Compléments d'algorithmique

Sami ABDUL SATER

17 mars 2021

Table des matières

Chapitre 1

Introduction

1.1 Algorithme

Un **algorithme** est une méthode de calcul qui, à partir d'un ensemble de valeurs d'entrée, permet d'obtenir des valeurs de sortie répondant à un problème donné (et pour lequel l'algorithme a été conçu).

1.1.1 Instance d'un problème

L'objectif de conception d'un algorithme est qu'il puisse résoudre un problème pour toute valeur d'entrée. De manière plus rigoureuse, on applique en réalité un algorithme à une *instance d'un problème* : il s'agit de l'ensemble des valeurs d'entrée. La sortie consiste donc en la solution de cette instance du problème.

Un algorithme doit donc être capable de résoudre toutes les instances du problème pour lequel il a été conçu.

1.1.2 Lien entre algorithme et programme

Un programme est simplement la transcription d'un algorithme pour un ordinateur. Là où les programmes que nous connaissons sont écrits avec un langage de programmation compréhensible par une machine (après compilation), les algorithmes sont **écrits en pseudo-code** et sont en général pas écrits pour un ordinateur mais, pour rester inclusif, pour un **modèle de calcul**. C'est l'analogue de l'ordinateur pour les programmes.

Programme	Algorithme
Langage de programmation	Pseudo-code
Ordinateur	Modèle de calcul

TABLE 1.1 – Comparaison entre programme et algorithme

1.1.3 Modèle de calcul – machine RAM

Le modèle de calcul sur lequel vont être appliqués nos algorithmes va être la **machine RAM** : il s'agit d'une machine à "accès aléatoire", c'est-à-dire qui nous donne accès à une mémoire de grande taille, sur laquelle nous pourrons effectuer nos opérations. Nous distinguons alors naturellement les instructions **élémentaires** des instructions **non-élémentaires** qui sont constituées d'instructions élémentaires.

- Élémentaires : lecture d'une donnée du tableau, écrire une valeur dans le tableau, instructions arithmétiques (addition, multiplication ...), de test (égalité, superiorité), et logiques (négation, et, ou).
- Non-élémentaires : tout le reste ! Par exemple, une insertion est non-élémentaire parce qu'il faut décaler les données avant d'insérer.

1.1.4 Qualités d'un algorithme

Nous nous attendons d'un algorithme :

- Qu'il se termine en un temps fini
- Qu'il soit efficace : qu'il y ait le moins d'instructions élémentaires possible
- Qu'il soit correct avec haute probabilité, c'est-à-dire qu'il renvoie une solution correcte pour toute instance du problème.

1.2 Complexité

1.2.1 Temps de calcul

Le **temps de calcul** d'un algorithme est la durée de son exécution. Elle dépend généralement de la taille du problème $\Rightarrow T(n)$. Comme on s'intéresse aux problèmes de grande taille, on ne conserve dans $T(n)$ que la forme asymptotique.

$$T(n) = 2n^2 + 10n + 5 \sim 2n^2 \quad \Rightarrow T(n) = 2n^2 + O(n)$$

1.2.2 Notations asymptotiques

$$\text{Grand-O} : f(n) \in O\left(g(n)\right) \Leftrightarrow \exists c, n_0 > 0 \text{ t.q } \boxed{f(n) \leq c \cdot g(n)} \ \forall n \geq n_0$$

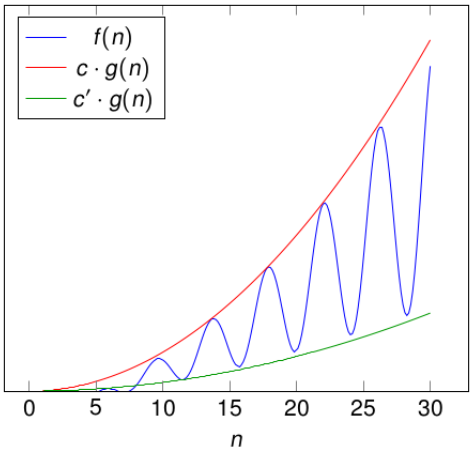
$$\text{Grand-}\Omega : f(n) \in \Omega\left(g(n)\right) \Leftrightarrow \exists c, n_0 > 0 \text{ t.q } \boxed{f(n) \geq c \cdot g(n)} \ \forall n \geq n_0$$

Enfin, nous définissons l'ensemble des fonctions en Θ d'une autre fonction :

$$f(n) \in \Theta(n)$$

$$\Longleftrightarrow$$

$$\exists c_1, c_2, n_0 > 0 \text{ t.q } \boxed{c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)} \ \forall n \geq n_0$$



1.2.3 Classification des comportements asymptotiques

En algorithmique, on cherche à évaluer le temps de calcul d'un algorithme et l'approcher en valeur asymptotique à une fonction "classique" parmi les suivantes, triées dans l'ordre croissant des tendances asymptotiques.

Notation grand-Θ	Nom
$\Theta(\log n)$	logarithmique
$\Theta(n)$	linéaire
$\Theta(n^2)$	quadratique
$\Theta(n^c)$	polynomial
$\Theta(2^n)$	exponentiel
$\Theta(n!)$	factoriel

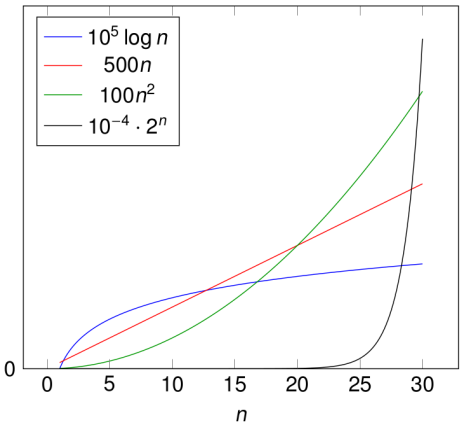


FIGURE 1.1 – Tendances asymptotiques

1.2.4 Complexité d'algorithme et de problème

La **complexité d'un algorithme** est le comportement asymptotique de son temps de calcul $T(n)$ pour la **pire entrée**.

La **complexité d'un problème** est la complexité du meilleur algorithme pour ce problème. On précise *"meilleur algorithme"* parce qu'un algorithme peut ne pas être optimal pour le problème qu'il résout.

Chapitre 2

Problème de recherche de pic

Le problème de la recherche de pic, d'abord à une dimension, puis à deux dimensions, consiste à chercher, dans un tableau, une valeur correspondant à un pic, c'est-à-dire plus grande que tous ses voisins contigus.

2.1 Problème à 1D

2.1.1 Approche naïve

L'apporche naïve consiste à parcourir toutes les valeurs une à une et pour chaque valeur, tester les deux voisins. C'est une approche qui marche mais qui est longue si chaque voisin de droite est plus grand que l'élément : dans le pire des cas, le temps est en $\Theta(n)$. La complexité de l'algorithme naïf est donc n.

2.1.2 Algorithme *divide-and-conquer*

L'algorithme *divide-and-conquer* , a.k.a "diviser pour régner", consiste à lire les deux valeurs en milieu de tableau : $a[\lfloor n/2 \rfloor]$ et $a[\lfloor n/2 \rfloor + 1]$, et de les comparer.

- Si $a[\lfloor n/2 \rfloor > a[\lfloor n/2 \rfloor - 1]$, c'est qu'il y a un pic dans le tableau entre les indices 0 et $\lfloor n/2 \rfloor - 1$. Il faut alors appliquer l'algorithme sur ce sous-tableau.

- Si non, lire $a[\lfloor n/2 \rfloor + 1]$ et appliquer le même raisonnement.

- Si non, c'est qu'il y a un pic à cet endroit-là.

Dans le pire des cas, le pic ne se trouve jamais à l'endroit où on regarde, et on divise donc en 2 le tableau à chaque fois, jusqu') se réduire un tableau de taille minuscule : $n \to n/2^t$. Un pic est forcément atteint lorsque la taille du tableau vaut 1 ou 2, ce qui arrive lorsque

$$n/2^t \approx 1 \iff t \approx \log n \quad \Longrightarrow \quad \boxed{T(n) = \Theta\left(\log(n)\right)}$$

2.2 Problème à 2D

Dans un tableau bidimensionnel, il faut vérifier 4 voisins :

$$a[i][j] \text{ est un pic} \iff a[i][j] \begin{array}{l} \geq a[i-1][j] \\ \geq a[i+1][j] \\ \geq a[i][j+1] \\ \geq a[i][j-1] \end{array}$$

2.2.1 Approche naïve – greedy ascent

L'approche naïve consiste à se déplacer dans le tableau dans la direction du voisin le plus grand. Encore une fois, ce n'est pas très économe parce que dans la pire des situations, dans un tableau de taille $n \times m$, on peut arriver à parcourir les nm valeurs du tableau : $T(n) = \Theta(n^2)$

2.2.2 *divide-and-conquer*

Encore cet algorithme ! Cette fois-ci, on commence par regarder la colonne du milieu en cherchant le maximum global de la colonne (temps en $\Theta(n)$). On compare ce maximum à ses voisins de gauche et de droite et on applique l'algorithme sur le tableau de gauche ou de droite selon quel voisin est plus grand que $a[i][\lfloor m/2 \rfloor]$ (le maximum de la colonne du milieu).

