x$ )
1   tant que  $\text{gauche}[x] \neq \text{NIL}$ 
2     faire  $x \leftarrow \text{gauche}[x]$ 
3   retourner  $x$ 

```

FIGURE 3.13 – Algorithme de recherche de la valeur minimale d'un arbre binaire (scan de [Cormen et al., 1994])

```

ARBRE-SUCCESSEUR( $x$ )
1   si  $droite[x] \neq \text{NIL}$ 
2     alors retourner ARBRE-MINIMUM( $droite[x]$ )
3      $y \leftarrow p[x]$ 
4     tant que  $y \neq \text{NIL}$  et  $x = droite[y]$ 
5       faire  $x \leftarrow y$ 
6        $y \leftarrow p[y]$ 
7   retourner  $y$ 

```

FIGURE 3.14 – Algorithme de recherche du successeur d'une valeur dans un arbre binaire (scan de [Cormen et al., 1994])

ARBRE-INSÉRER(T, z)

```

1    $y \leftarrow \text{NIL}$ 
2    $x \leftarrow racine[T]$ 
3   tant que  $x \neq \text{NIL}$ 
4     faire  $y \leftarrow x$ 
5       si  $clé[z] < clé[x]$ 
6         alors  $x \leftarrow gauche[x]$ 
7         sinon  $x \leftarrow droite[x]$ 
8    $p[z] \leftarrow y$ 
9   si  $y = \text{NIL}$ 
10    alors  $racine[T] \leftarrow z$             $\triangleright$  arbre  $T$  était vide
11    sinon si  $clé[z] < clé[y]$ 
12      alors  $gauche[y] \leftarrow z$ 
13      sinon  $droite[y] \leftarrow z$ 

```

FIGURE 3.15 – Algorithme d'insertion d'une valeur dans un arbre binaire (scan de [Cormen et al., 1994])

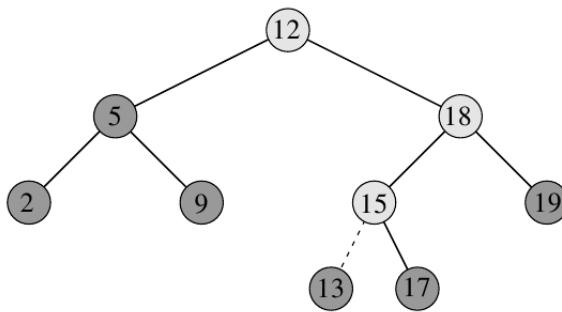
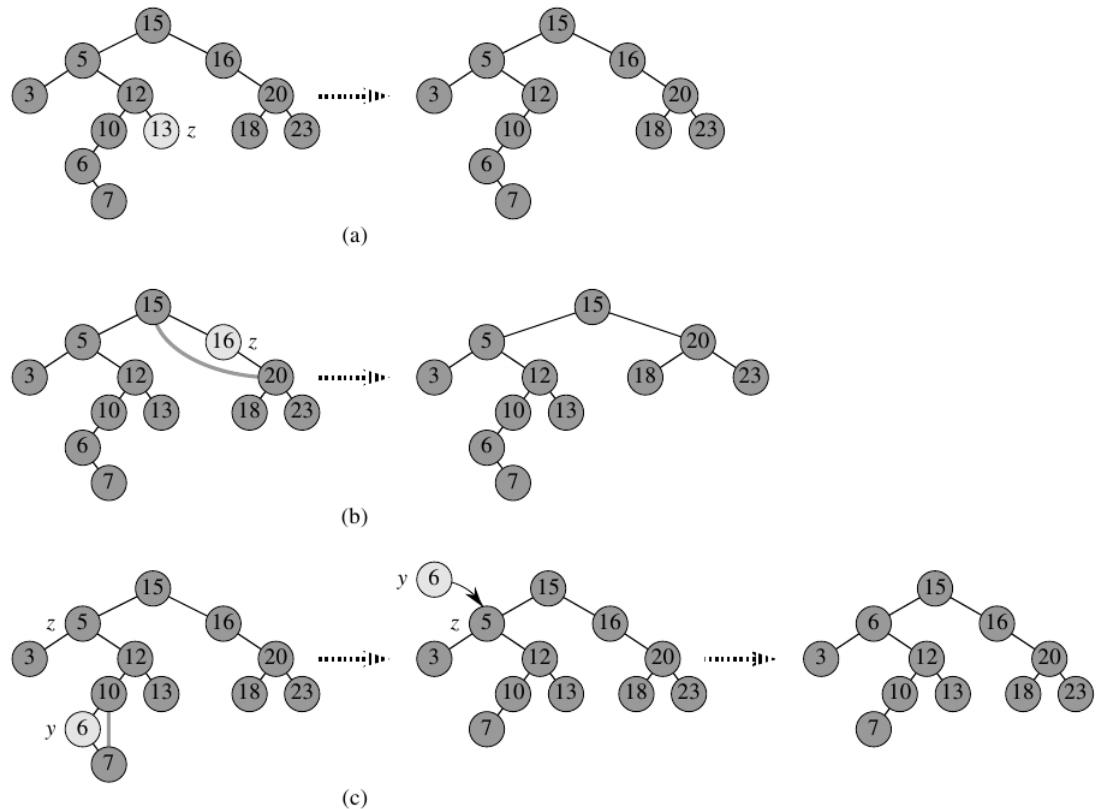


FIGURE 3.16 – Illustration de l'algorithme d'insertion d'une valeur dans un arbre binaire (scan de [Cormen et al., 1994])

