Enseignement de l'informatique

POLYCOPIE

Algorithmique Avancée et Complexité

LEBBAH Yahia, GHOMARI AbdelGhani
Chargés de cours, Université d'Oran

Département Informatique, Faculté des Science, Université d'Oran Es-Senia
B.P. 1524, El-M'Naouar Oran, Algérie

Version du 30 janvier 2010

Table des matières

Avant-propos

Algorithmique et Complexité

...

But de cet ouvrage

...

Chapitre 1

Introduction

Une des premières références historiques sur les algorithmes date du neuvième siècle, à Baghdad, 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., 2006]

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

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...$$

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 version exponentielle

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 :

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

Ceci veut dire que le nombre de pas croit aussi vite que la suite de Fibonacci. Comme la suite croit exponentiellement, $T(n)$ 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 comme ordinateur, l'ordinateur le plus puissant à l'heure actuelle, le NEC Earth Simulator, qui exécute 40 mille milliards 40×10^{12} instructions par seconde. Nous aurons besoin sur cet ordinateur de 2^{92} 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 $F(101)$, ... 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.

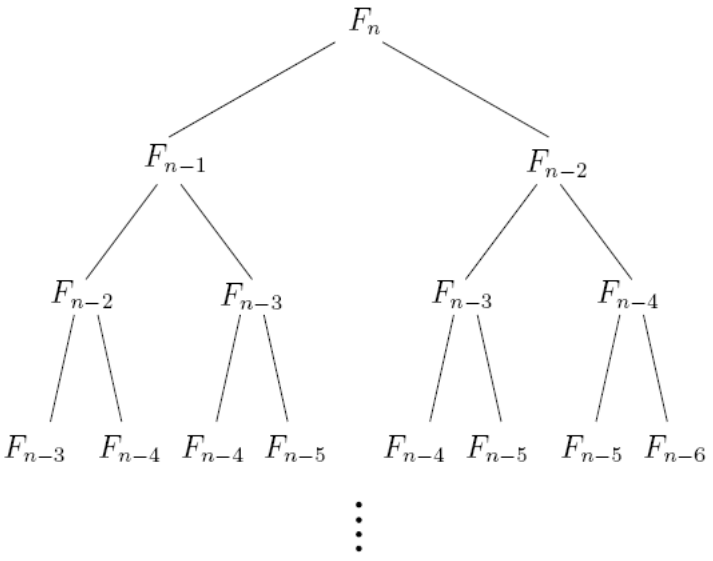


FIGURE 1.1 – Trace de FIB1(n) [Papadimitriou et al., 2006]

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.

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

Algorithm 2 FIB2(n)

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

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 Algorithmes [Cormen et al., 1994, 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 aboderons un problème posé fréquement 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 $\langle a_1, a_2, ..., a_n \rangle$.

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

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

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

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.

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

Algorithm 3 TRI-INSERTION(A)

Input: Une séquence de n nombres $A = \langle a_1, a_2, ..., a_n \rangle$.
Output: Une permutation $\langle a'_1, a'_2, ..., a'_n \rangle$ de la permutation de la suite d'entrée telle que $a_1 \leq a_2 \leq ... \leq a_n$.

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

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. On procédera comme suit :

Initialisation La propriété Invariant 1 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.

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 cas i, $i < j$ 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.

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 nombres d'arêtes. Pour chaque problème étudié, nous préciserons la 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 :

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

Si la suite est déjà ordonnée, on a

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) = (c_1 + c_2 + c_4 + c_5 + c_8)n + (c_2 + c_4 + c_5 + c_8).$$

On peut exprimer cette dernière formule sous la forme $an + b$, où a et b sont des constantes.

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 $j = 2, 3, ..., n$. 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

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n(n+1)/2 - 1) + c_6(n(n-1)/2) + c_7(n(n-1)/2) + c_8(n-1)$$

$$T(n) = (c_5/2 + c_6/2 + c_7/2)n^2 + (c_1 + c_2 + c_4 + c_5/2 + c_6/2 + c_7/2 + c_8)n.$$

Ce dernier coût est de la forme $an^2 + bn + c$ où a, b et c sont des constantes. 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., 2001]

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 illustation 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.

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.

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

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

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

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

```
1: if p < r then
2:    q ← ⌊(p + r)/2⌋
3:    TRI-FUSION(A, p, q)
4:    TRI-FUSION(A, q + 1, r)
5:    FUSIONNER(A, p, q, r)
6: end if
```

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 $anlog(n) + bn + c$, où a, b, et c sont des constantes. On notera cette forme $\Theta(nlog(n))$. On peut aussi remarquer que ce coût est nettement inférieur à celui du tri par insertion, car $\exists n_0, \forall n, n > n0, n^2 > nlog(n)$.

Chapitre 2

Principes de l'analyse des algorithmes

2.1 Mesure du coût [Beauquier et al., 1992, Cormen et al., 1994, 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 $(a_1, ..., a_n)$ 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 doit 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 :

1. Lorsque les opérations sont dans une séquence d'instructions, leurs nombres s'ajoutent.
2. 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.
3. 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 ième itération. Si le nombre d'itérations est difficile à calculer, on peut se contenter d'une bonne majoration.
4. 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.
5. 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.

2.1.1 La complexité dans le meilleur des cas

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

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

2.1.2 La complexité dans le pire des cas

Le coût $Max_A(n)$ d'un algorithme A dans le cas le plus défavorable ou dans le cas le pire[1] 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 sur les données de taille n est par définition

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

1. "worst-case" en anglais.

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 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.

2.2 Grandeurs des fonctions et notations de Landau : O, ω, ... [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 connaitre 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 $n > 8$. Quelles que soient les constantes multiplicatives k_1 et k_2 telles que $M_1(n) = k_1 n^2$ et $M_2(n) = k_2 n$, 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 $lim_{n\to\infty} g(n)/f(n) = 0$).

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

La Figure 2.1 met en évidence la différence de rapidité de croissance de certaines fonctions usuelles : les ordres de grandeur asymptotique des fonctions $1, log_2(n), nlog_2(n), n^2, n^3, 2^n$ vont en croissant strictement ; ces fonctions forment une échelle de comparaison.

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'odre de grandeur asymptotique des fonctions, on a l'habitude d'utiliser la notion suivante :

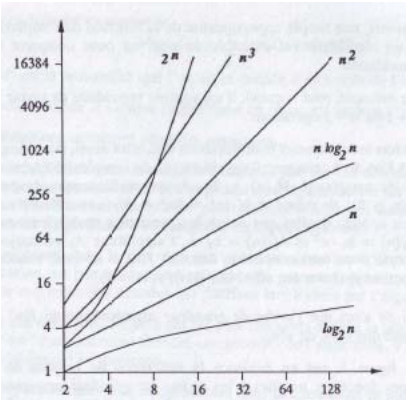


FIGURE 2.1 – Croissance de certaines fonctions usuelles [Gaudel et al., 1990]

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 aussi que

$$g = \Omega(f)$$

si et seulement si $f = 0(g)$.

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, mais elle n'est pas suffisante pour comparer entre elles les performances de différents algorithmes, car il faut connaitre 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).

On est donc amené à introduire la définition suivante :

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 $f = 0(g)$ 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 $\Theta(n^2)$. 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 $\Theta(nlog(n))$.

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 aymptotiques 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 sept 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), $log_2(n)$ (c'est-à-dire une fonction polynomiale d'ordre 3), 2^n (c'est-à-dire une fonction exponentielle).

Complexité / Taille	1	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
$n = 10^2$	$\simeq 1\mu s$	6,6 μs	0,1 ms	0,6 ms	10 ms	1 s	4×10^{16} a
$n = 10^3$	$\simeq 1\mu s$	9,9 μs	1 ms	9,9 ms	1 s	16,6 mn	∞
$n = 10^4$	$\simeq 1\mu s$	13,3 μs	10 ms	0,1 s	100 s	11,5 j	∞
$n = 10^5$	$\simeq 1\mu s$	16,6 μs	0,1 s	1,6 s	2,7 h	31,7 a	∞
$n = 10^6$	$\simeq 1\mu s$	19,9 μs	1 s	19,9 s	11,5 j	$31,7 \times 10^3$ a	∞

FIGURE 2.2 – Temps d'exécution et taille des données [Gaudel et al., 1990]

Le tableau de la Figure 2.2 donne une estimation du temps d'exéction 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.

