Résumé du cours d'algorithmique
Licence 3 informatique (Semestre 5)
Université de Lorraine

Sylvain Contassot-Vivier
D'après une base de cours de Dieter Kratsch

Table des matières

1 Introduction **3**
1.1 Description d'un algorithme . . . 3
1.2 Qu'est-ce qu'un algorithme ? . . . 3
1.3 Notion d'efficacité . . . 3
1.4 Le temps . . . 3
1.5 Analyse du temps d'un algorithme . . . 4
1.6 Notation $\mathcal{O}()$ . . . 4
1.7 Notation $\Omega()$ . . . 4
1.8 Notation $\Theta()$ . . . 4
1.9 Notations $o()$ et $\omega()$ . . . 5

2 Théorème Maître **6**
2.1 Diviser pour régner . . . 6
2.2 Énoncé du théorème . . . 6
2.3 Preuve du théorème pour les puissances de b . . . 7
2.4 Extension de la preuve aux autres valeurs de b . . . 7
2.5 Autres remarques sur le théorème maître . . . 8
2.5.1 Autre notation . . . 8
2.5.2 Décompositions entières . . . 8
2.5.3 Décompositions non régulières . . . 8

3 Tables de hachage **9**
3.1 Structures de données . . . 9
3.2 Application . . . 9
3.2.1 Problématique . . . 9
3.3 Table à adressage direct . . . 9
3.4 Table de hachage . . . 10
3.5 Hachage avec chaînage . . . 10
3.5.1 Analyse du temps moyen . . . 10
3.6 Fonctions de hachage . . . 10
3.6.1 Méthode de la division . . . 11
3.6.2 Méthode de la multiplication . . . 11
3.7 Hachage avec adressage ouvert . . . 11
3.7.1 Hachage uniforme . . . 12
3.7.2 Analyse du temps . . . 12

4 Rappels sur les arbres **13**
4.1 Arbres binaires de recherche . . . 13
4.2 AVL . . . 13
4.3 Tas . . . 13
4.3.1 Définition et construction . . . 13
4.3.2 Tri par tas . . . 14

4.3.3 Files de priorité . . . 15

5 Arbres rouge et noir 16
5.1 Arbres binaires de recherche . . . 16
5.2 Arbres binaires de recherche équilibrés . . . 16
5.3 Arbres rouge et noir . . . 17
5.4 Hauteur d'un arbre rouge et noir . . . 17
5.5 Modifications des arbres rouge et noir . . . 17
5.5.1 Insertion . . . 18
5.5.2 Suppression . . . 19

6 Stratégie gloutonne 21
6.1 Problème d'optimisation . . . 21
6.2 Classes de problèmes . . . 21
6.3 Algorithme glouton . . . 21
6.4 Application au choix d'activités . . . 22
6.4.1 Algorithme glouton pour le choix d'activités . . . 22
6.4.2 Analyse du temps de l'algorithme glouton obtenu . . . 22
6.4.3 Démonstration de validité d'un algorithme glouton . . . 23
6.4.4 Application au problème du choix des activités . . . 23
6.5 Caractéristiques nécessaires à la stratégie gloutonne . . . 23
6.6 Problème du sac-à-dos . . . 23

7 Programmation dynamique 24
7.1 Intérêt de la programmation dynamique . . . 24
7.2 Conception d'un algorithme de programmation dynamique . . . 24
7.3 Multiplication d'une suite de matrices . . . 24
7.3.1 Formalisation du problème d'optimisation . . . 24
7.4 Application de la programmation dynamique . . . 25
7.4.1 Identifier la structure d'une solution optimale . . . 25
7.4.2 Valeur d'une solution optimale sous forme récursive . . . 25
7.4.3 Algorithme de calcul des coûts optimaux . . . 25
7.4.4 Construction de la solution optimale . . . 25
7.5 Analyse du temps . . . 26

Chapitre 1

Introduction

Ce cours suit les grandes lignes du livre suivant :

```
«Introduction à l'algorithmique»
T.H. Cormen, C.E. Leiserson, R.L. Rivest.
Éditions Dunod, Paris, 1994. (MIT Press 1990).
```

1.1 Description d'un algorithme

Un algorithme est l'expression de la résolution d'un problème sous une forme calculable par un ordinateur.

Énoncé du problème → Algorithme universel → Programme lié à un langage

1.2 Qu'est-ce qu'un algorithme ?

Définition 1. *Un **algorithme** est une procédure de calcul bien définie, qui prend en entrée un ensemble de valeurs (éventuellement vide), et qui produit en sortie un ensemble de valeurs (éventuellement vide). Un algorithme est donc une séquence d'étapes de calcul permettant de passer de l'ensemble des entrées (données) à l'ensemble des sorties (résultats).*

1.3 Notion d'efficacité

Un algorithme sera dit ***plus efficace*** qu'un autre s'il a un coût temporel et/ou mémoire inférieur.

Définition 2. *Un **algorithme** est **optimal** selon un certain coût s'il atteint la borne théorique minimale de ce coût pour le problème traité.*

1.4 Le temps

Le ***temps d'exécution*** d'un algorithme sur une entrée particulière est donné par le ***nombre d'étapes*** ou ***d'opérations élémentaires*** exécutées.
On va donc décrire le ***temps d'exécution en fonction de la taille de l'entrée***.

1.5 Analyse du temps d'un algorithme

On utilise souvent le temps dans le ***pire cas***. C'est le temps d'exécution le plus long pour une ***entrée quelconque de taille fixée*** (n,...).

Définition 3. *Un algorithme A s'exécute en **temps** $f(n)$ **au pire cas** si son temps d'exécution pour une entrée quelconque de taille n est au plus $f(n)$.*

Il est important de pouvoir ***comparer l'évolution des temps d'exécution*** de différents algorithmes selon la taille de l'entrée afin d'en déduire le(s) plus efficace(s).

Dans la suite, nous considérons des fonctions $f, g : \mathbb{N} \to \mathbb{R}^+$, où le paramètre donne la taille de l'entrée et le résultat est le temps d'exécution (dans une unité quelconque).

1.6 Notation $\mathcal{O}(\,)$

Définition 4. *$\mathcal{O}(g(n))$ est l'ensemble des fonctions $f(n)$ pour lesquelles il existe des constantes $c > 0$ et $n_0 > 0$ telles que $0 \leq f(n) \leq c.g(n),\ \forall n \geq n_0$.*
On écrit alors $f(n) = \mathcal{O}(g(n))$.

1.7 Notation $\Omega(\,)$

Définition 5. *$\Omega(g(n))$ est l'ensemble des fonctions $f(n)$ pour lesquelles il existe des constantes $c > 0$ et $n_0 > 0$ telles que $0 \leq c.g(n) \leq f(n),\ \forall n \geq n_0$.*
On écrit alors $f(n) = \Omega(g(n))$.