```

ARBRE-SUPPRIMER( $T, z$ )
1   si  $gauche[z] = \text{NIL}$  ou  $droite[z] = \text{NIL}$ 
2     alors  $y \leftarrow z$ 
3     sinon  $y \leftarrow \text{ARBRE-SUCESSEUR}(z)$ 
4     si  $gauche[y] \neq \text{NIL}$ 
5       alors  $x \leftarrow gauche[y]$ 
6       sinon  $x \leftarrow droite[y]$ 
7     si  $x \neq \text{NIL}$ 
8       alors  $p[x] \leftarrow p[y]$ 
9     si  $p[y] = \text{NIL}$ 
10      alors  $\text{racine}[T] \leftarrow x$ 
11      sinon si  $y = gauche[p[y]]$ 
12        alors  $gauche[p[y]] \leftarrow x$ 
13        sinon  $droite[p[y]] \leftarrow x$ 
14   si  $y \neq z$ 
15     alors  $clé[z] \leftarrow clé[y]$ 
16     copier données satellites de  $y$  dans  $z$ 
17   retourner  $y$ 
  
```

FIGURE 3.17 – Algorithme de suppression d'une valeur dans un arbre binaire (scan de [Cormen et al., 1994])



Suppression d'un nœud z d'un arbre binaire de recherche. Pour savoir quel est le nœud à supprimer effectivement, on regarde le nombre d'enfants de z ; ce noeud est colorié en gris clair. (a) Si z n'a pas d'enfant, on se contente de le supprimer. (b) Si z n'a qu'un seul enfant, on détache z . (c) Si z a deux enfants, on détache son successeur y qui possède au plus un enfant et on remplace alors la clé et les données satellites de z par celles de y .

FIGURE 3.18 – Illustration de l'algorithme de suppression d'une valeur dans un arbre binaire (scan de [Cormen et al., 1994])

Proposition 8. *Toutes les opérations citées ci-haut sont en $O(h)$, où h est la hauteur de l'arbre binaire.*

3.4 Arbres AVL

[Références : [1](#) [2](#)]

Les arbres AVL sont des arbres binaires de recherche introduit par Adelson-Velskii et Landis en 1962.

Définition 4. Un arbre binaire de recherche est un arbre AVL si, pour n'importe lequel de ses noeuds, la différence absolue de hauteur entre ses deux fils diffère d'au plus un.

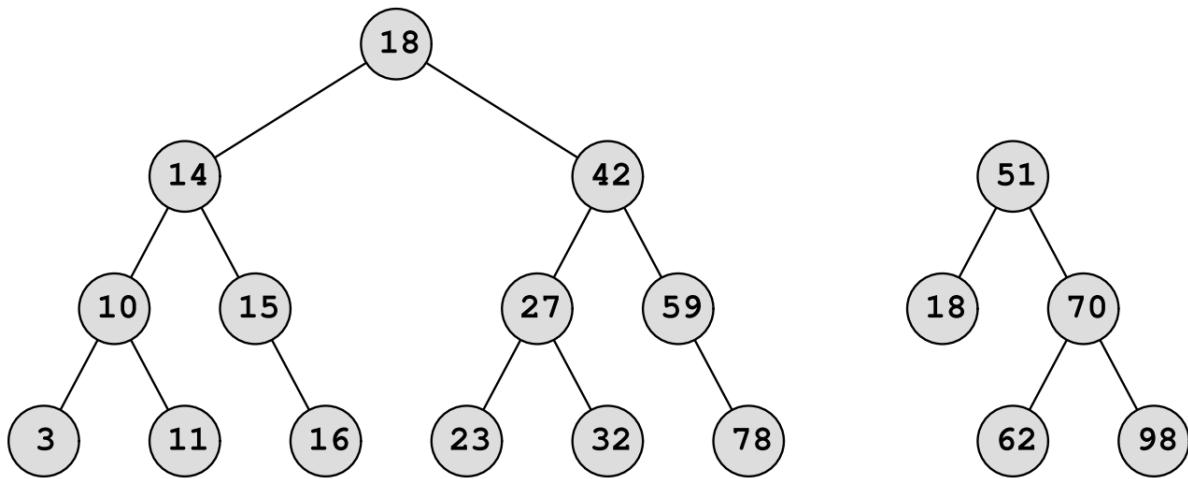


FIGURE 3.19 – Arbres AVL [1](#)

1. <http://stephane.glondu.net/projets/tipe/transparents.pdf>

2. <https://www.enseignement.polytechnique.fr/profs/informatique/Jean-Eric.Pin/PDF/Amphi9.pdf>

Proposition 9. Soit A un arbre AVL ayant n sommets et de hauteur h . Alors

$$\log_2(1+n) \leq 1+h \leq \alpha \log_2(2+n)$$

avec $\alpha \leq 1.44$.

Preuve On a toujours $n \leq 2^{h+1} - 1$, car le nombre de noeuds est toujours inférieur au max qu'on pourrait placer dans cet arbre. Et donc $\log_2(1+n) \leq 1+h$.

Soit u_h le nombre minimum de sommets d'un arbre AVL de hauteur h . Alors

$$u_h = u_{h-1} + u_{h-2} + 1 \text{ car la diff des hauteurs est } 1 \text{ au min (sinon max...)}$$

$$u_0 = 1, u_1 = 2$$

Posons $F_h = u_h + 1$. On a donc

$$F_h = F_{h-1} + F_{h-2} \text{ en ajoutant } +1 \text{ dans les deux membres ci-haut}$$

$$F_0 = 2, F_1 = 3$$

Cette dernière équation est une équation récurrente linéaire d'ordre 2 à coefficients constants. On peut donc appliquer le théorème 1. On obtient la solution :

$$F_h = c_1 \beta_1^h + c_2 \beta_2^h$$

$$\text{où } \beta_1 = \frac{1+\sqrt{5}}{2} \simeq 1.62, c_1 = \frac{5+2\sqrt{5}}{5} \simeq 1.89,$$

$$\beta_2 = \frac{1-\sqrt{5}}{2} \simeq -0.62 \text{ et } c_2 = \frac{5-2\sqrt{5}}{5} \simeq 0.11.$$

Et donc

$$u_h = c_1 \beta_1^h + c_2 \beta_2^h - 1$$

On a $-1 \leq \beta_2^h$, d'où $c_2 \beta_2^h \geq -c_2 \geq -0.11 \geq -1$. On peut poser

$$n \geq u_h = c_1 \beta_1^h + c_2 \beta_2^h - 1 > c_1 \beta_1^h - 2$$

$$h < \frac{\log_2(n+2)}{\log_2 \beta_1} - \frac{\log_2 c_1}{\log_2 \beta_1} \leq \frac{\log_2(n+2)}{\log_2 \beta_1} - 1.31 \dots$$

$$h \leq \frac{\log_2(n+2)}{\log_2 \beta_1} - 1$$

En considérant \leq et dominant $-\log_2 c_1$:

$$h+1 \leq \frac{1}{\log_2 \beta_1} \log_2(n+2)$$

□

Par exemple si $n = 100000$, alors $17 \leq h \leq 25$.

Les **rotations** gauche et droit transforment un arbre :

- Elles préservent l'ordre infixe,
- Elles se réalisent en temps constant.

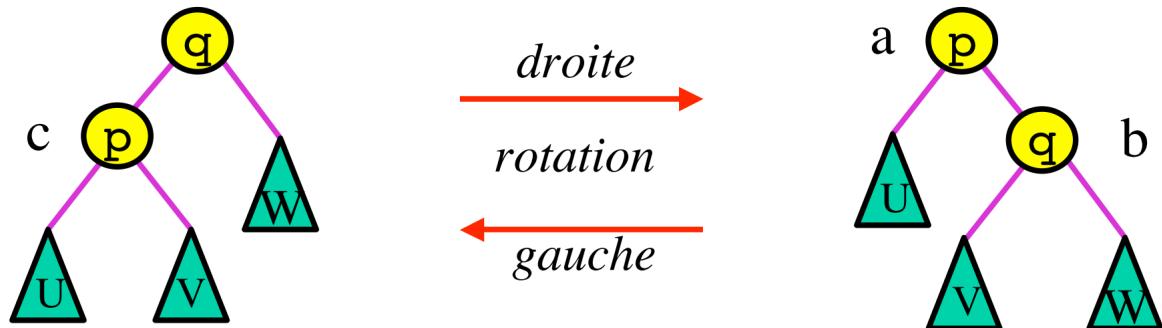


FIGURE 3.20 – Rotations des arbres AVL ²

Algorithm 18 ROTATIONGAUCHE(ARBRE a)

Input: un arbre AVL a

Output: arbre produit de la rotation "gauche" de a

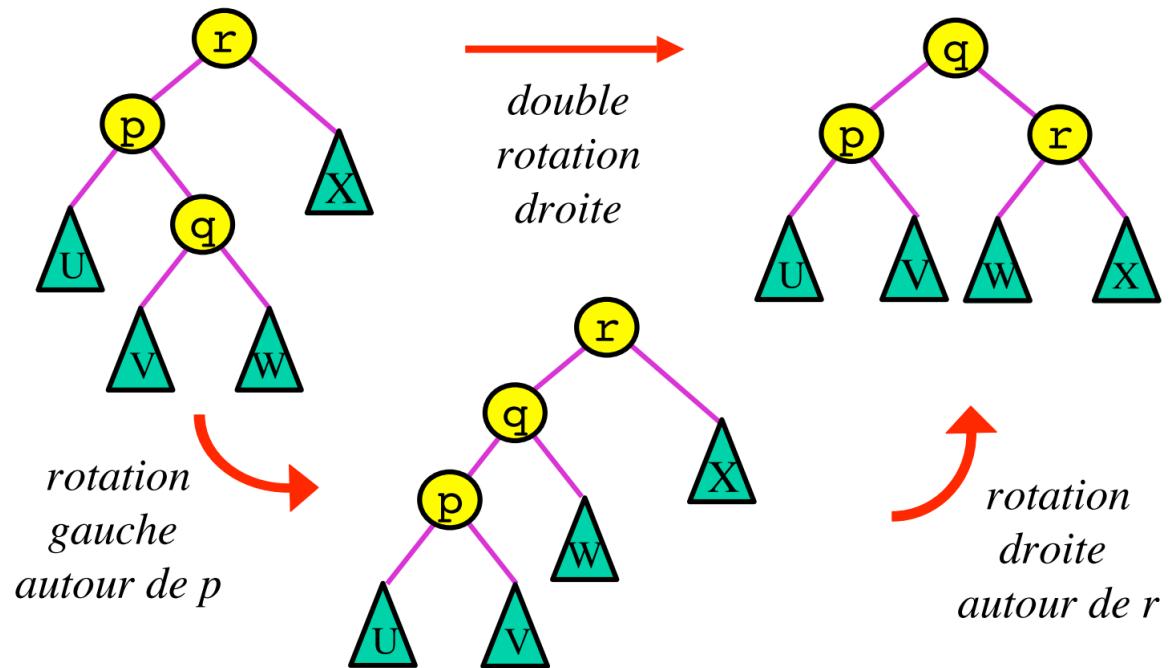
- 1: Arbre $b = a.filsDroit$
 - 2: $a.filsDroit = b.filsGauche$
 - 3: $b.filsGauche = a$
 - 4: **return** b
-

Algorithm 19 ROTATIONDROIT(ARBRE a)

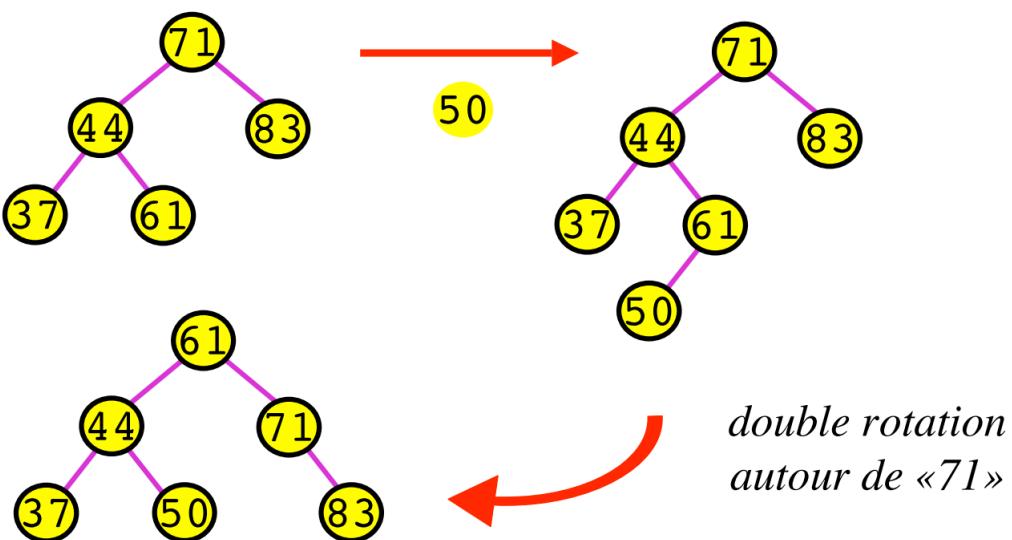
Input: un arbre AVL a

Output: arbre produit de la rotation "droit" de a

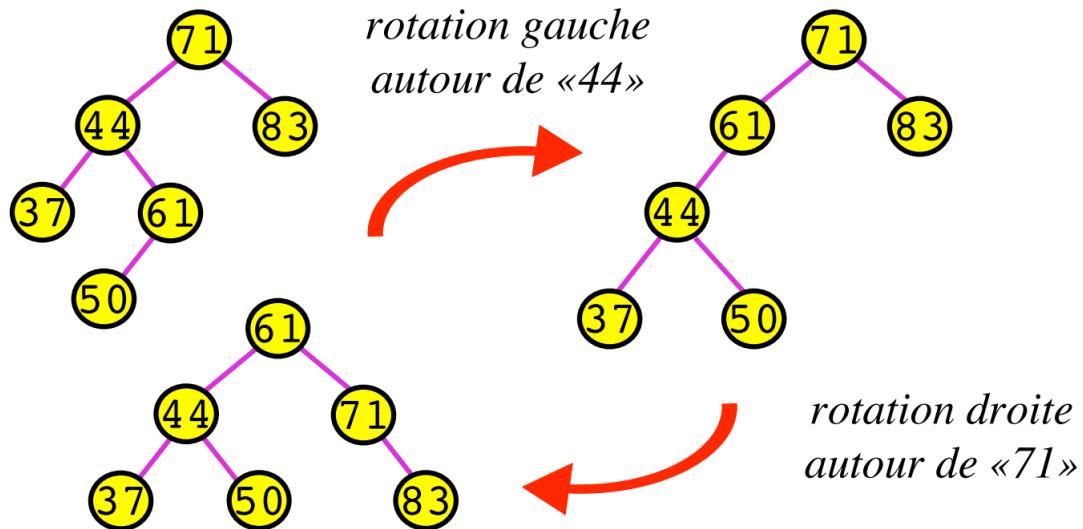
- 1: Arbre $b = a.filsGauche$
 - 2: $a.filsGauche = b.filsDroit$
 - 3: $b.filsDroit = a$
 - 4: **return** b
-

FIGURE 3.21 – Rotations des arbres AVL²

- Insertion ordinaire, suivie de rééquilibrage.

FIGURE 3.22 – Insertion suivie de rééquilibrage²

• Insertion ordinaire, suivie de rééquilibrage.

FIGURE 3.23 – Insertion suivie de rééquilibrage ²

Algorithm 20 EQUILIBRER(ARBRE A)

Input: un arbre AVL A , G et D sont ses sous-arbres gauche et droit

Output: Arbre équilibré de A

- 1: **if** $h(G) - h(D) = 2$ **then**
 - 2: Soient g et d les sous-arbres gauche et droit de G .
 - 3: **if** $h(g) < h(d)$ **then**
 - 4: On fait d'abord une rotation gauche de G .
 - 5: **end if**
 - 6: On fait une rotation droite
 - 7: **end if**
 - 8: **if** $h(G) - h(D) = -2$ **then**
 - 9: Opérations symétriques
 - 10: **end if**
-

Conséquences :

- Pour rééquilibrer un arbre AVL après une insertion, une seule rotation ou double rotation suffit **sur la racine**.
- Pour rééquilibrer un arbre AVL après une suppression, il faut jusqu'à h rotations ou double rotations **sur le chemin de la suppression** (h est la hauteur de l'arbre).

3.5 Arbres rouge et noir [Cormen et al., 1994, Cormen et al., 1994]

Un arbre binaire de recherche est un arbre rouge et noir s'il satisfait les propriétés suivantes :

1. chaque noeud est soit rouge, soit noir,
2. la racine est noire,
3. chaque feuille NIL est noire.
4. si un noeud est rouge, alors ses deux fils sont noirs,
5. chaque chemin simple reliant un noeud à une famille descendante de feuilles contient le même nombre de noeud noirs.

Un exemple d'un arbre rouge et noir et donné dans la Figure 3.24.

Proposition 10. *La hauteur moyenne d'un arbre binaire rouge et noir a une hauteur au plus égale à $O(2 \log n + 1)$.*

En conséquence de cette proposition, les algorithmes de recherche d'une valeur, du minimum, du successeur ont une complexité égale à $O(2 \log n + 1)$. Les algorithmes d'insertion (resp. de suppression) dans un arbre rouge et noir doit tenir compte de sa structure particulière (voir [Cormen et al., 1994]).

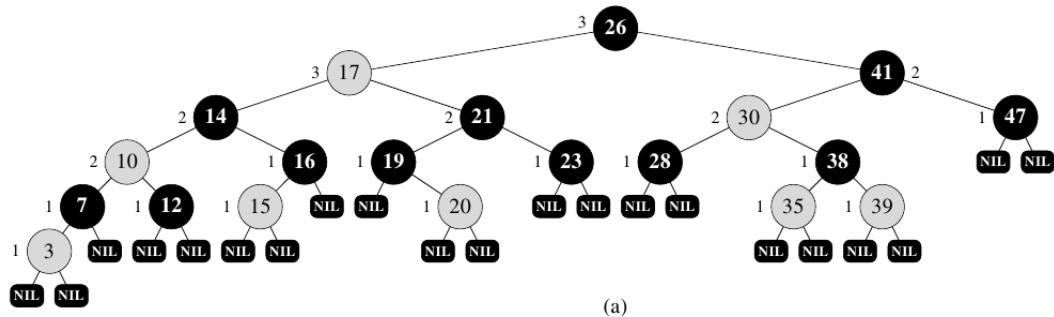


FIGURE 3.24 – Illustration d'un arbre rouge et noir (scan de [Cormen et al., 1994])

3.6 B-arbres [Cormen et al., 1994, Cormen et al., 2001]

Les B-arbres sont des arbres de recherche équilibrés conçus pour être efficaces sur des disques magnétiques ou d'autres unités de stockage secondaires à accès direct. La différence majeure entre les B-arbres et les arbres rouge et noir réside dans le fait que les noeuds des B-arbres peuvent avoir de nombreux fils, jusqu'à des milliers : le facteur de branchement d'un B-arbre peut être très grand.

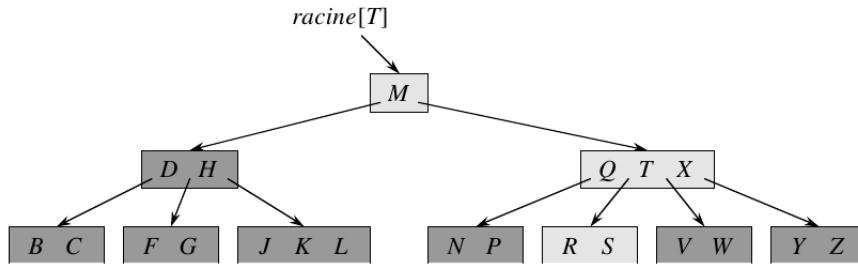


FIGURE 3.25 – Exemple d'un *B*-arbre (scan de [Cormen et al., 1994])

Un arbre binaire de recherche est un B-arbre s'il satisfait les propriétés suivantes :

1. Chaque noeud x contient les champs ci-dessous :
 - (a) $n[x]$, le nombre de clés conservées par le noeud x ,
 - (b) les $n[x]$ clés elles-mêmes, stockées par ordre croissant : $cl_1[x] \leq cl_2[x] \leq \dots \leq cl_n[x]$, et
 - (c) $feuille[x]$ vaut vrai si x est une feuille et faux sinon.
2. si x est un noeud interne, il contient également $n[x]+1$ pointeurs $c_1[x], c_2[x], \dots, c_{n[x]+1}[x]$ sur ses fils. Les feuilles n'ont aucun fils.
3. Les clés $cl_i[x]$ déterminent les intervalles de clés stockés dans chaque sous-arbre : si k_i est une clé quelconque stockée dans le sous-arbre de racine $c_i[x]$, alors

$$k_1 \leq cl_1[x] \leq k_2 \leq cl_2[x] \leq \dots \leq cl_{n[x]}[x] \leq k_{n[x]+1}$$
4. Toute les feuilles ont la même profondeur, qui est égale à h , la hauteur de l'arbre.
5. Il existe des bornes inférieures et supérieures sur le nombre de clés pouvant être contenues par un noeud. Ces bornes peuvent être exprimées en fonction d'un entier fixé $t \geq 2$, appelé le degré minimum du B-arbre :
 - (a) Tout noeud autre que la racine doit contenir au moins $t - 1$ clés. Tout noeud interne autre que la racine possède donc au moins t fils. Si l'arbre n'est pas vide, la racine doit posséder au moins une clé.
 - (b) Tout noeud peut contenir au plus $2t - 1$ clés. Un noeud interne peut donc posséder au plus $2t$ fils. On dit qu'un noeud est complet s'il contient exactement $2t - 1$ clés.

Un exemple d'un B -arbre est donné dans la Figure 3.26.

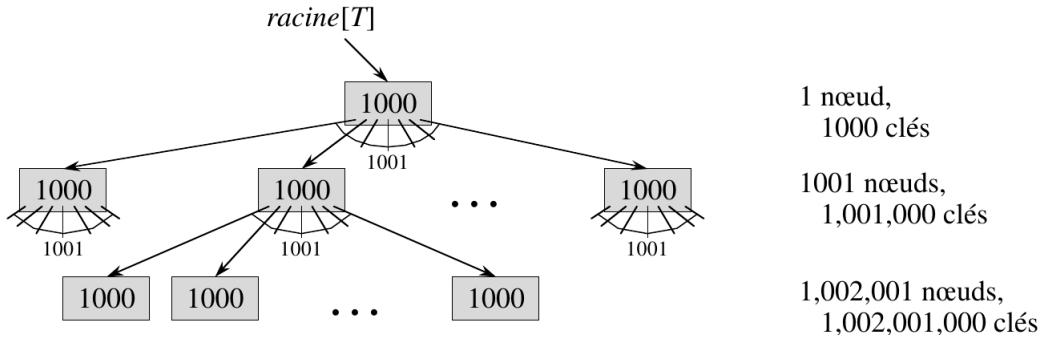


Figure Un B -arbre de hauteur 2 contenant plus d'un milliard de clés. Chaque nœud (feuilles comprises) contient 1 000 clés. Il existe 1 001 nœuds à la profondeur 1 et plus d'un millions de feuilles à la profondeur 2. On peut voir à l'intérieur de chaque nœud la valeur $n[x]$, qui représente le nombre de clés de x .

FIGURE 3.26 – Exemple d'un B -arbre (scan de [Cormen et al., 1994])

Théorème 3. Si $n \geq 1$, alors pour un B -arbre T à n clés, de hauteur h et de degré minimum $t \geq 2$,

$$h \leq \log_t(n + 1)/2.$$

RECHERCHER-B-ARBRE(x, k)

- 1 $i \leftarrow 1$
- 2 **tant que** $i \leq n[x]$ et $k > clé_i[x]$
- 3 **faire** $i \leftarrow i + 1$
- 4 **si** $i \leq n[x]$ et $k = clé_i[x]$
- 5 **alors retourner** (x, i)
- 6 **si** $feuille[x]$
- 7 **alors retourner** NIL
- 8 **sinon** LIRE-DISQUE($c_i[x]$)
- 9 **retourner** RECHERCHER-B-ARBRE($c_i[x], k$)

FIGURE 3.27 – Algorithmes pour les B-arbres (scan de [Cormen et al., 2001])

Pour plus de détails des algorithmes sur les B-arbres, on recommande la référence [Cormen et al., 1994].

3.7 Tas binomiaux [Cormen et al., 1994, Cormen et al., 2001]

Ici, nous abordons des structures de données connues sous le nom générique de tas fusionnables, qui supportent les cinq opérations suivantes :

CRÉERTAS() crée et retourne un nouveau tas sans éléments.

INSÉRER(T, x) insère dans le tas T un noeud x , dont le champ cl a déjà été rempli.

MINIMUM(T) retourne un pointeur sur le noeud dont la clé est minimum dans le tas T .

EXTRAIREMIN(T) supprime du tas T le noeud dont la clé est minimum, et retourne un pointeur sur ce noeud.

UNION(T_1, T_2) crée et retourne un nouveau tas qui contient tous les noeuds des tas T_1 et T_2 . Les tas T_1 et T_2 sont détruits par cette opération.

Ces structures de données utilisées ici supportent aussi les deux opérations suivantes :

DIMINUERCLÉ($T, x; k$) affecte au noeud x du tas T la nouvelle valeur de clé k , dont on suppose qu'elle est inférieure à sa valeur de clé courante.

SUPPRIMER(T, x) supprime le noeud x du tas T .

Nous donnons ci-dessous les complexités de ces opérations.

Procédure	Tas binaire(pire)	Tas binomial(pire)	Tas de Fibonacci(...)
INSÉRER	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$
EXTRAIREMIN	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
UNION	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$
DIMINUERCLÉ	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
SUPPRIMER	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

3.7.1 Arbres binomiaux

Un arbre binomial B_k est un arbre ordonné défini récursivement comme illustré dans la Figure 3.28 :

1. L'arbre B_0 contient un noeud unique.
2. L'arbre binomial B_k est constitué de deux arbres binomiaux B_{k-1} où la racine de l'un est l'enfant le plus à gauche de la racine de l'autre.

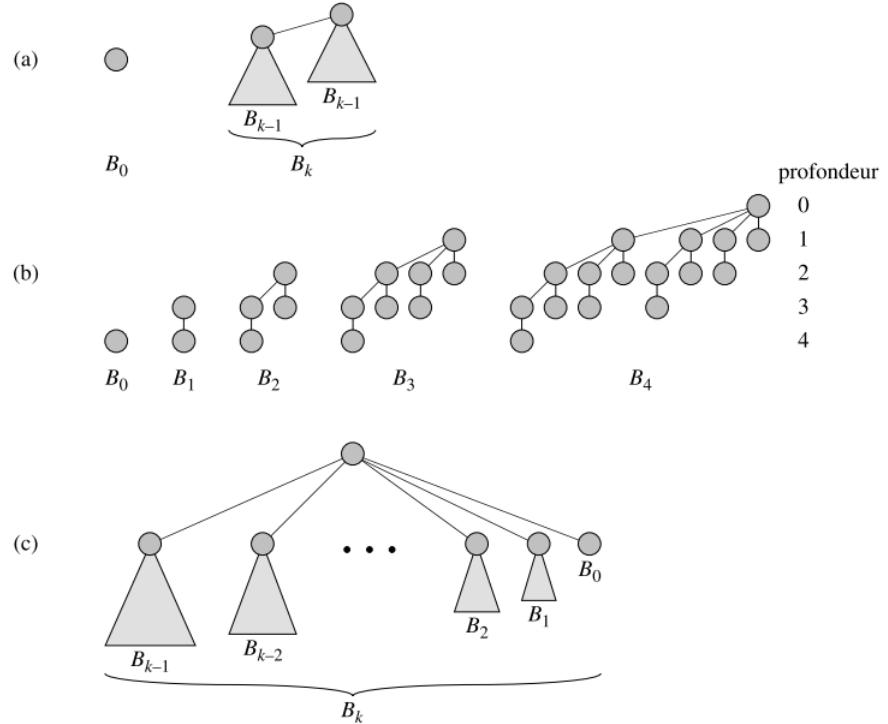


FIGURE 3.28 – Illustration des arbres binomiaux (scan de [Cormen et al., 2001])

Proposition 11. Pour un arbre binomial B_k :

1. il existe 2^k noeuds,
2. la hauteur de l'arbre est k ,
3. il existe $C_k^i = \binom{k}{i} = \frac{A_k^i}{i!} = \frac{k!}{(k-i)!i!}$ noeuds à la profondeur $i = 0, 1, \dots, k$.
4. la racine a le degré k qui est supérieur à celui de tout autre noeud ; si les enfants de la racine sont numérotés de la gauche vers la droite par $k-1, k-2, \dots, 0$, l'enfant i est la racine d'un sous-arbre B_i .

3.7.2 Définition d'un tas binomial

Un tas binomial T est un ensemble d'arbres binomiaux qui satisfait les propriétés des tas binomiaux :

1. Chaque arbre binomial de T obéit à la propriété de tas min : la clé d'un noeud est supérieure ou égale à la clé de son parent. On dit d'un tel arbre qu'il est trié en tas min.
2. Quel que soit l'entier k positif, il existe dans T un arbre binomial au plus dont la racine a le degré k .

La première propriété nous dit que la racine d'un arbre trié en tas min contient la plus petite clé de l'arbre. La deuxième propriété implique qu'un tas binomial T à n noeuds est constitué d'au plus $\lg(n) + 1$ arbres binomiaux. La représentation binaire de n comporte $\lg(n) + 1$ bits, notés par exemple $< b_{\lg(n)}, b_{\lg(n)-1}, \dots, b_0 >$, avec

$$n = \sum_{i=0..lg(n)} b_i 2^i.$$

Car si $n = \sum_{i=0..k} b_i 2^i$, alors $n \leq 2^{k+1}$, d'où $k \geq \ln(n)/\ln(2) - 1$, et on peut prendre ainsi $k = \ln(n)/\ln(2) = \lg(n)$. D'après la propriété des arbres binomiaux, l'arbre binomial B_i apparaît dans T si et seulement si le bit $b_i = 1$. Donc, le tas binomial T contient au plus $\lg(n) + 1$ arbres binomiaux.

La Figure 3.29 illustre la représentation d'un tas binomial contenant 13 éléments.

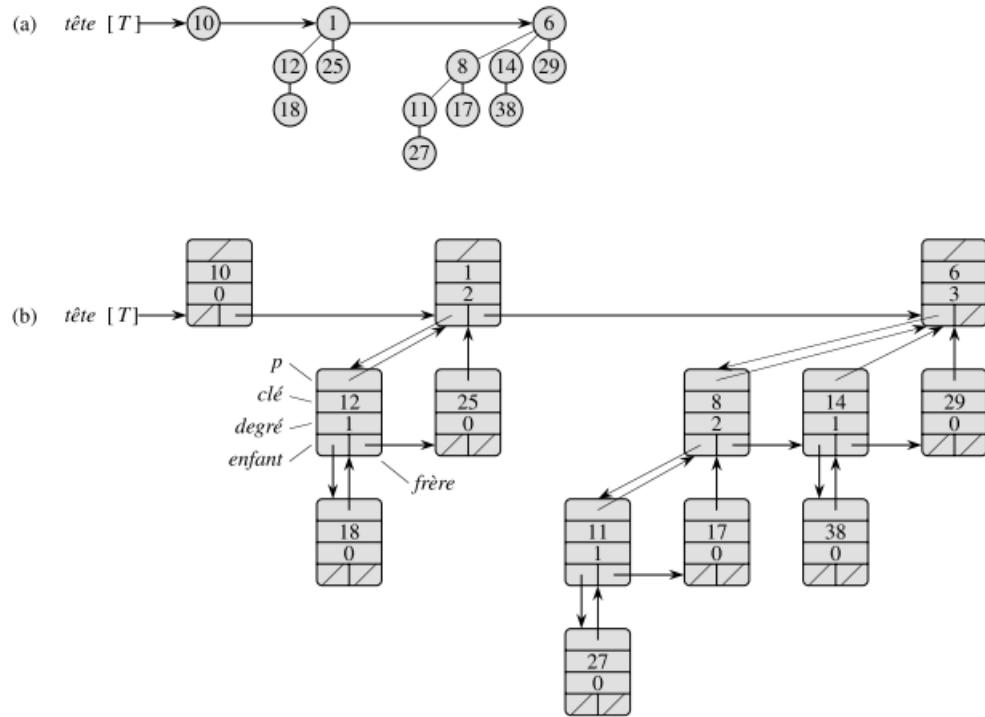


Figure Un tas binomial T à $n = 13$ nœuds. (a) Le tas est constitué des arbres binomiaux B_0 , B_2 et B_3 , qui comportent respectivement 1, 4 et 8 nœuds, pour un total de $n = 13$ nœuds. Comme chaque arbre binomial est trié en tas min, la clé d'un nœud quelconque est supérieure ou égale à celle de son parent. La liste des racines est aussi représentée ; c'est une liste chaînée de racines par ordre de degrés croissant. (b) Une représentation plus détaillée du tas binomial T . Chaque arbre binomial est stocké dans la représentation « enfant de gauche, frère de droite » et chaque nœud contient son degré.

FIGURE 3.29 – Illustration d'un tas binomial (scan de [Cormen et al., 2001])

Pour plus de détails, on recommande la référence [Cormen et al., 1994].

3.8 Analyse amortie [Robert et al., 2005]

On cherche à calculer la moyenne du coût de n opérations successives (à ne pas confondre avec la complexité en en moyenne).

L'analyse amortie est une méthode d'évaluation de la complexité temporelle des opérations sur une structure de données. Elle consiste à majorer le coût cumulé d'une suite d'opérations, et attribuer à chaque opération la moyenne de cette majoration, en prenant en compte le fait que les cas couteux surviennent peu et d'une façon isolée et compensent les cas non couteux.

- ³ L'analyse amortie diffère de l'analyse du cas moyen pour les raisons suivantes :
- dans l'analyse du cas moyen, on cherche à exploiter le fait que les cas onéreux sont peu probables en faisant des hypothèses sur la distribution des entrées ou sur les choix aléatoires effectués dans l'algorithme. Dans cette analyse, chaque opération est considérée indépendamment : il n'y a pas de notion d'état mémoire ;
 - dans l'analyse amortie, on ne fait pas appel au calcul des probabilités mais on cherche à exploiter le fait que, dans une suite d'opérations partant d'une donnée quelconque, les cas onéreux ne peuvent pas se produire très souvent, ou de manière très rapprochée et que surtout ils profitent aux autres. Cette analyse requiert donc d'identifier les cas les plus défavorables d'une suite d'opérations et de mesurer leur contribution à l'amélioration globale de l'efficacité de la dite suite. Dans l'analyse amortie, les opérations sont envisagées comme parties d'une séquence globale sur une donnée qu'elles modifient : il y a donc une notion d'état mémoire.

Etudions l'incrémentation d'un compteur à k bits :

Exemple : si $k = 6$

0	0	0	0	0	0
0	0	0	0	0	1
0	0	0	0	1	0
0	0	0	0	1	1
0	0	0	1	0	0

On définit le coût d'une incrémentation comme le nombre de bits qui changent lors de celle ci. Ce coût n'est pas constant pour chaque valeur de x . On remarque que ce coût équivaut au nombre de 1 successifs à droite du compteur plus un (qui correspond à la transformation du premier 0). Ainsi le coût des n incrémentations successives est majoré par nk . Cependant cette majoration peut être améliorée, en effet on remarque que le bit de poids faible change à toutes les étapes, que le second bit change une fois sur deux. Ainsi on peut majorer le coût des n opérations par :

$$n + n/2 + n/4 + \dots \leq 2n$$

Et ce, quelle que soit la valeur de k .

3. Comparaison copiée du site de Wikipedia https://fr.wikipedia.org/wiki/Analyse_amortie

3.9 Travaux dirigés III

3.9.1 Tables de hachage

Partie I

Considérer la fonction de hachage $h(c) = c \bmod 13$ et une table de hachage avec $m = 13$ adresses.

1. Insérer les clés 26,37,24,30, et 11 dans la table de hachage ci-dessus en utilisant la résolution des collisions par adressage ouvert et sondage linéaire avec la fonction $h(c, i) = (h(c) + i) \bmod m$.
2. Rajouter maintenant les clés 1,2,3,4,5,6,7,8,9,10. Quel problème rencontrez-vous ? Quelles solutions proposez-vous ?

Partie II

Cette exercice a pour but de se familiariser avec quelques fonctions de hachage en adressage ouvert. La taille de la table de hachage est notée m . Si p et q sont deux entiers naturels, $p \bmod q$ est le reste de la division euclidienne de p par q . On considère les fonctions de hachage

$$h_1(k) = k \bmod m, h_2(k) = k \bmod (m - 1).$$

1. Insérer successivement les clés 11, 22, 31, 4, 15, 28, 17, 88 et 59 dans une table de hachage de taille $m = 11$ générée en adressage ouvert en utilisant comme fonction de hachage principale $h(k, i) = (h_1(k) + i) \bmod m$.
2. Même question lorsque $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$.

3.9.2 Arbres binaires de recherche

1. Déssiner des arbres binaires de recherche de hauteur 2, 3, 4, 5, 6 pour le même ensemble de clés

$$\{1, 4, 5, 10, 16, 17, 21\}.$$

2. Quelle est la différence entre la structure des arbres binaires de recherche et la structure de tas ? Est-ce que la propriété de tas-min permet d'afficher en ordre trié et en temps $O(n)$ les clés d'un arbre de n noeuds ? Justifier la réponse.

3.9.3 Opérations sur les arbres binaires de recherche

Démontrer que si un noeud dans un arbre binaire de recherche a deux fils, alors son successeur n'a pas de fils gauche, et son prédécesseur n'a pas de fils droit.

3.9.4 Arbres AVL

Les arbres équilibrés AVL sont des arbres binaires tels que pour tout noeud de l'arbre, la différence entre les profondeurs du sous-arbre gauche et du sous-arbre droit est d'au plus 1. On suppose de plus que chaque noeud connaît la différence entre les profondeurs de ses sous-arbres droit et gauche (on rajoute un champ $n.\delta$

aux noeuds qui contient cette différence, et qui sera mis à jour lorsque l'on modifie l'arbre).

1. Démontrer que la hauteur d'un AVL est en $\theta(\log(n))$.
2. Proposer un algorithme d'insertion d'une clé dans un AVL en $O(\log(n))$ dans le pire des cas.
3. Proposer un algorithme de suppression d'une clé dans un AVL en $O(\log(n))$ dans le pire des cas.
4. Quelle la complexité finale de chacune des 7 opérations sur un AVL dans le pire des cas ?

Chapitre 4

Conception avancée

4.1 Programmation dynamique [Cormen et al., 1994, Cormen et al., 2001, Papadimitriou et al., 2006]

La programmation dynamique, comme la méthode diviser pour régner, résoud les problèmes en combinant les solutions de sous-problèmes (programmation dans ce contexte, fait référence à une méthode tabulaire, et non à l'écriture de code informatique.). Les algorithmes diviser pour régner partitionnent le problème en sous-problèmes indépendants, qu'ils résolvent récursivement, puis combinent leurs solutions pour résoudre le problème initial. La programmation dynamique est applicable lorsque les sous-problèmes ont en commun des sous-sous-problèmes. Dans ce cas l'algorithme diviser pour régner fait plus de travail que nécessaire, en résolvant plusieurs fois le sous-sous-problème commun. Un algorithme de programmation dynamique résoud chaque sous-sous-problème une seule fois, et mémorise sa solution dans un tableau, épargnant ainsi le recalcul de la solution chaque fois que le sous-sous-problème est rencontré. La programmation dynamique est en général appliquée aux problèmes d'optimisation.

La programmation dynamique s'appuie sur un principe simple, appelé le principe d'optimalité de Bellman : **toute solution optimale s'appuie elle-même sur des sous-problèmes résolus localement de façon optimale. Un problème possède une sous structure optimale si toute solution optimale contient la solution optimale aux sous-problèmes.**

La développement d'un algorithme de programmation dynamique peut être planifié dans une séquence de quatre étapes :

- **Approche initiale "diviser pour régner"**
- Caractériser la structure d'une solution optimale : **établir l'idée.**
- **Définir récursivement la valeur d'une solution optimale.**
- **Exploiter l'interaction entre les sous-problèmes**
- Calculer la valeur d'une solution optimale **en remontant progressivement jusqu'à l'énoncé du problème initial** (algorithme peut rester récursif).
- Construire une solution optimale pour les informations calculées **et stockées.**

D'un point de vue pratique¹ : La programmation dynamique est similaire à la méthode diviser et régner (voir la Figure 4.1). Une solution d'un problème dépend des solutions précédentes obtenues des sous-problèmes. La différence significative réside dans deux points :

Sous-problèmes redondants La programmation dynamique **permet aux sous-problèmes de se superposer**. Autrement dit, un sous-problème peut être **utilisé dans la solution de deux sous-problèmes différents**. Tandis que l'approche diviser et régner crée des sous-problèmes qui sont complètement séparés et peuvent être résolus **indépendamment l'un de l'autre**. Dans la figure 4.1, le problème à résoudre est à la racine, et les descendants sont les sous-problèmes, plus faciles à résoudre. Les feuilles de ce graphe constituent des sous-problèmes dont la résolution est triviale. Dans la programmation dynamique, ces feuilles constituent souvent les données de l'algorithme. La différence fondamentale entre ces deux méthodes devient alors claire : les **sous-problèmes dans la programmation dynamique peuvent être en interaction, alors dans la méthode diviser et régner, ils ne le sont pas**.

Sous-problèmes résolus de bas en haut La méthode diviser et régner est récursive, les calculs se font de haut en bas. Même si une solution par "programmation dynamique" est conçue d'une façon récursive, son **raisonnement est plutôt de bas en haut, car elle exploite l'interaction entre les sous-problèmes dans des branches différentes** : on commence par résoudre les plus petits sous-problèmes. En combinant leur solution, on obtient les solutions des sous-problèmes de plus en plus grands.

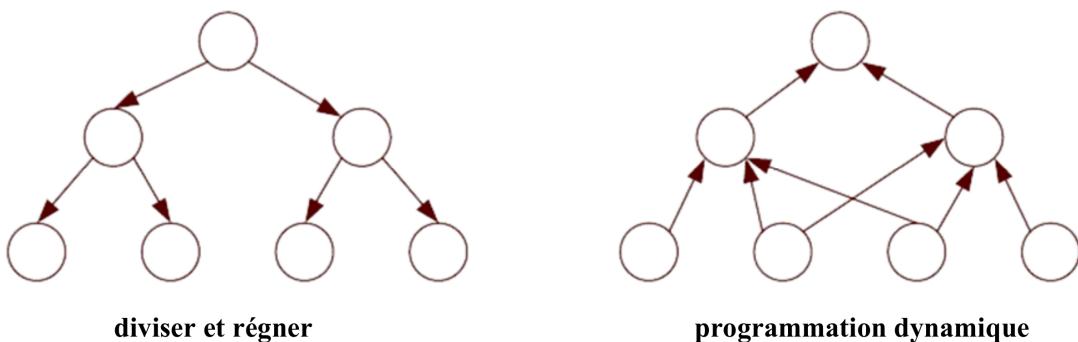


FIGURE 4.1 – Programmation-dynamique vs Diviser-pour-reigner (<http://www.uqac.ca/rebaine/8INF806/8INF806INDEX.htm>)

4.1.1 Multiplication d'une suite de matrices

Notre premier exemple est un algorithme qui résoud le problème de la multiplication d'une suite de matrices. On suppose qu'on a une suite (A_1, \dots, A_n) de n

1. Paragraphe basé sur des extraits de <http://www.uqac.ca/rebaine/8INF806/8INF806INDEX.htm>

matrices à multiplier, et qu'on souhaite calculer le produit

$$\mathbf{A_1 A_2 \dots A_n}.$$

La multiplication de matrices est associative, et tous les parenthésages aboutissent donc à la même valeur du produit. Par exemple, si la suite de matrices est (A_1, \dots, A_4) , le produit peut être complètement parenthésé de cinq façons distinctes :

$$(A_1(A_2(A_3A_4)))$$

...

La manière dont une suite de matrices est parenthésée peut avoir un impact crucial sur le coût d'évaluation du produit. Primo, si A est une matrice $p \times q$ et B une matrice $q \times r$, la matrice résultante C est une matrice $q \times r$. Le temps de calcul de C est dominé par **le nombre de multiplications qui vaut pqr** .

MULTIPLIERMATRICES(A, B)