15	16	10	18	20
14	11	9	9	7
13	14	13	10	9
3	16	3	9	8
3	4	5	6	7

FIGURE 2.1 – Application de l'algorithme *divide-and-conquer* sur un tableau de taille $n \times m$

L'analyse de la complexité de cet algorithme est simple :

1. Trouver un maximum global : $\Theta(n)$
2. Appliquer l'algorithme sur un tableau $n \times (m/2)$

$$\begin{array}{rcl} \text{Cas de base} & : & T(n,1) = \Theta(n) = C \\ & & T(n,m) = C + T(n,m/2) \end{array}$$

Cela veut dire qu'à chaque étape, on ajoute $C + T(n, m/2)$ jusqu'à arriver à $m = 2$ où $m/2 = 1$ et on revient au cas de base. La récurrence est donc facilement résolue :

$$\boxed{T(n,m) = \Theta\left(n \cdot \log m\right)}$$

Nous voyons donc qu'un algorithme *divide-and-conquer* permet d'améliorer grandement la complexité par rapport à une approche "naïve".

2.3 Réflexion algorithmique

1. Analyser la définition du problème : existence de solution ? cas particuliers ?
2. Considérer tous les cas possibles
3. Analyser les propriétés de l'alrogithme conçu : se termine-t-il ? est-il toujours correct ?
4. Analyser la complexité : le pire cas
5. Se demander si l'on peut faire mieux

Chapitre 3

Algorithmes de tri

3.1 Motivation

Il y a un besoin intrinsèque d'avoir des données triées. Que ça soit pour avoir un **ordre** dans les données (dictionnaire, annuaire), ou pour résoudre certains problèmes plus rapidement, ou pour des applications plus technologiques comme la compression de données et le rendu d'effets graphiques.

Pour un problème de tri, nous avons pour entrée un tableau a de taille n à ordonner. La sortie serait une permutation de tableau de sorte que

$$a'[0] \leq a'[1] \leq a'[2] \leq \cdots \leq a'[n-1]$$

3.2 Tri par insertion

3.2.1 Approche idiote

Une première idée, et sans doute la pire de l'histoire du tri, est de parcourir le tableau, valeur par valeur, et de placer chaque élément dans un nouveau tableau de sorte que ce-dernier soit trié. Non seulement il faut parcourir tout le tableau, mais en plus de ça le tri n'est **pas en place** (on travaille sur une copie du tableau), mais aussi : si le tableau est trié initialement dans l'ordre décroissant, il faudra décaler les $n-1$ éléments du tableau créé pour placer l'élément $a[n-1]$.

Meilleur cas

Notons que dans le meilleur cas, celui où le tableau est déjà trié, le coût de traitement de l'élément d'indice i, que nous allons noter C_i, est indépendant de i : il ne faut rien décaler, juste ajouter en fin de tableau.

Pire cas

Soit le pire des cas, celui bien précis où le tableau est initialement dans l'ordre inverse. À l'insertion de l'élément d'indice i, il faudra effectuer i **permutations**. En effet, vu que le tableau est dans l'ordre décroissant, à chaque fois qu'on va vouloir mettre une valeur dans le nouveau tableau copié, celle-ci sera inférieure à toutes les valeurs déjà dans le tableau : donc il faudra bouger chacune vers la droite, ce qui prend un temps constant

par élément, mais il faudra le faire n_i fois pour l'élément d'indice n_i ! Le temps de calcul vient alors, en notant C_i le coût de traitement de l'élément d'indice i :

$$C_i = c \cdot i \ ;$$

et pour l'ensemble des n éléments à trier :

$$T(n) = \sum_{i=1}^{n-1} C_i = c \cdot \frac{n(n-1)}{2} = \Theta(n^2)$$

3.2.2 Approche par recherche dichotomique

Une deuxième approche, mieux que la première, serait de regarder à chaque élément i, où est le meilleur endroit pour insérer : cela se fait en faisant une recherche dichotomique sur le nouveau tableau avec les $i-1$ éléments déjà dedans. Il faudra alors, à chaque insertion (il y en a n), effectuer $\log(i)$ comparaisons si nous insérons l'élément i. En revanche il faudra toujours effectuer le même nombre de permutations ! Le calcul suivant vient alors :

Comparaisons

On réduit le nombre total de comparaisons :

$$T'(n) = \sum_{i=1}^{n-1} \log(i) = \log\left(\prod_{i=1}^{n-1} i\right) = \log\left((n-1)!\right) = \Theta\left(n \cdot \log(n)\right)$$

Permutations

On ne change pas le nombre de permutations !

$$T(n) = \sum_{i=1}^{n-1} C_i = c \cdot \frac{n(n-1)}{2} = \Theta(n^2)$$

On l'aura compris, le but ultime est de réduire également le nombre de permutations. En tout cas pour le tri par insertion, quoique l'on fasse, on effectue n permutations et on revient à un temps en n^2.

Coût du tri par insertion : $T(n) = \Theta(n^2)$

3.3 Le tri par fusion – approche *divide-and-conquer*

3.3.1 Principe

Le problème du tri peut également être abordé par une approche *divide-and-conquer* . L'objectif serait d'appliquer récursivement le même algorithme sur une sous-partie du tableau et de recombiner tous les morceaux à la fin.

```
1: procedure SolveByDivideAndConquer
2:     if Problem small then
3:         Solve directly
4:     else
5:         Divide Problem in SubProblems
6:         for Each SubProblem do SolveByDivideAndConquer
7:         Reconstruct Solution from SubSolutions
```

FIGURE 3.1 – Canevas général d'un algorithme *divide-and-conquer*

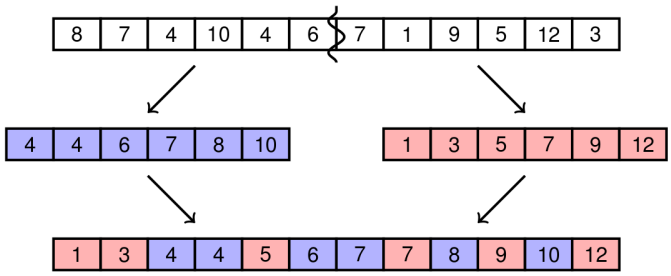


FIGURE 3.2 – Illustration du tri par fusion où les pseudo-codes sont donnés par les figures **??** et **??**.

3.3.2 Canevas général d'un algotithme *divide-and-conquer*

En toute généralité, un algorithme *divide-and-conquer* s'écrit de la manière qui suit :

3.3.3 Application au problème de tri

Pseudo-code

Pour l'appliquer **en place**, il faut travailler sur le même tableau et donc le trier entre certains indices, par exemple : trier le tableau entre les indices a et b, entre b et c, et puis combiner les deux sous-tableaux. On a donc naturellement le pseudo-code ci-contre, où il faut encore définir la procedure `merge` que nous voyons apparaître : comment fusionner deux sous-tableaux ?

```
1: procedure MergeSort(A, left, right)
2:     if left < right then
3:         mid = ⌈(left + right)/2⌉
4:         MergeSort(A, mid, right)
5:         MergeSort(A, left, mid − 1)
6:         Merge(A, left, mid, right)
```

FIGURE 3.3 – Pseudo-code du tri par fusion

Fusion de deux sous-tableaux triés

Vu que les deux sous-tableaux sont déjà triés, il sera facile de les fusionner en un tableau trié. Si un tableau A est donné en entrée et étant constitué de deux tels sous-tableaux, la première chose à faire sera de créer deux copies des sous-tableaux, pour appliquer les modifications sur A lui-même. Comme avant, pour que le tri se fasse entièrement en place, on va considérer en entrée la liste d'arguments : A, indice de gauche `left`, indice de droite `right` et indice de milieu `mid`. Le début du tableau de gauche serait alors $n_L =$`mid` - `left`, celui du droit serait $n_R =$ `right`-`mid`+1.