Le tableau de la Figure 2.3 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 0^6 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) ;
- logatithmique (par exemple la recherche dichotomique, ou les arbres binaires de recherche) ;
- linéaire (par exemple la recherche séquentielle) ;
- $n.log(n)$ (par exemple les bones 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 2.3 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 / 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 2.3 – Temps d'exécution et taille des données [Gaudel et al., 1990]

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×t	(10+ε)×t	100×t	1000×t	t^{10}
Evolution de la taille quand le temps est multiplé par 10	∞	n^{10}	10×n	(10−ε)×n	3,16×n	2,15×n	n+3,32

FIGURE 2.4 – 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.

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

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

2.2.2 Exemple : Recherche linéaire

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

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 probablité 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 apparait à la $i$ième 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

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

Taille → / Complexité ↓	20	50	100	200	500	1 000
$10^3.n$	0.02 s	0.05 s	0.1 s	0.2 s	0.5 s	1 s
$10^3.n.\log_2 n$	0.09 s	0.3 s	0.6 s	1.5 s	4.5 s	10 s
$100.n^2$	0.04 s	0.25 s	1 s	4 s	25 s	2 mn
$10.n^3$	0.02 s	1 s	10 s	1 mn	21 mn	27 h
$n^{\log_2 n}$	0.4 s	1.1 h	220 jours	12500 ans	5.10^{10} ans	--
$2^{n/3}$	0.0001 s	0.1 s	2.7 h	3.10^6 ans	--	--
2^n	1 s	36 ans	--	--	--	--
3^n	58 mn	2.10^{11} ans	--	--	--	--
$n!$	77 100 ans	--	--	--	--	--

FIGURE 2.5 – Temps d'exécution et taille des données [Prins, 1994]

Algorithm 5 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 ← 1 to n do
2:   for j ← 1 to n do
3:     c[i, j] ← 0
4:     for k ← 1 to n do
5:       c[i, j] ← c[i, j] + a[i, j] ∗ b[k, j]
6:     end for
7:   end for
8: end for
```

Algorithm 6 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 ← 1
2: while (j ≤ n) ∧ (L[j] ≠ X) do
3:   j ← j + 1
4: end while
5: if (j>n) then
6:   j ← 0
7: end if
```

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 connaitre 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 connait 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 connaitre 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 établit que la résolution du problème de la multiplication de deux matrices $n \times n$ nécessite au moins n^2 multiplications. Cependant, on ne connait 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 7 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 ← L[1]
2: for j ← 2 to n do
3:    if L[j] > M then
4:       M ← 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

$$Min_A(n) = Moy_A(n) = Max_A(n) = n - 1.$$

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' identité 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 :

$C_N = (1+\frac{1}{N}C_{N-1}+2$ pour $N>1$ avec $C_1 = 2$.
$C_N = C_{\lfloor N/2 \rfloor} + C_{\lceil N/2 \rceil} + N$ pour $N>1$ et $C_1 = 0$.
$C_N = N+1+\frac{1}{N}\sum_{1\leq j\leq N}(C_{j-1}+C_{N-j})$ pour $N>0$ avec $C_0 = 0$

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

$F_n = F_{n-1} + F_{n-2}$ pour $n>1$ avec $F_0 = 0$ 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	$a_n = f(a_{n-1}, a_{n-2}, ..., a_{n-t})$
complète	$a_n = n + a_{n-1} + a_{n-2}...a_1)$
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 équivanlente 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

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

La tableau de la Figure 2.6 montre quelques sommes discrètes fréquemment rencontrées.

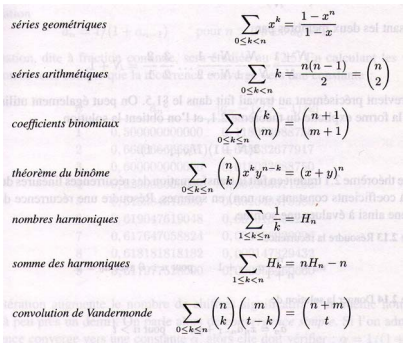


FIGURE 2.6 – Sommes discrètes élémentaires [Sedgewick and Flajolet, 1996]

Théorème 1 (Récurrences linéaires du premier ordre) *La récurrence*

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

pour $n > 0$ *avec* $a_0 0$*, a pour solution explicite*

$$a_n = y_n + \sum_{1 \leq j < n} y_j x_{j+1} x_{j+2} ... x_n.$$

Preuve
En divisant les deux membres par $x_n x_{n-1} ... x_1$ puis en itérant, il vient :

$$a_n = x_n x_{n-1} ... x_1 \sum_{1 \leq j \leq n} \frac{y_j}{x_j x_{j-1} ... x_1}$$

$$a_n = y_n + \sum_{1 \leq j \leq n} y_j x_{j+1} x_{j+2} ... x_n.$$

□

Le théorème 1 est un cas typique de la technique par changement de variables. Des changements de variables plus complexes peuvent être développés pour résoudre les récurrences. Par exemple, soit la récurrence

$$a_n = \sqrt{a_{n-1} a_{n-2}}$$

pour $n > 1$ avec $a_0 = 1$ et $a_1 = 2$. Si on effectue le changement de variable $b_n = \log a_n$, alors b_n vérifie

$$b_n = 1/2(b_{n-1} + b_{n-2})$$

pour $n > 1$ avec $b_0 = 0$ et $a_1 = 1$. i.e. une récurrence linéaire à coefficients constants.

2.4.2 Récurrences non linéaire du premier ordre [Sedgewick and Flajolet, 1996]

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 2.7.

n	a_n	$\lvert a_n - (\sqrt{5}-1)/2 \rvert$
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 2.7 – [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 procédési 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.

2.4.3 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 $a_{n-1}, a_{n-2}, a_{n-3}, ..., a_0$. 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 sufffit 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 2 (Récurrences linéaires à coefficients constants) *Les solutions de la récurrence*

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

pour $n \geq t0$, sont des combinaisons linéaires (dont les coefficients dépendent des conditions initiales $a_0, a_1, ..., a_{t-1}$) de termes de la forme $n^j \beta^n$ où β est une racine du polynôme caractéristique

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

et j est tel que $0 \leq j \leq v$ si β a pour multiplicité v.

Preuve Voir [Sedgewick and Flajolet, 1996].

□

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 sophistiquée. Le théorème 2 n'est plus applicable. Ici, on pourrait faire appel aux séries génératrices introduites dans la section 2.4.6.

Certaines récurrence 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 $a_n = n!F_n$.

2.4.4 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 répertoire, résolution par Rebouclage (bootstraping), résolution par perturbation, etc. Elles sont pour la plupart décrites dans la référence [Sedgewick and Flajolet, 1996]. Dans la suite, on va s'intéresser aux récurrences de type diviser pour régner. Par la suite, on développement la technique générale de résolution des récurrences en passant par les séries génératrices.

2.4.5 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 2.8 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 2.8 – [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 3 (Fonctions "diviser pour régner"[Sedgewick and Flajolet, 1996]) *Si la fonction $a(x)$ vérifie la récurrence*

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

pour $x > 1$ avec $a(x) = 0$ pour $x \leq 1$. alors

si $\alpha < \beta$ *alors* $a(x) \sim \frac{\beta}{\beta-\alpha}x$
si $\alpha = \beta$ *alors* $a(x) \sim x \log_\beta x$
si $\alpha > \beta$ *alors* $a(x) \sim \frac{\alpha}{\alpha-\beta}(\frac{\beta}{\alpha})^{\log_\beta \alpha} x^{\log_\beta \alpha}$

2.4.6 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 3 *Soit $a_0, a_1, ..., a_k, ...$ 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 ordinaire exponentielle (SGE) de cette suite. Le coefficient a_k est également noté $k![z^k]A(z)$.

Le tableau de la Figure 2.9 (resp. de la Figure 2.11) 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 2.10 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 2.12 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 $a_0, a_1, ..., a_k, ...$ Bien que ici, on se consacre uniquement aux noyaux z^k et $z^k/k!$. D'autres noyaux apparaissent parfois dans la littérature.