```

MATRIX-MULTIPLY( $A, B$ )
1  if  $columns[A] \neq rows[B]$ 
2    then error "incompatible dimensions"
3    else for  $i \leftarrow 1$  to  $rows[A]$ 
4      do for  $j \leftarrow 1$  to  $columns[B]$ 
5        do  $C[i, j] \leftarrow 0$ 
6        for  $k \leftarrow 1$  to  $columns[A]$ 
7          do  $C[i, j] \leftarrow C[i, j] + A[i, k] \cdot B[k, j]$ 
8    return  $C$ 

```

FIGURE 4.2 – Multiplication de deux matrices (scan de [Cormen et al., 2001])

Pour illustrer les différents coûts par différents parenthésages, on considère le problème d'une suite (A_1, A_2, A_3) de trois matrices, de dimensions respectives 10×100 , 100×5 , 5×50 . $((A_1A_2)A_3)$ nécessite 7500 multiplications. $A_1(A_2A_3)$ nécessite 75 000 multiplications. **Le premier parenthésage est 10 fois plus rapide !**

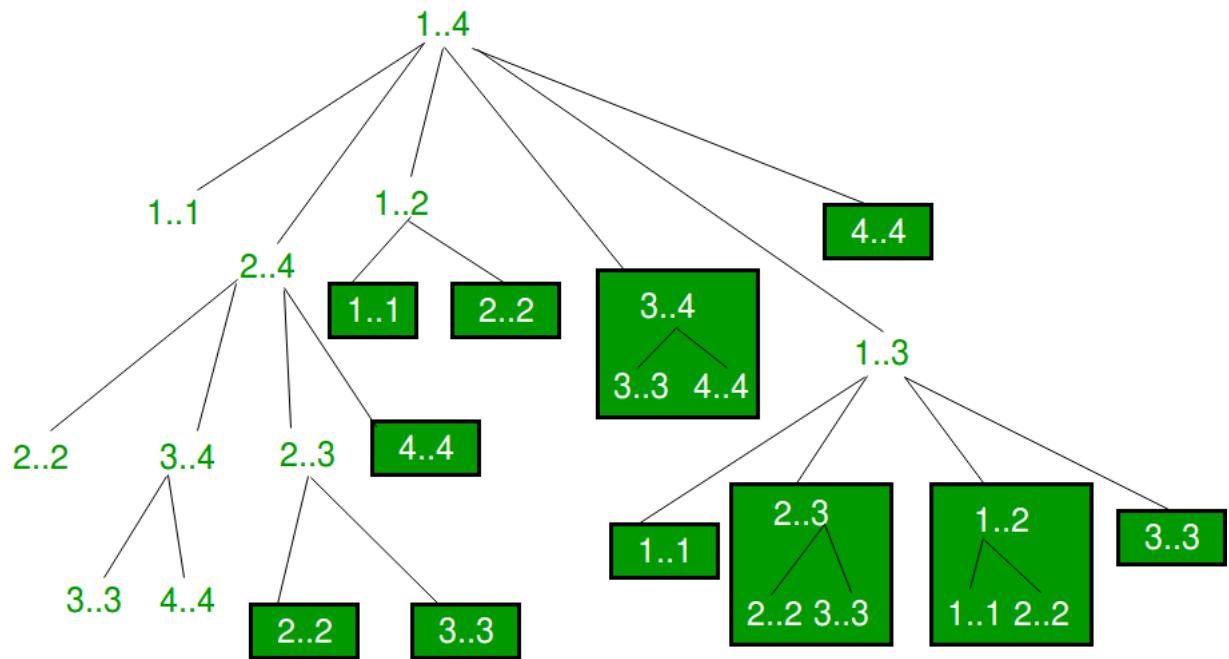


FIGURE 4.3 – Arbre des parenthésages et des redondances de $A_1 \times A_2 \times A_3 \times A_4$ (extrait du site <https://www.geeksforgeeks.org/matrix-chain-multiplication-dp-8/>)

Le nombre de parenthésage $P(n)$ est donné par la formule récurrente :

$$P(n) = \begin{cases} 1 & \text{si } n = 1, \\ \sum_{k=1..n-1} P(k)P(n-k) & \text{si } n \geq 2. \end{cases}$$

Une résolution de cette récurrence est la suite des nombres de Catalan :

$$P(n) = C(n-1),$$

où

$$C(n) = \frac{1}{n+1} C_{2n}^n$$

$$C(n) = \Omega(4^n/n^{3/2}).$$

Le nombre de solutions est donc exponentiel en n . La méthode directe consistant à effectuer une recherche exhaustive est donc une stratégie médiocre pour déterminer le parenthésage optimal d'une suite de matrices.

Structure d'un parenthésage optimal Soit $A_{i..j} = A_i A_{i+1} \dots A_j$. Un parenthésage optimal du produit $A_{1..n}$ sépare le produit entre $A_{1..k}$ et $A_{k+1..n}$ **pour un certain $k \in 1..n - 1$** . Il en résulte aussi que le parenthésage de $A_{1..k}$ et $A_{k+1..n}$ soient aussi optimal, sinon le parenthésage en k ne serait plus optimal. **La sous-structure optimal à l'intérieur de la solution optimale est l'une des garanties de l'applicabilité de la programmation dynamique.**

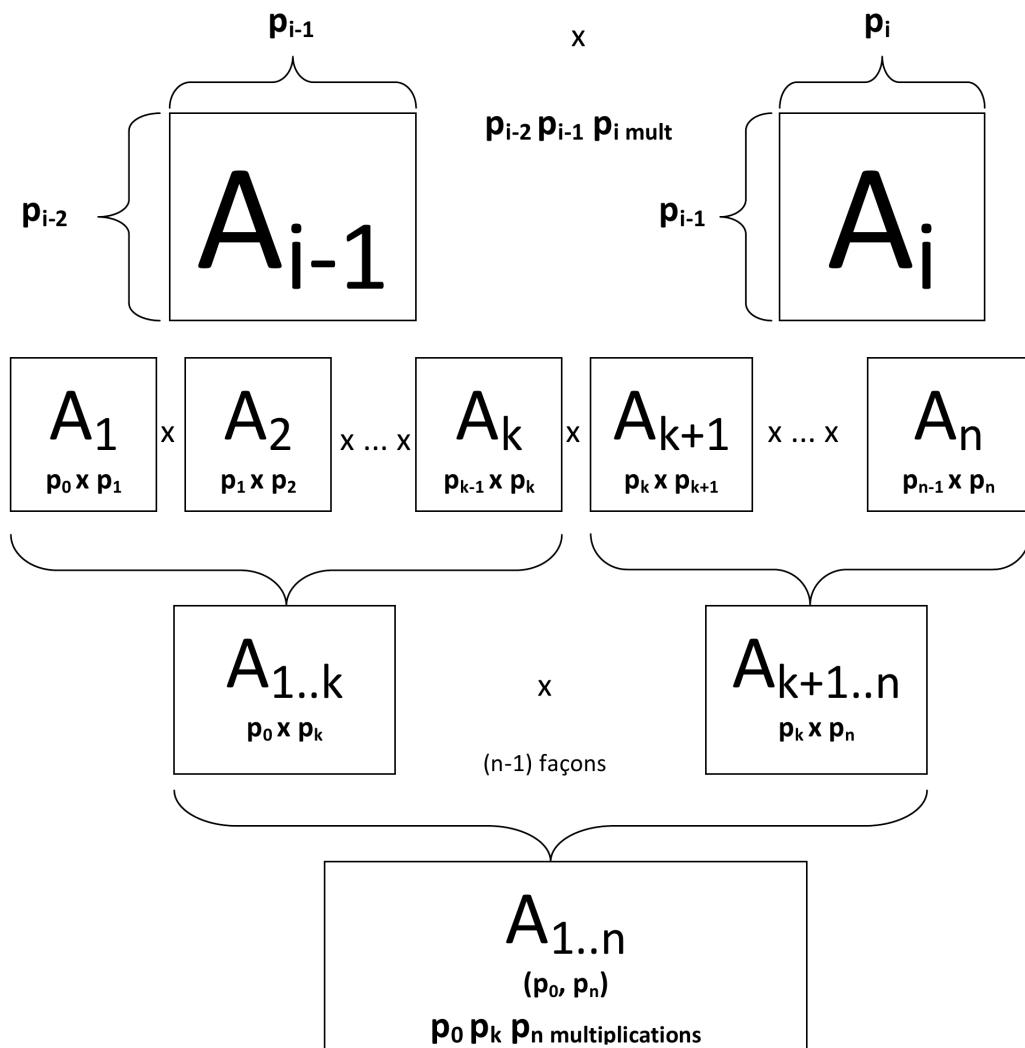


FIGURE 4.4 – Séquence de multiplications

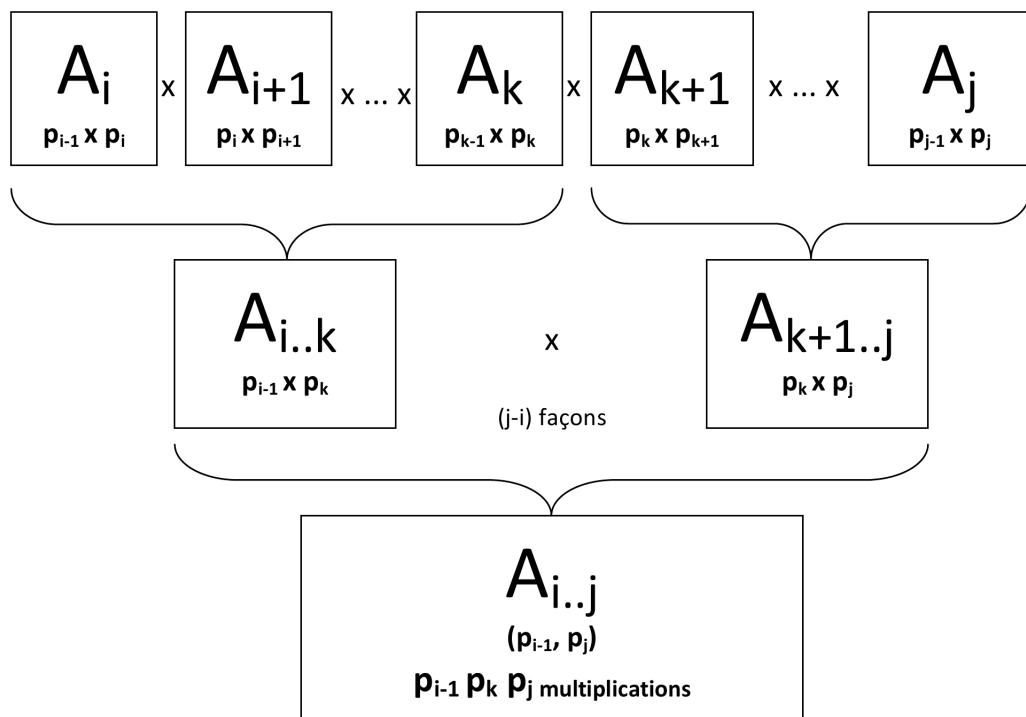


FIGURE 4.5 – Séquence de multiplications

Solution récursive La deuxième étape du paradigme de la programmation dynamique consiste à définir récursivement la valeur de la solution optimale, en fonction des solutions optimales aux sous-problèmes. Ici, on prend comme sous-problèmes les problèmes consistant à déterminer le coût minimum d'un parenthésage de $A_i \dots A_j$. Soit $m[i, j]$ le nombre minimum de multiplications nécessaire pour le calcul de la matrice $A_{i..j}$. La matrice A_i est de dimension $p_{i-1} \times p_i$. Pour $A_{1..n}$, ce sera $m[1, n]$.

$$m[i, j] = \begin{cases} 0 & \text{si } i = j, \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j\} & \text{si } i < j. \end{cases}$$

Pour disposer de l'indice de la solution optimale, appelons $s[i, j]$ une valeur de k , l'indice auquel on peut séparer le produit $A_{i..j}$ pour obtenir un parenthésage optimal.

Calcul des coûts optimaux Ici, on écrit un algorithme récursif, basé sur la solution réursive donnée précédemment.

L'algorithme RECURSIVE-MATRIX-CHAIN est de complexité $\Omega(2^n)$.

```

RECURSIVE-MATRIX-CHAIN( $p, i, j$ )
1   if  $i = j$ 
2     then return 0
3    $m[i, j] \leftarrow \infty$ 
4   for  $k \leftarrow i$  to  $j - 1$ 
5     do  $q \leftarrow \text{RECURSIVE-MATRIX-CHAIN}(p, i, k)$ 
         +  $\text{RECURSIVE-MATRIX-CHAIN}(p, k + 1, j)$ 
         +  $p_{i-1} p_k p_j$ 
6       if  $q < m[i, j]$ 
7         then  $m[i, j] \leftarrow q$ 
8   return  $m[i, j]$ 
```

FIGURE 4.6 – Solution réursive exponentielle (scan de [Cormen et al., 2001])

L'observation importante est que le nombre de sous-problèmes est assez réduit : un problème pour chaque choix de i et j satisfaisant $1 \leq i \leq j \leq n$, soit au total $C_n^2 + n = \Theta(n^2)$. L'algorithme récursif RECURSIVE-MATRIX-CHAIN peut rencontrer chaque sous-problème de nombreuses fois dans différentes branches.

Recensement Le principe ici est de recenser les actions naturelles, mais inefficaces, de l'algorithme récursif. On conserve dans un tableau des solutions aux sous-problèmes, mais la structure de remplissage du tableau est plus proche de l'algorithme récursif. Un algorithme récursif maintient à jour un élément du tableau pour la solution de chaque sous-problème. Chaque élément contient une valeur spéciale pour indiquer qu'il n'a pas été rempli. **Lorsque le sous-problème est rencontré pour la première fois, sa solution est calculée, puis stockée dans le tableau. A chaque confrontation avec ce sous-problème, la valeur stockée dans le tableau est simplement récupérée et retournée.**

```
MEMOIZED-MATRIX-CHAIN( $p$ )
1  $n \leftarrow \text{length}[p] - 1$ 
2 for  $i \leftarrow 1$  to  $n$ 
3   do for  $j \leftarrow i$  to  $n$ 
4     do  $m[i, j] \leftarrow \infty$ 
5 return LOOKUP-CHAIN( $p, 1, n$ )
```

```
LOOKUP-CHAIN( $p, i, j$ )
1 if  $m[i, j] < \infty$ 
2   then return  $m[i, j]$ 
3 if  $i = j$ 
4   then  $m[i, j] \leftarrow 0$ 
5 else for  $k \leftarrow i$  to  $j - 1$ 
6   do  $q \leftarrow \text{LOOKUP-CHAIN}(p, i, k)$ 
      +  $\text{LOOKUP-CHAIN}(p, k + 1, j) + p_{i-1} p_k p_j$ 
7   if  $q < m[i, j]$ 
8     then  $m[i, j] \leftarrow q$ 
9 return  $m[i, j]$ 
```

FIGURE 4.7 – Solution récursive polynomiale par programmation dynamique (scan de [Cormen et al., 2001])

On peut démontrer que l'algorithme MEMOIZED-MATRIX-CHAIN est en $O(n^3)$. Son espace de stockage est en $\Theta(n^2)$.

Il existe seulement n^2 sous-problèmes différents (chacun nécessitant au plus $O(n)$ comparaisons), et notre algorithme les résoud tous une seule fois. Notons qu'une solution de type "programmation dynamique" serait efficace si le nombre de sous-problème est réduit.

4.1.2 Plus longue sous-séquence commune [Cormen et al., 2001]

Soit l'exemple (pris de [Cormen et al., 2001]) de comparaison entre de deux séquences d'ADN

$$S_1 = ACCGGTCTGAGTGCGCCGAAGCCGGCCGAA$$

et

$$S_2 = GTCGTTCTGGAAATGCCGTTGCTCTGTAAA$$

La plus longue sous-séquence commune entre S_1 et S_2 permet d'évaluer à combien S_1 est similaire à S_2 .

Soit une séquence $X = \langle x_1, \dots, x_m \rangle$. Une séquence $Z = \langle z_1, \dots, z_k \rangle$ est dite une sous-séquence de X s'il existe une séquence strictement croissante $\langle i_1, \dots, i_k \rangle$ d'indices de X tels que $\forall j \in 1..k, x_{i_j} = z_j$. Par exemple, $Z = \langle B, C, D, B \rangle$ est une sous-séquence de $X = \langle A, B, C, B, D, A, B \rangle$, correspondant à la séquence d'indices $\langle 2, 3, 5, 7 \rangle$.

Etant données deux séquences X et Y , on dit qu'une séquence Z est une sous-séquence commune de X et Y , si Z est une sous-séquence de X et de Y . Par exemple, si $X = \langle A, B, C, B, D, A, B \rangle$ et $Y = \langle B, D, C, A, B, A \rangle$, la séquence $\langle B, C, A \rangle$ est une sous-séquence commune de X et de Y . Alors que, la séquence $\langle B, C, A \rangle$ n'est pas une plus longue sous-séquence commune (PLSC) de X et Y , puisqu'elle est de longueur 3 et que la séquence $\langle B, C, B, A \rangle$, qui est aussi commune à X et Y , est de longueur 4. La séquence $\langle B, C, B, A \rangle$ est une PLSC de X et Y , de même que la séquence $\langle B, D, A, B \rangle$, puisqu'il n'existe pas de sous-séquence commune de longueur supérieure ou égale à 5.

Dans le problème de la plus longue sous-séquence commune, on dispose au départ de deux séquences $X = \langle x_1, \dots, x_m \rangle$ et $Y = \langle y_1, \dots, y_n \rangle$. On souhaite trouver une sous-séquence commune à X et Y de longueur maximale.

Proposition 12 (Sous-structure optimale d'une PLSC)). *Soient deux séquences $X = \langle x_1, \dots, x_m \rangle$, et $Y = \langle y_1, \dots, y_n \rangle$, et soit $Z = \langle z_1, \dots, z_k \rangle$ une PLSC quelconque de X et Y .*

1. Si $x_m = y_n$, alors $z_k = x_m = y_n$ et Z_{k-1} est une PLSC de X_{m-1} et Y_{n-1} .
2. Si $x_m \neq y_n$, alors $z_k \neq x_m$ implique que Z est une PLSC de X_{m-1} et Y .
3. Si $x_m \neq y_n$, alors $z_k \neq y_n$ implique que Z est une PLSC de X et Y_{n-1} .

Preuve : voir [Cormen et al., 2001].

Soit la mise en oeuvre de cette stratégie divisor-pour-régnier proposée dans la proposition 12 donnant lieu à l'algorithme 21.

Algorithm 21 PLSCDIV(m, n)