Ensuite, il suffira de comparer les éléments du tableau de gauche aux éléments du tableau de droite et de placer dans A le plus petit d'entre les deux. Si on arrive à comparer

deux "infinis", c'est-à-dire des éléments qu'on a placé expressément à la fin des deux sous-tableaux pour indiquer qu'on s'y est baladé jusqu'à la fin, c'est qu'on a terminé le tri.

```
1:  procedure MERGE(A, left, mid, right)
2:      n_L = mid − left, n_R = right − mid + 1        ▷ Taille des sous-tableaux
3:      copy A[left : mid − 1] into L[0 : n_L − 1]
4:      copy A[mid : right] into R[0 : n_R − 1]
5:      L[n_L] = ∞, R[n_R] = ∞                         ▷ Sentinelle, marque la fin du tableau
6:      i_L = 0, i_R = 0
7:      for i = left → right do                        ▷ Copie les éléments un par un dans l'ordre
8:          if L[i_L] ≤ R[i_R] then
9:              A[i] = L[i_L], i_L = i_L + 1
10:         else
11:             A[i] = R[i_R], i_R = i_R + 1
```

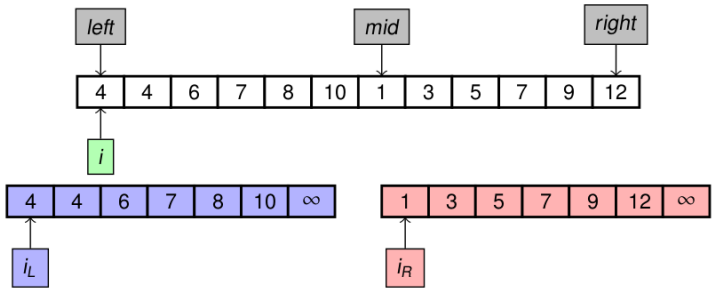


FIGURE 3.4 – Pseudo-code de la fusion de deux sous-tableaux triés, utile pour le tri par fusion.

Complexité de `merge`

Faire un `merge` nécessite deux choses :

- faire une copie de chaque sous-tableau : $\Theta(n_L + n_R) = \Theta(n)$,
- faire des affectations simples (indices n_L, n_R, ...) : $\Theta(c_2)$
- faire une boucle entre `left` et `right` soit n itérations. Pendant chaque itération, des opérations qui prennent un temps constant sont faites $\Rightarrow \Theta(c_1 n)$.

Le coût total d'un `merge` est donc

$$T(n) = c_1 \cdot n + c_2 \ .$$

Complexité de `mergesort`

La complexité du tri par fusion s'obtient en résolvant une récurrence et ceci est fait en détails dans le TP. Dans les grandes lignes, on fait un tri par fusion en :

- séparant le tableau en deux sous-tableaux : affectation d'indices, temps constant c_4,
- en appliquant l'algorithme sur le tableau de taille $\lfloor n/2 \rfloor$: $T(\lfloor n/2 \rfloor)$
- en appliquant l'algorithme sur le tableau de taille $\lceil n/2 \rceil$: $T(\lceil n/2 \rceil)$

– effectuer un `merge` : $c_1 \cdot n + c_2$

Le total est donc :

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + c_1 \cdot n + c_2 + c_4$$

La récurrence est donc à résoudre avec le cas de base $T(1) = c_3$. Les feuilles de l'arbre de récurrence sont donc de valeur c_3 (que nous allons noter c). Pour **simplifier**, on ignore les termes constants négligeables, on suppose que n est une puissance de 2 et on pose $c = \max(c_1, c_3)$:

$$\boxed{T(n) = 2 \cdot T(n/2) + c \cdot n}$$

En résolvant l'arbre de récurrence, on obtient : $c \cdot n$ pour le total de chaque étage, et une hauteur de $\log n$, donc en comptant la racine de l'arbre :

$$\text{Coût du tri par fusion : } T(n) = \Theta(n \cdot \log n)$$

3.4 Conclusion

L'approche *divide-and-conquer* apporte encore un gain asymptotique par rapport au tri par insertion. Le problème ici est que le tri par fusion n'est **pas en place**, il requiert un espace auxiliaire en mémoire de taille $\Theta(n)$.

Cependant, le facteur multiplicatif devant le $\log(n)$ n'est pas rassurant vu qu'il s'agit d'un n. Il faudra mieux alors **appliquer le tri par insertion pour les tableaux de petite taille**, et le tri par fusion pour les tableaux de grande taille.

3.5 Tri par tas

Le tri par tas est une méthode de tri qui a été introduite en même temps que la structure de données qu'on appelle "**tas**", présentée dans le chapitre suivant. De n'importe quel tableau il est possible de former un **tas**, en place (sans devoir réattribuer de la place en mémoire) et sans surcoût lié à la taille du tableau car la création d'un tas de taille n est en $\Theta(n)$. La particularité d'un tas est que le maximum se trouve toujours en première position : on peut le mettre en fin de tableau avec une méthode en $\Theta(\log(n))$. Le tri par tas a un coût en $\Theta(n \cdot \log(n))$ et est en place.

3.6 Quicksort

Le *quicksort* consiste en une méthode de type *divide-and-conquer* "alternative" : on réordonne le tableau **autour d'un élément**, le pivot, de sorte que les éléments à gauche du pivot lui soient inférieurs et que les éléments à droite lui soient supérieurs.

$$A[0 : n-1] \longrightarrow A[q] \quad \text{t.q} \quad \begin{cases} \forall i \in [0, q-1] : & A[i] < A[q] \\ \forall i \in [q+1, n-1] : & A[i] > A[q] \end{cases}$$

3.6.1 Partitionnement

Principe

Le tri rapide consiste à trier des sous tableaux avec un appel récursif à chaque fois. Chaque appel récursif est de la forme QUICKSORT(A, LEFT, RIGHT). À chaque appel, on effectue une **partition du tableau** : c'est une méthode qui prend comme pivot l'élément à droite du tableau et qui

- place tous les éléments inférieurs à A[RIGHT] à gauche dans le tableau,
- place tous les éléments supérieurs à A[RIGHT] à droite dans le tableau,
- place le pivot à la séparation entre les deux zones.

Cela se fait en déterminant la valeur de A[RIGHT], en comparant chaque élément, de A[LEFT] à A[RIGHT-1], et en définissant un indice i correspondant à la frontière entre les zones "plus petit" et "plus grand". i est donc incrémenté de 1 à chaque fois qu'une valeur plus **petite** que A[RIGHT] est parcourue. Alors, cette valeur est échangée avec l'élément $i+1$ et i s'incrémente de 1.

Pseudo-code

```
1: function PARTITION(A, left, right)
2:     x = A[right]
3:     i = left − 1
4:     for j = left → right − 1 do          ▷ Traitement des éléments uns par uns
5:         if A[j] ≤ x then
6:             i = i + 1
7:             Swap A[i] with A[j]
8:     Swap A[i + 1] with A[right]          ▷ Placement du pivot au bon endroit
9:     return i + 1                         ▷ Renvoi de la position finale du pivot
```

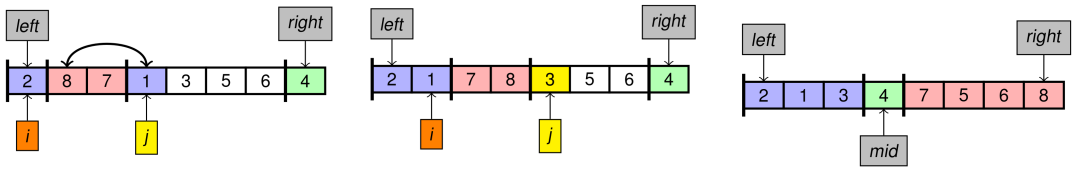


FIGURE 3.5 – Pseudo-code et illustration de la partition d'un tableau sur base d'un indice de gauche et un indice de droite. Valeur de retour : la position du pivot après partition.

Coût du partitionnement

Pour le **pire partitionnement**, qui est le premier, le programme parcourt tout le tableau. A chaque fois, il effectue des opérations constantes comme une comparaison, etc. Et en prime il y a quelques assignations dans le programme. Le cout pour partitionner le tableau est donc en $\Theta(n)$

$$T(n) = T(c_1 \cdot n + c_2) = \Theta(n)$$

3.6.2 L'algorithme du quicksort

L'algorithme du quicksort consiste à commencer par partitionner le tableau avec comme indices LEFT et RIGHT : ceci retournera la position du **pivot** (initialement l'argument

passé dans RIGHT de la fonction) une fois que les échanges de la figure **??** auront été effectués. Après ça, un quicksort sera appliqué sur le tableau de gauche puis sur le tableau de droite, sans préférence particulière.