1.8 Notation $\Theta(\,)$

Définition 6. *$\Theta(g(n))$ est l'ensemble des fonctions $f(n)$ pour lesquelles il existe des constantes $c_1 > 0$, $c_2 > 0$ et $n_0 > 0$ telles que $c_1.g(n) \leq f(n) \leq c_2.g(n),\ \forall n \geq n_0$.*
On écrit alors $f(n) = \Theta(g(n))$.

On donne en Figure 1.1 une illustration des trois notations.

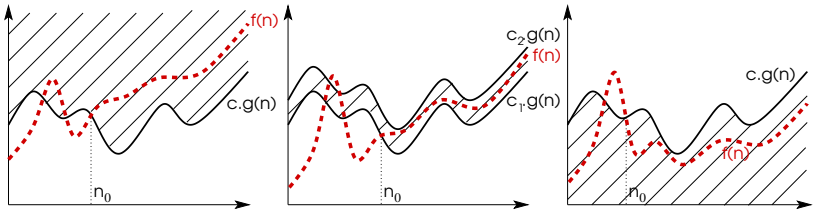


Figure 1.1 – Ensembles $\Omega(g(n))$, $\Theta(g(n))$ et $\mathcal{O}(g(n))$ de gauche à droite.

1.9 Notations $o(\,)$ et $\omega(\,)$

Définition 7. *$o(g(n))$ est l'ensemble des fonctions $f(n)$ pour lesquelles $\forall c > 0$, il existe une constante $n_0 > 0$ telle que $0 \leq f(n) < c.g(n),\ \forall n \geq n_0$.*
On écrit alors $f(n) = o(g(n))$.

Définition 8. *$\omega(g(n))$ est l'ensemble des fonctions $f(n)$ pour lesquelles $\forall c > 0$, il existe une constante $n_0 > 0$ telle que $0 \leq c.g(n) < f(n),\ \forall n \geq n_0$.*
On écrit alors $f(n) = \omega(g(n))$.

Finalement, nous pouvons en déduire ***quelques règles utiles*** pour la comparaison de fonctions :

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

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \in \mathbb{R}^+ \Leftrightarrow f(n) = \mathcal{O}(g(n))$$

Pour simplifier la comparaison de f et g, on peut utiliser la règle de l'hôpital :

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

Chapitre 2

Théorème Maître

Il établit une ***complexité globale*** à partir d'une complexité exprimée sous la ***forme récursive*** suivante :

$$T(n) = a.T\left(\frac{n}{b}\right) + f(n) \quad \text{avec} \quad a \geq 1,\ b > 1 \text{ et } f : \mathbb{N} \to \mathbb{R}^+$$

2.1 Diviser pour régner

La méthode de diviser pour régner ***(DPR)*** nécessite trois étapes pour obtenir un algorithme récursif :

- **Diviser :** diviser un problème de taille n en a problèmes de taille $\frac{n}{b}$.
- **Régner :** résoudre récursivement les a sous-problèmes de taille $\frac{n}{b}$, chacun en temps $T\left(\frac{n}{b}\right)$.
- **Combiner :** combiner les résultats des sous-problèmes (de même niveau) pour obtenir le résultat final.

2.2 Énoncé du théorème

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

$$T(n) = a.T\left(\frac{n}{b}\right) + f(n) \quad \text{où} \quad \frac{n}{b} \text{ est } \left\lceil \frac{n}{b} \right\rceil \text{ ou } \left\lfloor \frac{n}{b} \right\rfloor$$

Alors $T(n)$ peut être borné asymptotiquement selon les expressions suivantes de $f(n)$:

1. *Si $f(n) = \mathcal{O}\left(n^{log_b(a)-\epsilon}\right)$ pour une constante $\epsilon > 0$, alors :*
$$T(n) = \Theta\left(n^{log_b(a)}\right)$$
2. *Si $f(n) = \Theta\left(n^{log_b(a)}\right)$, alors :*
$$T(n) = \Theta\left(n^{log_b(a)}.log_2(n)\right)$$
3. *Si $f(n) = \Omega\left(n^{log_b(a)+\epsilon}\right)$ pour une constante $\epsilon > 0$, et si $a.f\left(\frac{n}{b}\right) \leq c.f(n)$ pour une constante $c < 1$ et n suffisamment grand, alors :*
$$T(n) = \Theta\left(f(n)\right)$$

On voit que le théorème maître repose sur la comparaison polynomiale de $f(n)$ avec $n^{log_b(a)}$. En effet, on a les correspondances suivantes :

1. Cas où $f(n)$ est polynomialement inférieure à $n^{log_b(a)}$ (facteur n^ϵ).
2. Cas où $f(n)$ est polynomialement égale à $n^{log_b(a)}$
3. Cas où $f(n)$ est polynomialement supérieure à $n^{log_b(a)}$ (facteur n^ϵ).

Il est important de noter que ce théorème n'est ***pas toujours applicable*** et comporte donc certains ***cas non recouverts*** :

2.3 Preuve du théorème pour les puissances de b

L'analyse de la récurrence $T(n) = a.T\left(\frac{n}{b}\right) + f(n)$, avec n une puissance de b ($n = b^k$), s'appuie sur l'***arbre récursif*** qui en découle, donné en Figure 2.1.

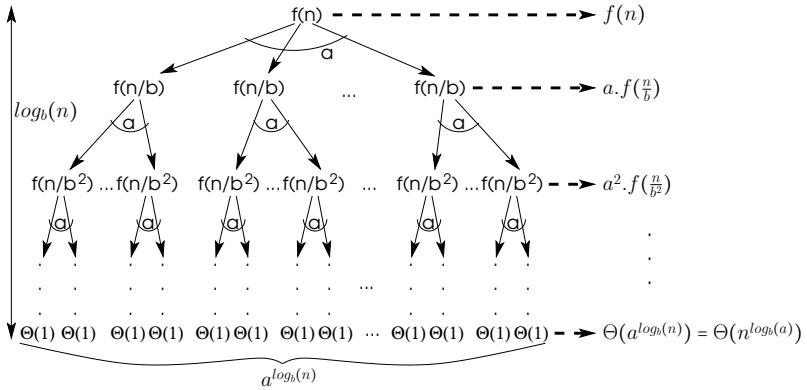


Figure 2.1 – Arbre récursif généré par $T(n)$.