```

1: if ( $m = 0$ )  $\vee$  ( $n = 0$ ) then
2:   return ""
3: else if ( $x_m = y_n$ ) then
4:   return concat(PLSCDIV( $m - 1, n - 1$ ),  $x_m$ )
5: else
6:    $P1 \leftarrow$  PLSCDIV( $m - 1, n$ )
7:    $P2 \leftarrow$  PLSCDIV( $m, n - 1$ )
8:   if ( $|P1| > |P2|$ ) then
9:     return  $P1$ 
10:  else
11:    return  $P2$ 
12:  end if
13: end if
```

Soit T_{n+m} la complexité de PLSCDIV dans le pire cas ($x_m \neq y_n$).

$$T_{n+m} = 2T_{n+m-1} + 1$$

et

$$T_0 = 1$$

Soit $T_{n+m} = S_{n+m} - 1$. On a

$$S_{n+m} = 2S_{n+m-1}$$

$$S_0 = 2$$

Une suite géométrique, d'où :

$$S_{n+m} = 2^{n+m+1}$$

D'où finalement

$$T_{n+m} = 2^{n+m+1} - 1$$

La complexité de PLSCDIV est exponentielle.

En fait, l'origine de l'exponentielle est le fait que PLSCDIV est appelé et rappelé un nombre exponentiel de fois. L'idée est de stocker les calculs dans une matrice c .

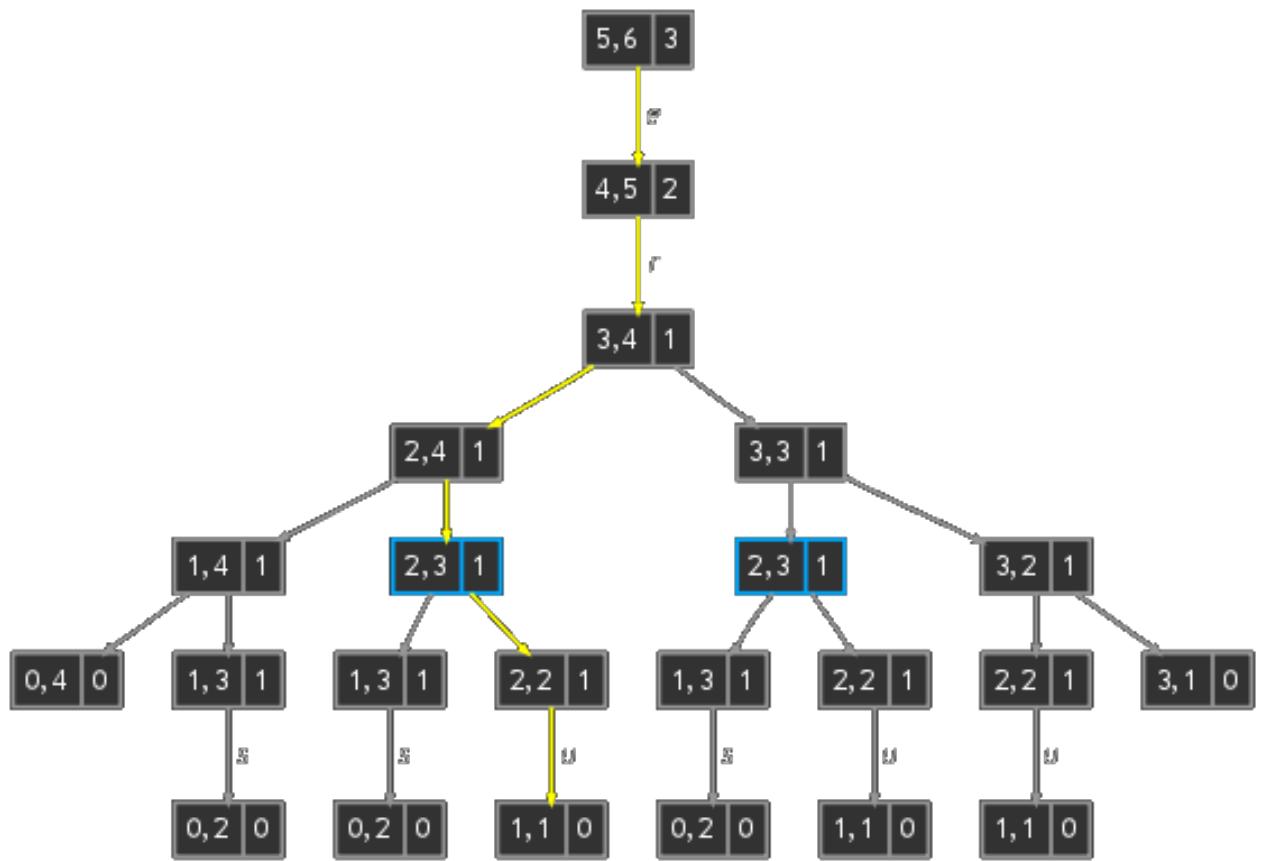


FIGURE 4.8 – Arbre des appels récursifs sur les chaines "sucre" et "lustre" (<https://zanotti.univ-tln.fr/ALGO/I51/PLSC.html>)

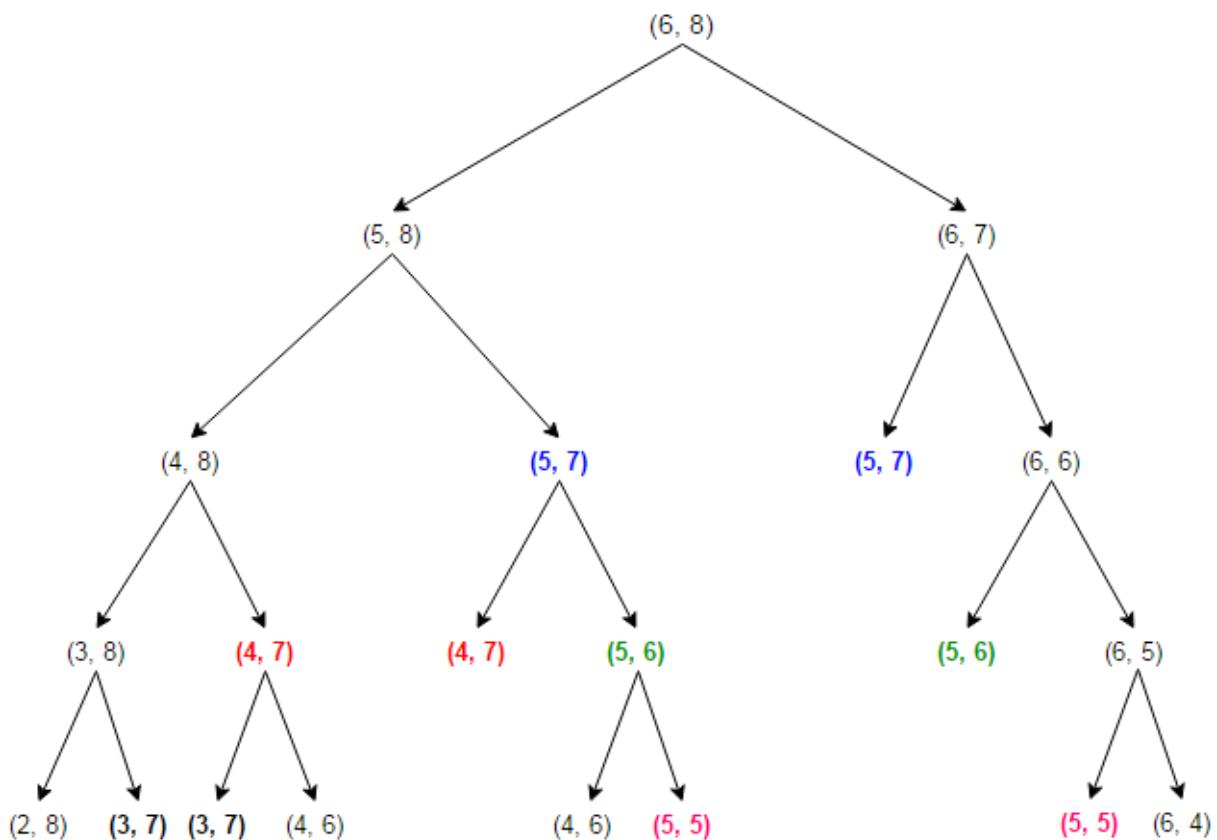


FIGURE 4.9 – Arbre des appels récursifs sur les chaînes "ABCBDAB" et "BDCABA" (<https://www.techiedelight.com/fr/longest-common-subsequence/>)

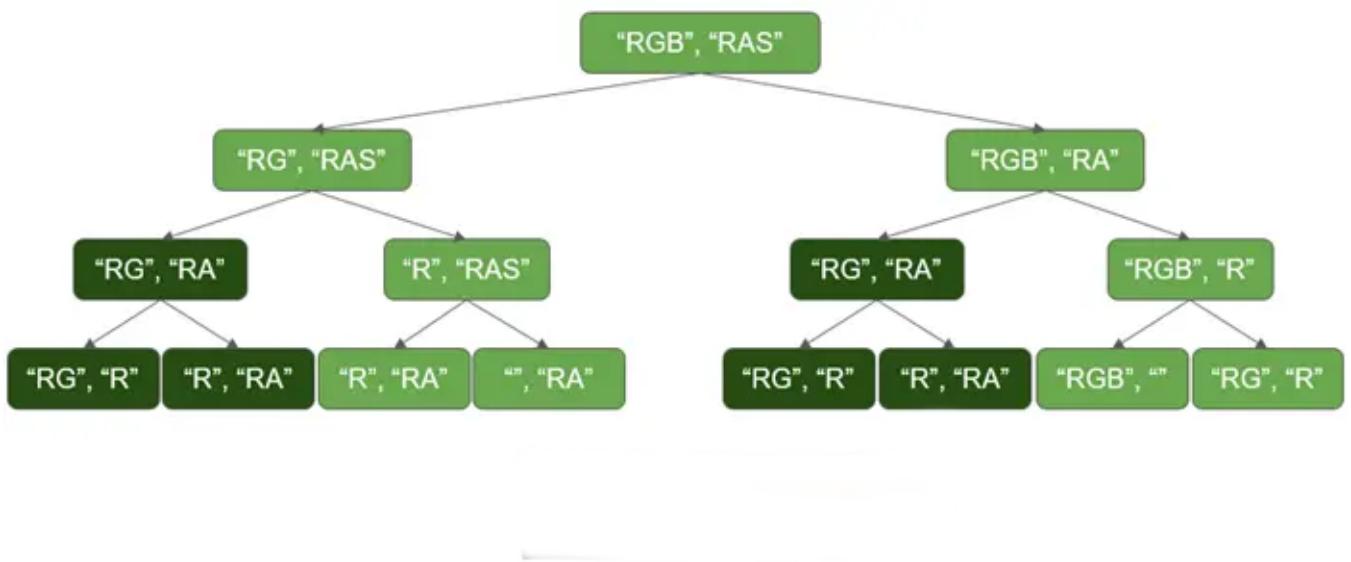


FIGURE 4.10 – Arbre des appels récursifs (<https://www.guru99.com/fr/longest-common-subsequence.html>)

Soit $c[i, j]$ la longueur d'une PLSC des séquences $X_i = < x_1, \dots, x_i >$ et $Y_j = < y_1, \dots, y_j >$. A partir de la proposition précédente, on peut établir la formule de la sous-structure optimale du problème de la PLSC :

$$c[i, j] = \begin{cases} 0 & \text{si } i=0 \text{ ou } j=0, \\ c[i-1, j-1] + 1 & \text{si } i,j>0, \text{ et } x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]) & \text{si } i,j>0, \text{ et } x_i \neq y_j. \end{cases}$$

Soit la reformulation de l'algorithme pour stocker les calculs (i.e., sans recalculer à chaque fois) :

Algorithm 22 PLSCDYN(m, n)

```

1: for  $i \in 1..m$  do
2:   for  $j \in 1..n$  do
3:      $c[i, j] \leftarrow -1$ 
4:   end for
5: end for
6: return PLSCDYN2( $m, n$ ) % voir l'algorithme 23

```

Algorithm 23 PLSCDYN2(m, n)

```

1: if  $c[m, n] > -1$  then
2:   return  $c[m, n]$ 
3: end if
4: if ( $m = 0$ )  $\vee$  ( $n = 0$ ) then
5:    $c[m, n] \leftarrow 0$ 
6: else if ( $x_m = y_n$ ) then
7:   PLSCDIV( $m - 1, n - 1$ )
8:    $c[m, n] \leftarrow c[m - 1, n - 1] + 1$ 
9:    $b[m, n] \leftarrow "↖"$ 
10: else
11:   PLSCDYN2( $m - 1, n$ )
12:   PLSCDYN2( $m, n - 1$ )
13:   if ( $c[m - 1, n] > c[m, n - 1]$ ) then
14:      $c[m, n] \leftarrow c[m - 1, n]$ 
15:      $b[m, n] \leftarrow "↑"$ 
16:   else
17:      $c[m, n] \leftarrow c[m, n - 1]$ 
18:      $b[m, n] \leftarrow "←"$ 
19:   end if
20: end if
21: return  $c[m, n]$ 

```

Il est évident que PLSCDYN explore chaque case $c[i, j]$ une et une seule fois, d'où sa complexité $O(n \times m)$. En effet PLSCDYN est un algorithme qui adopte le paradigme de la programmation dynamique en faisant le sondage des calculs, mais sa

structure algorithmique récursive cache une stratégie de bas vers le haut, explicitée dans la Figure 4.11, qui calcule la matrice c des longueurs optimales.

Pour faire valoir la structure du-bas-vers-le-haut de l'algorithme PLSCDYN, l'algorithme de la Figure 4.11 exploite à la fois la formule de sous-structure optimale, la superposition des problèmes, en faisant le recensement dans la matrice c .

```

LONGUEUR-PLSC( $X, Y$ )
1    $m \leftarrow \text{longueur}[X]$ 
2    $n \leftarrow \text{longueur}[Y]$ 
3   pour  $i \leftarrow 1$  à  $m$ 
4     faire  $c[i, 0] \leftarrow 0$ 
5     pour  $j \leftarrow 0$  à  $n$ 
6       faire  $c[0, j] \leftarrow 0$ 
7     pour  $i \leftarrow 1$  à  $m$ 
8       faire pour  $j \leftarrow 1$  à  $n$ 
9         faire si  $x_i = y_j$ 
10            alors  $c[i, j] \leftarrow c[i - 1, j - 1] + 1$ 
11             $b[i, j] \leftarrow \ll \nwarrow \gg$ 
12            sinon si  $c[i - 1, j] \geq c[i, j - 1]$ 
13              alors  $c[i, j] \leftarrow c[i - 1, j]$ 
14               $b[i, j] \leftarrow \ll \uparrow \gg$ 
15            sinon  $c[i, j] \leftarrow c[i, j - 1]$ 
16               $b[i, j] \leftarrow \ll \leftarrow \gg$ 
17   retourner  $c$  et  $b$ 
```

FIGURE 4.11 – Calcul de la longueur d'une PLSC entre deux séquences X et Y (scan de [Cormen et al., 2001])

La Figure 4.12 illustre le fonctionnement de l'algorithme de calcul des longueurs PLSC entre deux instances de séquences.

j	0	1	2	3	4	5	6
i	y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0
1	A	0	↑	↑	↑	↖	↖
2	B	0	1	↖	↖	↑	↖
3	C	0	1	1	↖	↑	↑
4	B	0	1	1	2	↑	↖
5	D	0	1	2	2	↑	↑
6	A	0	1	2	2	3	4
7	B	0	1	2	2	3	4

Figure Les tableaux c et b calculés par LONGUEUR-PLSC sur les séquences $X = \langle A, B, C, B, D, A, B \rangle$ et $Y = \langle B, D, C, A, B, A \rangle$. Le carré à l'intersection de la ligne i et de la colonne j contient la valeur de $c[i, j]$ et la flèche appropriée pour la valeur de $b[i, j]$. La valeur 4 de $c[7, 6]$ (coin en bas à droite dans le tableau) est la longueur d'une PLSC $\langle B, C, B, A \rangle$ de X et Y . Pour $i, j > 0$, l'élément $c[i, j]$ ne dépend que de l'égalité éventuelle entre x_i et y_j et des valeurs des éléments $c[i - 1, j]$, $c[i, j - 1]$ et $c[i - 1, j - 1]$, qui sont calculés avant $c[i, j]$. Pour raccorder les éléments d'une PLSC, il suffit de suivre les flèches $b[i, j]$ à partir du coin inférieur droit ; le chemin est indiqué en gris. Chaque « ↖ » du chemin correspond à un élément (auréolé) pour lequel $x_i = y_j$ est un membre d'une PLSC.

FIGURE 4.12 – Illustration du calcul de la longueur d'une PLSC entre les deux séquences $X = \langle A, B, C, B, D, A, B \rangle$ et $Y = \langle B, D, C, A, B, A \rangle$ (scan de [Cormen et al., 2001])

La Figure 4.13 imprime la PLSC entre deux séquences X et Y .

```

IMPRIMER-PLSC( $b, X, i, j$ )
1  si  $i = 0$  ou  $j = 0$ 
2  alors retourner
3  si  $b[i, j] = \nwarrow$ 
4    alors IMPRIMER-PLSC( $b, X, i - 1, j - 1$ )
5      imprimer  $x_i$ 
6  sinon si  $b[i, j] = \uparrow$ 
7    alors IMPRIMER-PLSC( $b, X, i - 1, j$ )
8  sinon IMPRIMER-PLSC( $b, X, i, j - 1$ )

```

FIGURE 4.13 – Affichage de la PLSC entre deux séquences X et Y (scan de [Cormen et al., 2001])

4.1.3 Triangulation optimale de polygones

Micro-projet

4.2 Algorithmes gloutons [Cormen et al., 1994, Cormen et al., 2006, Papadimitriou et al., 2006]

Les algorithmes servant à résoudre les problèmes d'optimisation parcourrent en général une série d'étapes, au cours lesquelles ils sont confrontés à un ensemble d'options. Pour de nombreux problèmes d'optimisation, la programmation dynamique est une approche trop lourde pour déterminer les meilleures options : d'autres algorithmes, plus simples et efficaces y arriveront. **Un algorithme glouton fait toujours le choix qui semble meilleur sur le moment. Autrement dit, il fait un choix optimal localement dans l'espoir que ce choix mènera à la solution optimale globalement. Les algorithmes gloutons n'aboutissent pas toujours à des solutions optimales, mais ils y arrivent dans de nombreux cas.** Nous verrons des exemples d'application de ce type d'algorithmes. Typiquement, plusieurs algorithmes de graphes (e.g., Dijkstra, Kruskal, Prim, ...) sont des exemples typiques illustrant la démarche gloutonne. Puis, nous aborderons les matroïdes pour lesquelles un algorithme glouton produit toujours une solution optimale.

4.2.1 Le problème du choix d'activités

Soit un ensemble $S = \{1, 2, \dots, n\}$ de n activités concurrentes, qui souhaitent utiliser une ressource, comme une salle de cours, qui ne peut être utilisée que pour une activité à la fois. Chaque activité i possède un horaire de début d_i et un horaire de fin f_i , avec $d_i \leq f_i$. Si elle est choisie, l'activité i a lieu pendant l'intervalle de temps $[d_i, f_i]$. Les activités i et j sont compatibles si les intervalles $[d_i, f_i]$ et $[d_j, f_j]$ ne se superposent pas (i.e., $d_i \geq f_j$ ou $d_j \geq f_i$). **Le problème du choix d'activités est de choisir un ensemble le plus grand possible d'activités compatibles entre elles.**

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

FIGURE 4.14 – Solution gloutonne pour l'ordonnancement (scan de [Cormen et al., 2001])

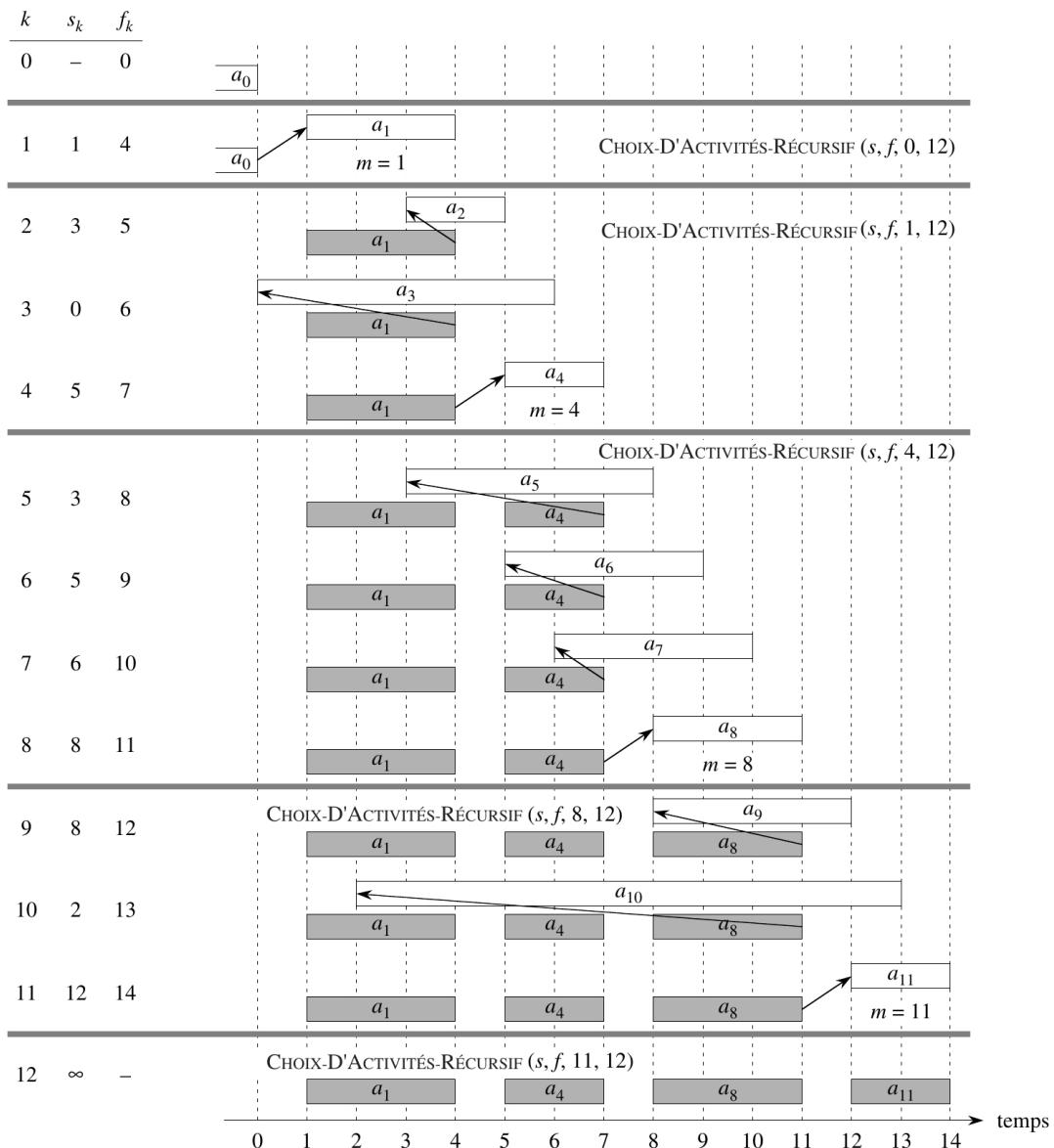
Un algorithme glouton résolvant le problème du choix d'activités est donné par le pseudo-code suivant. On suppose que les activités entrées sont triées par ordre d'horaire de fin croissant :

$$f_i \leq f_2 \leq \dots \leq f_n.$$