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

$$a(z) = \frac{z}{1-5z+6z^2} = \frac{z}{(1-3z)(1-2z)} = \frac{1}{1-3z} - \frac{1}{1-2z}$$

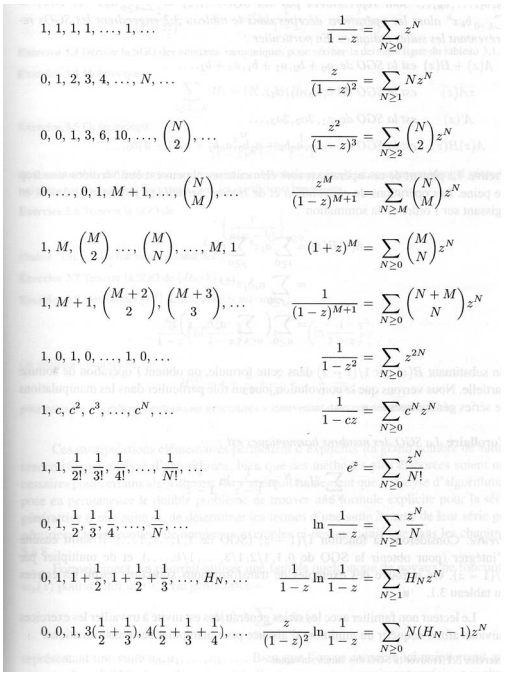
par conséquent $a_n = 3^n - 2^n$.

Les SGOs (resp. SGEs) ont plusieurs propriétés mathématiques que l'on pourrait exploiter pour raisonner sur les SGOs en vue de résoudre les récurrences.

Théorème 5 (SGOs de récurrence linéaire) *Si a_n vérifie la récurrence*

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

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 $g(z) = 1 - x_1 z - x_2 z^2 - ... - x_t z^t$ et le polynôme numérateur est déterminé par les conditions initiales $a_0, a_1, ..., a_{t-1}$.



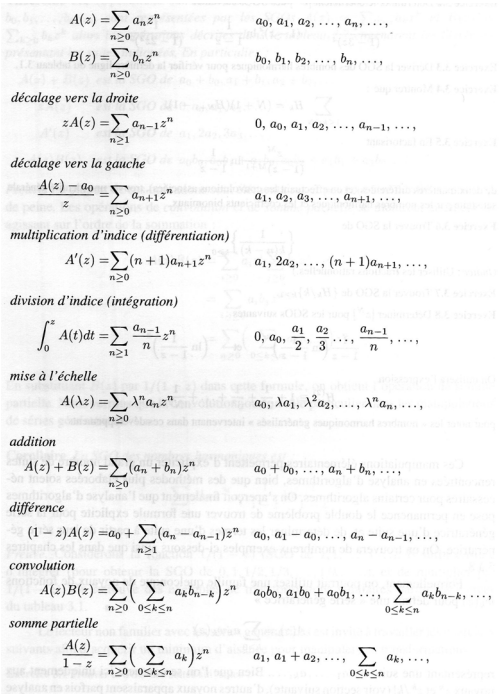
$1, 1, 1, 1, \ldots, 1, \ldots$	$\frac{1}{1-z} = \sum_{N\geq 0} z^N$
$0, 1, 2, 3, 4, \ldots, N, \ldots$	$\frac{z}{(1-z)^2} = \sum_{N\geq 1} N z^N$
$0, 0, 1, 3, 6, 10, \ldots, \binom{N}{2}, \ldots$	$\frac{z^2}{(1-z)^3} = \sum_{N\geq 2} \binom{N}{2} z^N$
$0, \ldots, 0, 1, M+1, \ldots, \binom{N}{M}, \ldots$	$\frac{z^M}{(1-z)^{M+1}} = \sum_{N\geq M} \binom{N}{M} z^N$
$1, M, \binom{M}{2} \ldots, \binom{M}{N}, \ldots, M, 1$	$(1+z)^M = \sum_{N\geq 0} \binom{M}{N} z^N$
$1, M+1, \binom{M+2}{2}, \binom{M+3}{3}, \ldots$	$\frac{1}{(1-z)^{M+1}} = \sum_{N\geq 0} \binom{N+M}{N} z^N$
$1, 0, 1, 0, \ldots, 1, 0, \ldots$	$\frac{1}{1-z^2} = \sum_{N\geq 0} z^{2N}$
$1, c, c^2, c^3, \ldots, c^N, \ldots$	$\frac{1}{1-cz} = \sum_{N\geq 0} c^N z^N$
$1, 1, \frac{1}{2!}, \frac{1}{3!}, \frac{1}{4!}, \ldots, \frac{1}{N!}, \ldots$	$e^z = \sum_{N\geq 0} \frac{z^N}{N!}$
$0, 1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \ldots, \frac{1}{N}, \ldots$	$\ln \frac{1}{1-z} = \sum_{N\geq 1} \frac{z^N}{N}$
$0, 1, 1+\frac{1}{2}, 1+\frac{1}{2}+\frac{1}{3}, \ldots, H_N, \ldots$	$\frac{1}{1-z} \ln \frac{1}{1-z} = \sum_{N\geq 1} H_N z^N$
$0, 0, 1, 3(\frac{1}{2}+\frac{1}{3}), 4(\frac{1}{2}+\frac{1}{3}+\frac{1}{4}), \ldots$	$\frac{z}{(1-z)^2} \ln \frac{1}{1-z} = \sum_{N\geq 0} N(H_N - 1) z^N$

FIGURE 2.9 – [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.



$$A(z) = \sum_{n\geq 0} a_n z^n \qquad a_0, a_1, a_2, \ldots, a_n, \ldots,$$

$$B(z) = \sum_{n\geq 0} b_n z^n \qquad b_0, b_1, b_2, \ldots, b_n, \ldots,$$

décalage vers la droite

$$zA(z) = \sum_{n\geq 1} a_{n-1} z^n \qquad 0, a_0, a_1, a_2, \ldots, a_{n-1}, \ldots,$$

décalage vers la gauche

$$\frac{A(z) - a_0}{z} = \sum_{n\geq 0} a_{n+1} z^n \qquad a_1, a_2, a_3, \ldots, a_{n+1}, \ldots,$$

multiplication d'indice (différentiation)

$$A'(z) = \sum_{n\geq 0} (n+1) a_{n+1} z^n \qquad a_1, 2a_2, \ldots, (n+1)a_{n+1}, \ldots,$$

division d'indice (intégration)

$$\int_0^z A(t)dt = \sum_{n\geq 1} \frac{a_{n-1}}{n} z^n \qquad 0, a_0, \frac{a_1}{2}, \frac{a_2}{3} \ldots, \frac{a_{n-1}}{n}, \ldots,$$

mise à l'échelle

$$A(\lambda z) = \sum_{n\geq 0} \lambda^n a_n z^n \qquad a_0, \lambda a_1, \lambda^2 a_2, \ldots, \lambda^n a_n, \ldots,$$

addition

$$A(z) + B(z) = \sum_{n\geq 0} (a_n + b_n) z^n \qquad a_0 + b_0, \ldots, a_n + b_n, \ldots,$$

différence

$$(1-z)A(z) = a_0 + \sum_{n\geq 1} (a_n - a_{n-1}) z^n \qquad a_0, a_1 - a_0, \ldots, a_n - a_{n-1}, \ldots,$$

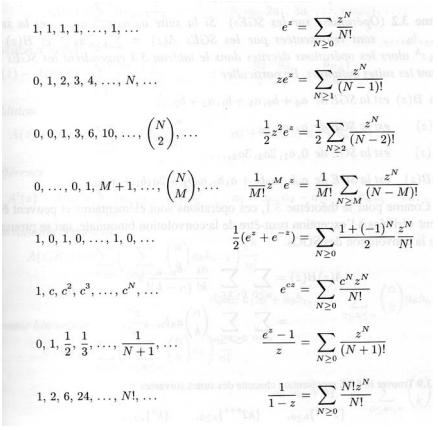
convolution

$$A(z)B(z) = \sum_{n\geq 0} \Big(\sum_{0\leq k\leq n} a_k b_{n-k} \Big) z^n \qquad a_0 b_0, a_1 b_0 + a_0 b_1, \ldots, \sum_{0\leq k\leq n} a_k b_{n-k}, \ldots,$$

somme partielle

$$\frac{A(z)}{1-z} = \sum_{n\geq 0} \Big(\sum_{0\leq k\leq n} a_k \Big) z^n \qquad a_1, a_1 + a_2, \ldots, \sum_{0\leq k\leq n} a_k, \ldots,$$