Si l'on fait la somme des termes à droite, on obtient le coût total :

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

Les trois cas du théorème correspondent aux trois répartitions possibles des coûts des feuilles et de nœuds dans le coût total de l'arbre :

1. Le coût est dominé par les feuilles
2. Le coût est équilibré entre les nœuds et les feuilles
3. Le coût est dominé par les nœuds (les plus hauts)

2.4 Extension de la preuve aux autres valeurs de b

Pour avoir une preuve complète, il nous faut maintenant considérer les deux cas suivants :

$$T(n) = a.T\left(\left\lceil \frac{n}{b} \right\rceil\right) + f(n) \tag{2.1}$$

$$T(n) = a.T\left(\left\lfloor \frac{n}{b} \right\rfloor\right) + f(n) \tag{2.2}$$

En fait, on peut établir un théorème qui donne l'équivalence entre :

$$T(n) = a.T\left(\left\lceil \frac{n}{b} \right\rceil\right) + n^c$$

définie sur les entiers positifs et

$$t(x) = a.T\left(\frac{x}{b}\right) + x^c$$

définie sur les réels positifs, pour a, b, c des constantes. Ainsi, on a $T(n) = \Theta(t(x))$, et cela est également vrai avec $\lfloor\rfloor$.

2.5 Autres remarques sur le théorème maître

2.5.1 Autre notation

Une autre formulation du théorème maître fait apparaître explicitement la forme de $f()$, avec les constantes positives a, b, c :

$$T(n) = a.T\left(\frac{n}{b}\right) + \mathcal{O}(n^c)$$

On a alors :

- Si $c < log_b(a)$, alors $T(n) = \Theta(n^{log_b(a)})$
- Si $c = log_b(a)$, alors $T(n) = \Theta(n^c.log_2(n))$
- Si $c > log_b(a)$, alors $T(n) = \Theta(n^c)$

2.5.2 Décompositions entières

Le théorème maître couvre également les cas de récurrence de la forme :

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + f(n)$$

Il suffit de constater que

$$2.T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + f(n) \leq T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + f(n) \leq 2.T\left(\left\lceil \frac{n}{2} \right\rceil\right) + f(n)$$

$T(n)$ est donc encadré par deux formulations qui sont toutes les deux en $\Theta\left(2.T\left(\frac{n}{2}\right) + f(n)\right)$.

2.5.3 Décompositions non régulières

Le théorème maître peut être étendu aux récurrences avec une division en a sous-problèmes de tailles différentes :

$$T(n) = \sum_{i=1}^{a} T(\alpha_i.n) + f(n)$$

avec $0 \leq \alpha_i \leq 1$, $a \geq 1$ et $f(n) = \Theta(n^c)$ pour $c > 0$. On a alors :

- Si $\sum_{i=1}^{a} \alpha_i^c < 1$, alors $T(n) = \Theta(n^c)$
- Si $\sum_{i=1}^{a} \alpha_i^c = 1$, alors $T(n) = \Theta(n^c.log_2(n))$
- Si $\sum_{i=1}^{a} \alpha_i^c > 1$, alors $T(n) = \Theta(n^\gamma)$ avec γ tel que $\sum_{i=1}^{a} \alpha_i^\gamma = 1$

Chapitre 3

Tables de hachage

3.1 Structures de données

Définition 9. *Un **dictionnaire** est un ensemble dynamique qui supporte les opérations suivantes :*

- *Insérer(S, x) : Ajoute à S l'élément x.*
- *Rechercher(S, k) : Requête qui retourne un élément y de S de clé k, ou une valeur neutre (Nil) s'il n'y en a pas.*
- *Supprimer(S, x) : Enlève l'élément x de S s'il est présent.*

3.2 Application

3.2.1 Problématique

On a les hypothèses suivantes :

- Données : ensemble de ***clés*** $U = \{0, 1, 2, |U| - 1\}$ (univers)
- Contraintes :
  - Chaque élément de l'ensemble dynamique a une ***clé unique***
  - Le nombre n d'éléments effectivement stockés reste limité ($n << |U|$).

Et l'on cherche :

- Une structure de données offrant les fonctions de ***dictionnaire*** de manière ***efficace***.

3.3 Table à adressage direct

Définition 10. *Une **table à adressage direct** est un tableau de taille $|U|$ dans lequel chaque position (alvéole) correspond à une clé de U et l'absence de valeur dans une alvéole est représenté par une valeur spécifique (Nil).*
Une telle table n'est utilisable que si la taille de U n'est pas trop grande.

3.4 Table de hachage

Si l'on suppose que $n << |U|$, une table de hachage requiert moins de place mémoire qu'une table à adressage direct. Le coût mémoire peut alors être réduit à $\Theta(n)$.

Définition 11. *Une **table de hachage** est définie par :*

- *Une capacité $m << |U|$ telle que $m = \Theta(n)$*
- *Un tableau T de taille m (numéroté à partir de 0)*
- *Une fonction de hachage : $h : U \to \{0, 1, ..., m-1\}$*
- *Un placement des éléments dans le tableau selon leur clé et la fonction $h()$:*
  - *Tout élément de clé k est placé (haché) dans l'alvéole $h(k)$ (valeur de hachage de la clé k).*

Le principal ***inconvénient*** du hachage réside dans les ***collisions*** entre clés différentes hachées dans la même alvéole. Cela est inévitable puisque la fonction $h()$ est définie d'un ensemble de départ plus grand que son ensemble d'arrivée. Nous allons voir deux solutions pour contourner ce problème.

3.5 Hachage avec chaînage

Dans le hachage avec chaînage, chaque case du tableau T contient une ***liste chaînée*** afin de stocker dans une même case les éléments qui ont cet indice de case comme valeur de hachage. Lorsqu'aucun élément n'a été haché dans une case donnée, alors la liste de cette case est vide (Nil).

3.5.1 Analyse du temps moyen

Définition 12. *Pour un nombre n d'éléments conservés dans une table de hachage de taille m, on définit le **facteur de remplissage** α par : $\alpha = \frac{n}{m}$.*
Il représente le nombre moyen d'éléments stockés dans une alvéole de T.

Définition 13. *L'hypothèse de **hachage uniforme simple** (HUS) implique que chaque clé a la même probabilité d'être placée dans n'importe quelle alvéole de la table, indépendamment des autres clés.*

Théorème 2. *Dans une table de hachage avec chaînage, une opération de recherche est effectuée en temps $\Theta(1+\alpha)$ en moyenne, sous l'hypothèse de hachage uniforme simple.*

3.6 Fonctions de hachage

Une bonne fonction de hachage doit vérifier l'hypothèse HUS, ce qui n'est pas si simple. En pratique, on fait appel à des techniques heuristiques pour construire de telles fonctions.