```
GREEDY-ACTIVITY-SELECTOR( $s, f$ )
1    $n \leftarrow \text{length}[s]$ 
2    $A \leftarrow \{a_1\}$ 
3    $i \leftarrow 1$ 
4   for  $m \leftarrow 2$  to  $n$ 
5       do if  $s_m \geq f_i$ 
6           then  $A \leftarrow A \cup \{a_m\}$ 
7            $i \leftarrow m$ 
8   return  $A$ 
```

FIGURE 4.15 – Solution gloutonne pour l'ordonnancement (scan de [Cormen et al., 2001])

La Figure 4.16 illustre le fonctionnement de cet algorithme glouton.



Fonctionnement de $\text{CHOIX-D'ACTIVITÉS-RÉCURSIF}$ sur les 11 activités précédemment données. Les activités traitées dans chaque appel récursif apparaissent entre des traits horizontaux. L'activité fictive a_0 finit à l'heure 0 et, dans l'appel initial $\text{CHOIX-D'ACTIVITÉS-RÉCURSIF}(s, f, 0, 11)$, l'activité a_1 est sélectionnée. Dans chaque appel récursif, les activités qui ont été déjà sélectionnées sont estompées, et l'activité en blanc est celle qui est en cours de traitement. Si l'heure de début d'une activité a lieu avant l'heure de fin de l'activité ajoutée en dernier (la flèche entre elles pointe vers la gauche), elle est refusée. Sinon (la flèche pointe directement vers le haut ou vers la droite), elle est sélectionnée. Le dernier appel récursif, $\text{CHOIX-D'ACTIVITÉS-RÉCURSIF}(s, f, 11, 11)$, retourne \emptyset . L'ensemble d'activités sélectionnées résultant est $\{a_1, a_4, a_8, a_{11}\}$.

FIGURE 4.16 – Choix glouton des activités (scan de [Cormen et al., 2001])

4.2.2 Eléments de la stratégie gloutonne

Un algorithme glouton détermine une solution optimale pour un problème après avoir effectué une série de choix. Pour chaque point de décision de l'algorithme, le choix qui semble le meilleur à cet instant est effectué. Cette stratégie heuristique ne produit pas toujours une solution optimale, mais comme nous l'avons vu dans le problème du choix d'activités, c'est parfois le cas.

Comment peut-on savoir si un algorithme glouton saura résoudre un problème d'optimisation particulier ? C'est en général impossible, mais il existe deux caractéristiques qu'un problème peut avoir, et qui se prêtent à une stratégie gloutonne : la propriété du choix glouton, et une sous-structure optimale.

Propriété du choix glouton On peut arriver à une solution globalement optimale en effectuant un choix localement optimal (glouton). C'est en cela que les algorithmes gloutons diffèrent de la programmation dynamique. En programmation dynamique, on fait un choix à chaque étape, mais ce choix dépend de la solution des sous-problèmes. Dans un algorithme glouton, on fait le choix qui semble le meilleur sur le moment, puis on résoud les sous-problèmes qui surviennent une fois que le choix est fait. Ainsi, contrairement à la programmation dynamique, qui résout les sous-problèmes de manière ascendante, une stratégie gloutonne progresse de manière descendante, en faisant se succéder les choix gloutons, pour ramener itérativement chaque instance du problème à une instance plus petite.

Pour montrer qu'un algorithme glouton donné trouve la solution optimale, il faut démontrer qu'un choix glouton à chaque étape engendre une solution optimale globalement, et c'est là qu'un peu d'astuces peut s'avérer utile.

Sous-structure optimale Un problème fait apparaître une sous-structure optimale si une solution optimale du problème contient la solution optimale de sous-problèmes. Cette propriété est un indice important de l'applicabilité de la programmation dynamique comme des algorithmes gloutons.

4.2.3 Codages de Huffman

Les codages de Huffman constituent une technique largement utilisée et très efficace pour la compression de données ; des économies de 20% à 90% sont courantes. L'algorithme glouton de Huffman utilise un tableau contenant les fréquences d'apparition de chaque caractère pour établir une manière optimale de représenter chaque caractère par une chaîne binaire.

Soit un fichier de données de 100 000 caractères qu'on souhaite conserver de manière compacte. On observe que les caractères du fichier apparaissent avec la fréquence suivante :

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>
Fréquence	45	13	12	16	9	5
Mot de code de longueur fixe	000	001	010	011	100	101
Mot de code de longueur variable	0	101	100	111	1101	1100

Le codage de longueur fixe demande 300 000 bits. Le codage de longueur variable demande 224 000 bits. Cet encodage résulte en une économie de l'ordre de 25%.

Codages préfixes Ici, on ne considère que les codages où

aucun mot de code n'est aussi préfixe d'un autre mot du code.

Ce codage est simple : il suffit de concaténer les mots de code représentant chaque caractère du fichier. Le décodage est aussi très simple, car comme aucun mot de code n'est préfixe d'un autre, le mot de code qui commence un fichier encodé n'est pas ambigu. Par exemple, la chaîne 001011101 ne peut être interprétée que comme 0.0.101.1101, ce qui donne *aabe*. D'ailleurs, il est démontré que la compression de données maximale accessible à l'aide d'un codage de caractères peut toujours être obtenue avec un codage préfixe.

Un arbre binaire dont les feuilles sont les caractères donnés est bien adapté. On interprète le mot de code binaire pour un caractère comme le chemin allant de la racine à ce caractère, où 0 signifie "bifurquer vers le fils gauche", et 1 signifie "bifurquer vers le fils droit". Étant donné un arbre T correspondant à un codage préfixe, il suffit de calculer le nombre de bits nécessaire pour encoder un fichier. Pour chaque caractère c de l'alphabet C , soit $f(c)$ la fréquence de c , et soit $d_T(c)$ la profondeur de la feuille c dans l'arbre. On remarque que $d_T(c)$ est aussi la longueur du mot de code pour le caractère c . Le nombre de bits requis pour encoder un fichier vaut

$$B(T) = \sum_{c \in C} f(c)d_T(c),$$

ce qu'on définit comme le coût de l'arbre T .

Construction d'un codage de Huffman Huffman a inventé un algorithme glouton qui construit un codage préfixe optimal appelé codage de Huffman. L'algorithme construit du bas vers le haut l'arbre T correspondant au codage optimal. Il commence avec un ensemble de $|C|$ feuilles et effectue une série de $|C| - 1$ fusions pour créer l'arbre final. L'algorithme 24 montre les étapes du codage de Huffman. La Figure 4.17 illustre clairement le fonctionnement de l'algorithme de Huffman.

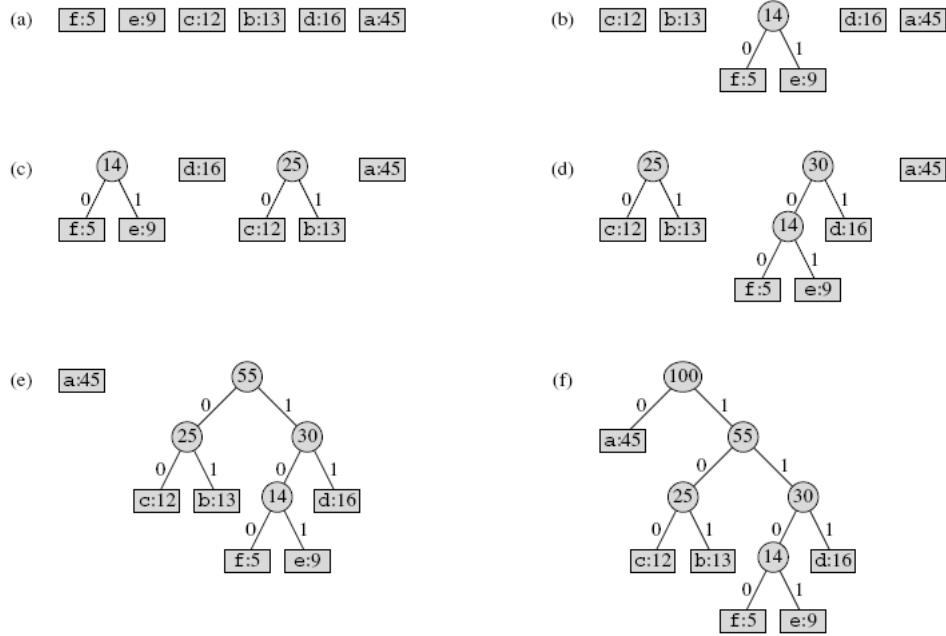


FIGURE 4.17 – Codage de Huffman (scan de [Cormen et al., 2001])

Algorithm 24 HUFFMAN(C)**Input:** C l'alphabet**Output:** Arbre binaire T du codage préfixe optimal