FIGURE 2.10 – [Sedgewick and Flajolet, 1996]



$$1, 1, 1, 1, \ldots, 1, \ldots \qquad e^z = \sum_{N\geq 0} \frac{z^N}{N!}$$

$$0, 1, 2, 3, 4, \ldots, N, \ldots \qquad ze^z = \sum_{N\geq 1} \frac{z^N}{(N-1)!}$$

$$0, 0, 1, 3, 6, 10, \ldots, \binom{N}{2}, \ldots \qquad \frac{1}{2} z^2 e^z = \frac{1}{2} \sum_{N\geq 2} \frac{z^N}{(N-2)!}$$

$$0, \ldots, 0, 1, M+1, \ldots, \binom{N}{M}, \ldots \qquad \frac{1}{M!} z^M e^z = \frac{1}{M!} \sum_{N\geq M} \frac{z^N}{(N-M)!}$$

$$1, 0, 1, 0, \ldots, 1, 0, \ldots \qquad \frac{1}{2}(e^z + e^{-z}) = \sum_{N\geq 0} \frac{1 + (-1)^N}{2} \frac{z^N}{N!}$$

$$1, c, c^2, c^3, \ldots, c^N, \ldots \qquad e^{cz} = \sum_{N\geq 0} \frac{c^N z^N}{N!}$$

$$0, 1, \frac{1}{2}, \frac{1}{3}, \ldots, \frac{1}{N+1}, \ldots \qquad \frac{e^z - 1}{z} = \sum_{N\geq 0} \frac{z^N}{(N+1)!}$$

$$1, 2, 6, 24, \ldots, N!, \ldots \qquad \frac{1}{1-z} = \sum_{N\geq 0} \frac{N! z^N}{N!}$$

FIGURE 2.11 – [Sedgewick and Flajolet, 1996]

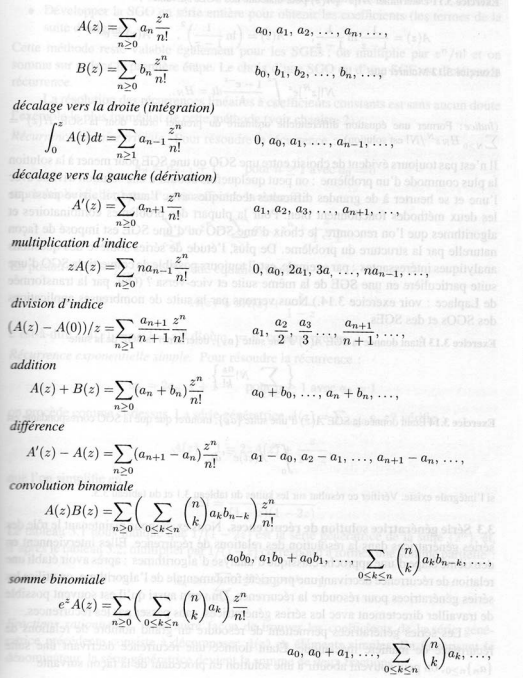


FIGURE 2.12 – [Sedgewick and Flajolet, 1996]

Les tas

La structure de tas (binaire) est un object tabulé qui peut être vu comme un arbre binaire presque complet (voir la Figure 2.13).

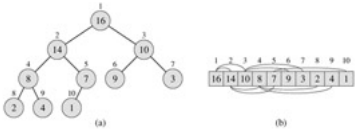


FIGURE 2.13 – [Cormen et al., 1994, Cormen et al., 2001]

Les tas nécessitent trois opérations de base :

PERE(i)
    retourner $\lfloor i/2 \rfloor$
GAUCHE(i)
    retourner $2i$
DROIT(i)
    retourner $2i + 1$

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

$$A[Pere[i]] \geq A[i] \tag{2.1}$$

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. On peut démontrer que la hauteur d'un tas de n éléments est égale à $\Theta(\log n)$, car le tas est un arbre binaire complet (exercice).

A chaque étape, on détermine le plus grand des éléments $A[i]$, $A[Gauche(i)]$, et $A[Droit(i)]$, et son indice est rangé dans max. Si $A[i]$ est le plus grand, alors le sous-arbre enraciné au noeud i est

Algorithm 8 ENTASSER(A)

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 ← GAUCHE(i)
2: r ← DROIT(i)
3: if (l ≤ taille[A]) ∧ (A[l] > A[i]) then
4:    max ← l
5: else
6:    max ← i
7: end if
8: if (r ≤ taille[A]) ∧ (A[r] > A[max]) then
9:    max ← r
10: end if
11: if (max ≠ i) then
12:    Echanger A[i] ↔ A[max]
13:    ENTASSER(A, max)
14: end if
```

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 sur un sous-arbre de taille n en un noeud i peut s'écrire avec la récurrence

$$T(n) \leq T(2n/3) + \Theta(1).$$

La solution à cette équation de partition est $O(\log n)$. (Exercice !)

Algorithm 9 CONSTRUIRETAS(A)

Input: Un tableau A
Output: Construire le tas de A.

```
1: taille[A] ← longueur[A]
2: for i ← ⌊longueur[A]/2⌋ à 1 do
3:    ENTASSER(A, i)
4: end for
```

Comme la procédure ENTASSER est en $O(\log n)$, et il existe $O(n)$ appels, alors le temps d'exécution est au plus $O(n \log n)$. Cette borne supérieure, quoique correcte, n'est pas approchée asymptotiquement.

On peut démontrer que le temps d'exécution est en $O(n)$ (voir [Cormen et al., 1994, Cormen et al., 2001]).

L'algorithme 15 du tri par tas commence par se servir de CONSTRUIRETAS pour construire un tas sur le tableau d'entrée $A[1..n]$, où $n = longueur[A]$. 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 $taille[A]$), on observe que $A[1..(n-1)]$ peut facilement être transformé en tas, en relancant 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.

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 cl. Une file de priorité supporte les opérations

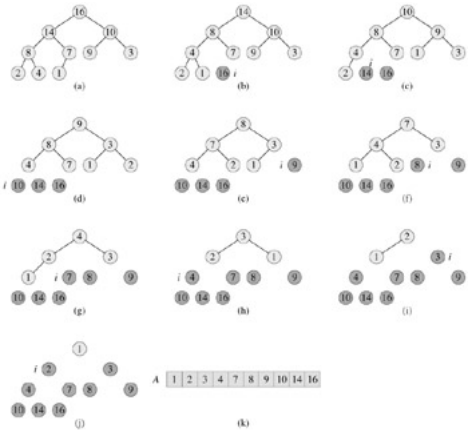


FIGURE 2.14 – [Cormen et al., 1994, Cormen et al., 2001]

Algorithm 10 TRITAS(A)

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

```
1: ConstruireTas(A)
2: for i ← longueur[A] à 2 do
3:     Echanger A[1] ↔ A[i]
4:     taille[A] ← taille[A] − 1
5:     Entasser(A, 1)
6: end for
```

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 11 INSÉRER(S, x)

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

```
1: taille[A] ← taille[A] + 1
2: i ← taille[A]
3: while (i > 1) ∧ (A[Pere(i)] < cl) do
4:     A[i] ← A[Pere(i)]
5:     i ← Pere(i)
6: end while
```

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 $A[q+1..r]$.

Algorithm 12 MAXIMUM(S)

Input: ensemble S
Output: Retourne l'élément de S ayant la plus grande clé.
1: **return** $A[1]$

Algorithm 13 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

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 14 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: $q \leftarrow$ PARTITIONNER(A, P, R)
3: TRIRAPIDE(A, p, q)
4: TRIRAPIDE($A, q+1, r$)
5: **end if**

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

Dans le pire des cas, le partionnement 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 :

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

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 $\Theta(n \log n)$.

Le tri rapide a une complexité en moyenne $\Theta(n \log n)$. (exercice en cours)

Algorithm 15 PARTIOTIONNER(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 $A[p..q]$ et $A[q+1..r]$ soient triés.