Pseudo-code

```
1: procedure QUICKSORT(A, left, right)
2:     if left < right then
3:         mid =PARTITION(A, left, right)
4:         QUICKSORT(A, left, mid − 1)
5:         QUICKSORT(A, mid + 1, right)
```

FIGURE 3.6 – Pseudo-code de l'algorithme de tri rapide.

3.6.3 Complexité du tri rapide

A chaque appel de QUICKSORT, un appel à PARTITION est fait, un algorithme qui prend un temps $c_1 \cdot n + c_2$, et deux appels sont faits à QUICKSORT. Nous avons ainsi por la complexité totale :

$$\begin{array}{rlll} T(n) = & T_1(n) & = & c_1 \cdot n + c_2 \\ & +T_2(n) & & +T(n_{\text{left}}) \\ & +T_3(n) & & +T(n_{\text{right}}) \end{array}$$

où $n_{\text{left}} = \text{MID} - \text{LEFT}$ et $n_{\text{right}} = \text{RIGHT} - \text{MID}$, et MID est justement déterminé par l'appel à PARTITION. Autrement dit, le temps mis par l'algorithme dépend de la position du pivot : il dépend de l'instance du problème.

Complexité – meilleur cas

Etudions la complexité dans le meilleur des cas : c'est celui où le partitionnement est **parfaitement équilibré** $\Rightarrow n_{\text{left}} \approx n_{\text{right}} \approx n/2$. Alors, le temps mis par l'algorithme est donné par :

$$T(n) = 2 \cdot T(n/2) + c \cdot n = \Theta(n \cdot \log(n))$$

après résolution de l'arbre de récurrence.

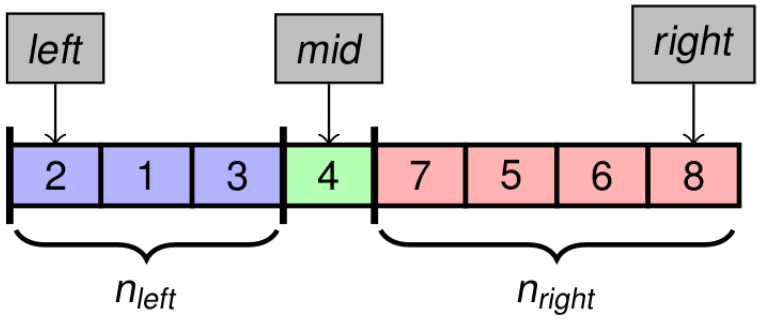


FIGURE 3.7 – Signification des variables n_{left} et n_{right} après un appel à PARTITION qui renvoie la valeur de MID.

Complexité – pire cas

Le pire cas correspond à la situation où le partionnement est **complètement déséquilibré**, et ce, à **chaque étape du tri** : $n_{\text{left}} = n - 1$ et $n_{\text{right}} = 0$ comme on peut l'imaginer sur la figure **??**. Dans ce genre de situations, la récurrence s'écrit :

$$T(n) = T(n-1) + c \cdot n$$

et l'arbre de récurrence est un arbre avec des branches d'un seul côté, à chaque noeud. La taille de l'arbre étant de n, on peut trouver facilement le coût de l'algorithme, car chaque étage est n décrémenté de 1. Plus précisément, pour le calcul de la complexité, on part du cas de base (le bas de l'arbre) $T(1) = c$, et on remonte chaque branche en incrémentant la hauteur de 1, et en ajoutant un $c \cdot i$ jusqu'à atteindre $c \cdot n$. Ainsi :

$$T(n) = \sum_{i=1}^{n} c \cdot i = c \cdot \frac{n(n+1)}{2} = \Theta(n^2)$$

Nous observons alors que pour un partionnement très déséquilibré, ce qui correspond par ailleurs à un tableau déjà trié (!), la complexité est en $\Theta(n^2)$. Pour le tri par insertion, le tri d'un tableau déjà trié prend un temps $\Theta(n)$... Que devons-nous conclure de ça ? Venons-nous d'apprendre une méthode de tri inutile ?

Nous allons montrer dans la section suivante qu'en **moyenne**, la complexité de l'algorithme QUICKSORT est en $\Theta(n \cdot \log(n))$.

Complexité – cas moyen

Si nous supposons que nous avons un partitionnement *légèrement déséquilibré*, du style $n_r = n/10$ et $n_l = 9n/10$, alors le raisonnement de la figure **??** mène à une complexité en $O(n \cdot \log(n))$. Nous constatons en effet que nous avons des étages à $c \cdot n$ qui finissent par

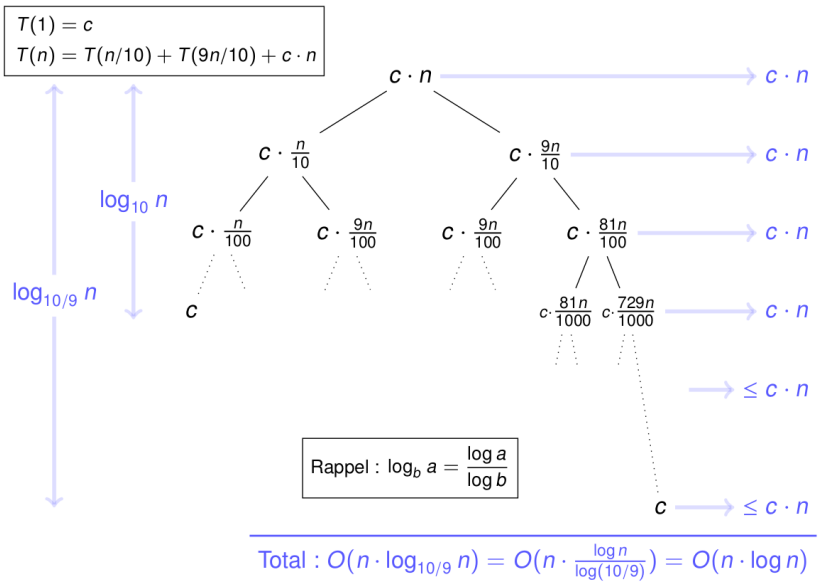


FIGURE 3.8 – Arbre de récurrence de l'algorithme QUICKSORT si le partitionnement est tel que $n_r = n/10$ et $n_l = 9n/10$ **à chaque étape**.

se terminer à une hauteur de $\log_{10}(n)$ tandis que d'autres branches continuent jusqu'à

Pseudo-code : Partitionnement randomisé

```
1: function RANDOMIZED-PARTITION(A, left, right)
2:     i =RANDOM(left, right)                    ▷ Tirage aléatoire du pivot
3:     Swap A[i] with A[right]                   ▷ Placement du pivot à la fin du tableau
4:     Return PARTITION(A, left, right)
```

Pseudo-code : Quicksort randomisé

```
1: procedure RANDOMIZED-QUICKSORT(A, left, right)
2:     if left < right then                      ▷ Code identique à la version déterministe
3:         mid =RANDOMIZED-PARTITION(A, left, right)
4:         RANDOMIZED-QUICKSORT(A, left, mid − 1)
5:         RANDOMIZED-QUICKSORT(A, mid + 1, right)
```

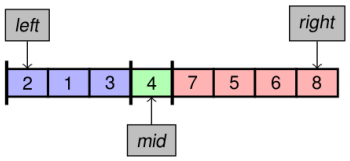


FIGURE 3.9 – Pseudo-code de l'algorithme de tri rapide avec partitionnement aléatoire.

une hauteur $\log_{10/9}(n)$: il y a moins d'un total de $c \cdot n$ à ce dernier étage. Nous pouvons donc borner supérieurement à l'aide d'un O le temps mis par l'algorithme :

$$T(n) = O(n \log_{10/9}(n)) = O(n \cdot \log(n)) \ .$$

Si nous voulons nous **assurer** d'avoir un partitionnement meilleur que $(9/10, 1/10)$, il faut que le pivot soit soit plus grand soit plus petit que 10% des éléments : cela arrivera dans 80% des cas.