```

1:  $n \leftarrow |C|$ 
2:  $F \leftarrow C$ 
3: for  $i \leftarrow 1$  à  $n - 1$  do
4:    $z \leftarrow AllouerNoeud()$ 
5:    $x \leftarrow gauche[z] \leftarrow ExtraireMin(F)$ 
6:    $y \leftarrow droit[z] \leftarrow ExtraireMin(F)$ 
7:    $f[z] \leftarrow f[x] + f[y]$ 
8:    $Insrer(F, z)$ 
9: end for
10: return  $ExtraireMin(F)$ 
```

L'analyse du temps d'exécution de l'algorithme de Huffman suppose que F est implémenté comme un tas binaire. Pour un ensemble C de n caractères, l'initialisation de F à la ligne 2 peut s'effectuer en $O(n)$ à l'aide de la procédure CONSTRUIRETAS. La boucle **for** est exécuté en $|n| - 1$ fois, et comme chaque opération de tas en $O(\log n)$, la boucle contribue pour $O(n \log n)$ au temps d'exécution. Le temps global est donc $O(n \log n)$.

La puissance du codage de Huffman est exprimée dans la proposition suivante :

Proposition 13. *La procédure de Huffman produit un codage préfixe optimal.*

Preuve : voir [Cormen et al., 1994, Cormen et al., 2001].

4.3 Travaux dirigés IV

4.3.1 Algorithme glouton vs prog. dyn. : pièces de monnaies

² On considère le problème où l'on doit rendre la monnaie pour x euros avec le moins possible de pièces de monnaies.

1. On a des pièces de 1, 2, 5, 10. Ecrire l'algorithme glouton qui donne une solution optimale.
2. Généraliser la solution proposée sur un ensemble de pièces $c^0, c^1, c^2, \dots, c^k$ pour $c > 1$, et $k \geq 1$. L'algorithme est-il optimal ? Sinon, donner un contre-exemple !
3. Proposer une solution optimale de type diviser-pour-régner. Calculer sa complexité.
4. Peut-on l'améliorer par programmation dynamique ?

4.3.2 Algorithme glouton 2 : ordonnancement

³ Le problème de l'ordonnancement de tâches sur un seul processeur se compose :

- d'un ensemble de tâches $S = \{1..n\}$ de durée 1,
- avec des dates limites d_1, \dots, d_n : la tâche i doit finir avant la date d_i , sinon on doit payer la pénalité w_i .
- Problème : comment ordonner pour minimiser la somme des pénalités ?
- Définition : Un ensemble A de tâches est dit "indépendant" si il est possible de les ordonner tel que personne ne soit en retard.
- $N_t(A)$: nombre de tâches de l'ensemble A dont la date limite est $\leq t$.
- Montrer que les 3 propriétés suivantes sont équivalentes :
 - A est indépendant
 - Pour $t = 1..n$, on a $N_t(A) \leq t$.
 - Si les tâches de A sont ordonnancées dans l'ordre croissant de leur dates limites, alors aucune tâche n'est en retard.
- Déduire un algorithme glouton pour tester l'indépendance d'un ensemble A .
- Donner la complexité de l'algorithme proposé.

- Application numérique : soient les 7 tâches :
- | i | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|-------|---|---|---|---|---|---|---|
| d_i | 4 | 2 | 4 | 3 | 1 | 4 | 6 |
| w_i | 7 | 6 | 5 | 4 | 3 | 2 | 1 |

4.3.3 Programmation dynamique 1 : course

⁴ Vous participez à une course à bicyclette. Le long du parcours, vous retrouvez N points de ravitaillement (incluant les points de départ et d'arrivée). À chaque point de ravitaillement, vous pouvez louer une bicyclette que vous pouvez rendre à n'importe quel point de ravitaillement suivant du parcours. Or, il peut arriver qu'une location de i à j coûte plus cher qu'une série de locations plus courtes. Dans un tel

2. <http://www.lix.polytechnique.fr/~bournez/cours/CoursMaitrise/> de Olivier BOURNEZ

3. <http://www.lix.polytechnique.fr/~bournez/cours/CoursMaitrise/> de Olivier BOURNEZ

4. <http://www.polymtl.ca/etudes/cours/details.php?sigle=INF4705> de Gilles Pesant

cas, vous avez intérêt à rendre la bicyclette à un point de ravitaillement k entre i et j , et à louer une autre bicyclette pour continuer la course

1. En adoptant une approche de programmation dynamique, donnez une expression du coût minimal pour effectuer le trajet au complet (c'est-à-dire en franchissant les N points de ravitaillement).
2. Quel est le temps de calcul nécessaire pour évaluer ce coût minimal en fonction de N ?

4.3.4 Programmation dynamique 2 : arbres binaires

⁵ Si un arbre binaire de recherche (voir la figure 4.18) est utilisé pour de la vérification orthographique, avec les mots pour clés, il est avantageux de tenir compte des fréquences des mots dans la langue pour faire apparaître les plus fréquents vers la racine. Plus précisément, étant données n clés $x_1 < \dots < x_n$ et leurs fréquences f_1, \dots, f_n , il s'agit de construire un arbre binaire de recherche sur ces clés qui minimise le coût $f_1d_1 + \dots + f_nd_n$, où d_i représente la profondeur du nœud contenant la clé x_i (la profondeur de la racine étant 1).

Écrire le pseudo-code d'une procédure calculant par programmation dynamique le coût de l'arbre binaire optimal, et estimer sa complexité. [Indication : les clés appartenant à un même sous-arbre sont consécutives.]

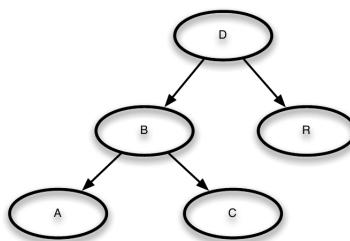


FIGURE 4.18 – Arbre binaire d'un texte

4.3.5 Programmation dynamique 3 : politique de vente

⁶ Une entreprise met en place une nouvelle politique de vente dans trois secteurs numérotés I, II et III. Elle peut embaucher au plus six représentants. Le directeur commercial prend la décision compte-tenu des bénéfices espérés et de la politique de développement définie par la direction générale. Les estimations des volumes des ventes dans chaque secteur, en fonction du nombre de représentants qui y sont affectés, sont précisées dans le tableau suivant :

Représentants	0	1	2	3	4	5	6
Secteur I	19	20	24	29	33	36	42
Secteur II	20	21	25	30	33	37	41
Secteur III	30	32	34	39	45	51	55

5. <https://www.enseignement.polytechnique.fr/informatique/INF431/X11-2012-2013/> de Benjamin Werner et François Pottier

6. <http://bdesgraupes.pagesperso-orange.fr/UPX/Master1/>

Donner les politiques optimales selon le nombre total de représentants embau-chés. Estimer sa complexité algorithme. En particulier, indiquer dans quels cas il y a deux répartitions équivalentes ?

4.3.6 Programmation dynamique 4 : sac-à-dos et PVC

1. Etant donné l'algorithme classique de programmation dynamique dédié au problème du sac-à-dos.

Rappel. L'algorithme évalue $P_{i,j}$, le gain max des i premiers objets dont la somme des poids est j .

$$P_{i,j} = 0 \text{ if } i = 0 \vee j = 0$$

$$P_{i,j} = P_{i-1,j} \text{ if } j - w_i < 0 \vee i > 0$$

$$P_{i,j} = \max\{P_{i-1,j}, P_{i-1,j-w_i} + v_i\}$$

où w_i est le poids de l'objet i , et v_i est son gain.

Evaluer sa complexité algorithme.

2. Etant donné l'algorithme classique de programmation dynamique dédié au problème du voyageur de commerce.

Rappel. L'algorithme évalue $D[i, S]$, la distance d'un plus court chemin partant de i , passant par tous les points de S , une et une seule fois, et se terminant au sommet 1.

$$D[i, S] = \min_{j \in S} \{d_{i,j} + D[j, S - \{j\}]\}$$

où $d_{i,j}$ est la distance entre i et j . La solution optimale est donnée par $D[1, V - 1]$

Evaluer sa complexité algorithme.

Chapitre 5

Eléments sur la théorie de la complexité

5.1 Problèmes faciles et problèmes difficiles [Prins, 1994]

5.1.1 Problèmes faciles

Exploration d'un graphe Donnée : Un graphe orienté $G = (X, U)$, deux sommets s et t de X . Question : Existe-t-il un chemin de s à t ? Algorithme en $O(M)$.

Chemin de coût minimal Données : $G = (X, U, C)$ un graphe orienté valué, et deux sommets s et t de X . Question : Trouver un chemin de coût minimal de s à t . Algorithme de Bellman en $O(NM)$. Algorithme de Dijkstra en $O(N^2)$.

Flot maximal Données : Un réseau de transport $G = (X, U, C, s, t)$. Question : Maximiser le débit du flot qui peut s'écouler dans le réseau entre s et t . Algorithme de Ford-Fulkerson en $O(NM^2)$.

Arbre recouvrant de poids minimal Données : $G = (X, E, W)$ un graphe simple valué. Question : Trouver un arbre recouvrant de poids minimal. Algorithme de Prim en $O(N^2)$. Algorithme de Kruskal en $O(M \log N)$. Si G est orienté, on pourrait calculer une arborescence recouvrante en $O(MN)$ avec l'algorithme de Edmonds.

Couplage Données : Soit $G = (X, E)$ un graphe simple. Un couplage de G (matching) est un sous-ensemble d'arêtes tel que deux quelconques d'entre elles n'aient aucun sommet commun. Question : Trouver un couplage de cardinalité maximale.

Parcours eulériens et chinois Données : Un parcours eulérien passe une fois par chaque arc ou arête. Le problème d'existence est solvable en $O(M)$. Un parcours chinois visite au moins une fois chaque arête.

Test de planarité Données : Un graphe simple $G = (X, E)$. Question : G est-il planaire, c'est-à-dire dessinable dans le plan sans croisement d'arêtes ? Un algorithme en $O(M)$ dû à Hopcroft et Tarjan.

Test de bipartisme On peut démontrer qu'un graphe est biparti si et ne contient pas de cycle impair. Algorithme en $O(M)$.

Recherche d'une information parmi N Il s'agit de la recherche d'un élément dans un tableau de N éléments.

Tri de N nombre Solvable avec l'algorithme du tri par tas en $O(n \log n)$.

Programmation linéaire Il s'agit de résoudre le problème d'optimisation :

$$\begin{cases} \min c.x \\ A.x \leq b \\ x \in \mathbb{R}^n, x \geq 0 \end{cases}$$

L'algorithme du simplexe est performant en moyenne, mais exponentiel dans le pire des cas. Karmarkar a proposé en 1984 un algorithme polynomial en $O(n^{3.5}L)$ où L est le nombre de bits pour coder A, b et c .

5.1.2 Problèmes difficiles

Stable maximal Données : Un graphe simple $G = (X, E)$. Un sous-ensemble de sommets S est un ensemble stable si il n'y a pas d'arête entre deux sommets quelconques de S .

Transversal minimal Données : Un graphe simple $G = (X, E)$. Un sous-ensemble de sommets T est un Transversal si toute arête de G a au moins une extrémité dans T .

Clique maximale Données : Un graphe simple $G = (X, E)$. Un sous-ensemble de sommets Q est une clique si toute paire de sommets de Q est reliée par une arête. Q engendre donc un graphe complet.

Coloration minimale Données : Un graphe simple $G = (X, E)$. G est k -colorable si on peut colorer ses sommets avec k couleurs distinctes, sans que deux sommets voisins aient la même couleur. Le plus petit k pour lequel G est k -colorable est le nombre chromatique de G .

Problèmes hamiltoniens Données : $G = (X, U, C)$ un graphe orienté valué. Le problème d'existence d'un parcours hamiltonien dans un graphe G est difficile. Le problème du voyageur de commerce consiste à trouver un circuit ou un cycle hamiltonien, de coût minimal, dans un graphe valué complet.

Problème SAT Données : une formule clausale. Question : Peut-on affecter à chaque variable propositionnelle de façon à rendre toutes les clauses vraies.

Sac à dos en variables entières Il s'agit de résoudre le problème :

$$\begin{cases} \min c.x \\ a.x \leq b \\ x \text{ entier} \end{cases}$$

Bin packing On donne n objets de poids a_i et un nombre non limité de boîtes de capacité b . Le but est de répartir les objets en un nombre minimal de boîtes.

5.2 NP-complétude et les classes P et NP [Prins, 1994]

Certains problèmes d'optimisation combinatoire disposent d'algorithmes polynomiaux, tandis que d'autres n'en ont toujours pas. Existe-t-il réellement une classe de problèmes combinatoires pour lesquels on ne trouvera jamais d'algorithme polynomiaux, ou est ce que les problèmes difficiles ont en fait de tels algorithmes, mais non encore découverts ? On conjecture depuis longtemps l'existence d'une classe de problèmes intrinsèquement difficiles, car plusieurs problèmes difficiles (comme le problème du voyageur de commerce) résistent depuis plus de 50 ans à l'assaut des travaux de recherche : malgré ces efforts, aucun algorithme polynomial n'a été trouvé.

La théorie de la complexité a été développée dans les années 1970 pour répondre à cette question. Le principal résultat est que tous les problèmes difficiles sont liés : la découverte d'un algorithme polynomial pour un seul problème difficile permettrait de déduire des algorithmes polynomiaux pour tous les autres.

La théorie de la complexité ne traite que des problèmes d'existence, à réponse oui/non. Ceci n'est pas gênant pour les problèmes d'optimisation. Un algorithme efficace pour le problème d'existence peut être utilisé pour résoudre efficacement son problème d'optimisation associé, par une simple dichotomie sur les valeurs de la fonction objectif.

5.3 La classe P et NP [Prins, 1994, Cori et al., 2001]

Etant donné une fonction $f : \mathbb{N} \rightarrow \mathbb{N}$, on dira qu'un problème appartient à la classe $DTIME(f)$ s'il existe une machine déterministe qui sur tout entrée de longueur n , résout le problème en $O(f(n))$ pas de calcul.

L'ensemble des problèmes d'existence qui admettent des algorithmes polynomiaux forment la classe P. On peut poser $P = \bigcup_{k \geq 0} DTIME(n \mapsto n^k)$. Il faut pouvoir vérifier en temps polynomial une proposition de réponse Oui.

La classe NP est celles des problèmes d'existence dont une proposition de solution Oui est vérifiable polynomialement. On définit également la classe $NP = \bigcup_{k \geq 0} NTIME(n \mapsto n^k)$, où N signifie non-déterministe. Le modèle de calcul non-déterministe est enrichi par une instruction de "choix" où il existe une façon immédiate pour conduire à la solution.

Les problèmes qui ne sont pas dans NP existent, mais ne présentent qu'un intérêt théorique pour la plupart. NP inclut P.

Pour un problème sans algorithme efficace, il faut procéder comme suit pour prouver l'appartenance à NP :

1. proposer un codage de la solution (appelé certificat) ;
2. proposer un algorithme qui va vérifier la solution au vu des données du certificat ;
3. montrer que cet algorithme a une complexité polynomiale.

Considérons le problème suivant : **étant donné un ensemble S de n nombres entiers et un entier b, existe-t-il un sous-ensemble T de S dont la somme des éléments est égale à b ?** On ne connaît pas d'algorithme polynomial pour résoudre ce problème. Il n'empêche qu'il est dans NP, car **vérifier qu'une somme d'un ensemble d'entiers T, sous-ensemble d'un ensemble S de cardinalité n, est égale à b, est en $O(n^2)$** . Dans cette vérification, il faut s'assurer que tous les éléments de T sont dans S, qui nécessitera au plus n^2 tests.

5.4 Les problèmes NP-complets [Prins, 1994, Cori et al., 2000]

Il s'agit des problèmes les plus difficiles de NP, le "noyau dur". La notion de problème NP-complet est basée sur celle de transformation polynomiale d'un problème. Un problème d'existence P_1 se transforme polynomialement en un autre P_2 s'il existe un algorithme polynomial A transformant toute donnée pour P_1 en une pour P_2 , en conservant la réponse Oui et Non. Par exemple, un stable d'un graphe simple G est une clique dans le graphe complémentaire G_c .

Un problème NP-complet est un problème de NP en lequel se transforme polynomialement tout autre problème de NP. La classe des problèmes NP-complets est notée NPC. On rencontre dans la littérature le terme NP-difficile pour les problèmes d'optimisation : un problème d'optimisation combinatoire est NP-difficile si le problème d'existence associé est NP-complet.

La conséquence pratique forte de la classe NPC : si on trouverait un algorithme polynomial A pour un seul problème NP-complet X , on pourrait en déduire un autre pour tout autre problème difficile Y de P. Il suffit de transformer polynomialement les données de Y en données pour X , puis exécuter l'algorithme pour X .

Une question immédiate est de savoir si de tels problèmes existent réellement dans NP. Le logicien Cook (et Levin) a montré en 1970 que **le problème SAT est NP-complet**. Depuis cette date, les travaux de recherche ont montré que la plupart des problèmes d'existence associés aux problèmes difficiles sans algorithmes polynomiaux connus sont NP-complets.

Problème SAT

Données Soit un ensemble de variables $\{x_1, \dots, x_n\}$. Soit une formule logique F sous forme normale conjonctive $F = C_1 \wedge C_2 \wedge \dots \wedge C_l$, avec chaque clause $C_i = (y_{i,1} \vee y_{i,2} \vee \dots \vee y_{i,k_i})$, où chaque $y_{i,j}$ est un littéral, c'est-à-dire $y_{i,j} = x_i$ ou $y_{i,j} = \neg x_i$.

Résultat "oui"ssi F est satisfaisable, qu'il existe une affectation de valeurs de vérités aux variables qui rende F vraie.

Proposition 14. *Le problème SAT est NP-complet.*

Preuve : Premier problème démontré NP-complet. Preuve réalisée par Cook et Levin [Cook, 1971, Levin, 1973]. \square

5.5 Comment démontrer qu'un problème est NP-complet ? [Prins, 1994, Cori et al., 2001]

Etant donné un problème P , on passe au problème d'existence associé. Si P est un problème d'optimisation, on passe au problème d'existence associé en remplaçant la recherche d'une solution optimale par une de coût au plus k , k étant un entier qu'on ajoute aux données.

La technique qu'on utilise pour prouver qu'un problème X de NP est NP-complet consiste à **montrer qu'un problème NP-complet connu Y peut se transformer polynomialement en X (et pas le contraire, erreur fréquente !)**.

La NP-complétude d'un problème P s'obtient dans la quasi-totalité des cas de la façon suivante, dite technique de réduction :

1. $P \in NP$: on **vérifie en temps polynomial le certificat** accompagnant une instance I de P .
2. P est complet : on réduit à partir d'un problème NP-complet connu P' . On transforme une instance I_1 de P' en une instance I_2 de P :
 - (a) $\text{taille}(I_1) = P(\text{taille}(I_2))$
 - (b) " I_1 a une solution" \iff " I_2 a une solution"

5.5.1 3-SAT [Cori et al., 2001]

Problème 3-SAT

Données Soit un ensemble de variables $\{x_1, \dots, x_n\}$. Soit une formule logique F sous forme normale conjonctive $F = C_1 \wedge C_2 \wedge \dots \wedge C_p$, avec chaque clause $C_i = (y_{i,1} \vee y_{i,2} \vee y_{i,3})$, où chaque $y_{i,j}$ est un littéral, c'est-à-dire $y_{i,j} = x_i$ ou $y_{i,j} = \neg x_i$.

Résultat "oui" ssi F est satisfaisable, qu'il existe une affectation de valeurs de vérités aux variables qui rende F vraie.

Proposition 15. *3-SAT est NP-complet.*

Preuve :

Prouvons tout d'abord que $3\text{-SAT} \in NP$. Soit I une instance de 3-SAT. $\text{taille}(I) = O(n + 3p)$. Certificat : valeurs des x_i . Vérification en $O(p)$. D'où $3\text{-SAT} \in NP$.

Montrons maintenant que $3\text{-SAT} \in NPC$. Réduction à partir de SAT. Soit I_1 une instance de SAT.

1. — n variables x_1, \dots, x_n .
— p clauses C_1, \dots, C_p de longueurs k_1, \dots, k_p .
— $\text{taille}(I_1) = O(n + \sum_{i=1..p} k_i)$
2. Soit C_i une clause de P dans I_1 . Nous avons plusieurs cas suivant le nombre de littéraux dans C_i :
 - (a) 1 variables y : soit a_i et b_i deux nouvelles variables.

$$\begin{aligned} & y \vee a_i \vee b_i \\ \wedge \quad & y \vee \overline{a_i} \vee b_i \\ \wedge \quad & y \vee a_i \vee \overline{b_i} \\ \wedge \quad & y \vee \overline{a_i} \vee \overline{b_i} \end{aligned}$$

(b) 2 variables $y_1 \vee y_2$: soit c_i une nouvelle variable

$$\begin{aligned} & (y_1 \vee y_2 \vee c_i) \\ & \wedge (y_1 \vee y_2 \vee \overline{c_i}) \end{aligned}$$

(c) 3 variables : aucun changement

(d) k variables, $k > 3$, $C_i = y_1 \vee y_2 \vee \dots \vee y_k$

Soient $z_1^{(i)}, z_2^{(i)}, \dots, z_{k-2}^{(i)}$ nouvelles variables.

$$\begin{aligned} & (y_1 \vee y_2 \quad \textcolor{red}{\vee z_1^{(i)}}) \\ & \wedge (y_2 \quad \textcolor{red}{\vee z_2^{(i)}} \quad \textcolor{red}{\vee \overline{z_1^{(i)}}}) \\ & \wedge \dots \\ & \wedge (y_{i-1} \quad \textcolor{red}{\vee z_{i-1}^{(i)}} \quad \textcolor{red}{\vee \overline{z_{i-2}^{(i)}}}) \\ & \wedge (y_i \quad \textcolor{red}{\vee z_i^{(i)}} \quad \textcolor{red}{\vee \overline{z_{i-1}^{(i)}}}) \\ & \wedge (y_{i+1} \quad \textcolor{red}{\vee z_{i+1}^{(i)}} \quad \textcolor{red}{\vee \overline{z_i^{(i)}}}) \\ & \wedge \dots \\ & \wedge (y_{k-2} \quad \textcolor{red}{\vee z_{k-2}^{(i)}} \quad \textcolor{red}{\vee \overline{z_{k-3}^{(i)}}}) \\ & \wedge (y_{k-1} \vee y_k \quad \textcolor{red}{\vee z_{k-2}^{(i)}}) \end{aligned}$$

3. taille(I_2) linéaire en taille(I_1)

4. (\Rightarrow) : si I_1 a une solution c'est-à-dire une instanciation des x_i telle que $\forall j C_j$ soit vrai. Une solution pour I_2 est :

- $x_i \rightsquigarrow$ inchangé
- $a_i \rightsquigarrow$ vrai
- $b_i \rightsquigarrow$ faux
- $c_i \rightsquigarrow$ vrai

— $y_1 \vee \dots \vee y_k$: soit y_i le 1er littéral vrai :

- $z_1, \dots, z_{i-2} \rightsquigarrow$ vrai
- $z_{i-1}, \dots, z_{k-3} \rightsquigarrow$ faux

5. (\Leftarrow) : si I_2 a une solution :

- Instanciation des $y_i, a_i, b_i, c_i, z_j^i \rightsquigarrow$ vrai/faux
- Les x_1, \dots, x_n marchent pour la formule de départ.
- Raison : I_2 ne peut être V avec tous les y_i faux. Sinon, soit pour $k = 7$, on aura :

$$C = (\textcolor{red}{z_1}) \wedge (\textcolor{red}{\overline{z_1}} \vee z_2) \wedge (\textcolor{red}{\overline{z_2}} \vee z_3) \wedge (\textcolor{red}{\overline{z_3}} \vee z_4) \wedge (\textcolor{red}{\overline{z_4}} \vee z_5) \wedge (\textcolor{red}{\overline{z_5}})$$

plus généralement, on aura :

$$\overline{z_{k-2}^{(i)}} \wedge z_{k-2}^{(i)}$$

contradictoire ...

Exemple : soit C une clause de F , par exemple

$$C = y_1 \vee y_2 \vee y_3 \vee y_4 \vee y_5 \vee y_6 \vee y_7$$

On introduit de nouvelles variables z_1, z_2, z_3, z_4, z_5 associées à cette clause, et on remplace C par la formule 3-SAT :

$$C = (y_1 \vee y_2 \textcolor{red}{\vee z_1}) \wedge (\textcolor{red}{\overline{z_1}} \vee y_2 \vee z_2) \wedge (\textcolor{red}{\overline{y_2}} \vee y_3 \textcolor{red}{\vee z_3}) \wedge (\textcolor{red}{\overline{z_3}} \vee y_4 \vee z_4) \wedge (\textcolor{red}{\overline{z_4}} \vee y_5 \vee z_5) \wedge (\textcolor{red}{\overline{z_5}} \vee y_6 \vee y_7)$$

□

5.5.2 Autres instances de SAT dans NPC [Cori et al., 2001]

Problème NAESAT (Not All Equal SAT)

Données Soit un ensemble de variables $\{x_1, \dots, x_n\}$. Soit une formule logique F sous forme normale conjonctive $F = C_1 \wedge C_2 \wedge \dots \wedge C_l$, avec chaque clause $C_i = (y_{i,1} \vee y_{i,2} \vee \dots \vee y_{i,k_i})$, où chaque $y_{i,j}$ est un littéral, c'est-à-dire $y_{i,j} = x_i$ ou $y_{i,j} = \neg x_i$.

Résultat Décider s'il existe une affectation de valeurs de vérité aux x_i de sorte que chaque clause contienne au moins un littéral vrai et au moins un littéral faux ; c'est-à-dire $\forall i, \exists j, k, y_{i,j} = V, y_{i,k} = F$.

Problème NAE3SAT