```
1: x ← A[p]
2: i ← p − 1
3: j ← r + 1
4: while true do
5:    repeat
6:       j ← j − 1
7:    until (A[j] ≤ x)
8:    repeat
9:       i ← i + 1
10:   until (A[i] ≥ x)
11:   if (i < j) then
12:      échanger A[i] ↔ A[j]
13:   else
14:      return j
15:   end if
16: end while
```

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 dans le pire des cas avec la notation Θ.

2.6.2 Algorithme de tri

Exprimer le tri par insertion d'une façon récursive : pour trier un tableau $A[1..n]$ l'algorithme tri récursivement $A[1..n-1]$, puis place au bon endroit l'élément $A[n]$ dans le tableau trié de $A[1..n-1]$. Formuler l'équation de récurrence exprimant le coût de cet algorithme.

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 $\Theta(nlog(n))$ résolvant ce problème.

2.6.4 Récurrence

Dans le cas où n est une puissance exacte de 2, démontrer par induction que la solution de l'équation de récurrence

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

est $T(n) = nlog(n)$.

2.6.5 Grandeurs des fonctions

Exprimer $n^3/1000 - 100n^2 - 100n + 3$ avec la notation Θ. Les deux identités suivantes, sont elles correctes ?

$$2^{n+1} = O(2^n)$$
$$2^{2n} = O(2^n)$$

2.6.6 Extension à deux paramètres

Etant donnée une fonction $g(n, m)$, on dénote par $O(g(n, m))$ l'ensemble des fonctions

$$O(g(n,m)) = \{f(n,m) : \begin{array}{l}\text{il existe les constantes } c, n_0, \text{ et } m_0 \\ \text{telles que } 0 \leq f(n,m) \leq cg(n,m) \\ \text{pour tout } n \geq n_0 \text{ et } m \geq m_0\}.\end{array}$$

Définissez les fonctions correspondantes à $\Theta(g(n, m))$ et $\Omega(g(n, m))$.

2.7 Travaux dirigés II

2.7.1 Notations asymptotiques

Démontrer que pour tout a, b réels, où $b > 0$ que

$$(n + a)^n = \theta(n^b).$$

2.7.2 Récurrences linéaires

Résoudre la récurrence

$$C_N = (1 + 1/N)C_{n-1} + 2,$$

pour $n > 1$ avec $C_1 = 2$.

2.7.3 Récurrences d'ordre supérieur

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. Etudier les solutions de la récurrence

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

   pour $n \geq 3$ avec $a_0 = c_0$, $a_1 = c_1$, $a_2 = c_2$.
3. Résoudre la récurrence

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

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

2.7.4 Séries génératrices

1. Résoudre la récurrence

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

   pour $n \geq 1$ avec $a_0 = 0$.
2. Résoudre la récurrence

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

   pour $n \geq 1$ avec $a_0 = 1$.
3. Résoudre la récurrence

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

   pour $n > 1$ avec $a_0 = 0$, $a_1 = 1$.
4. Résoudre la récurrence

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

   pour $n > 1$ avec $a_0 = 1$, $a_1 = 1$, $a_2 = 1$.
5. Résoudre la récurrence

   $$a_n = 5a_{n-1} - 8a_{n-2} + 4a_{n-3}$$

   pour $n > 2$ avec $a_0 = 0$, $a_1 = 1$, $a_2 = 4$.

2.7.5 Tri par tas

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. Démonter 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 $\Omega(n \log n)$.
7. Illustrer les opérations ExtraireMax et Inserer sur le tas $< 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 >$.

2.7.6 Tri rapide

1. Quelle est la complexité du tri rapide si tous ses éléments sont égaux ?

2.8 Mini projets

Référence des mini projets : [Cormen et al., 1994, Cormen et al., 2001].
Chaque étudiant doit choisir un thème algorithmique parmi les thèmes donnés ci-dessous.

Théorie des nombres et cryptographie

A1 Test de primalité

A2 Factorisation des entiers

A3 L'algorithme RSA

Recherche de motifs

B1 L'algorithme naïf

B2 Algorithme de Rabin-Karp

B3 Recherche de motifs au moyen d'automates finis

B4 Algorithme de Knutt-Morris-Pratt

B5 Algorithme de Boyer-Moore

Géométrie algorithmique

C1 Proriétés des ségments de droites

C2 Déterminer une intersection dans un ensemble de ségments

C3 Recherche d'une enveloppe convexe

C4 Recherche des deux points les plus rapprochés

L'étudiant doit remettre un rapport décrivant les algorithmes du thème choisi, et plus particulièrement :

- Fonctionnement des algorithmes : Le rapport doit décrire les algorithmes avec un pseudo code, en le faisant comprendre avec un texte dédié. Cette partie doit aussi faire comprendre les algorithmes sur un jeu simple de données.
- Analyse de la correction des algorithmes : Le rapport doit démontrer que les algorithmes donnés sont corrects, en définissant si nécessaire les assertions logiques (e.g., invariant, pré-condition, post-condition, ...).
- Analyse de la complexité des algorithmes : Une analyse de coût est exigée en démontrant les complexités obtenues. L'analyse dans le pire des cas est incontournable. L'analyse en moyenne est souhaitable si l'étudiant trouve les références nécessaires.
- Démonstration logicielle sur les mises en oeuvre des algorithmes : Cette partie est une partie technique. L'étudiant doit faire une recherche pour repérer un environnement de programmation qui exploite les algorithmes étudiés. Le rapport doit donc comprendre une description de l'environnement repréré. L'étudiant a aussi la possibilité de coder les algorithmes.

Chapitre 3

Structures de données

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

Piles

PileVide(P)

Empiler(P, x)

Dépiler$(\mathbf{P})$

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 $O(1)$.

DIRECT-ADDRESS-SEARCH(T, k)
  return $T[k]$

DIRECT-ADDRESS-INSERT(T, x)
  $T[key[x]] \leftarrow x$

DIRECT-ADDRESS-DELETE(T, x)
  $T[key[x]] \leftarrow$ NIL

Figure 3.1 – Algorithmes pour les dictionnaires (scan de [Cormen et al., 2001])

CHAINED-HASH-INSERT(T, x)
insert x at the head of list $T[h(key[x])]$

CHAINED-HASH-SEARCH(T, k)
search for an element with key k in list $T[h(k)]$

CHAINED-HASH-DELETE(T, x)
delete x from the list $T[h(key[x])]$

FIGURE 3.2 – Algorithmes pour les dictionnaires (scan de [Cormen et al., 2001])

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

FIGURE 3.3 – Algorithmes pour les dictionnaires (scan de [Cormen et al., 2001])

```
HASH-SEARCH(T, k)
1  i ← 0
2  repeat  j ← h(k, i)
3          if T[j] = k
4            then return j
5          i ← i + 1
6    until T[j] = NIL or i = m
7  return NIL
```

FIGURE 3.4 – Algorithmes pour les dictionnaires (scan de [Cormen et al., 2001])

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 $U = \{0, 1, ..., m-1\}$, où m n'est pas trop grand. On supposera que deux éléments ne peuvent pas partager la même clé.

On utilise un tablea $T[0, 1, ..., m-1]$, dans lequel chaque position, ou alvéol, correspond à une clé dans l'univers U. (page 216). Chaque position de $T[0, 1, ..., m-1]$ (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 $T[k] = nil$. Le problème de cet adressage est évident : si l'univers U est grand !

L'ensemble des opérations RechercheAdressageDirect(T, k), InsererAdressageDirect(T, x),Supprime est rapide : le temps est en $O(1)$. (page 245)

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 $T[0..m-1]$.

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

On dit qu'un élément de clé k est haché dans l'alvéole $h(k)$; $h(k)$ est aussi dite valeur de hachage (page 218).

L'inconvénient de cette idée est que deux clés peuvent être hachées dans la même alvéole - ce qu'on appelle collision. Eviter complétement 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 chainée (page 219).

Les opérations de dictionnaire sur une table de hachage T sont : (page 219)

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.

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 chaine. 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 raisonable : chaque clé a la même chance d'être haché dans toute alvéole. 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 2 *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

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

Par exemple, supposons que les clés soient des nombres réels k aléatoires, répartis indépendamment et uniformément dans l'intervelle $0 \leq k < 1$. 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 $\{0, 1, 2, ...\}$. Par exemple, une clé sous forme de chaine 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 mlvéoles 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 $n = 2000$ chaines, 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

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

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 fonction de hachage est de la forme

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

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

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

soit une permutation de $< 0, 1, ..., m-1 >$, 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.

InsérerHachage(T, k)

RechercherHachage(T, k)

La suppression se fait en marquant 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 $i = 0, 1, ..., m-1$. 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 $i = 0, 1, ..., m-1$..

Le double hachage

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

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

Proposition 3 *Etant donnée une table de hachage en adressage ouvert avec un facteur de remplissage $\alpha = n/m < 1$, 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].

3.3 Arbres binaires [Cormen et al., 1994, Cormen et al., 2001]

Les arbres de recherche sont des structures de données pouvant supporter nombre d'opérations sur les ensembles dynamiques : Rechercher, Minimum, Maximum, Predecesseur, Successeur, Inserer, et Supprimer.

Pour un arbre binaire complet à n noeuds, ces opérations s'exécutent en $\Theta(\log n)$ dans le pire des cas, alors qu'une recherche est en $\Theta(n)$. Nous verrons que la hauteur d'un arbre binaire de recherche construit aléatoirement est $O(\log n)$ qui est à l'origine de la complexité en $O(\log n)$.

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2     then INORDER-TREE-WALK(left[x])
3          print key[x]
4          INORDER-TREE-WALK(right[x])
```