3.6.4 Tri rapide avec aléatoire

Comme on vient de le voir, il n'est pas toujours intéressant d'appliquer le tri rapide quand on prend comme pivot l'élément le plus à droite du tableau. Pour assurer la légitimité du tri rapide, nous allons introduire le fait qu'une **analyse probabiliste rigoureuse non-vue au cours** permet d'assurer que *si le pivot est choisi au hasard parmi tous les éléments du tableau, alors en moyenne, la complexité de l'agorithme de tri rapide est en* $\Theta(n \cdot \log(n))$*, et ce quel que soit l'ordre initial du tableau.*

3.7 Tris par comparaison – borne inférieure pour le tri

Faisons un récapitulatif des algorithmes de tri vus jusqu'à présent :

	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n \cdot \log(n))$
Pire cas	Tri par insertion		Mergesort Heapsort
Meilleur cas		Tri par insertion	
En moyenne			Quicksort

3.7.1 Peut-on encore faire mieux ?

C'est ambitieux de vouloir un temps plus court que du $\Theta(n \cdot \log(n))$: on a déjà vu 4 algorithmes et 3 de ceux-ci rivalisent très clairement mais s'alignent selon ce temps.

3.7.2 Arbres de décision

En réalité, on peut montrer que **si le tri est un tri par comparaison**, c'est-à-dire si l'ordre final est déterminé uniquement par comparaisons antre éléments, alors un algorithme de tri en $\Theta(n \cdot \log(n))$ est optimal. C'est en dressant l'arbre de décision du tri que l'on détermine le nombre de comparaisons à effectuer au minimum pour trier les éléments d'un tableau qu'on se rend compte que cette borne inférieure est $n \cdot \log(n)$.

Dans cet arbre, nous observerons les éléments suivants :

- h la hauteur du tableau : la hauteur de la plus basse feuille. C'est aussi le nombre **maximum de comparaisons** à effectuer pour trier un tableau.
- $l \leq 2^h$ le nombre de feuilles, car l'arbre est un arbre **binaire**
- $n! \leq l$ le nombre d'ordres finaux possibles pour n éléments : $n! = l$ pour les algorithmes déterministes.

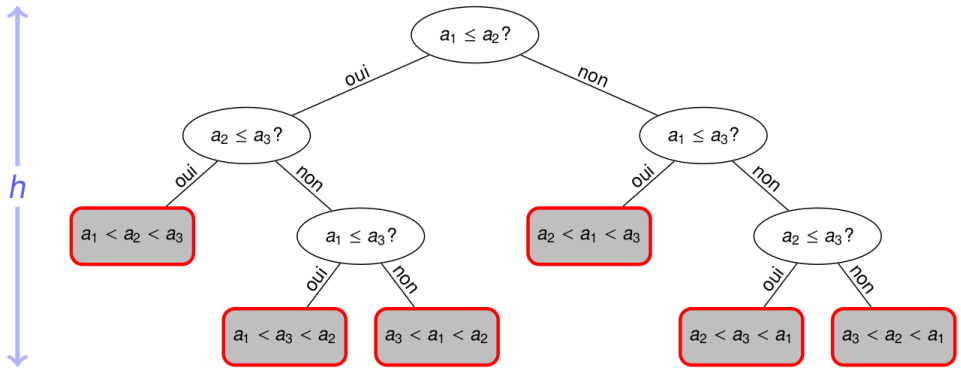


FIGURE 3.10 – Arbre de décision lors du tri d'un tableau de 3 éléments. Nombre d'ordres finaux possibles : $n!$

A la lumière des notions introduites, nous pouvons écrire sereinement

$$h \geq \log(n!) \ ,$$

et étant donné le théorème suivant très important pour le raisonnement, nous pourrons conclure sur une borne inférieure pour le tri par comparaison.

$$\text{Théorème : } \log(n!) = \Theta(n \cdot \log(n))$$

Ainsi,

$$h \geq \log(n!) = \Theta(n \cdot \log(n))$$

$\Rightarrow$ tout algorithme requiert donc $\Omega(n \cdot \log(n))$ comparaisons pour faire un tri par comparaison.

3.7.3 Conclusion

Tous les algorithmes de tri doivent faire au moins $\Omega(n \cdot \log(n))$ comparaisons. La complexité est donc en $\Omega(n \cdot \log(n))$, ce qui consitue une **borne inférieure**. Or pour certains algorithmes, comme MERGESORT ou HEAPSORT, la complexité est en $\Theta(n \cdot \log(n))$ dans le pire des cas : ils sont donc **optimaux pour le modèle de tri par comparaison**.

3.8 Tri linéaire

Le tri linéaire n'est pas un tri par comparaison. Comme son nom l'entend, son but est de trier un tableau en un temps linéaire.

3.8.1 Tri par comptage (counting sort)

Très brièvement, le but est d'ici trier un tableau en comptant les valeurs en doublon dans le tableau, et ensuite de créer un tableau trié dans lequel on copie les éléments du premier tableau. Attention : ici on suppose qu'un élément du tableau est un objet et pas juste un entier, il faut donc s'assurer de copier l'objet lui-même et pas juste la valeur de la clé !

Nous allons appliquer cet algorithme sur un tableau de n objets où les clés prennent k valeurs différentes.

3.8.2 Pseudo-code du tri par comptage

Le pseudo code consiste en 4 boucles :

1. La première boucle itère sur toutes les k valeurs que peuvent prendre les clés : on initie le tableau de comptage $C \Rightarrow$ coût en $\Theta(k)$.
2. La deuxième boucle est une itération sur tous les éléments du tableau, et on incrémente $C[A[j]]$ pour tous les j allant de 0 jusqu'à $n-1$: c'est un coût en $\Theta(n)$.
3. La troisième boucle est assez bizarre : elle fait contenir dans $C[i]$ le nombre d'éléments dont la clé est inférieure à i. Ca a l'air bizarre mais c'est utile pour la boucle d'après $\Rightarrow$ coût en $\Theta(k)$.
4. Enfin, la dernière boucle place $A[j]$ dans le tableau trié : $B[C[A[j]] - 1] = A[j]$, et puis décrémente $C[A[j]]$ de 1 pour "mettre à jour la place du prochain élément" $\Rightarrow$ coût en $\Theta(n)$.

Le coût total est en $\Theta(n+k)$, ce qui revient à $\Theta(n)$ si $k = O(n)$. C'est donc possible d'avoir un coût plus faible que $\Theta(n \cdot \log(n))$ car les informations sont obtenues autrement que par comparaison d'éléments.

Il s'agit d'un tri **stable**, c'est-à-dire que si deux objets se suivaient dans un certain ordre dans le tableau initial et qu'ils ont la même clé, ils se suivront dans le même ordre dans le tableau trié.

Cepedant, on peut remarquer que ce tri n'est intéressant que pour les k petits, sinon la complexité évolue en $\Omega(k^2)$, et que le tri n'est pas en place.

Pseudo-code

```
1:  procedure CountingSort(A, B, n, k)
2:      for i = 0 → k − 1 do                          ▷ Initialisation du tableau de comptage
3:          C[i] = 0
4:      for j = 0 → n − 1 do                          ▷ C[i] va contenir le nombre d'éléments dont la clé vaut i
5:          C[A[j]] = C[A[j]] + 1
6:      for i = 1 → k − 1 do                          ▷ C[i] va contenir le nombre d'éléments dont la clé est ≤ i
7:          C[i] = C[i] + C[i − 1]
8:      for j = n − 1 downto 1 do
9:          B[C[A[j]] − 1] = A[j]                     ▷ Placement de A[j] dans le tableau trié B
10:         C[A[j]]] = C[A[j]] − 1                    ▷ Mise-à-jour de la place du prochain élément = i
```

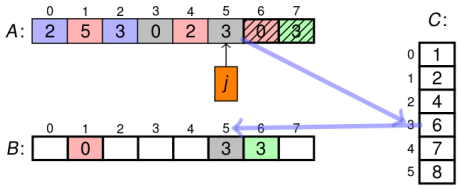


FIGURE 3.11 – Pseudo-code du tri linéaire.

3.9 Tri Radix

Chapitre 4

Structures de données

4.1 Définitions

Structure de données	**manière d'encoder** un ensemble (dynamique) d'objets en mémoire de sorte à pouvoir implémenter certaines opérations.
Type de données abstrait	**spécification d'un ensemble d'opérations** pouvant être supportées par une structure de données.