3.6.1 Méthode de la division

La fonction de hachage la plus simple est celle qui utilise l'opérateur modulo :

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

Cependant, certaines recommandations sont nécessaires sur la valeur de m à utiliser dans cette fonction :

- Éviter les puissances de 2 ($m = 2^t$) : seul le sous-mot binaire de taille t est pris en compte.
- Éviter les puissances de 10 ($m = 10^t$) : si les clés de l'application sont des nombres décimaux.
- Privilégier les nombres premiers éloignés des puissances de 2.

3.6.2 Méthode de la multiplication

Une fonction de hachage un peu plus complexe est la suivante :

$$h(k) = \lfloor m.(k.A \mod 1) \rfloor, \quad \text{avec } 0 < A < 1$$

où $x \mod 1$ représente la partie fractionnaire de x, c'est-à-dire : $x - \lfloor x \rfloor$.

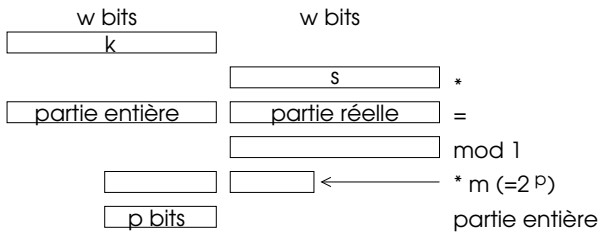


Figure 3.1 – Hachage par multiplication

3.7 Hachage avec adressage ouvert

Dans ce type de hachage, tous les éléments sont conservés dans le tableau T (pas de liste). Ainsi, le facteur de remplissage vérifie nécessairement $\alpha \leq 1$. Lors d'une collision, on ***sonde*** (cherche) un autre alvéole pour cette clé. La fonction de hachage nécessite un second paramètre, qui permet de générer une ***séquence de sondage***. Elle est définie par :

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

Pour la clé k, on a la séquence $h(k, 0), h(k, 1), ..., h(k, m-1)$, qui est une permutation de $(0, 1, ..., m-1)$. Ainsi, toutes les cases du tableau T peuvent être parcourues lors d'un sondage.

3.7.1 Hachage uniforme

Définition 14. *L'hypothèse de hachage uniforme implique que les $m!$ permutations possibles de $\{0, 1, ..., m-1\}$ ont la même probabilité de constituer la séquence de sondage.*

Sondage linéaire

Définition 15. *Le **sondage linéaire** est défini par :*

$$h(k,i) = (h'(k) + i) \mod m, \quad 0 \leq i < m$$

où $h'(k)$ est une fonction de hachage auxiliaire.

Sondage quadratique

Définition 16. *Le **sondage quadratique** est défini pour deux constantes positives c_1 et c_2, par :*

$$h(k,i) = (h'(k) + c_1.i + c_2.i^2) \mod m, \quad 0 \leq i < m$$

où $h'(k)$ est une fonction de hachage auxiliaire.

Double hachage

Définition 17. *Le **double hachage** est défini par :*

$$h(k,i) = (h_1(k) + i.h_2(k)) \mod m, \quad 0 \leq i < m$$

où $h_1(k)$ et $h_2(k)$ sont deux fonctions de hachage auxiliaires.

Afin d'assurer que toute la table puisse être parcourue, il est nécessaire que $h_2(k)$ soit premier avec m.

3.7.2 Analyse du temps

Théorème 3. *Étant donnée une table de hachage en adressage ouvert avec un facteur de remplissage $\alpha = \frac{n}{m} < 1$ et un hachage uniforme, le **nombre maximal de sondages** lors d'une **recherche infructueuse** est : $min\left(\frac{1}{1-\alpha}, m\right)$*

Un ***corollaire*** de ce théorème est que ***l'insertion d'un élément*** requiert au plus $\frac{1}{1-\alpha}$ sondages en moyenne.

Théorème 4. *Étant donnée une table de hachage en adressage ouvert avec un facteur de remplissage $\alpha = \frac{n}{m} < 1$ et un hachage uniforme, le **nombre maximal de sondages** lors d'une **recherche fructueuse** est : $min\left(\frac{1}{\alpha} ln\left(\frac{1}{1-\alpha}\right), m\right)$*

Chapitre 4

Rappels sur les arbres

4.1 Arbres binaires de recherche

Définition 18. *Dans une **arbre binaire de recherche**, toutes les valeurs contenues dans le **sous-arbre gauche** sont **inférieures ou égales** à la valeur de la racine et toutes les valeurs du **sous-arbre droit** sont **supérieures ou égales** à la valeur de la racine. On peut éventuellement supprimer l'égalité à la racine pour un des sous-arbres.*

Définition 19. *Le **facteur d'équilibre** est défini par :*
fe(a) = hauteur(filsDroit(a)) - hauteur(filsGauche(a))
*On dit qu'un arbre est **équilibré** si $|fe(a)| \leq 1$*

4.2 AVL

Définition 20. *Un **AVL** (G.Adelson-Velsky et E.Landis) est un arbre binaire **équilibré**. C'est à dire que pour chaque noeud de l'arbre, les hauteurs des deux sous-arbres différent d'au plus 1. Ainsi, tout sous-arbre d'un AVL est un AVL.*

4.3 Tas

4.3.1 Définition et construction

Définition 21. *La structure de **tas (binaire)** est un objet tabulé (représenté par un tableau) dont le contenu est organisé comme un arbre binaire presque complet. Tous les niveaux de l'arbre sont remplis au maximum sauf le dernier qui est rempli de gauche à droite. La hauteur d'un tas de n éléments est donc en $\mathcal{O}(log_2(n))$.*

On définit trois fonctions d'accès faisant le lien entre les cases du tableau et la structure d'arbre binaire :

- `père(ind : entier) : ret entier` // renvoie l'indice du père de `ind`
- `fg(ind : entier) : ret entier` // renvoie l'indice du fils gauche de `ind`
- `fd(ind : entier) : ret entier` // renvoie l'indice du fils droit de `ind`

Il existe deux types de tas (numérotation à partir de 0) :

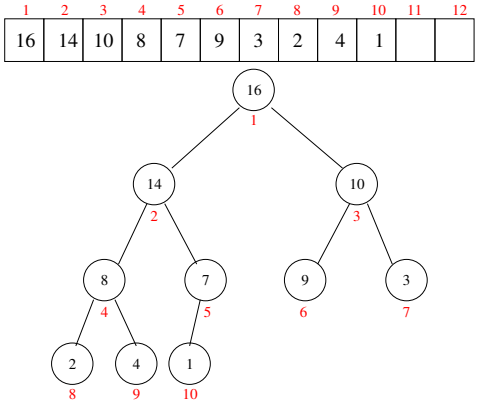


Figure 4.1 – Exemple de tas

- tas ***max*** : $\forall i > 0,\ tas.valeurs[père(i)] \geq tas.valeur[i]$
- tas ***min*** : $\forall i > 0,\ tas.valeurs[père(i)] \leq tas.valeur[i]$

L'opération qui permet de maintenir la propriété de tas de la structure lors des modifications du contenu est la ***fonction d'entassement***. Elle s'applique à un nœud particulier (`ind`) et suppose que les deux sous-arbres (`fg(ind)` et `fd(ind)`) sont des tas. Par contre, la valeur du nœud (`ind`) peut ne pas respecter la propriété de tas max/min. La fonction modifie donc le contenu pour rétablir la propriété sur le sous-arbre de racine `ind`.