Figure 3.5 – Algorithmes pour les arbres binaires (scan de [Cormen et al., 2001])

Définition 4 *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 $cl[x] \leq cl[y]$.*

ArbreRechercher(x, k) +
ArbreRechercherItérative(x, k) +
ArbreMinimum(x) +

```
TREE-SEARCH(x, k)
1  if x = NIL or k = key[x]
2     then return x
3  if k < key[x]
4     then return TREE-SEARCH(left[x], k)
5     else  return TREE-SEARCH(right[x], k)
```

FIGURE 3.6 – Algorithmes pour les arbres binaires (scan de [Cormen et al., 2001])

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ key[x]
2        do if k < key[x]
3              then x ← left[x]
4              else  x ← right[x]
5  return x
```

FIGURE 3.7 – Algorithmes pour les arbres binaires (scan de [Cormen et al., 2001])

```
TREE-MINIMUM(x)
1  while left[x] ≠ NIL
2        do x ← left[x]
3  return x
```

FIGURE 3.8 – Algorithmes pour les arbres binaires (scan de [Cormen et al., 2001])

```
TREE-SUCCESSOR(x)
1  if right[x] ≠ NIL
2     then return TREE-MINIMUM(right[x])
3  y ← p[x]
4  while y ≠ NIL and x = right[y]
5        do x ← y
6           y ← p[y]
7  return y
```

FIGURE 3.9 – Algorithmes pour les arbres binaires (scan de [Cormen et al., 2001])

```
TREE-INSERT(T, z)
 1  y ← NIL
 2  x ← root[T]
 3  while x ≠ NIL
 4        do y ← x
 5           if key[z] < key[x]
 6             then x ← left[x]
 7             else  x ← right[x]
 8  p[z] ← y
 9  if y = NIL
10     then root[T] ← z                ▷ Tree T was empty
11     else  if key[z] < key[y]
12              then left[y] ← z
13              else  right[y] ← z
```

FIGURE 3.10 – Algorithmes pour les arbres binaires (scan de [Cormen et al., 2001])

```
TREE-DELETE(T, z)
 1  if left[z] = NIL or right[z] = NIL
 2     then y ← z
 3     else  y ← TREE-SUCCESSOR(z)
 4  if left[y] ≠ NIL
 5     then x ← left[y]
 6     else  x ← right[y]
 7  if x ≠ NIL
 8     then p[x] ← p[y]
 9  if p[y] = NIL
10     then root[T] ← x
11     else  if y = left[p[y]]
12              then left[p[y]] ← x
13              else  right[p[y]] ← x
14  if y ≠ z
15     then key[z] ← key[y]
16          copy y's satellite data into z
17  return y
```

FIGURE 3.11 – Algorithmes pour les arbres binaires (scan de [Cormen et al., 2001])

ARBREMAXIMUM(x) +

ARBREINSÉRER(T, z) +

ARBRESUPPRIMER(T, z) +

Proposition 4 *Toutes les opérations citées ci-haut sont en $O(h)$, où h est la hauteur de l'arbre binaire.*

3.4 Arbres rouge et noir [Cormen et al., 1994, Cormen et al., 2001]

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. chaque feuille NIL est noire.
3. si un noeud est rouge, alors ses deux fils sont noirs,
4. chaque chemin simple reliant un noeud à une famille descendante contient le même nombre de noeud noirs.

Proposition 5 *La hauteur moyenne d'un arbre binaire rouge et noir a une hauteur au plus égale à $O(2\log n + 1)$.*

```
LEFT-ROTATE(T, x)
 1  y ← right[x]            ▷ Set y.
 2  right[x] ← left[y]      ▷ Turn y's left subtree into x's right subtree.
 3  if left[y] ≠ nil[T]
 4     then p[left[y]] ← x
 5  p[y] ← p[x]             ▷ Link x's parent to y.
 6  if p[x] = nil[T]
 7     then root[T] ← y
 8     else if x = left[p[x]]
 9             then left[p[x]] ← y
10             else right[p[x]] ← y
11  left[y] ← x             ▷ Put x on y's left.
12  p[x] ← y
```

FIGURE 3.12 – Algorithmes pour les arbres rouge et noir (scan de [Cormen et al., 2001])

ROTATIONGAUCHE(T, x) -

RNINSÉRER(T, x) -

RNSUPPRIMER(T, z) -

RNSUPPRIMERCORRECTION(T, x) -

3.5 Extension d'une structure de données [Cormen et al., 1994, Cormen et al., 2001]

Micro-projet !

```
RB-INSERT(T, z)
 1  y ← nil[T]
 2  x ← root[T]
 3  while x ≠ nil[T]
 4        do y ← x
 5           if key[z] < key[x]
 6              then x ← left[x]
 7              else  x ← right[x]
 8  p[z] ← y
 9  if y = nil[T]
10     then root[T] ← z
11     else  if key[z] < key[y]
12              then left[y] ← z
13              else  right[y] ← z
14  left[z] ← nil[T]
15  right[z] ← nil[T]
16  color[z] ← RED
17  RB-INSERT-FIXUP(T, z)
```

FIGURE 3.13 – Algorithmes pour les arbres rouge et noir (scan de [Cormen et al., 2001])

```
RB-INSERT-FIXUP(T, z)
 1  while color[p[z]] = RED
 2      do if p[z] = left[p[p[z]]]
 3            then y ← right[p[p[z]]]
 4                 if color[y] = RED
 5                   then color[p[z]] ← BLACK                    ▷ Case 1
 6                        color[y] ← BLACK                       ▷ Case 1
 7                        color[p[p[z]]] ← RED                   ▷ Case 1
 8                        z ← p[p[z]]                            ▷ Case 1
 9                   else  if z = right[p[z]]
10                           then z ← p[z]                       ▷ Case 2
11                                LEFT-ROTATE(T, z)              ▷ Case 2
12                        color[p[z]] ← BLACK                    ▷ Case 3
13                        color[p[p[z]]] ← RED                   ▷ Case 3
14                        RIGHT-ROTATE(T, p[p[z]])               ▷ Case 3
15            else  (same as then clause
                     with "right" and "left" exchanged)
16  color[root[T]] ← BLACK
```

FIGURE 3.14 – Algorithmes pour les arbres rouge et noir (scan de [Cormen et al., 2001])

```
RB-DELETE(T, z)
 1  if left[z] = nil[T] or right[z] = nil[T]
 2     then y ← z
 3     else y ← TREE-SUCCESSOR(z)
 4  if left[y] ≠ nil[T]
 5     then x ← left[y]
 6     else x ← right[y]
 7  p[x] ← p[y]
 8  if p[y] = nil[T]
 9     then root[T] ← x
10     else if y = left[p[y]]
11             then left[p[y]] ← x
12             else right[p[y]] ← x
13  if y ≠ z
14     then key[z] ← key[y]
15          copy y's satellite data into z
16  if color[y] = BLACK
17     then RB-DELETE-FIXUP(T, x)
18  return y
```

FIGURE 3.15 – Algorithmes pour les arbres rouge et noir (scan de [Cormen et al., 2001])