TABLE 4.1 – Différence entre structure de données et type de données abstrait

Identifier l'ensemble d'opérations devant être supportées (= le type de données abstrait) est, dans la conception d'un modèle de données, l'étape prérequise avant de choisir la structure de données la plus appropriée. Par exemple, pour concevoir une intelligence artificielle efficace capable d'imaginer une vingtaine de tours à l'avance sur un plateau de jeu, il faut que ce plateau soit encodé selon une structure de données qui implémente les opérations `insert`, `delete`, `maximum`, `predecessor` en un temps, idéalement, en $O(\log(n))$. En effet :

- Il faut pouvoir insérer un pion dans un tableau fictif pour imaginer le coup à l'avance
- Il faut pouvoir supprimer ce pion pour revenir en arrière dans nos décisions
- Il faut pouvoir chercher le maximum pour sélectionner le coup le plus avantageux si chaque case du plateau de jeu sera marquée d'un attribut de "qualité"
- Il faut pouvoir chercher le prédecesseur d'une case si jamais une case relevant de la recherche du maximum amène dans une situation peu avantageuse (pour prendre le deuxième maximum)
- ...

Sur la figure **??** on peut voir les structures de données que nous connaissons grâce aux TPs. On remarque qu'elles ne permettent pas de tout faire en un temps $\Theta(\log(n))$.

Nous allons dans ce cours voir trois nouvelles structures de données : les **tas-max**, les **arbres binaires de recherche**, et les **tables de hachage**.

	Insert	Delete	Search	Minimum	Maximum	Predecessor	Successor
Tableau	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Tableau trié	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Liste simplement chaînée	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Liste doublement chaînée	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Liste simplement chaînée triée	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Liste doublement chaînée triée	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$

FIGURE 4.1 – Comparaison des coûts des opérations classiques entre les structures de données connues.

4.2 Tas-max et queue de priorité

Disclaimer

Quand on dit "max" et "maximum", tout le raisonnement se fait également en disant "min" et "minimum".

4.2.1 Queue de priorité max

Une queue de priorité **max** est un type de données abstrait dont les opérations à faire supporter sont les suivantes :

`Insert(S,x)`	Insérer l'élément `x` dans la queue `S`.
`Maximum(S)`	Renvoie l'élément de `S` avec la plus grande clé.
`Extract-Max(S)`	Renvoie et supprime l'élément de `S` avec la plus grande clé.
`Increase-Key(S,x,k)`	Augmente la valeur de la clé de l'élément `x` jusqu'à la nouvelle valeur `k` (devant être supérieure à l'ancienne valeur).

Aucune structure de données déjà vue (présente sur la structure **??**) permet d'effectuer ces opérations en un temps acceptable (constant ou $\Theta(\log(n))$). Il nous faut une nouvelle structure de données, et il est temps d'introduire notre première structure de donnée inédite : les **tas-max**.

4.2.2 Tas

Un tas est une structure de données implémentée par un tableau et qui peut être visualisé sous la forme d'un arbre binaire (presque complet). Pour le parcourir, la notice est présentée à la figure **??**. Cette structure de donnée permettra, une fois complétée de certaines propriétés, d'effectuer **toutes les opérations de la queue de priorité en un temps** $O(\log(n))$.

Parcourir l'arbre

- Racine de l'arbre : $i = 1$ (premier élément du tableau)
- PARENT$(i) = \lfloor i/2 \rfloor$ (parent du noeud i)
- LEFT$(i) = 2i$ (enfant de gauche du noeud i)
- RIGHT$(i) = 2i + 1$ (enfant de droite du noeud i)
- HEIGHT$(i) = \lfloor \log n - \log i \rfloor$ (longueur du plus long chemin vers une feuille)

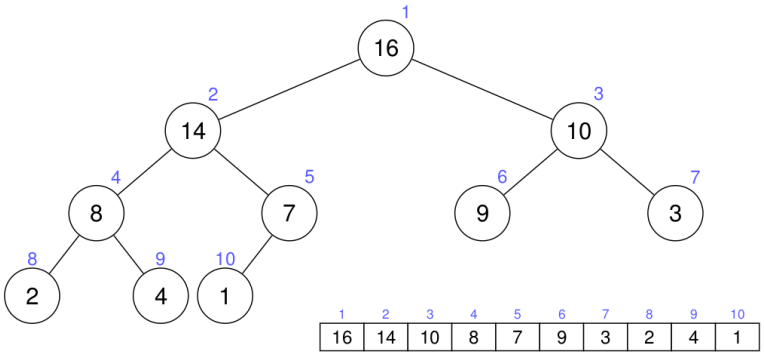


FIGURE 4.2 – Parcours d'un tax

4.2.3 Tas-max

Un tax-max est un tas tel que

$$A\,[\text{PARENT}(i)] \geq A[i] \quad \forall i \neq \text{ROOT}$$

Opérations à implémenter

Construction et maintien d'un tas-max	
BUILD-MAX-HEAP	construit un tas-max à partir d'un tableau désordonné.
MAX-HEAPIFY	corrige une violation unique de la prorpriété de tas

Opérations de queue de priorité	
HEAP-MAXIMUM	renvoie l'élément avec la plus grande clé.
HEAP-EXTRACT-MAX	renvoie et supprimer l'élément avec la plus grande clé

Opérations de tri	
HEAPSORT	trier un tableau en place.

MAX-HEAPIFY

Cette méthode consiste à corriger une violation **unique** de la propriété de tas. Un noeud peut vérifier la propriété de tas sans pour autant que ses enfants la vérifient !

Si un noeud i viole la propriété de tas-max, c'est qu'un de ses enfants est plus grand que lui. L'idée sera alors d'échanger l'élément au noeud i par son plus grand enfant, et vérifier qu'on n'a pas ce faisant généré une violation au noeud échangé. La complexité

```
1:  procedure MAX-HEAPIFY(A, i)
2:      l =LEFT((i))
3:      r =RIGHT((i))
4:      if l ≤ A.heap-size and A[l] > A[i] then
5:          largest = l
6:      else
7:          largest = i
8:      if r ≤ A.heap-size and A[r] > A[largest] then
9:          largest = r
10:     if largest ≠ i then                    ▷ Si il y a violation de la propriété de tas
11:         Swap A[i] with A[largest]
12:         MAX-HEAPIFY(A, largest)            ▷ Appel récursif
```

FIGURE 4.3 – Pseudo-code de l'opération de correction d'une violation unique de la propriété de tas-max, MAX-HEAPIFY

de cette méthode s'évalue facilement : on a une récurrence

$$\begin{aligned} \text{Cas de base :} \quad T(0) &= c \\ T(h) &= c + T(h-1) \\ \Rightarrow T(h) &= c + h \cdot c \end{aligned}$$

où h est la hauteur du noeud i. Celle-ci est toujours inférieure à $\log(n)$, il est donc correct de conclure en disant que MAX-HEAPIFY a une complexité en $O(\log n)$, car

$$\text{HEIGHT}(i) = \lfloor \log(n) - \log(i) \rfloor \leq \log(n)$$

BUILD-MAX-HEAP

Le but est ici de corriger dans un tas **toutes les violations** de la propriété de tas, pas juste celle à un noeud. On va alors parcourir toutes les feuilles et appliquer la méthode MAX-HEAPIFY sur chacun des noeuds parents pour vérifier que la propriété est respectée et de la faire respecter si non. On va donc commencer depuis le **parent du dernier élément**, à savoir

$$\text{PARENT}(n) = \lfloor n/2 \rfloor$$

On va donc appliquer MAX-HEAPIFY sur chacun des noeuds de l'arbre, soit $\lfloor n/2 \rfloor$ noeuds. Le coût de MAX-HEAPIFY vaut dans le pire des cas $\Theta(h)$ pour un noeud à la hauteur h, donc $O(\log(n))$. La complexité de BUILD-MAX-HEAP est donc en $O(n \cdot \log(n))$.

Une analyse plus fine est obtenue, comme avant, en sommant ce qu'il y a à sommer. À chaque étage de hauteur h, il y a environ $\frac{n}{2^{h+1}}$ noeuds. Et pour chaque noeud à la hauteur h, le traitement de ce noeud prend un temps linéaire selon h : $T_h = c \cdot h$. Le coût total est alors obtenu en sommant sur chaque hauteur, de $h = 1$ à $h = \log(n) - 1$.