Données Même donnée qui NAESAT sur des clauses ternaires.

Résultat Voir NAESAT.

Nous avons également les problèmes NPC :

Problème k -SAT, $k > 2$

Problème MAX2SAT et MAXSAT Etant donné l , existe-t-il une affectation de valeurs de vérité qui permet de satisfaire au moins l clauses de la formule ?

5.5.3 Autres instances de SAT dans P [Cori et al., 2001]

Problème 2-SAT

Problème HORN-SAT Toutes les clauses doivent être des clauses de Horn, c'est-à-dire de la forme $(x_1 \wedge x_2 \wedge \dots \wedge x_n \Rightarrow y)$, ou encore $(\overline{x_1} \vee \overline{x_2} \vee \dots \vee \overline{x_n} \vee y)$.

Problème SAT ONE Le cas de SAT où chaque variable apparaît au plus deux fois sur l'ensemble des clauses est polynomial.

Problème 3-SAT ONE Le cas de 3SAT où chaque variable apparaît au plus trois fois sur l'ensemble des clauses est également polynomial.

5.5.4 Stable [Cori et al., 2001]

Problème STABLE

Données Soit un graphe $G = (V, E)$ non orienté et un entier k .

Résultat Décider s'il existe $V' \subset V$, $\text{card}(V') = k$, tel que $\forall u, v \in V' \Rightarrow (u, v) \notin E$.

Proposition 16. STABLE est NP-complet.

Preuve :

STABLE est dans NP, car la donnée de V' est un certificat aisément vérifiable en temps polynomial.

On va réduire le problème 3-SAT à STABLE, c'est-à-dire étant donné une formule booléenne F du type 3-SAT, construire en temps polynomial un graphe G , de sorte

que l'existence d'un stable dans G soit équivalente à l'existence d'une affectation de valeurs de vérité satisfaisant F .

Soit $F = \bigwedge_{1 \leq j \leq p} (x_{1,j} \vee x_{2,j} \vee x_{2,j})$. Définissons un graphe G . G a $3k$ sommets, un pour chaque occurrence d'un littéral dans une clause. Contraintes liées aux littéraux : pour tout i , G a une arête entre chaque sommet associé à un littéral x_i et chaque sommet associé à un littéral \bar{x}_i (ainsi, un stable de G correspond à une affectation de valeurs de vérité à une partie des variables). Contraintes associées aux clauses : si par exemple $C = (x_1 \vee \bar{x}_2 \vee x_3)$ est une clause de F , alors G a les arêtes $\{x_1, \bar{x}_2\}$, $\{\bar{x}_2, x_3\}$, $\{x_3, x_1\}$ formant un triangle (ainsi un stable de G contient au plus un des trois associés à la clause C).

Soit k le nombre de clauses de F . On démontre maintenant que F est satisfaisable ssi G possède un stable de taille k .

Si F est satisfaisable, on considère une affectation des variables satisfaisant F . Pour chaque clause C de F , on choisit l_c un littéral de C rendu vrai par l'affectation. Ceci définit k sommets formant un stable de G .

Inversement, si G a un stable de taille k , alors il a exactement un sommet dans chaque triangle ; ce sommet correspond à un littéral rendant la clause associée vraie, et forme une affectation des variables cohérente par construction des arêtes formant les contraintes de cohérence. Cette réduction est manifestement polynomiale.

□

5.5.5 Clique [Cori et al., 2001]

Problème CLIQUE

Données Soit un graphe $G = (V, E)$ non orienté et un entier k .

Résultat Décider s'il existe $V_0 \subset V$, $\text{card}(V_0) = k$, tel que $\forall u, v \in V_0 \Rightarrow (u, v) \in E$.

Proposition 17. *CLIQUE est NP-complet.*

Preuve :

Passage au complémentaire.

□

5.5.6 Recouvrement par sommets [Cori et al., 2001]

Problème RECOUVREMENT

Données Soit un graphe $G = (V, E)$ non orienté et un entier k .

Résultat Décider s'il existe $V_0 \subset V$, $\text{card}(V') = k$, tel que toute arête de E ait au moins une extrémité dans V_0 ?

Proposition 18. *RECOUVREMENT est NP-complet.*

Preuve :

Complémentaire du stable de G .

□

5.5.7 Circuit hamiltonien [Cori et al., 2001]

Problème CIRCUIT HAMILTONIEN

Données Soit un graphe $G = (V, E)$.

Résultat Décider s'il existe un circuit hamiltonien, c'est-à-dire un chemin de G passant une fois et une seule fois par chacun des sommets et revenant à son point de départ ?

Proposition 19. *CIRCUIT HAMILTONIEN est NP-complet.*

Preuve :

On va démontrer ce fait en réduisant RECOUVREMENT DE SOMMETS à CIRCUIT HAMILTONIEN. La technique consiste à construire, partant d'une instance de RECOUVREMENT DE SOMMETS, un graphe dans lequel chaque arête initiale sera remplacée par un "motif" admettant exactement deux chemins hamiltoniens, c'est-à-dire des chemins visitant une fois et une seule fois chaque sommet. L'un de ces deux chemins correspondra au cas où le sommet correspondant appartient à la couverture, l'autre au cas où il n'appartient pas. Il reste ensuite à préciser comment recoller ces différents motifs pour qu'un circuit hamiltonien global corresponde exactement à une réunion de chemins hamiltoniens à travers chaque motif, et pour assurer qu'on obtient bien le bon résultat au problème RECOUVREMENT DE SOMMETS initial en résolvant CIRCUIT HAMILTONIEN dans ce graphe.

Notons $(G = (V, E), k)$ l'instance de RECOUVREMENT DE SOMMETS étudiée. Le motif que nous allons utiliser est donné dans la Figure 5.1.

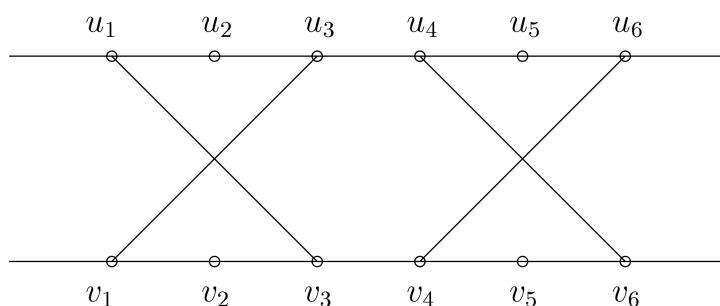


FIGURE 5.1 – RECOUVREMENT DE SOMMETS à CIRCUIT HAMILTONIEN, page 92 [Cori et al., 2001]

Pour obtenir un parcours de ce motif traversant une fois et une seule fois chaque sommet, seules deux solutions sont possibles : soit un passage en deux fois, une fois $u_1u_2u_3u_4u_5u_6$ puis $v_1v_2v_3v_4v_5v_6$ (dans un sens ou dans l'autre) ; ou alors un passage en une seule fois $u_1u_2u_3v_1v_2v_3v_4v_5v_6u_4u_5u_6$ (ou le même en inversant les u et les v). A chaque arête (u, v) du graphe de départ, on associe un motif de ce type, en faisant correspondre le sommet u au côté (u) et le sommet v au côté (v) . On raccorde ensuite entre eux bout-à-bout tous les côtés de tous les motifs correspondant à un même sommet ; on forme donc une chaîne associée à un sommet donné, avec encore deux sorties libres. On raccorde alors chacune de ces deux sorties à k nouveaux sommets s_1, \dots, s_k . On note le graphe ainsi construit H dans la suite.

Supposons maintenant donné un recouvrement du graphe initial de taille k , dont les sommets sont $\{g_1, \dots, g_k\}$. On peut alors construire un circuit hamiltonien de H de la façon suivante :

- partir de s_1 ;
- parcourir la chaîne g_1 de la façon suivante. Quand on traverse une arête (g_1, h) , si h est aussi la couverture, on traverse simplement le côté g_1 , sinon, on parcourt les deux côtés simultanément.
- une fois la chaîne g_1 finie, on revient en s_2 et on repart par g_2 ; ainsi de suite. Il est clair que tous les sommets s_k sont atteints une fois et une seule.

Considérons un sommet h du motif correspondant à une arête (u, v) . On peut toujours supposer que u est dans la couverture, disons $u = g_1$. Il s'ensuit que si h est du côté de u , h sera atteint une fois au moins. On voit en vertu du second item qu'il ne sera plus atteint dans la suite. Si h est du côté de v , et que v n'est pas dans la couverture, h est parcouru lors du parcours de u . Si $v = g_i$, h est parcouru lors du parcours de la chaîne correspondant à g_i et à ce moment-là seulement. On a donc bien un circuit hamiltonien.

Réciproquement, supposons que l'on dispose d'un circuit hamiltonien de H . La construction de notre motif impose que venant d'un sommet s_i , on traverse entièrement une chaîne u puis l'on passe à un autre des sommets s_j . On traverse ainsi k chaînes ; les k sommets correspondants forment alors une couverture. En effet, si (u, v) est une arête, le motif correspondant est parcouru par le chemin hamiltonien ; or il ne peut l'être que lors du parcours d'une boucle correspondant à une des deux extrémités de l'arête.

Enfin, cette réduction est trivialement polynomiale. CIRCUIT HAMILTONIEN est donc bien NP-complet.

□

5.5.8 Voyageur de commerce [Cori et al., 2001]

Problème VOYAGEUR DE COMMERCE

Données Soit un couple (n, M) , où M est une matrice $n \times n$ de réels et un réel k .

Résultat Décider s'il existe une permutation π de $[1, n]$ telle que $\sum_{1 \leq i \leq n-1} M_{\pi(i), \pi_{i+1}} + M_{\pi(n), \pi_1} \leq k$?

On peut voir ce problème comme l'établissement de la tournée d'un voyageur de commerce devant visiter n villes, dont les distances sont données par la matrice M , de façon à faire au moins k kilomètres.

Proposition 20. *VOYAGEUR DE COMMERCE est NP-complet.*

Preuve :

On réduit CIRCUIT HAMILTONIEN à VOYAGEUR DE COMMERCE. Pour ce faire, étant donné $G = (V, E)$, on pose $V = \{x_1, \dots, x_n\}$. On considère alors la matrice $n \times n$ telle que

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

Montrons alors que :

"VOYAGEUR DE COMMERCE(, M, n) = CIRCUIT HAMILTONIEN(V, E)"

S'il existe un circuit hamiltonien dans G , on peut en effet construire la permutation π comme décrivant l'ordre dans lequel les sommets de G sont parcourus. La construction de M implique alors que $\sum_{1 \leq i \leq n-1} M_{\pi(i), \pi_{i+1}} + M_{\pi(n), \pi_1} = n$. Inversement, étant donné une permutation avec cette propriété, le fait que les $n - 1$ termes de la somme soient au moins égaux à 1, implique qu'ils sont en fait tous égaux à 1, et que donc les arêtes $(x_{\pi(i)}, x_{\pi(i+1)})$ existent dans le graphe G ; on a donc bien construit un circuit hamiltonien.

□

5.5.9 Autres instances des graphes dans NPC [Cori et al., 2001]

Problème COUPURE MAXIMALE

Données Soit un graphe $G = (V, E)$ non orienté et un entier k .

Résultat Décider s'il existe une partition $V_1 \cup V_2$ telle que le nombre d'arêtes entre V_1 et V_2 soit au moins k ?

Problème CIRCUIT LE PLUS LONG Existe-t-il un circuit de G ne passant pas deux fois par le même sommet donc la longueur est $\geq r$?

Problème HAMILTONIEN CHEMIN et CIRCUIT restent NP-complets même si le graphe est supposé planaire et de degré 3, ou si le graphe est supposé biparti. Reste NP-complet si les distances sont choisies dans $\{1, 2\}$.

5.5.10 Autres instances des graphes dans P [Cori et al., 2001]

Problème

5.5.11 Colorabilité dans NPC [Cori et al., 2001]

Problème 3-COLORABILITE même si le graphe est planaire, même si l'on suppose le degré ≤ 4 .

Problème k -COLORABILITE pour tout $k \geq 3$.

5.5.12 Colorabilité dans C [Cori et al., 2001]

Problème 2-COLORABILITE

Problème 4-COLORABILITE PLANAIRES

Problème k -COLORABILITE pour les graphes de degré ≤ 3 .

5.5.13 Somme de sous-ensemble [Cori et al., 2001]

SOMME DE SOUS-ENSEMBLE

Donnée un ensemble fini d'entiers E et un entier $t \in \mathbb{N}$

Résultat existe-t-il $E' \subset E$ tel que $\sum_{x \in E'} x = t$?

NP-Complétude SOMME DE SOUS-ENSEMBLE est NP-complet

SAC-A-DOS

Donnée un ensemble de poids a_1, \dots, a_n , un ensemble de valeurs v_1, \dots, v_n , un poids limite A , et un entier V .

Résultat existe-t-il une suite $\epsilon_i \in \{0, 1\}$ telle que $\sum \epsilon_i v_i \geq V$?

NP-Complétude SAC-A-DOS est NP-complet

RANGEMENT OPTIMAL

Donnée n objets de poids $s_i \in \mathbb{Z}$, une capacité B et un entier k .

Résultat est-il possible de ranger les objets dans k boîtes de capacité B ?

NP-Complétude RANGEMENT OPTIMAL est NP-complet

SOMME DE SOUS-ENSEMBLE, PARTITION et SAC-A-DOS sont dans P si les éléments de l'ensemble (les deux premiers cas) et les valeurs des objets (SAC-A-DOS) sont bornés.

5.5.14 Programmation entière [Cori et al., 2001]

PROGRAMMATION ENTIERE

Donnée une matrice $m \times n$ notée A , un vecteur de dimension m noté b .

Résultat existe-t-il des entiers x_1, \dots, x_n positifs tels que $Ax = b$?

NP-Complétude PROGRAMMATION ENTIERE est NP-complet

La variante où les variables sont dans $\{0, 1\}$ est dans NPC.

5.6 La conjecture $P \neq NP$ [Prins, 1994]

La question cruciale de la théorie de la complexité est de savoir si les problèmes NP-complets peuvent être résolus polynomialement, auquel cas $P=NP$, ou bien si on ne leur trouvera jamais d'algorithmes polynomiaux, auquel cas $P \neq NP$. Cette question n'est pas encore tranchée.

Comme des centaines de problèmes NP-complets résistent en bloc à l'assaut des travaux de recherche, on conjecture aujourd'hui que $P \neq NP$.

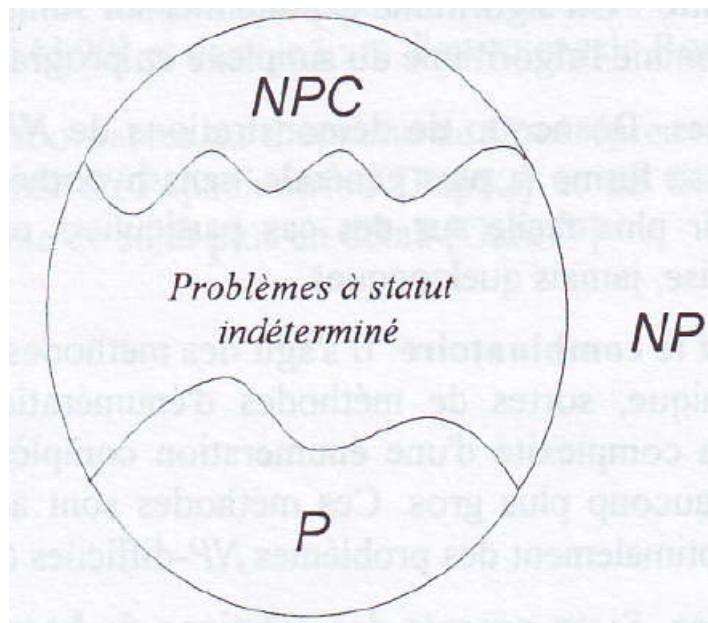


FIGURE 5.2 – Illustration des classes P, NP et NPC (scan [Prins, 1994])

La figure 5.2 illustre les différentes classes de complexité. On note particulièrement les problèmes à statut indéterminé, qui sont dans NP, font partie des problèmes difficiles à résoudre (les meilleurs algorithmes disponibles sont de complexité exponentielle), mais on ne dispose pas encore d'une preuve de leur appartenance à NPC. On peut citer l'exemple du problème d'isomorphisme 2 graphes. Deux graphes G et H sont isomorphes si on peut transformer l'un en l'autre par renumérotation des sommets. Le problème d'isomorphisme de graphes n'a pas d'algorithme polynomial connu, mais personne n'a réussi à montrer qu'il est NP-complet.

5.7 Impact sur l'approche pratique d'un problème réel [Prins, 1994]

Il existe des problèmes combinatoires qu'il ne faut pas s'acharner à résoudre. Si vous avez à résoudre un problèmes combinatoire, vous avez intérêt à consulter un répertoire de problèmes NP-complets pour voir si le problème d'existence associé n'est pas dedans. Un répertoire d'environ 300 problèmes peut être trouvé dans le livre de Garey et Jonhson [Garey and Jonhson, 1979]. Si votre problème est NP-complet, il vaut mieux abandonner l'espoir de trouver un algorithme polynomial.

Le problème de résolution se pose pour tout problème pour lequel on ne dispose pas d'algorithmes polynomiaux (problèmes NP-complets essentiellement). En pratique, on résout le problèmes en tirant parti :

De la taille des données L'énumération complète peut être valable sur de petits cas.

De la complexité moyenne Un algorithme exponentiel sur son pire des cas peut être assez rapide en moyenne (comme l'algorithme du simlexe).

De la nature des données Les démonstrations de NP-complétude considèrent un problème dans sa forme la plus générale, sans hypothèses sur les données. Le problème peut devenir plus facile sur des cas particuliers ou pour les données particulières d'une entreprise, jamais quelconques. On peut citer les nombreux travaux de recherche qui visent à dresser les classes polynomiales de complexités de problèmes difficiles.

Des méthodes diminuant la combinatoire Il s'agit des méthodes arborescentes, de programmation dynamique, et de programmation par contraintes, sortes de méthodes d'énumération intelligentes.

Des méthodes approchées Si on accepte des solutions de bonne qualité mais sans garantie d'optimalité, on peut trouver des algorithmes rapides de recherche locale. En entreprise, une solution est intéressante dès qu'elle permet des économies par rapport aux solutions existantes ou méthodes manuelles, même si elle n'est pas optimale.

5.8 Problèmes de décision et langages [Alliot and Schiex, 1999]

De façon informelle, un problème P est défini par l'ensemble des propriétés que doivent vérifier ses solutions. Généralement, ces propriétés font apparaître un ou plusieurs paramètres non spécifiés. Une instance I d'un problème P est obtenue en spécifiant des valeurs particulières pour tous les paramètres du problème. On dira qu'un algorithme résout un problème P s'il peut être appliqué à toute instance I de P pour fournir la solution de l'instance I en un temps fini.

Les problèmes qui nous intéressent ici sont des problèmes de décision i.e., des problèmes qui n'ont que deux solutions Oui et Non. Si l'on note D_P l'ensemble des instances possibles d'un problème P , on peut distinguer le sous-ensemble Y_P (pour Yes) des instances pour lesquelles la réponse est Oui, N_P pour les réponses Non.

Exemple d'un problème de décision : un entier positif k étant donné, k est-il premier ?

Le complément P^c d'un problème de décision P est défini par la question opposée à celle définissant P : $D_P = D_{P^c}$, $Y_P = N_{P^c}$, $N_P = Y_{P^c}$.

En fait, la raison de cette limitation aux problèmes de décision est qu'il existe une contrepartie naturelle, mais formelle, à un problème de décision, il s'agit du problème de reconnaissance d'un mot dans un langage.

La correspondance entre problèmes de décision et langages s'effectue au moyens d'une fonction de codage e , permettant de traduire toute instance d'un problème de décision P en un mot sur un alphabet Σ . L'ensemble des mots de Σ^* est alors partitionné par e et P en :

- l'ensemble des mots qui ne sont pas des codages d'une instance de P , qui ne nous intéressent pas ;
- l'ensemble des mots qui résultent du codage d'une instance de N_P , pour laquelle la réponse est Non ;
- l'ensemble des mots, notés $L[P, e]$, qui résultent du codage d'une instance de Y_P , pour laquelle la réponse est Oui.

La notion informelle d'algorithme permettant de résoudre un problème de décision peut ensuite être ramenée à la notion formelle de programme de machine de Turing acceptant un mot $x \in \Sigma^*$, x résultant du codage d'une instance I . Plus précisément, nous considérerons des machines de Turing utilisant un alphabet Γ contenant Σ et possédant deux états d'arrêt z_Y et z_N . Appliquée à un mot x , une telle machine \mathcal{M} peut :

- s'arrêter dans l'état z_Y , on dira que le mot est accepté par la machine. L'ensemble des mots acceptés est noté $L(\mathcal{M})$;
- s'arrêter dans l'état z_N , on dira que le mot est rejeté par la machine ;
- ne pas s'arrêter.

Cependant, et pour que le programme de machine de Turing corresponde précisément à la notion d'algorithme qui résout un problème, nous considérerons que des programmes qui s'arrêtent sur toute entrée $x \in \Sigma^*$.

Définition 5 (Complexité temporelle pour une machine de Turing déterministe). *La complexité temporelle d'un programme de machine de Turing déterministe \mathcal{M} est la fonction de n (taille d'un mot codé du langage) :*

$$T_{\mathcal{M}}(n) = \max_{x \in \Sigma^*, |x|=n} \{k \mid k \text{ longueur de calcul sur l'entrée } x\}$$

Définition 6 (Complexité temporelle pour une machine de Turing non déterministe). *La complexité temporelle d'un programme de machine de Turing non déterministe \mathcal{M} est la fonction de n (taille d'un mot codé du langage) :*

$$T_{\mathcal{M}}(n) = \max_{x \in \Sigma^*, |x|=n} \{\min\{k \mid k \text{ longueur de calcul sur l'entrée } x\}\}$$

La définition de la complexité temporelle d'un programme d'une machine de Turing non déterministe est plus délicate. On prend le maximum sur l'ensemble des mots de taille n , mais pour chaque mot, on considère que l'on suit le chemin le plus court menant à une acceptation dans l'arbre de résolution de la machine de Turing non déterministe. Ceci est dû au caractère indéterministe d'un calcul où il existe plusieurs chemins d'exécutions possibles.

5.9 Compléments sur la classification des problèmes

On définit $DTIME(f(n))$ l'ensemble des langages qu'on peut reconnaître en temps avec un nombre d'opération $f(n)$ sur une machine de Turing déterministe. On définit $NTIME(f(n))$ l'ensemble des langages qu'on peut reconnaître en temps avec un nombre d'opération $f(n)$ sur une machine de Turing non-déterministe. On définit $DSPACE(f(n))$ l'ensemble des langages qu'on peut reconnaître avec une consommation mémoire $f(n)$ sur une machine de Turing déterministe. On définit $NSPACE(f(n))$ l'ensemble des langages qu'on peut reconnaître avec une consommation mémoire $f(n)$ sur une machine de Turing non-déterministe.

$$\begin{aligned} P &= \bigcup_{k \geq 0} DTIME(n^k) \\ NP &= \bigcup_{k \geq 0} NTIME(n^k) \\ PSPACE &= \bigcup_{k \geq 0} DSPACE(n^k) \\ NPSPACE &= \bigcup_{k \geq 0} NSPACE(n^k) \\ EXPTIME &= \bigcup_{k \geq 0} DTIME(2^{n^k}) \end{aligned}$$

Etant donnée une classe de langages C , nous définissons $\text{co-}C$ l'ensemble des langages dont les compléments sont dans C . Nous avons donc les classes co-P , co-NP , co-PSPACE et co-NPSPACE . On a $P = \text{co-NP}$.

Intuitivement : PSPACE est la classe des problèmes qui sont solvables en espace polynomial ; EXPTIME est l'ensemble des problèmes solvable en temps exponentiel.

Le théorème de Savitch annonce que la machine de Turing non-déterministe peut être simulée avec une machine de Turing déterministe en espace quadratique. Du coup, $\text{PSPACE} = \text{NPSPACE}$. Plus généralement nous avons :

$$P \subseteq NP \subseteq PSPACE = NPSPACE \subseteq EXPTIME.$$

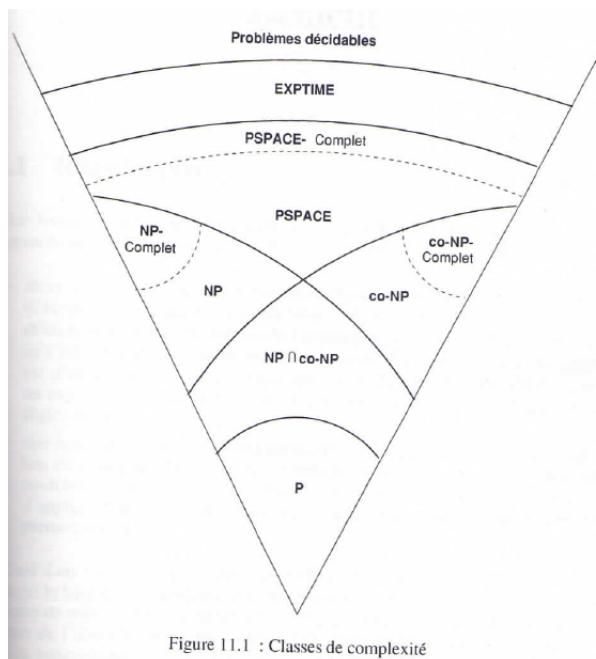


Figure 11.1 : Classes de complexité

FIGURE 5.3 – Illustration des classes de complexités (scan [Alliot and Schiex, 1993])

5.10 Travaux dirigés V

Reprendre pas à pas les preuves de NP-complétude des problèmes :

- 3SAT
- Stable
- Clique
- Recouvrement par sommet
- Circuit hamiltonien
- Voyageur de commerce

Chapitre 6

Algorithmes d'approximation [Robert et al., 2005]

On connaît un certain nombre de problèmes NP-complets. En fait, la plupart d'entre eux sont des problèmes de décision associés à des problèmes d'optimisation. A défaut de calculer la solution optimale, peut-on l'approximer ?

Définition 7. *Un algorithme de λ -approximation est un algorithme polynomial qui renvoie une solution approchée garantie au pire cas à un facteur λ .*

6.1 Vertex cover

6.2 Voyageur de commerce : TSP

6.3 Bin Packing : BP

6.4 2-Partition

Chapitre 7

Différents sujets algorithmiques

- Fondements théoriques des méthodes gloutonnes : matroïdes
- Analyse amortie [Cormen et al., 1994, Cormen et al., 2001]
- Automates [Cormen et al., 1994, Cormen et al., 2001, Beauquier et al., 1992]
- Motifs [Cormen et al., 1994, Cormen et al., 2001, Beauquier et al., 1992]
- FFT [Cormen et al., 1994, Cormen et al., 2001]
- Algorithmes de la théorie des nombres [Cormen et al., 1994, Cormen et al., 2001, Papadimitriou et al., 2006]
- Géométrie algorithmique [Cormen et al., 1994, Cormen et al., 2001, Beauquier et al., 1992]
- Algorithmes approchés [Cormen et al., 1994, Cormen et al., 2001, Prins, 1994]
- Aspects sur la théorie de la complexité [Cormen et al., 1994, Cormen et al., 2001, Prins, 1994, Papadimitriou, 1995, Papadimitriou et al., 2006]

Chapitre 8

Ressources d'auto-formation

8.1 LeetCode

LeetCode is one of the best platform to help you enhance your skills, expand your knowledge and prepare for technical interviews.

— Top Interview 150

<https://leetcode.com/studyplan/top-interview-150/>

Chapitre 9

Annexe : écriture des algorithmes

Nous avons les trois écritures :

Alternatives