```
RB-DELETE-FIXUP(T, x)
 1  while x ≠ root[T] and color[x] = BLACK
 2        do if x = left[p[x]]
 3              then w ← right[p[x]]
 4                   if color[w] = RED
 5                     then color[w] ← BLACK                          ▷ Case 1
 6                          color[p[x]] ← RED                         ▷ Case 1
 7                          LEFT-ROTATE(T, p[x])                      ▷ Case 1
 8                          w ← right[p[x]]                           ▷ Case 1
 9                   if color[left[w]] = BLACK and color[right[w]] = BLACK
10                     then color[w] ← RED                            ▷ Case 2
11                          x ← p[x]                                  ▷ Case 2
12                     else if color[right[w]] = BLACK
13                             then color[left[w]] ← BLACK            ▷ Case 3
14                                  color[w] ← RED                    ▷ Case 3
15                                  RIGHT-ROTATE(T, w)                ▷ Case 3
16                                  w ← right[p[x]]                   ▷ Case 3
17                          color[w] ← color[p[x]]                    ▷ Case 4
18                          color[p[x]] ← BLACK                       ▷ Case 4
19                          color[right[w]] ← BLACK                   ▷ Case 4
20                          LEFT-ROTATE(T, p[x])                      ▷ Case 4
21                          x ← root[T]                               ▷ Case 4
22              else (same as then clause with "right" and "left" exchanged)
23  color[x] ← BLACK
```

FIGURE 3.16 – Algorithmes pour les arbres rouge et noir (scan de [Cormen et al., 2001])

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.

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 conservés par le noeud x,
   (b) les $n[x]$ clés elles-mêmes, stockées par ordre croissant : $cl_1[x] \leq cl_1[x] \leq ... \leq cl_n[x]$, et
   (c) $feuille[x]$, une valeur booléenne qui vaut vrai si x est une feuille et faux si x est un noeud interne.
2. si x est un noeud interne, il contient également $n[x]+1$ pointeurs $c_1[x], c_2[x], ..., c_{n[x]+1}[x]$ sur ses fils. Les feuilles n'ont aucun fils, et leurs champs c_i ne sont donc pas définis.
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
$$k_1 \leq cl_1[x] \leq k_2 \leq cl_2[x] \leq ... \leq cl_{n[x]}[x] \leq k_{n[x]+1}$$
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.

Théorème 6 *Si $n \geq 1$, alors pour un B-arbre T à n clés, de hauteur h et de degré minimum $t \geq 2$,*
$$h \leq log_t(n+1)/2.$$

```
B-TREE-SEARCH(x, k)
1  i ← 1
2  while i ≤ n[x] and k > key_i[x]
3        do i ← i + 1
4  if i ≤ n[x] and k = key_i[x]
5     then return (x, i)
6  if leaf[x]
7     then return NIL
8     else DISK-READ(c_i[x])
9          return B-TREE-SEARCH(c_i[x], k)
```

FIGURE 3.17 – Algorithmes pour les B-arbres (scan de [Cormen et al., 2001])

BARBRERECHERCHER(x, k) +
CRÉERBARBRE(T) -
BARBREDÉCOUPERFILS(x, i, y) -
BARBREINSÉRER(T, k) -
BARBREINSÉRERINCOMPLET(x, k) -
SUPPRESSION D'UNE CLÉ DANS B-ARBRE -

```
B-TREE-CREATE(T)
1  x ← ALLOCATE-NODE()
2  leaf[x] ← TRUE
3  n[x] ← 0
4  DISK-WRITE(x)
5  root[T] ← x
```

FIGURE 3.18 – Algorithmes pour les B-arbres (scan de [Cormen et al., 2001])

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 noveau 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.

Les structures de données utilisées ici supportent 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(amorti)
CréerTas	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
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	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

Opérations de base :

MinimumTasBinomial(T) -

LienBinomial(y, z) -

UnionTasBinomiaux(T_1, T_2) -

TasBinomiauxInsérer(T, x) -

TasBinomiauxExtraireMin(T) -

TasBinomiauxDiminuerClé(T, x, k) -

TasBinomiauxSupprimer(T, x) -

ACMTasFusionnable(G) -

3.8 Tas de Fibonacci [Cormen et al., 1994, Cormen et al., 2001]

Micro-projet !

Les tas de Fibonacci bénéficient de bornes meilleures que les tas binomiaux pour certaines opérations, avec une analyse amortie. Besoin de notions sur l'analyse amortie ...

TasFibInsérer(T, x) -
UnionTasFib(T_1, T_2) -
TasFibExtraireMin(T) -
TasFibConsolider$(T$ -
TasFibRelier(T, y, x) -
TasFibDiminuerClé(T, x, k) -
TasFibCouper(T, x, y) -
TasFibCoupeEnCascade(T, y) -
TasFibSupprimer(T, x) -
TasFibCoupeEnCascade(T, y) -

3.9 Structures de données pour les ensembles disjoints

Micro-projet !

3.10 Travaux dirigés II

3.10.1 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 ?

3.10.2 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.

3.10.3 Tables de hachage

Supposons qu'on utilise une fonction de hachage aléatoire h pour hacher n clés distinctes dans un tableau T de longueur m. Quel est le nombre attendu de collisions ? Plus précisément, quel est le cardinal attendu de
$$\{(x, y)|h(x) = h(y)\} \quad ?$$

3.10.4 Fonction de hachage

On considère une variante de la méthode de la division dans laquelle $h(k) = k \mod m$, où $m = 2^p - 1$ et k est une chaine de caractères interprétée en base 2^p. Montrer que si la chaine x peut être déduite de la chaine y par permutation de ses caractères, alors x et y ont même valeur de hachage. Donner un exemple d'application pour laquelle cette propriété de la fonction de hachage serait indésirable.

Chapitre 4

Conception avancée et techniques d'analyse

4.1 Programmation dynamique [Cormen et al., 1994, Cormen et al., 20 Papadimitriou et al., 2006]

La programmation dynamique, comme la méthode diviser pour reigner, 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 reigner 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 solition 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 développement d'un algorithme de programmation dynamique peut être planifié dans une séquence de quatre étapes :

1. Caractériser la structure d'une solution optimale.
2. Définir récursivement la valeur d'une solution optimale.
3. Calculer la valeur d'une solution optimale en remontant progressivement jusqu'à l'énoncé du problème initial.
4. Construire une solution optimale pour les informations calculées.

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 $(A_1, ..., A_n)$ de n matrices à multiplier, et qu'on souhaite calculer le produit

$$A_1 A_2 ... A_n.$$

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 $(A_1, ..., A_4)$, 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] ≠ rows[B]
2     then error "incompatible dimensions"
3     else  for i ← 1 to rows[A]
4                 do for j ← 1 to columns[B]
5                       do C[i, j] ← 0
6                          for k ← 1 to columns[A]
7                              do C[i, j] ← C[i, j] + A[i, k] · B[k, j]
8           return C
```

FIGURE 4.1 – 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. $((A_1 A_2)A_3)$ nécessite 7500 multiplications. $A_1(A_2 A_3)$ nécessite 75 000 ,multiplications. Le premier parenthésage est 10 fois plus rapide !

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 $A_{i..j} = A_i A_{i+1} ... A_j$. 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.

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 optimale aux sous-problèmes. Ici, on prend comme sous-problèmes les problèmes consistant à déterminer le coût minimum d'un parenthésage de $A_i ... A_j$. 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 coût optimaux Ici, on écrit un algorithme récursif, basé sur la solution récursive donnée précédemment. L'observation importante

ORDONNERCHAINEDEMATRICESRÉCURSIF(p)

MULTIPLIERCHAINEDEMATRICE(A, s, i, j)

L'algorithme ORDONNERCHAINEDEMATRICESRÉCURSIF est de complexité $\Omega(2^n)$.

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
1  if i = j
2     then return 0
3  m[i, j] ← ∞
4  for k ← i to j − 1
5        do q ← RECURSIVE-MATRIX-CHAIN(p, i, k)
                + 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] ← q
8  return m[i, j]
```

FIGURE 4.2 – Solution récursive exponentielle (scan de [Cormen et al., 2001])

```
MEMOIZED-MATRIX-CHAIN(p)
1  n ← length[p] − 1
2  for i ← 1 to n
3       do for j ← i to n
4             do m[i, j] ← ∞
5  return LOOKUP-CHAIN(p, 1, n)

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

FIGURE 4.3 – Solution récursive polynomiale par programmation dynamique (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 ORDONNERCHAINEDEMATRICESRÉCURSIF 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.

RecensementChaineDeMatrices(p)

RecupérationChaînes(p, i, j)

On peut démontrer que cet algorithme est en $O(n^2)$.

4.1.2 Eléments de programmation dynamique

Sous-structure optimale

Sous-problèmes superposés

Recensement

4.1.3 Plus longue sous-séquence commune

Micro-projet

4.1.4 Triangulation optimale de polygones

Micro-projet

4.2 Algorithmes gloutons [Cormen et al., 1994, Cormen et al., 2001, Papadimitriou et al., 2006]

Les algorithmes servant à résoudre les problèmes d'optimisation parcourent 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 $S = \{1, 2, ..., n\}$ 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.