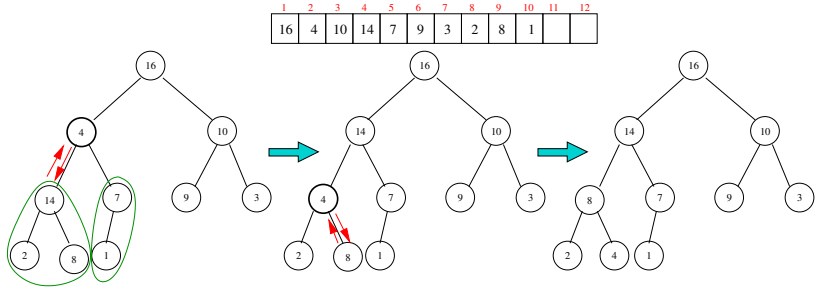


Figure 4.2 – entasser(tas, 2)

Transformation d'un tableau en tas :

Le principe est d'utiliser la fonction `entasser()` en remontant des feuilles vers les ancêtres. Ainsi, lorsque l'on traite un nœud, on est sûr que ses deux sous-arbres sont déjà des tas, ce qui respecte les contraintes de la fonction d'entassement. Le ***temps de construction*** du tas est en $\Theta(n.log_2(n))$ puisque l'on a une boucle en $\Theta(n)$ dont les itérations sont en $\Theta(log_2(n))$.

4.3.2 Tri par tas

La propriété des tas max implique que la valeur la plus grande est à la racine du tas. Le tri croissant par tas utilise cette propriété, il extrait ce premier élément

(en l'échangeant avec le dernier élément du tas) et recréé un tas avec les éléments restants, et ainsi de suite.

Un des avantages de ce tri est qu'il est ***sur place*** et ne consomme donc pas de mémoire supplémentaire. Sa complexité est en $\Theta(n.log_2(n))$ puisque l'on effectue $\Theta(n)$ entassements de complexité $\Theta(log_2(n))$.

4.3.3 Files de priorité

Les tas représentent une structure de données adaptée à la gestion des priorités parmi un ensemble d'événements telle que la planification de tâches sur un ordinateur à ressources partagées :

- Lorsqu'une tâche est interrompue ou en attente : choix de la tâche de plus grande priorité.
- Lorsqu'une tâche est soumise, il faut la mettre dans la file des priorités.

Trois opérations sont nécessaires à cette gestion :

- `insérerTas(tas : typeTas, x : typeEvt) : ret booléen`
  Insertion d'un événement `x` dans la liste des événements en conservant la propriété de tas.
- `maxTas(tas : typeTas) : ret typeEvt`
  Retourne l'évènement de plus grande priorité dans le tas (file des priorités).
- `extraireMax(tas : typeTas, max : typeEvt) : ret booléen`
  Supprime l'événement de plus grande priorité dans le tas.

Chapitre 5

Arbres rouge et noir

5.1 Arbres binaires de recherche

Les arbres binaires de recherche (ou ordonnés) permettent de supporter de manière efficace un certain nombre d'opérations sur des ***ensembles totalement ordonnés***, telles que :

— Rechercher, Insérer, Supprimer, Min, Max, Prédécesseur, Successeur

Dans le contexte de stockage basé sur des clés uniques, nous avons la définition suivante :

Définition 22. *Dans un **arbre binaire de recherche**, tout nœud x vérifie les propriétés suivantes :*

- *pour tout y dans le sous-arbre gauche de x, on a : clé(y) ≤ clé(x)*
- *pour tout z dans le sous-arbre droit de x, on a : clé(z) ≥ clé(x)*

Théorème 5. *Les sept opérations de base des ensembles dynamiques peuvent être réalisées en temps $\mathcal{O}(h)$ sur un arbre binaire de recherche de hauteur h.*

5.2 Arbres binaires de recherche équilibrés

Définition 23. *On dit qu'un arbre binaire de recherche comportant n nœuds est **approximativement équilibré** si sa hauteur est $\mathcal{O}(log_2(n))$.*

5.3 Arbres rouge et noir

Définition 24. *Un **arbre rouge et noir** est un arbre de recherche comportant une information de couleur (rouge ou noir) dans chaque nœud, et vérifiant les propriétés suivantes :*

1. *Chaque nœud est soit rouge, soit noir.*
2. *Chaque feuille (Nil) est noire.*
3. *Un nœud rouge a ses deux fils noirs.*
4. *Tous les chemins entre un nœud et ses feuilles descendantes ont le même nombre de nœuds noirs.*

Une ***convention importante*** pour la suite est que les ***valeurs Nil*** sont considérées comme des ***nœuds externes*** et le nœuds comportant des clés sont les ***nœuds internes***.

5.4 Hauteur d'un arbre rouge et noir

Définition 25. *La **hauteur noire** d'un nœud x, notée $hn(x)$, est le nombre de nœuds noirs présents dans un chemin quelconque descendant d'un fils de x vers une feuille.*

Ainsi, tout chemin de la racine à une feuille descendante ne peut être plus de deux fois plus long qu'un autre chemin. Cela limite donc le déséquilibre potentiel de l'arbre et l'inclut dans les arbres approximativement équilibrés.

Théorème 6. *Le sous-arbre enraciné à un nœud x d'un arbre rouge et noir contient au moins $2^{hn(x)} - 1$ nœuds internes.*

Théorème 7. *La **hauteur d'un arbre noir et rouge** comportant n nœuds internes (clés) est au plus $2.log_2(n+1)$.*

5.5 Modifications des arbres rouge et noir

Définition 26. *Une **rotation** est une opération locale à un nœud d'un arbre de recherche, qui préserve l'ordre infixe des clés. Elle s'exécute en temps $\mathcal{O}(1)$.*

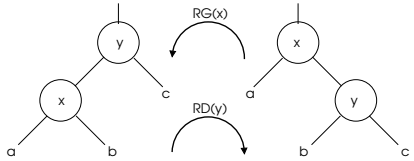


Figure 5.1 – Rotations dans un arbre de recherche

5.5.1 Insertion

L'insertion dans un arbre rouge et noir se déroule en trois étapes :

1. Insertion classique (arbre de recherche) du nœud x dans l'arbre T.
2. Coloriage en rouge de x (préserve la propriété 4).
3. Correction éventuelle de l'arbre si le père est rouge (re-coloriage et rotations)

Pour l'étape 3, on distingue six cas différents où x et $p(x)$ sont tous les deux rouge. Cependant, la symétrie gauche-droite nous permet de n'en retenir que trois.