Pseudo-code

```
1: procedure BUILD-MAX-HEAP(A)
2:     n = A.heap-size
3:     for i = ⌊n/2⌋ downto 1 do          ▷ Pour tous les noeuds sauf les feuilles
4:         MAX-HEAPIFY(A, i)
```

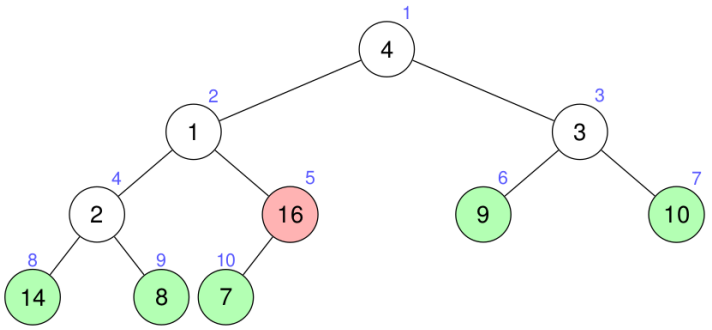


FIGURE 4.4 – Pseudo-code de l'opération BUILD-MAX-HEAP qui consiste à construire un tas-max à partir de n'importe quel tas désordonné.

$$\begin{aligned} T(n) &= \sum_{h=1}^{\log(n)-1} \frac{n}{2^{h+1}} \cdot c \cdot h \\ &= \frac{c \cdot n}{2} \sum_{h=1}^{\log(n)-1} \frac{h}{2^h} \\ &= \Theta(n) \end{aligned}$$

Et c'est ainsi qu'on démontre non seulement que la vraie complexité de la construction d'un tas-max est $\Theta(n)$ et non $O(n \cdot \log(n))$ et que la création d'un tas-max prend exactement le même temps que créer un tableau désordonné de n éléments.

EXTRACT-MAX

Le maximum se trouve à la racine. On pourrait le supprimer directement en créant un trou dans l'arbre, mais ce n'est pas trop ce qu'on a envie de faire. Pour ne pas créer de trou, on préfère remplacer la racine par le dernier élément, puis d'appliquer MAX-HEAPIFY sur la racine. La complexité de cette méthode s'évalue simplement : une permutation de deux éléments $\Theta(1)$ et appeler MAX-HEAPIFY sur la racine de hauteur $\log(n)$ d'où un coût en $\Theta(\log(n))$.

4.2.4 Tri par tas (heapsort)

Le tri par tas consiste à profiter de la structure d'un tas-max pour trier un tableau désordonné. Comme on l'a déjà vu, pour le transformer en tas, il faut appliquer BUILD-MAX-HEAP ce qui prend un temps $\Theta(n)$. Ensuite, pour trier ce tas, on a vu qu'on pouvait extraire le maximum avec EXTRACT-MAX en un temps $\Theta(\log(n))$ en l'échangeant avec le dernier élément du tableau (dernier en termes d'indices). En appliquant EXTRACT-MAX n fois, on aura trié le tableau.

La complexité d'un tri par tas est donc en $\Theta(n \cdot \log(n))$ soit la même que le tri par fusion. L'avantage ici est que le tri est 100% en place.

Pseudo-code

```
1: procedure HEAP-EXTRACT-MAX(A)
2:     n = A.heap-size
3:     Swap A[1] and A[n]                    ▷ Place le max en dernière position
4:     max = A[n]
5:     A.heap-size = A.heap-size − 1         ▷ Supprime le dernier élément du tas
6:     MAX-HEAPIFY(A, 1)
7:     return max
```

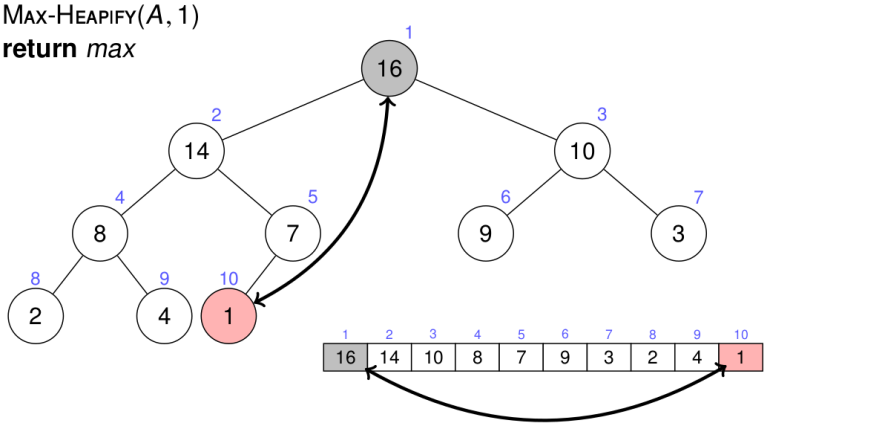


FIGURE 4.5 – Pseudo-code de la méthode EXTRACT-MAX

Tri par tas : $T(n) = \Theta(n \cdot \log(n))$ et en place

4.3 Choix et conception d'une structure de données

À chaque application sa structure de données propre. Afin de la choisir, il faut, dans l'ordre :

1. Déterminer les opérations nécessaires (spécification du type de données abstrait)
2. Choisir une structure de données appropriée (qui limite le coût des opérations nécessaires)
3. Implémenter la structure de données
4. Débuguer

Pour débuguer, il faut faire appel à la notion d'**invariant de représentation**.

Invariant de représentation

Un invariant de représentation est une propriété d'une structure de données qui reste vérifiée à tout moment de l'exécution. Ca peut être l'ordre pour un tableau trié, la propriété de tas pour un tas-max, ...

Une erreur lors de l'implantation d'une opération peut violer cet invariant et provoquer d'autres bugs par la suite. Une bonne stratégie de débugage serait alors d'implémenter une méthode qui vérifie si l'invariant est toujours respecté, et de l'appliquer avant et après certaines opérations pour localiser où surviendrait la potentielle couille qui rend le potage légèrement aigre.

4.4 Taille de tableau

Généralement, on crée un tableau (sur base d'une certaine structure) et puis on le complète. On lui fixe une taille dans la mémoire à sa création, mais il se peut qu'on soit un peu trop prévoyant et qu'on prévoie m places pour au final remplir $n \ll m$ éléments. En outre, si nous voulons ajouter un élément et qu'on se rend compte qu'on dépasse la taille prévue, il faut créer un autre tableau plus grand en mémoire et copier l'ancien tableau dans le nouveau, ce qui prend un temps $\Theta(n)$. De même si l'on supprime beaucoup d'éléments : on aura peut-être pour finir un gros emplacement non utilisé en mémoire, ce dont il faut s'occuper en créant un tableau plus petit. On se retrouve encore avec du $\Theta(n)$.

Dans cette section, on cherche à optimiser à la fois la place occupée en mémoire et le temps passé à faire ces opérations de création de nouveaux tableaux.

4.4.1 Stratégie lors de l'insertion – doublement de tableau

Il faut déterminer une règle pour redimensionner le tableau.

Approche naïve

L'approche naïve consiste à dire qu'à chaque insertion on crée un tableau de taille m' tel que

$$m' = m + 1$$

Cela veut dire qu'à chaque insertion, on doit effectuer un redimensionnement et donc une copie du tableau vers un nouveau en mémoire. En commançant avec un tableau de taille 1, puis 2, puis $2 + \cdots + n$ pour ajouter n éléments. On calcule le temps pris pour l'insertion de n éléments :

$$\Theta(1 + 2 + \cdots + n) = \Theta\left(\frac{n(n+1)}{2}\right) = \Theta(n^2)$$

Cela donne donc en moyenne $\Theta(n)$ par insertion.

Doublement de tableau

Le doublement de tableau consiste à **redimensionner le tableau à chaque fois que n est multiplié par 2**. À chaque fois :

$$m' = 2 \cdot m \ .$$

Ainsi, le surcoût dû aux redimensionnements est calculé selon :

$$\Theta(1 + 2 + 4 + \cdots + n) = \Theta\left(\sum_{i=0}^{\log(n)} 2^i\right) = \Theta(n)$$

Donc un coût "moyen" par insertion de $\Theta(1)$. On dit que la méthode INSERT a un coût **amorti** de $\Theta(1)$. Le redimensionnement n'implique pas de surcoût en moyenne.

4.4.2 Stratégie à la suppression

Il faut également redimensionner le tableau lors de la suppression d'un grand nombre d'éléments. Typiquement, on aimerait, pour un tableau de taille m rempli de n éléments, maintenir la propriété suivante :

$$C \cdot m \leq n \leq m \quad , \, C < 1 \, ,$$

et il faut choisir cette constante.
Si nous choisissons $C = 1/2$, alors la taille du tableau diminue de moitié lorsque la moitié manquera. Cependant, si nous voulons ajouter un élément juste après, alors il faudra réagrandir le tableau si on suit le critère précédent. n suppresions/insertions provoquera donc n redimensionnements, soit un coût total en $\Theta(n^2)$ ou un coût amorti en $\Theta(n)$.