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 :

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

ChoixActivitéGlouton(s, f)

La Figure 4.5 illustre le fonctionnement de cet algorithme glouton.

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

FIGURE 4.4 – Solution gloutonne pour l'ordonnancement (scan de [Cormen et al., 2001])

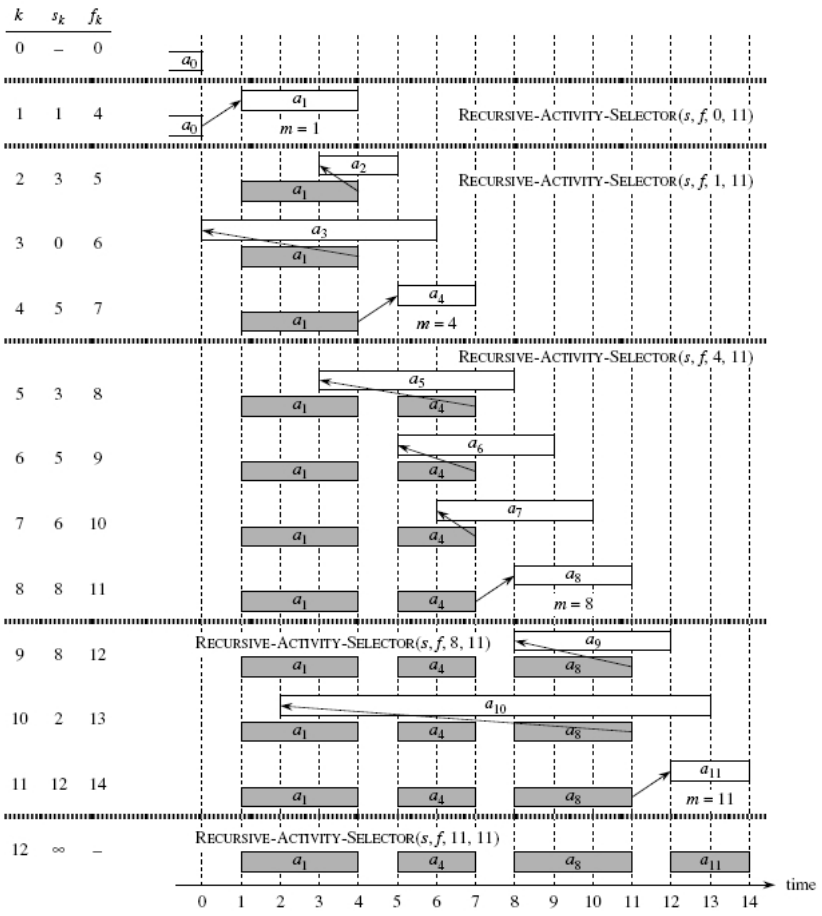


FIGURE 4.5 – Choix glouton des activités (scan de [Cormen et al., 2001])

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 apparaitre 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.

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 chaine 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 :

	a	*b*	*c*	*d*	*e*	*f*
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 considére 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 concacténer 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 chaine 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". Etant 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 16 montre les étapes du codage de Huffman. La Figure 4.6 illustre clairement le fonctionnement de l'algorithme de Huffman.

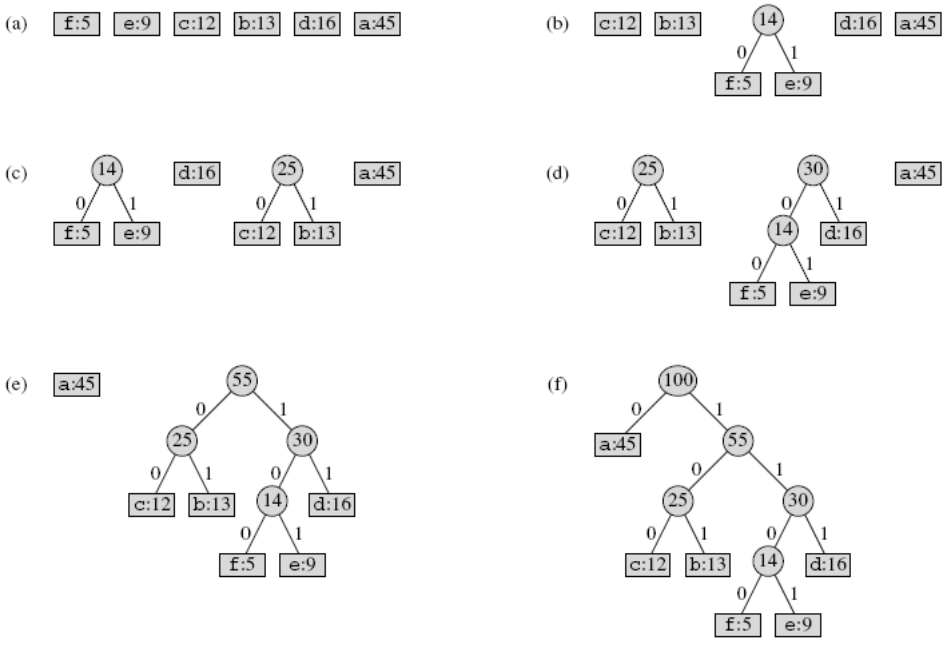


FIGURE 4.6 – Codage de Huffman (scan de [Cormen et al., 2001])

Algorithm 16 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: $y \leftarrow droit[z] \leftarrow ExtraireMax(F)$
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 6 *La procédure de Huffman produit un codage préfixe optimal.*

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

4.2.3 Fondements théoriques des méthodes gloutonnes : matroïdes

Projet !

4.3 Analyse amortie [Cormen et al., 1994, Cormen et al., 2001]

Projet !

Chapitre 5

Algorithmes sur les graphes

Voir [Lebbah Yahia, 2009]. On prend comme "exercices" la preuve de correction et de complexité des algorithmes introduits.

Chapitre 6

Différents sujets algorithmiques

6.1 Automates [Cormen et al., 1994, Cormen et al., 2001, Beauquier et

Micro-projet !

6.2 Motifs [Cormen et al., 1994, Cormen et al., 2001, Beauquier et al.,

Micro-projet !

6.3 FFT [Cormen et al., 1994, Cormen et al., 2001]

Micro-projet !

6.4 Algorithmes de la théorie des nombres [Cormen et al., 1994, Cormen et al., 2001, Papadimitriou et al., 2006]

Micro-projet !

6.5 Géométrie algorithmique [Cormen et al., 1994, Cormen et al., 2001 Beauquier et al., 1992]

Micro-projet !

6.6 Algorithmes approchés [Cormen et al., 1994, Cormen et al., 2001, Prins, 1994]

Micro-projet !

6.7 Aspects sur la théorie de la complexité [Cormen et al., 1994, Cormen et al., 2001, Prins, 1994, Papadimitriou, 1995, Papadimitri

Projet !

Chapitre 7

Annexes

7.1 Ecriture des algorithmes

Nous avons les trois écritures :
Alternatives

if B **then**
$P_1; P_2; ...; P_k$
end if

et

if B **then**
$P_1; P_2; ...; P_k$
else
$Q_1; Q_2; ...; Q_l$
end if

et

if B_1 **then**
$P1_1; P1_2; ...; P1_k$
else if B_2 **then**
$P2_1; P2_2; ...; P2_l$ ...
else
$Pm_1; Pm_2; ...; Pm_l$
end if

Itérations bornées

for $i = 1$ to n **do**
$P_1; P_2; ...; P_k$
end for

ou aussi

for $i = n$ downto 1 **do**
$P_1; P_2; ...; P_k$
end for

Itérations non bornées

while B **do**
$P_1; P_2; ...; P_k$
end while

ou aussi

repeat
$P_1; P_2; ...; P_k$
until B

Bibliographie

[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/Elements.html), Paris.

[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.

[Gaudel et al., 1990] Gaudel, M.-C., Froidevaux, C., and Soria, M. (1990). *Types de données et algorithmes*. McGraw-Hill, Paris.

[Lebbah Yahia, 2009] Lebbah Yahia, Belalem Ghelem, N. O. E. K. (2009). Introduction aux algorithmes de la théorie des graphes. Technical report, Support de cours, Université d'Oran Es-Sénia.

[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.

[Sedgewick and Flajolet, 1996] Sedgewick, R. and Flajolet, P. (1996). *Introduction à l'analyse des algorithmes*. Thomson Publishing.