Cas 1 : oncle rouge

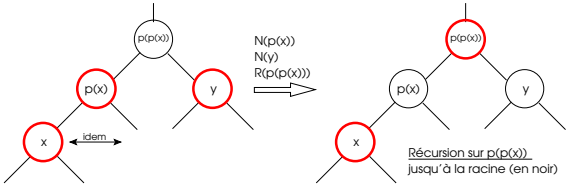


Figure 5.2 – Traitement du cas 1 de conflit rouge - noir

Cas 2 : oncle noir et x sur branche interne de l'arbre

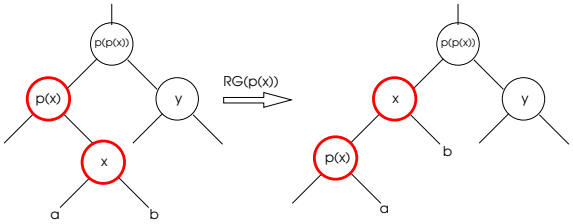


Figure 5.3 – Traitement du cas 2 de conflit rouge - noir

Cas 3 : oncle noir et x sur branche externe de l'arbre

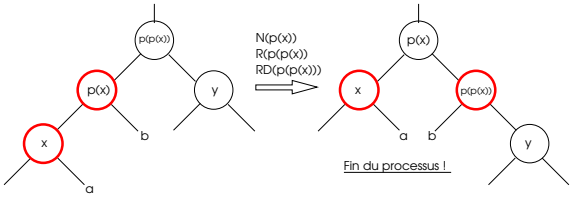


Figure 5.4 – Traitement du cas 3 de conflit rouge - noir

Analyse du temps

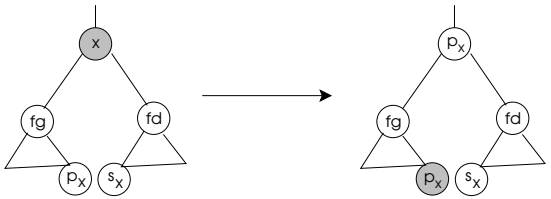
En sommant les coûts d'insertion et des cas 1, 2 et 3 successifs, on obtient le coût maximal :

$$\mathcal{O}(log_2(n)) + \mathcal{O}(log_2(n)) + \mathcal{O}(1) + \mathcal{O}(1) = \mathcal{O}(log_2(n))$$

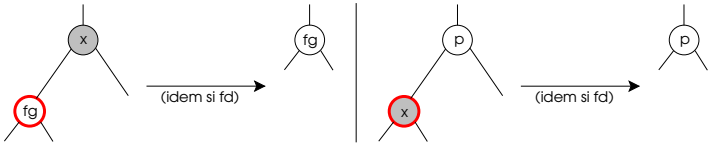
5.5.2 Suppression

La suppression d'un nœud va également générer des conflits par rapport aux propriétés 3 et 4. Comme pour l'insertion, il y a plusieurs cas possibles à traiter. La distinction se fait sur le nombre de fils Nil du nœud x à supprimer :

1. x a ***deux fils non Nil*** :

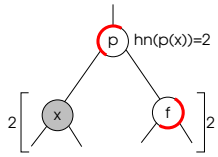


2. x a ***un fils non Nil*** :



Cas 2 Cas 3r

3. x a ***deux fils Nil*** :
   - Si x est ***rouge***, on le remplace simplement par Nil qui est toujours noir (Cas 3r)
   - Si x est ***noir***, sa suppression va entraîner un déséquilibre des hauteurs noires entre les deux branches de son père :



On a alors ***quatre cas*** selon les couleurs de son frère et des fils de son frère :

a) Le ***frère est noir avec deux fils noir*** :
b) Le ***frère est noir avec un fils rouge*** :
c) Le ***frère est noir avec deux fils rouge*** :
d) Le ***frère est rouge*** :

En ce qui concerne la ***procédure de ré-équilibrage***, on considère un nœud x dont la hauteur noire est inférieure de 1 à celle de son frère.

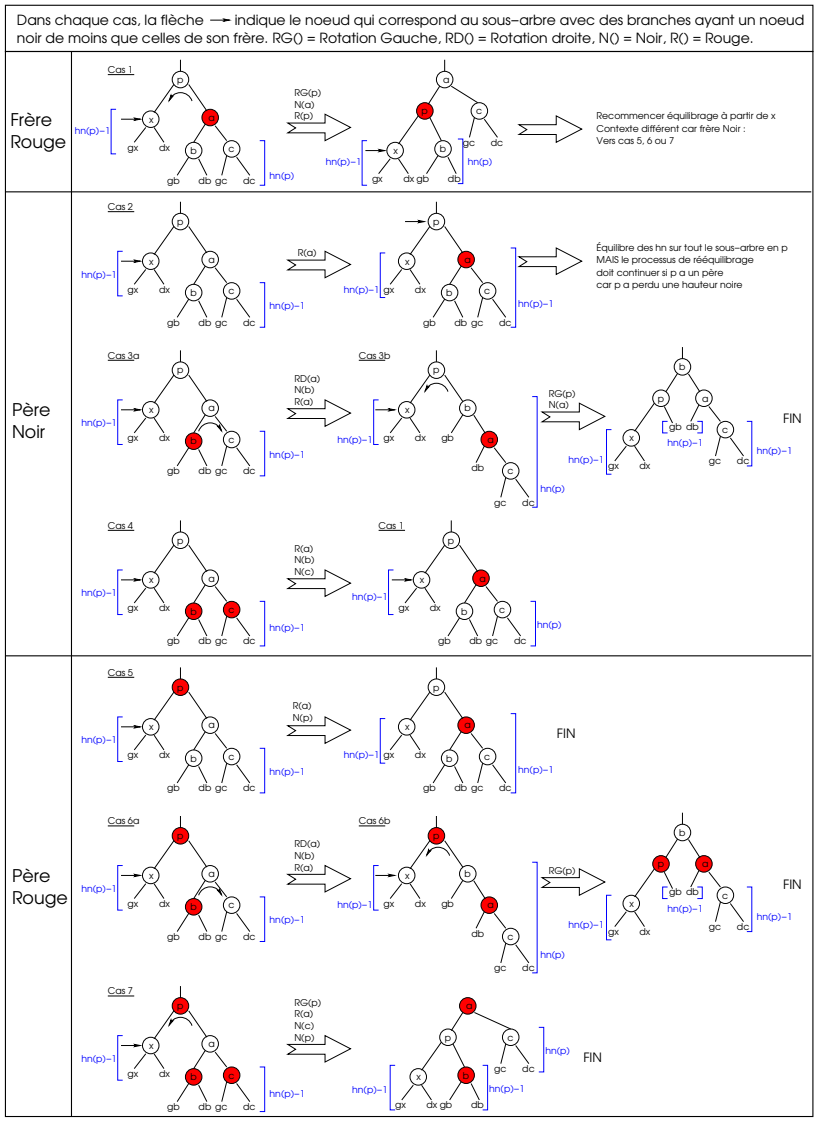


Figure 5.5 – Ré-équilibrage d'un nœud x dont les branches ont un nœud noire de moins que celles de son frère.