```
if B then
     $P_1; P_2; \dots; P_k$ 
end if
    et
if B then
     $P_1; P_2; \dots; P_k$ 
else
     $Q_1; Q_2; \dots; Q_l$ 
end if
    et
if  $B_1$  then
     $P_{11}; P_{12}; \dots; P_{1k}$ 
else if  $B_2$  then
     $P_{21}; P_{22}; \dots; P_{2l} \dots$ 
else
     $P_{m1}; P_{m2}; \dots; P_{ml}$ 
end if
```

Itérations bornées

```
for  $i = 1$  to  $n$  do
     $P_1; P_2; \dots; P_k$ 
end for
    ou aussi
for  $i = n$  downto 1 do
     $P_1; P_2; \dots; P_k$ 
end for
```

Itérations non bornées

```
while B do
     $P_1; P_2; \dots; P_k$ 
end while
```

ou aussi

repeat

$P_1; P_2; \dots; P_k$

until B

Chapitre 10

Annexe : compléments sur les récurrences

10.1 Récurrences non linéaire du premier ordre [Sedgewick a]

Lorsque la récurrence est une fonction nonlinéaire de a_n et a_{n-1} , il n'y a pas de solution générale sous forme close.

On calcule souvent les valeurs initiales pour la bonne raison que certaines récurrences apparemment compliquées convergent vers une constante. En effet soit l'équation

$$a_n = 1/(1 + a_{n-1})$$

pour $n > 0$ avec $a_0 = 1$.

En calculant les valeurs initiales, on se rend compte que la récurrence converge vers une constante, illustré dans la Figure 10.1.

n	a_n	$ a_n - (\sqrt{5} - 1)/2 $
1	0,500000000000	0,118033988750
2	0,666666666667	0,048632677917
3	0,600000000000	0,018033988750
4	0,625000000000	0,006966011250
5	0,615384615385	0,002649373365
6	0,619047619048	0,001013630298
7	0,617647058824	0,000386929926
8	0,618181818182	0,000147829432
9	0,617977528090	0,000056460660

FIGURE 10.1 – [Sedgewick and Flajolet, 1996]

Si l'on admet que la récurrence converge vers une constante α , alors elle doit vérifier $\alpha = 1/(1+\alpha)$, ou $1-\alpha-\alpha^2 = 0$. C'est-à-dire que $\alpha = (\sqrt{5}-1)/2 \approx 0.6180334$.

On peut aussi faire appel à des procédés d'analyse numérique pour étudier la convergence de la suite récurrente, et aussi pour la résoudre. Typiquement, on peut faire appel à la méthode de Newton qui a la bonne propriété de convergence quadratique.

10.2 Séries génératrices [Sedgewick and Flajolet, 1996]

Cette section a comme objectif de montrer le rôle essentiel des récurrences en analyse d'algorithmes, et que de nombreuses récurrences se résolvent facilement à l'aide des séries génératrices. Les séries génératrices est un outil analytique pour calculer les estimations.

Définition 8. Soit $a_0, a_1, \dots, a_k, \dots$ une suite.

La série

$$A(z) = \sum_{k \geq 0} a_k z^k$$

est appelée série génératrice ordinaire (SGO) de cette suite. Le coefficient a_k est également noté $[z^k]A(z)$.

La série

$$A(z) = \sum_{k \geq 0} a_k \frac{z^k}{k!}$$

est appelée série génératrice exponentielle (SGE) de cette suite. Le coefficient a_k est également noté $k![z^k]A(z)$.

Le tableau de la Figure 10.2 (resp. de la Figure ??) présente quelques séries génératrices (resp. séries génératrices exponentielles) élémentaires et les suites correspondantes.

Théorème 4. Soient deux SGOs $A(z) = \sum_{k \geq 0} a_k z^k$ et $B(z) = \sum_{k \geq 0} b_k z^k$. Les opérations décrites dans la Figure 10.3 engendrent les SGOs présentant les suites indiquées.

Soient deux SGEs $A(z) = \sum_{k \geq 0} a_k \frac{z^k}{k!}$ et $B(z) = \sum_{k \geq 0} b_k \frac{z^k}{k!}$. Les opérations décrites dans la Figure ?? engendrent les SGEs présentant les suites indiquées.

En fait, formellement, on pourrait utiliser une famille quelconque de noyaux de fonctions $w_k(z)$ pour définir une "série génératrice"

$$A(z) = \sum_{k \geq 0} a_k w_k(z)$$

représentant une suite $a_0, a_1, \dots, a_k, \dots$ Bien que ici, on se consacre uniquement aux noyaux z^k et $z^k/k!$. D'autres noyaux apparaissent parfois dans la littérature.

$1, 1, 1, 1, \dots, 1, \dots$	$\frac{1}{1-z} = \sum_{N \geq 0} z^N$
$0, 1, 2, 3, 4, \dots, N, \dots$	$\frac{z}{(1-z)^2} = \sum_{N \geq 1} Nz^N$
$0, 0, 1, 3, 6, 10, \dots, \binom{N}{2}, \dots$	$\frac{z^2}{(1-z)^3} = \sum_{N \geq 2} \binom{N}{2} z^N$
$0, \dots, 0, 1, M+1, \dots, \binom{N}{M}, \dots$	$\frac{z^M}{(1-z)^{M+1}} = \sum_{N \geq M} \binom{N}{M} z^N$
$1, M, \binom{M}{2}, \dots, \binom{M}{N}, \dots, M, 1$	$(1+z)^M = \sum_{N \geq 0} \binom{M}{N} z^N$
$1, M+1, \binom{M+2}{2}, \binom{M+3}{3}, \dots$	$\frac{1}{(1-z)^{M+1}} = \sum_{N \geq 0} \binom{N+M}{N} z^N$
$1, 0, 1, 0, \dots, 1, 0, \dots$	$\frac{1}{1-z^2} = \sum_{N \geq 0} z^{2N}$
$1, c, c^2, c^3, \dots, c^N, \dots$	$\frac{1}{1-cz} = \sum_{N \geq 0} c^N z^N$
$1, 1, \frac{1}{2!}, \frac{1}{3!}, \frac{1}{4!}, \dots, \frac{1}{N!}, \dots$	$e^z = \sum_{N \geq 0} \frac{z^N}{N!}$
$0, 1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \dots, \frac{1}{N}, \dots$	$\ln \frac{1}{1-z} = \sum_{N \geq 1} \frac{z^N}{N}$
$0, 1, 1 + \frac{1}{2}, 1 + \frac{1}{2} + \frac{1}{3}, \dots, H_N, \dots$	$\frac{1}{1-z} \ln \frac{1}{1-z} = \sum_{N \geq 1} H_N z^N$
$0, 0, 1, 3(\frac{1}{2} + \frac{1}{3}), 4(\frac{1}{2} + \frac{1}{3} + \frac{1}{4}), \dots$	$\frac{z}{(1-z)^2} \ln \frac{1}{1-z} = \sum_{N \geq 0} N(H_N - 1) z^N$

FIGURE 10.2 – [Sedgewick and Flajolet, 1996]

$$A(z) = \sum_{n \geq 0} a_n z^n \quad a_0, a_1, a_2, \dots, a_n, \dots,$$

$$B(z) = \sum_{n \geq 0} b_n z^n \quad b_0, b_1, b_2, \dots, b_n, \dots,$$

décalage vers la droite

$$zA(z) = \sum_{n \geq 1} a_{n-1} z^n \quad 0, a_0, a_1, a_2, \dots, a_{n-1}, \dots,$$

décalage vers la gauche

$$\frac{A(z) - a_0}{z} = \sum_{n \geq 0} a_{n+1} z^n \quad a_1, a_2, a_3, \dots, a_{n+1}, \dots,$$

multiplication d'indice (différentiation)

$$A'(z) = \sum_{n \geq 0} (n+1) a_{n+1} z^n \quad a_1, 2a_2, \dots, (n+1)a_{n+1}, \dots,$$

division d'indice (intégration)

$$\int_0^z A(t) dt = \sum_{n \geq 1} \frac{a_{n-1}}{n} z^n \quad 0, a_0, \frac{a_1}{2}, \frac{a_2}{3}, \dots, \frac{a_{n-1}}{n}, \dots,$$

mise à l'échelle

$$A(\lambda z) = \sum_{n \geq 0} \lambda^n a_n z^n \quad a_0, \lambda a_1, \lambda^2 a_2, \dots, \lambda^n a_n, \dots,$$

addition

$$A(z) + B(z) = \sum_{n \geq 0} (a_n + b_n) z^n \quad a_0 + b_0, \dots, a_n + b_n, \dots,$$

différence

$$(1-z)A(z) = a_0 + \sum_{n \geq 1} (a_n - a_{n-1}) z^n \quad a_0, a_1 - a_0, \dots, a_n - a_{n-1}, \dots,$$

convolution

$$A(z)B(z) = \sum_{n \geq 0} \left(\sum_{0 \leq k \leq n} a_k b_{n-k} \right) z^n \quad a_0 b_0, a_1 b_0 + a_0 b_1, \dots, \sum_{0 \leq k \leq n} a_k b_{n-k}, \dots,$$

somme partielle

$$\frac{A(z)}{1-z} = \sum_{n \geq 0} \left(\sum_{0 \leq k \leq n} a_k \right) z^n \quad a_1, a_1 + a_2, \dots, \sum_{0 \leq k \leq n} a_k, \dots,$$

FIGURE 10.3 – [Sedgewick and Flajolet, 1996]

Les séries génératrices permettent de résoudre un grand nombre de relations de récurrence de manière mécanique. Etant donnée une récurrence décrivant une suite $(a_n)_{n \geq 0}$ on peut souvent aboutir à une solution en procédant de la façon suivante :

Multiplier les deux membres de la récurrence par z^n et sommer sur n .

Evaluer les sommes pour en tirer une équation vérifiée par la SGO.

Résoudre cette équation pour obtenir une formule explicite de la SGO.

Développer la SGO en série entière pour obtenir les coefficients, c'est-à-dire les termes de la suite d'origine.

(Cette méthode reste valable également pour les SGEs.)

Soit par exemple, la récurrence

$$a_n = 5a_{n-1} - 6a_{n-2}$$

pour $n > 1$, avec $a_0 = 0$ et $a_1 = 1$.

La série génératrice $a(z) = \sum_{n \geq 0} a_n z^n$ devient

$$a(z) = \frac{z}{1 - 5z + 6z^2} = \frac{z}{(1 - 3z)(1 - 2z)} = \frac{1}{1 - 3z} - \frac{1}{1 - 2z}$$

par conséquent $a_n = 3^n - 2^n$.

Les SGOS (resp. SGEs) ont plusieurs propriétés mathématiques que l'on pourrait exploiter pour raisonner sur les SGOS en vue de résoudre les récurrences.

Fibonacci numbers. The generating function $F(z) = \sum_{k \geq 0} F_k z^k$ for the Fibonacci sequence

$$F_n = F_{n-1} + F_{n-2} \quad \text{for } n > 1 \text{ with } F_0 = 0 \text{ and } F_1 = 1$$

satisfies

$$F(z) = zF'(z) + z^2F(z) + z.$$

This implies that

$$F(z) = \frac{z}{1 - z - z^2} = \frac{1}{\sqrt{5}} \left(\frac{1}{1 - \phi z} - \frac{1}{1 - \hat{\phi} z} \right)$$

by partial fractions, since $1 - z - z^2$ factors as $(1 - z\phi)(1 - z\hat{\phi})$ where

$$\phi = \frac{1 + \sqrt{5}}{2} \quad \text{and} \quad \hat{\phi} = \frac{1 - \sqrt{5}}{2}$$

are the reciprocals of the roots of $1 - z - z^2$. Now the series expansion is straightforward from Table 3.4:

$$F_n = \frac{1}{\sqrt{5}} (\phi^n - \hat{\phi}^n).$$

FIGURE 10.4 – Solution de la récurrence de Fibonacci
[Sedgewick and Flajolet, 1996]

Théorème 5 (SGOs de récurrence linéaire). *Si a_n vérifie la récurrence*

$$a_n = x_1 a_{n-1} + x_2 a_{n-2} + \dots + x_t a_{n-t}$$

pour $n \geq t$, alors la série génératrice $a(z) = \sum_{n \geq 0} a_n z^n$ est une fraction rationnelle $a(z) = f(z)/g(z)$ où le polynôme dénominateur est $g(z) = 1 - x_1 z - x_2 z^2 - \dots - x_t z^t$ et le polynôme numérateur est déterminé par les conditions initiales a_0, a_1, \dots, a_{t-1} .

Simple example. To solve the recurrence

$$a_n = 2a_{n-1} + a_{n-2} - 2a_{n-3} \quad \text{for } n > 2 \text{ with } a_0 = 0 \text{ and } a_1 = a_2 = 1$$

we first compute

$$g(z) = 1 - 2z - z^2 + 2z^3 = (1 - z)(1 + z)(1 - 2z)$$

then, using the initial conditions, we write

$$\begin{aligned} f(z) &= (z + z^2)(1 - 2z - z^2 + 2z^3) \pmod{z^3} \\ &= z - z^2 = z(1 - z). \end{aligned}$$

This gives

$$a(z) = \frac{f(z)}{g(z)} = \frac{z}{(1 + z)(1 - 2z)} = \frac{1}{3} \left(\frac{1}{1 - 2z} - \frac{1}{1 + z} \right),$$

so that $a_n = \frac{1}{3}(2^n - (-1)^n)$.

FIGURE 10.5 – Illustration du théorème 5 [Sedgewick and Flajolet, 1996]

10.2.1 Exercices

- Résoudre la récurrence

$$a_n = 2a_{n-1} + 1$$

pour $n \geq 1$ avec $a_0 = 1$.

- Résoudre la récurrence

$$a_n = 2a_{n-1} + a_{n-2} - 2a_{n-3}$$

pour $n > 1$ avec $a_0 = 1, a_1 = 1, a_2 = 1$.

- Résoudre la récurrence

$$a_n = 5a_{n-1} - 8a_{n-2} + 4a_{n-3}$$

pour $n > 2$ avec $a_0 = 0, a_1 = 1, a_2 = 4$.

Chapitre 11

Annexe : préliminaires mathématiques

Note : Le texte de ce document rassemble des extraits du livre de Guessarian [Arnold and Guessarian, 1997] et du handbook d'Allen [Allen B. Tucker, 1996].

11.1 Calcul ensembliste

11.1.1 Langage ensembliste

Soit E un ensemble et e un élément, $e \in E$ signifie que e est un élément qui est dans l'ensemble E et se lit e appartient à E . La négation de cette relation s'écrit $e \notin E$. L'ensemble vide est un ensemble qui ne contient aucun élément, il est noté \emptyset .

- $A \subseteq B$ ssi $\forall x \in A \Rightarrow x \in B$
 - $A = B$ ssi $A \subseteq B$ et $B \subseteq A$
 - $P(E)$ l'ensemble des parties de E ; $A \in P(E)$ ssi $A \subseteq E$
 - $E \times F = \{(x, y) | x \in E \text{ et } y \in F\}$
 - $E_1, \dots, E_n = \{(x_1, \dots, x_n) | x_1 \in E_1, \dots, x_n \in E_n\}$
- Soit A et B deux sous-ensembles de E , nous avons
- $A \cap B = \{e \in E | e \in A \text{ et } e \in B\}$
 - $A \cup B = \{e \in E | e \in A \text{ ou } e \in B\}$
 - $A \setminus B = \{e \in E | e \in A \text{ et } e \notin B\}$
 - $\overline{A} = E \setminus A = \{e \in E | e \notin A\}$

11.1.2 Fonctions

Soit E et F deux ensembles, on appelle correspondance de E dans F tout triplet $f = (E, F, \Gamma)$ où Γ est une partie de $E \times F$. On dit que E est l'ensemble de départ, F l'ensemble d'arrivée et Γ le graphe de f . L'ensemble

$$Dom(f) = \{x \in E | \exists y \in F, (x, y) \in \Gamma\}$$

est le domaine ou l'ensemble de définition de f et l'ensemble

$$Im(f) = \{y \in F | \exists x \in E, (x, y) \in \Gamma\}$$

est l'image de f .

Pour $y \subseteq F$, on note

$$f^{-1}(Y) = \{x \in E \mid \exists y \in Y, (x, y) \in \Gamma\}$$

l'ensemble des antécédents par f des éléments de Y .

On dit qu'une correspondance $f = (E, F, \Gamma)$ est une fonction si tout élément de E a au plus une image par f . Pour tout $x \in E$, son image est noté $f(x)$. La fonction f est notée $f : E \rightarrow F$. On dit qu'une fonction $f : E \rightarrow F$ est une application si son domaine est l'ensemble E tout entier.

- $id_E : E \rightarrow E$ est l'application identité de E définie par $f(x) = x, \forall x$
- $\chi_A : E \rightarrow \{0, 1\}$ est définie par $\chi_A(e) = \begin{cases} 1 & \text{si } e \in A \\ 0 & \text{sinon} \end{cases}$
- f est injectivessi $\forall x, y \in E, (f(y) = f(x)) \Rightarrow x = y$
- f est surjectivessi $\forall y \in E, \exists x \in E, y = f(x)$
- f est bijectivessi $\forall y \in E, !\exists x \in E, y = f(x)$

Soit $f : E \rightarrow F$ et $g : E \rightarrow G$ deux applications, la composition de f et g est l'application $gOf : E \rightarrow G$ définie par $gOf(x) = g(f(x))$.

11.1.3 Cardinaux

Pour tout entier n , on note $[n]$ l'ensemble $\{1, \dots, n\}$ des entiers compris entre 1 et n . Cet entier n est alors unique, s'appelle cardinal de E et se note $|E|$.

Un ensemble E est dénombrable s'il est en bijection avec l'ensemble \mathbb{N} des entiers naturels. Signalons qu'il existe des ensembles non dénombrables, qui proviennent essentiellement de la propriété :

$$|E| < |P(E)|$$

11.1.4 Opérateurs et relations

Un opérateur Φ sur un ensemble E est une application $\Phi : E^n \rightarrow E$. On dit que n est l'arité, ou le rang, de Φ , ou encore que Φ est une opération n -aire.

Une opération binaire $*$ (ou bien de composition interne) $* : E \rightarrow E$. On définit :

- $*$ est associative si $\forall a, b, c \in E, a * (b * c) = (a * b) * c$
- $*$ est commutative si $\forall a, b \in E, a * b = b * a$
- $*$ admet l'élément λ pour élément neutre si $\forall e \in E, e * \lambda = \lambda * e = e$

Une relation sur un ensemble E est la donnée d'une partie R de $E \times E$. Pour indiquer qu'une paire (e, e') de $E \times E$ est dans cette partie R , on utilisera, selon les cas, l'une des notations suivantes : $(e, e') \in R, e R e', R(e, e')$.

On dit que la relation R est :

- réflexive si $\forall e \in E, e R e'$
- irréflexive si $\forall e, e' \in E, e R e' \Rightarrow R$
- symétrique si $\forall e, e' \in E, e R e' \Rightarrow e' R e$
- antisymétrique si $\forall e, e' \in E, e R e' \text{ et } e' R e \Rightarrow e = e'$
- transitive si $\forall e, e', e'' \in E, e R e' \text{ et } e' R e'' \Rightarrow e R e''$

Définition 9 (relation d'équivalence). *Une relation d'équivalence est une relation réflexive, symétrique et transitive.*

Définition 10 (classe d'équivalence). *Soit R une relation d'équivalence sur E , et e un élément de E . L'ensemble $\{e' \in E | e R e'\}$, noté $[e]_R$, est appelé la classe d'équivalence de e .*

11.2 Ensembles ordonnés, récursion et induction

11.2.1 Les relations d'ordre

Définition 11.

- Une relation d'ordre est une relation réflexive, antisymétrique, et transitive.
- Une relation d'ordre stricte est une relation irréflexive et transitive.
- Si la relation R vérifie $\forall e, e' \in E, e \neq e' \Rightarrow (e R e' \text{ ou } e' R e)$, on dit que R est un ordre total. Sinon on dit que R est un ordre partiel.
- Une relation de préordre est une relation transitive.
- Un ensemble ordonné (E, \leq) est un ensemble muni d'une relation d'ordre \leq .
- Soient (E_1, \leq_1) et (E_2, \leq_2) deux ensembles ordonnés. Une application f de E_1 dans E_2 est dite monotone si

$$\forall x, y \in E_1, x \leq_1 y \Rightarrow f(x) \leq_2 f(y).$$

- f est donc un homéomorphisme de (E_1, \leq_1) dans (E_2, \leq_2) .
- (E_1, \leq_1) et (E_2, \leq_2) sont isomorphes s'il existe une bijection b entre E_1 et E_2 telle que b et b^{-1} soient monotones.
- Un ensemble ordonné (E, \leq) est totalement ordonné si \leq est un ordre total. Sinon, il est partiellement ordonné.
- Soit (E, \leq) un ensemble ordonné. Un sous-ensemble ordonné de (E, \leq) est un ensemble ordonné (E', \leq') tel que $E' \subseteq E$ et $\leq' = \leq \cap (E' \times E')$, c'est-à-dire $\forall x, y \in E', x \leq' y \Leftrightarrow x \leq y$.
- Une chaîne de E est un sous-ensemble totalement ordonné de E .
- Une chaîne est maximale si elle n'est pas incluse dans une autre chaîne.
- Une antichaine E' de E est un sous-ensemble de E tel que

$$\leq \cap (E' \times E') = Id_E$$

Deux éléments quelconques d'une antichaine sont incomparables.

- Soit E' une partie d'un ensemble ordonné (E, \leq) . Un élément x de E est un majorant de E' (resp. minorant) si $\forall y \in E', y \leq x$ (resp. $x \leq y$). Nous noterons $Maj(E')$ l'ensemble des majorants de E' et $Min(E)$ l'ensemble des minorants.
- Si $Maj(E') \cap E'$ n'est pas vide l'unique élément de cet ensemble est appelé le maximum de E' . De même avec les prédictats.
- La borne supérieure (resp. inférieure) d'une partie E' de E est le minimum de $Maj(E')$ (resp. du maximum de $Min(E')$). On notera $Sup(E')$ et $Inf(E')$ ses bornes supérieure et inférieure, si elles existent.

11.2.2 Ensembles bien fondés et induction

Définition 12. Une relation d'ordre \leq sur un ensemble E est bien fondée si il n'y a pas de suite infinie strictement décroissante d'éléments de E . Un bon ordre est un ordre total bien fondé.

Proposition 21. Un ensemble ordonné (E, \leq) est bien fondéssi toute partie vide de E admet au moins un élément minimal.

Théorème 6. Soit \leq un ordre bien fondé sur un ensemble E et P une proposition dépendant d'un élément x de E . Si la propriété suivante est vérifiée :

$$(I) \forall x \in E, ((\forall y < x, P(y)) \Rightarrow P(x))$$

alors $\forall x \in E, P(x)$.

11.2.3 Induction sur les entiers

Théorème 7 (premier principe d'induction). Soit $P(n)$ un prédictat (une propriété) dépendant de l'entier n . Soient les deux propriétés

$$(B) P(0) \text{ est vraie}$$

$$(I) \forall n \in \mathbb{N}, (P(n) \Rightarrow P(n + 1))$$

alors $\forall n \in \mathbb{N}, P(n)$ est vraie.

Théorème 8 (deuxième principe d'induction). Soit $P(n)$ un prédictat (une propriété) dépendant de l'entier n . Soi les deux conditions suivantes sont vérifiées :

$$(I') \forall n \in \mathbb{N}, (\forall k < n, P(k) \Rightarrow P(n + 1))$$

alors $\forall n \in \mathbb{N}, P(n)$ est vraie.

11.2.4 Définitions inductives, et preuves par induction structurelle

Une définition par induction structurelle d'un ensemble X se présente sous la forme générique intuitive suivante :

(B) : Certains éléments de l'ensemble X sont donnés explicitement (base de la définition récursive)

(I) : les autres éléments de l'ensemble X sont définis en fonction d'éléments appartenant déjà à l'ensemble X (étapes inductives de la définition récursive). Formellement

Définition 13. Soit E un ensemble. Une définition inductive d'une partie X de E consiste en la donnée :

- d'un sous-ensemble B de E
- d'un ensemble K d'opérateurs $\Phi : E^{a(\Phi)} \rightarrow E$, où $a(\Phi) \in \mathbb{N}$ est l'arité de Φ .

X est défini comme étant le plus petit ensemble vérifiant les assertions (B) et (I) suivantes :

$$(B) \quad B \subseteq X$$

$$(I) \quad \forall \Phi \in K, \forall x_1, \dots, x_{a(\Phi)} \in X, \Phi(x_1, \dots, x_{a(\Phi)}) \in X.$$

L'ensemble défini est donc

$$X = \bigcap_{Y \in F} Y$$

où $F = \{Y \subseteq E \mid B \subseteq Y, \text{ et } Y \text{ vérifie (I)}\}$

Par la suite nous noterons une définition inductive sous la forme suivante :

$$(B) \quad x \in X \ (\forall x \in B)$$

$$(I) \quad x_1, \dots, x_{a(\Phi)} \in X \Rightarrow \Phi(x_1, \dots, x_{a(\Phi)}) \in X \ (\forall \Phi \in K).$$

Exemple 1.

Soit

(B)	$0 \in X$
(I)	$n \in X \Rightarrow n + 1 \in X.$

X n'est rien d'autre que \mathbb{N} tout entier

Exemple 2. L'ensemble E des expressions entièrement parenthésées formées à partir d'identificateurs pris dans un ensemble A et des opérateurs $+$ et \times est la partie $(A \cup \{+, \times\})^*$ définie inductivement par

$$(B) \quad A \subseteq E$$

$$(I) \quad \text{Si } e \text{ et } f \text{ sont dans } E \text{ alors } (e + f) \text{ et } (e \times f) \text{ sont aussi dans } E.$$

On peut remarquer que les définitions syntaxiques sont presque toujours inductives. L'ensemble E provient en fait de la définition syntaxique :

$$E ::= A | (e + f) | (e \times f)$$

Théorème 9. Si X est défini par les conditions (B) et (I), tout élément de X peut s'obtenir à partir de la base en appliquant un nombre fini d'étapes inductives.

Proposition 22 (preuve par induction structurelle). Soit X un ensemble défini inductivement, et soit $P(X)$ un prédicat exprimant une propriété de l'élément x de X . Si les conditions suivantes sont vérifiées :

$$(B) \quad P(x) \text{ est vraie pour chaque } x \in B,$$

$$(I) \quad (P(x_1), \dots, P(x_{a(\Phi)})) \Rightarrow P(\Phi(x_1, \dots, x_{a(\Phi)})) \text{ pour chaque } \Phi \in K$$

alors $P(x)$ est vraie pour tout $x \in X$.

11.2.5 Treillis et points fixes

Définition 14. Un ensemble (E, \leq) est appelé un treillis si toute paire d'éléments de E admet une borne supérieure et une borne inférieure. On notera $x \sqcup y$, au lieu de $\sup(\{x, y\})$, la borne supérieure de x et y et $x \sqcap y$ la borne inférieure.

Si E est un treillis, on peut donc considérer que E est un ensemble muni de deux opérations binaires \sqcup et \sqcap .

Définition 15. Un treillis E est dit complémenté si

- il a un élément minimum \perp et un élément maximum \top , avec $\perp \neq \top$,
- il existe une application v de E dans E telle que

$$\forall x \in E, x \sqcap v(x) = \perp,$$

$$\forall x \in E, x \sqcup v(x) = \top.$$

Définition 16. Un ensemble ordonné (E, \leq) est appelé un treillis complet si toute partie de E admet une borne supérieure et une borne inférieure.

Proposition 23. Un ensemble ordonné (E, \leq) est un treillis completssi tout sous-ensemble de E a une borne supérieure.

Définition 17. Une application f d'un ensemble ordonné (E_1, \leq_1) dans un ensemble ordonné (E_2, \leq_2) est dite monotone, (ou plus précisément sup-continue), si elle préserve les bornes supérieures des parties non vides : si la partie $E' \neq \emptyset$ a une borne supérieure $e = \sup(E')$, alors $f(E') = \{f(x) | x \in E'\}$ a aussi une borne supérieure qui est égale à $f(e)$.

Théorème 10. Si f est un application monotone d'un treillis complet dans lui-même, alors f a un plus grand point fixe et un plus petit point fixe.

Théorème 11. Si f est un application monotone d'un treillis complet dans lui-même, alors le plus petit point fixe de f est égal à

$$\sup(\{f^n(\perp) | n \in \mathbb{N}\}).$$

Théorème 12. Si E est un ensemble ordonné fini admettant un élément minimum \perp , pour toute fonction monotone f de E dans lui-même il existe $k \leq \text{card}(E)$ tel que le plus petit point fixe de f est $f^k(\perp)$.

11.3 Logique

11.3.1 Logique propositionnelle

Une proposition est un fait qui peut être exclusivement vrai ou faux. Une proposition individuelle est dite “un atome”. Les formules logiques sont construites à partir d'un ensemble d'atomes \mathcal{A} , un ensemble de connecteurs, et un ensemble de valeurs de vérité comme suit :

- Les atomes et les connecteurs sont des formules.
- Si μ est un connecteur n -aire et si F_1, F_2, \dots, F_n sont des formules, alors $\mu(F_1, F_2, \dots, F_n)$ est une formule.

L'expression $(A \vee \neg B \Leftrightarrow (B \Rightarrow \text{true}))$ est un exemple avec une seule constante, deux atomes A et B , un opérateur unaire \neg (négation), et 3 opérateurs binaires \vee (ou), \Leftrightarrow (iff), et \Rightarrow (if). Un littéral est un atome ou la négation d'un atome.

La sémantique d'une formule logique consiste en ses valeurs de vérité. Soit Δ l'ensemble des valeurs de vérité, alors un connecteur n -aire est une fonction qui va

de Δ^n vers Δ . Dans la logique classique, $\Delta = \{\text{true}, \text{false}\}$; et on dispose tout particulièrement du connecteur unaire \neg de négation, et des deux connecteurs binaires \vee de disjonction et \wedge de conjonction.

Une formule est sous forme NNF “negative normal form” si les conjonctions et les disjonctions sont les seuls opérateurs binaires, et les négations sont au niveau atomique. Une formule est sous forme CNF “conjunctive normal form”, si elle est une conjonction de clauses, où une clause est une disjonction de littéraux.

Une interprétation est une fonction qui associe à chaque atome de \mathcal{A} une valeur dans Δ . Dans la logique classique, une formule est dite satisfiable si elle s'évalue à *true* dans une interprétation donnée. Dans une autre logique, la satisfiabilité est déterminée par un ensemble Δ^* donné ($\Delta^* = \{\text{true}\}$ dans la logique classique). Une formule C est dite une conséquence logique de F si toute interprétation I satisfaisant F , satisfait aussi C : dans ce cas nous écrivons

$$F \models C$$

11.3.2 Inférence et déduction

La notion d'inférence est au cœur de l'implantation de toute logique. Les règles d'inférence sont écrites sous la forme

$$\frac{\text{Premisses}}{\text{Conclusion}} \quad (11.1)$$

où “Prémisses” est un ensemble de formules et “Conclusion” est une formule. La déduction, ou l'inférence, de la formule C à partir d'un ensemble de formules S est une suite C_0, C_1, \dots, C_n , tels que $C_0 \in S$, $C_n = C$, et pour tout i , $1 \leq i \leq n$, C_i satisfait une des conditions suivantes :

- $C_i \in S$
- Il existe une règle d'inférence (*Premisses/Conclusion*) telle que *Premisses* $\subseteq \{C_0, \dots, C_{i-1}\}$ et $C_i \equiv \text{Conclusion}$.

Nous utilisons la notation $S \vdash C$ pour désigner la déduction de C à partir de S . Soit la règle

$$\frac{\text{Premisses}}{\text{La terre est ronde !}}$$

Tout le monde accepte cette inférence, et pourtant d'un point de vue *logique*, on ne peut pas inférer une formule d'une façon détachée d'un environnement logique. Pour éviter ce genre de liberté incontrôlée, il existe deux propriétés importantes qui permettent de juger la pertinence des règles d'inférence :

- Cohérence (soundness) : Soit $F \vdash C$, alors $F \models C$.
- Complétude (completeness) : Soit $F \models C$, alors $F \vdash C$.

La première propriété est la plus importante : la possibilité d'inférer seulement les formules valides est plus important que de pouvoir couvrir toutes les formules valides. En pratique, les logiciens sont plus intéressés par la complétude de la réfutation : la possibilité de vérifier qu'une formule insatisfiable est vraiment insatisfiable. Cependant, il existe des logiques qui violent la propriété de cohérence, comme par exemple la logique non-monotone.

11.3.3 La logique du premier ordre

Les atomes sont appelés les prédictats qui nécessitent des arguments. Par exemple, si H est le prédictat “est un homme”, alors $H(x)$ peut être interprété comme “ x est un homme”. Ainsi $H(Socrates)$, $H(7)$, $H(f(x))$ sont bien formés. Ainsi, les prédictats ont un nombre fini d’arguments et tout argument peut être substitué par un terme. Les termes sont récursivement définis comme suit : les variables et les constantes sont des termes ; si t_1, t_2, \dots, t_n sont des termes, alors si f est une fonction n -aire alors $f(t_1, t_2, \dots, t_n)$ est un terme.

Les formules d’ordre 1 sont essentiellement semblables à celles de la logique propositionnelle, sauf que les atomes sont remplacés par les prédictats. Cependant, les formules d’ordre 1 peuvent être quantifiées.

Dans l’exemple $\forall x \exists y (P(x, y) \vee \neg Q(y, z, c))$, nous avons une seule constante c , une variable x quantifiée universellement, et y existentiellement, et z est une variable libre.

Les interprétations d’ordre 1 dépendent du domaine de discours. Si F est une formule avec n variables libres, alors si I est une interprétation et D est le domaine de discours, alors I fait correspondre F à une fonction allant de D^n à Δ . Une valuation est une affectation de valeurs de D aux variables. Etant donnée une interprétation I et une valuation V , une formule F fournit une valeur de vérité. Deux formules sont dites équivalentes si elles s’évaluent en la même valeur de vérité pour toute interprétation.

On finit cette section de logique sur les interprétations de Herbrand qui jouent un rôle central dans le développement des techniques d’inférences et est au coeur du développement du langage de programmation Prolog. Les interprétations de Herbrand ont comme domaine de discours l’espace de Herbrand. Un espace de Herbrand est défini récursivement comme suit :

- Soit F une formule ;
- H_0 est l’ensemble des constantes apparaissant dans F ; s’il existe aucune constante, soit a une constante arbitraire et $H_0 = \{a\}$;
- Pour $n = 0, 1, 2, \dots$, H_{n+1} est l’union de H_n et tous les termes de la forme $f(t_1, \dots, t_m)$, où $t_i \in H_n$ pour $i = 0, 1, 2, \dots, m$.
- L’espace de Herbrand est

$$H = \bigcup_{i=0}^{\infty} H_i.$$

L’intérêt de l’univers de Herbrand est résumé dans le théorème suivant sur la complétude de la réfutation :

Théorème 13. *Une formule F est insatisfiablessi F est insatisfiable dans les interprétations de Herbrand.*

Ce théorème est à l’origine du développement des techniques d’inférence notamment celle utilisée dans Prolog.

Bibliographie

- [Allen B. Tucker, 1996] Allen B. Tucker, J. (1996). *The computer science and engineering handbook*. CRC Press, ACM.
- [Alliot and Schiex, 1993] Alliot, J.-M. and Schiex, T. (1993). *Intelligence Artificielle et Informatique Théorique*. Cepadues, ISBN : 2-85428-324-4.
- [Arnold and Guessarian, 1997] Arnold, A. and Guessarian, I. (1997). *Mathématiques pour l'informatique*. Masson.
- [Beauquier et al., 1992] Beauquier, D., Berstel, J., and Chrétienne, P., editors (1992). *Éléments d'algorithmique*. Masson - (<http://www-igm.univ-mlv.fr/~berstel/Elements.html>), Paris.
- [Cook, 1971] Cook, S. A. (1971). The complexity of theorem-proving procedures. In Harrison, M. A., Banerji, R. B., and Ullman, J. D., editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM.
- [Cori et al., 2001] Cori, R., Hanrot, G., and Steyaert, J.-M. (2001). Conception et analyse des algorithmes. Technical report, Ecole polytechnique.
- [Cormen et al., 1994] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. (1994). *Introduction à l'algorithmique (traduit par Xavier Cazin)*. Dunod.
- [Cormen et al., 2001] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to Algorithms, Second Edition*. The MIT Press.
- [Garey and Jonhson, 1979] Garey, M. and Jonhson, D. (1979). *Computers and Intractability : A guide to the theory of NP-Completeness*. W.H. Freeman.
- [Gaudel et al., 1990] Gaudel, M.-C., Froidevaux, C., and Soria, M. (1990). *Types de données et algorithmes*. McGraw-Hill, Paris.
- [Levin, 1973] Levin, L. A. (1973). Universal search problems (? ??????????????????????????). *Problems of Information Transmission* (? ??????????????????????????????????????), 9(3).
- [Papadimitriou, 1995] Papadimitriou, C. (1995). *Computational Complexity*. Addison-Wesley.
- [Papadimitriou et al., 2006] Papadimitriou, C. H., Dasgupta, S., and Vazirani, U. (2006). *Algorithms*. McGraw Hill.
- [Prins, 1994] Prins, C. (1994). *Algorithmes de graphes*. Eyrolles.
- [Robert et al., 2005] Robert, Y., Caniou, Y., and Thierry, E. (2005). Algorithmique - cours et travaux dirigés. Technical report, Ecole Normale Supérieure de Lyon.

[Sedgewick and Flajolet, 1996] Sedgewick, R. and Flajolet, P. (1996). *Introduction à l'analyse des algorithmes*. Thomson Publishing.

[Skiena, 2008] Skiena, S. S. (2008). *The Algorithm Design Manual*. Springer.