Chapitre 6

Stratégie gloutonne

6.1 Problème d'optimisation

Définition 27. *Un problème d'**optimisation** est défini par les éléments suivants :*

- *Les données du problème*
- *La caractérisation d'une solution au problème (ensemble de taille > 1)*
- *Une valeur associée à chaque solution (coût)*
- *Un objectif de solution avec une **valeur optimale** (min ou max)*

6.2 Classes de problèmes

Définition 28. *La **classe P** contient les problèmes que l'on sait **résoudre en temps polynomial** $\mathcal{O}(n^k)$ pour une taille n de donnée du problème et k une constante.*

Définition 29. *La **classe NP** contient les problèmes que l'on sait **vérifier en temps polynomial**. Pour un candidat donné, on peut vérifier en temps polynomial de sa taille s'il est solution du problème.*

Définition 30. *La **classe NP-complet** contient les problèmes de NP qui sont **aussi difficiles que tout autre problème de NP**. S'il existe un problème NP-complet qui peut être résolu en temps polynomial, alors tous les problèmes NP peuvent être résolus en temps polynomial.*

6.3 Algorithme glouton

Définition 31. *Un **algorithme glouton** résout un problème d'optimisation en effectuant une succession de **choix localement optimaux** permettant de réduire le problème jusqu'à un cas simple.*

Définition 32. *Une **heuristique** est un algorithme d'optimisation qui permet d'obtenir une **solution approchée** de la valeur optimale, mais pas systématiquement l'optimale. Il n'y a donc pas de garantie d'optimalité.*

Dans la suite, nous considérons les algorithmes gloutons ***valides***, c'est-à-dire ceux dont nous pouvons prouver qu'ils produisent toujours une

6.4 Application au choix d'activités

Le problème du choix d'activité est posé comme suit :

- Donnée : Ensemble $A = \{1, ..., n\}$ de n activités concurrentes définies par un horaire de début (d_i) et un horaire de fin (f_i) pour $1 \geq i \geq n$, tels que $d_i \leq f_i$.
- Solution : Choix C d'*activités compatibles* entre elles
- Valeur : Nombre d'activités ($|C|$)
- Objectif : Trouver un ensemble C de taille maximale

Définition 33. *Deux activités $i, j \in A$ vérifiant $i \neq j$ sont dites **compatibles** si les intervalles $[d_i, f_i]$ et $[d_j, f_j]$ ne se superposent pas : $f_i < d_j$ ou $f_j < d_i$.*

6.4.1 Algorithme glouton pour le choix d'activités

On peut accélérer le choix de la prochaine activité en *triant* les activités dans l'*ordre croissant des horaires de fin* :

$$f_1 \leq f_2 \leq ... \leq f_{n-1} \leq f_n$$

Algorithme 6.1 – Fonction ChoixActivités(d, f)

```
C ← {1} // La 1ère activité est nécessairement celle qui finit le plus tôt
a ← 1   // Dernière activité choisie
pour i de 2 à n faire // Parcours des activités restantes
  si f_a < d_i alors     // Si l'activité suivante est compatible
    C ← C ∪ {i}          // avec la courante, on la choisit car
    a ← i                // c'est celle qui finit le plus tôt
  fsi
fpour
retourner C
```

Théorème 8. *Soit un sous-problème $A_k = \{a_i \in A : d_i > f_k\}$ non vide, et a_m une activité dans A_k avec le plus petit horaire de terminaison. Alors, a_m est incluse dans un sous-ensemble de taille maximale de A_k d'activités compatibles.*

6.4.2 Analyse du temps de l'algorithme glouton obtenu

En analysant l'Algorithme 6.1, on constate qu'il n'y a que des opérations en temps constant et une boucle d'ordre n. Ainsi, le temps de l'algorithme glouton pour le choix des activités est en $\Theta(n)$ sur des entrées déjà triées. S'il faut trier les activités, le temps devient $\Theta(n.log_2(n))$.

6.4.3 Démonstration de validité d'un algorithme glouton

Pour ***démontrer*** qu'un algorithme glouton est ***valide***, on peut utiliser la méthode suivante :

1. On suppose qu'il y a une entrée pour laquelle l'algorithme glouton génère une ***solution*** s_a ***non optimale***.
2. On considère s_o ***une solution optimale*** *particulière* pour cette même entrée (maximisant ou minimisant un critère donné).
3. On effectue des échanges entre les deux solutions s_a et s_o pour obtenir une ***nouvelle solution*** s qui aboutie à une ***contradiction*** du choix glouton ou du choix de s_o.

6.4.4 Application au problème du choix des activités

Théorème 9. *L'Algorithme 6.1 s'exécute en temps $\Theta(n)$ et produit une solution avec le plus grand nombre possible d'activités compatibles.*

6.5 Caractéristiques nécessaires à la stratégie gloutonne

Il y a deux caractéristiques nécessaires pour qu'un problème puisse être résolu par un algorithme glouton :

- ***Propriété du choix glouton :*** On peut arriver à une solution optimale en effectuant des choix localement optimaux.
- ***Sous-structure optimale :*** Une solution optimale d'un problème contient les solutions optimales des sous-problèmes.

6.6 Problème du sac-à-dos

Le problème d'optimisation du sac-à-dos (*knapsack*) est défini comme suit :

- Donnée : $O = \{1, 2, ..., n\}$ ensemble de n objets où chaque objet i vaut v_i euros et pèse p_i grammes. Un sac-à-dos d'une capacité C grammes.
- Solution : ensemble S d'objets tel que
$$\sum_{i \in S} p_i \leq C$$
- Valeur : la valeur d'une solution S est définie par :
$$val(S) = \sum_{i \in S} v_i$$
- Objectif : trouver une solution S de valeur maximale

Il y a ***deux variantes*** de ce problème selon que l'on considère les objets divisibles ou non :

- ***Version discrète :*** les objets ne sont pas divisibles et on doit donc laisser l'objet ou le prendre complètement.
- ***Version continue :*** les objets sont divisibles et l'on peut prendre seulement une fraction de son poids.

Il existe un algorithme glouton valide pour la version continue (basé sur les valeurs massiques $\frac{v_i}{p_i}$ décroissantes), mais pas pour la version discrète.

Chapitre 7

Programmation dynamique

7.1 Intérêt de la programmation dynamique

On a vu précédemment que certains problèmes d'optimisation ne peuvent être résolus par stratégie gloutonne (version discrète du sac-à-dos) bien qu'ils aient une sous-structure optimale.

Contrairement à la stratégie gloutonne où l'on fait un choix localement optimal puis on résout les sous-problèmes qui en découlent, avec la programmation dynamique on fait également un ***choix à chaque étape***, mais qui ***dépend de la solution des sous-problèmes***.

7.2 Conception d'un algorithme de programmation dynamique

La conception d'un algorithme de programmation dynamique peut être décomposée en quatre parties :

1. Déterminer la structure d'une solution optimale
2. Exprimer la valeur d'une solution optimale sous forme récursive
3. Calculer la valeur d'une solution optimale en allant des cas terminaux jusqu'au problème initial (***approche ascendante***)
4. Construire la solution optimale à partir des éléments déjà calculés

7.3 Multiplication d'une suite de matrices

7.3.1 Formalisation du problème d'optimisation

Le problème du coût minimum de la multiplication d'une séquence de matrices est défini par :

- Donnée : Séquence $A_1, A_2, ..., A_n$ de matrices et une séquence d'entiers $p_0, p_1, ..., p_n$ telles que chaque matrice A_i a les dimensions $p_{i-1} \times p_i$.
- Solution : Parenthésage P du produit $A_1.A_2...A_n$.
- Valeur : Nombre d'opérations effectuées en suivant le parenthésage P.
- Objectif : Trouver un parenthésage P obtenant la valeur minimale.

Pour une séquence donnée de taille n, la ***recherche exhaustive*** n'est ***pas raisonnable*** car en temps exponentiel.

7.4 Application de la programmation dynamique

7.4.1 Identifier la structure d'une solution optimale

Le produit d'une séquence de matrices se termine nécessairement par la multiplication de deux matrices. Ces deux matrices correspondent respectivement au produit d'une ***sous-séquence préfixe*** et d'une ***sous-séquence suffixe***, de la séquence initiale.

$$(A_1...A_k)(A_{k+1}...A_n)$$

Et le ***coût total du produit*** est la somme des coûts suivants :

- produit de la sous-séquence préfixe $A_1...A_k$
- produit de la sous-séquence suffixe $A_{k+1}...A_n$
- multiplication des deux matrices résultantes

On remarque que pour un k donné, les coûts des produits des sous-séquences doivent être minimaux (sinon on peut les remplacer par des versions moins coûteuses), ce qui implique de trouver des parenthésages optimaux pour ces deux sous-séquences. On vient d'identifier une ***sous-structure optimale*** du problème.

7.4.2 Valeur d'une solution optimale sous forme récursive

Soit $m_{i,j}$ le coût d'une solution optimale d'une sous-séquence $A_i...A_j$. On sait qu'il existe une valeur k pour laquelle le parenthésage $(A_i...A_k)(A_{k+1}...A_j)$ a le coût minimal. Il suffit donc de chercher le meilleur k parmi les possibles, ce qui nous mène à la récurrence suivante :

$$m_{i,j} = \begin{cases} 0 & \text{si} \quad i = j \\ \min_{k=i}^{j-1}(m_{i,k} + m_{k+1,j} + p_{i-1}p_k p_j) & \text{si} \quad i < j \end{cases}$$

7.4.3 Algorithme de calcul des coûts optimaux

C'est à cette étape que se révèle toute la puissance de la programmation dynamique. En effet, cette ***approche ascendante*** part des cas terminaux et remonte vers le problème initial en ***explorant une seule fois les combinaisons possibles***, et en ne retenant que celles qui présentent des coûts minimums dans les niveaux intermédiaires.

Pour mettre en œuvre notre algorithme, on utilise un tableau m de taille $n \times n$ pour stocker les coûts $m_{i,j}$. On utilise également un tableau s de même taille, dans lequel on stockera dans la case $[i, j]$ l'indice de séparation optimale k du sous-problème $A_i...A_j$.

Le principe de l'algorithme est donc de remplir les deux tableaux **en commençant par les cas terminaux**, puis de continuer en augmentant progressivement la taille de la séquence (3, 4,...) jusqu'à n matrices. La Figure 7.1 indique les cases de m utilisées pour déduire les coûts $m_{1,3}$ et $m_{1,4}$. Mais on a également besoin des p_i liés aux matrices de ces sous-séquences.

7.4.4 Construction de la solution optimale

Une fois les tableaux m et s remplis, la case du haut de chacun de ces tableaux (celle d'indice $[1, n]$), contient le coût minimum (tableau m) et la meilleure séparation en deux sous-séquences (tableau s). Si $s[1, n] = k$ alors on parcourt les cases $s[1, k]$ et $s[k + 1, n]$ pour trouver les séparations des deux sous-séquences, et ainsi de suite.

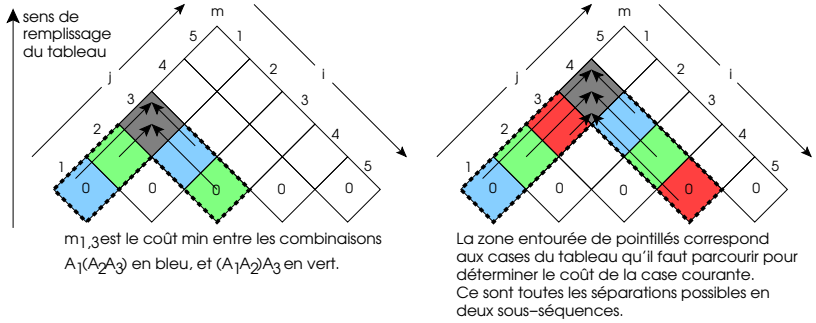


Figure 7.1 – Remplissage du tableau des coûts

7.5 Analyse du temps

Sur la Figure 7.1, on constate que pour chaque case $m_{i,j}$ (il y en a $\Theta(n^2)$), on effectue un calcul à coût constant pour chacune des $j-i$ séparations possibles (il y en a $\mathcal{O}(n)$). On obtient donc un ***temps en*** $\mathcal{O}(n^3)$.

On peut également estimer la complexité mémoire de cet algorithme en comptant le nombre de valeurs stockées. On voit que l'algorithme utilise les tableaux m et s qui sont de taille n^2. Cet algorithme a donc une complexité ***mémoire en*** $\mathcal{O}(n^2)$.