Si nous choisissons $C = 1/4$, alors on a quelque chose de bien plus beau, car il faudra réinsérer $m'/2$ éléments avant de devoir redimensionner. Le coût amorti restera donc en $\Theta(1)$.

4.5 Arbres de recherche

4.6 Tables de hachage

Première partie

Techniques de programmation

Chapitre 5

Backtracking

Nous revenons au but de la vie de l'ingénieur : résoudre des problèmes. Quand on accepte que les ordinateurs fassent partie de notre formidable vie, on fait aussi le choix de les laisser gérer nos problèmes du quotidien pour nous (sauf ceux avec les femmes, malheureusement aucun algorithme assez puissant peut nous sortir de la merde avec elles). Si il existe plusieurs solutions pour répondre à un problème, l'algorithme va **toutes les trouver**. Mais s'il y a plusieurs possibilités sans pour autant qu'elles soient toutes possibles, l'algorithme va quand même devoir toutes les essayer, ce qui prendrait un temps fou ! À moins qu'interviennent les humains intelligents qui mettent en place des techniques splendides pour optimiser le calcul. Parmi elles, **le backtracking**.

Très brièvement, le backtracking consiste à rechercher une solution en essayant plusieurs voies, et revenir en arrière lorsqu'une de ces voies s'avère incorrecte, et alors choisir une autre.

5.1 Génération exhaustive

Ce concept est le plus idiot et chronophage possible, et on se demande à quoi elle sert parce qu'il ne s'agit pas de résoudre un problème, il s'agit de lister tous les candidats possibles et imaginables. Le pseudo-code de la génération exhaustive est présent sur la figure **??**. On utilise déjà là en réalité le backtracking puisqu'à chaque position on

Canevas général

```
1: procedure GENERATE(i)
2:     if Finished then
3:         Print generated element
4:     else
5:         for Choice in PossibleChoices do
6:             Build partially generated element according to Choice
7:             GENERATE(i + 1)
8:             Undo last choice
```

FIGURE 5.1 – Pseudo-code de la génération exhaustive, en ayant en tête une chaîne que l'on remplit progressivement jusqu'à une taille n. Lorsque $i = n$ on est dans la situation FINISHED

entame une nouvelle version de la chaîne à partir de l'élément d'après (GENERATE(i+1))

mais on fait un UNDO pour revenir en arrière et prendre un nouveau choix qui va mener à une différente situation d'arrivée.

5.2 Recherche exhaustive

La **recherche exhaustive** est premièrement un poil plus exigeante que la génération exhaustive car elle ne demande d'imprimer que les candidats qui sont solutions. Ensuite, il advient en toute logique que nous n'aimerions pas explorer des pistes qui ne mènent de toute façon pas à de bons résultats. Sur base de notre situation, nous seront amenés à définir des critères qui consisteront à dire si un choix est admissible ou non. Un canevas de recherche exhaustive optimisée est représenté sur la figure **??**.

Canevas général

```
1:  procedure TEST(i)
2:      if Finished then
3:          if Candidate is Solution then
4:              Print Candidate
5:      else
6:          for Choice in PossibleChoices do
7:              if Choice is admissible then      ▷ On ne teste que les choix admissibles
8:                  Build partial Candidate according to Choice
9:                  TEST(i + 1)
10:                 Undo Choice
```

FIGURE 5.2 – Pseudo-code de la recherche exhaustive, où la notion de **choix admissible** apparaît, contrairement à la génération exhaustive où la seule notion intéressante était la notion de **choix possible**.

5.3 Recherche d'une seule solution

On commence ici à modifier les pseudo-codes précédents en y incluant la possibilité qu'une solution soit trouvée. Pour rappel, jusque là, on appliquait la méthode TEST récursivement sur la position i de la chaîne en augmentant $i + 1$ à chaque appel, jusqu'à arriver à $i = n$ où la chaine est terminée et on vérifie si la proposition candidate est solution ou non.

Maintenant, on ajoute les **retours** de la fonction : à chaque fois qu'on arrive à $i = n$, si on a une solution on renvoie `true`, sinon on renvoit `false`. Ensuite, pour chaque appel récursif TEST$(i + 1)$, on retourne `true` si le choix qu'on a effectué mène à un chemin qui mène à une solution, false sinon. Le canevas général de la recherche d'une solution est affiché sur la figure **??**

Intégration dans le canevas général

```
1:  function TEST(i)
2:      if Finished then
3:          if Candidate is Solution then
4:              Print Candidate
5:              return True                    ▷ Renvoie vrai dès qu'une solution est trouvée
6:      else
7:          for Choice in PossibleChoices do
8:              if Choice is admissible then
9:                  Build partial Candidate according to Choice
10:                 if TEST(i + 1) then
11:                     return True▷ Renvoie vrai si une solution basée sur ce choix est trouvée
12:                 Undo Choice
13:     return False                           ▷ Renvoie faux si aucune solution n'a été trouvée
```

FIGURE 5.3 – Pseudo-code de la recherche exhaustive d'une solution en général.

Exemple

Pour le problème de remplissage d'un sac, la figure ci-dessous reprend le pseudo-code de la recherche d'une solution. On remarque par exemple que le critère d'admission d'un choix se base sur le débordement ou non du sac lors de l'ajout de l'objet en question. S'il peut être ajouté, alors on l'ajoute et on vérifie que ça mène peut-être à une solution. Si ça mène en effet à une solution (TEST$(i+1)$ renvoie `true`), la fonction s'arrête en renvoyant `true`. Si non, on fait une **marche arrière** (lignes 12 et 13) et on enlève l'objet du sac. On vérifie que ne pas mettre l'objet dans le sac permet en effet de résoudre le problème et si c'est le cas la fonction renvoie `true`, sinon elle renvoie `false`.

Pseudo-code : génération d'une solution

```
1:  function TEST(n, i)                        ▷ Fonction à valeur de retour booléenne
2:      if i == n then
3:          if VolTot == VolSack then
4:              Print s
5:              return True                    ▷ Renvoie vrai dès qu'une solution est trouvée
6:      else
7:          if VolTot + Vol[i] ≤ VolSack then
8:              s[i] = 1
9:              VolTot = VolTot + Vol[i]
10:             if TEST(n, i + 1) then
11:                 return True    ▷ Renvoie vrai si une solution basée sur ce choix est trouvée
12:             VolTot = VolTot − Vol[i]
13:         s[i] = 0
14:         if TEST(n, i + 1) then
15:             return True        ▷ Renvoie vrai si une solution basée sur ce choix est trouvée
16:     return False                           ▷ Renvoie faux si aucune solution n'a été trouvée
```

5.4 Recherche de la meilleure solution

Si on désire rechercher la meilleure solution, il suffira de sauvegarder chaque solution trouvée dans une variable temporaire et de l'écraser par une nouvelle solution arrivante

si cette-dernière est meilleure. Le pseudo-code serait alors celui de la figure **??**

```
Canevas général
1: procedure TEST(i)
2:     if Finished then
3:         if Candidate is Solution and Better Solution then
4:             Save Candidate                ▷ On sauvegarde la solution si elle est meilleure
5:     else
6:         for Choice in PossibleChoices do
7:             if Choice is admissible then
8:                 Build partial Candidate according to Choice
9:                 TEST(i + 1)
10:                Undo Choice
```

FIGURE 5.4 – Pseudo-code de la recherche de la meilleure solution d'un problème.

5.5 Branch-and-bound

La recherche d'une meilleure solution par recherche exhaustive passe tout de même par un semblant de génération exhaustive d'un nombre potentiellement exponentiel de candidats. Le coût est donc **typiquement exponentiel**. Comme on est jamais content, on est pas content là non plus. Il faut venir avec une nouvelle technique.

Le **branch-and-bound** est une technique qui consiste à réduire le coût en évitant de générer certains candidats, typiquement ceux qui ne mèneraient pas à une meilleure solution. On ne vérifie donc pas seulement que le choix est admissible, mais qu'il ne puisse pas donner un résultat moins bon que la meilleure solution initialement trouvée. Si il peut mener à une meilleure solution, alors on applique TEST à $i + 1$.

```
Canevas général
1: procedure TEST(i)
2:     if Finished then
3:         if Candidate is Solution and Better Solution then
4:             Save Candidate
5:     else
6:         for Choice in PossibleChoices do
7:             if Choice is admissible then
8:                 Build partial Candidate according to Choice
9:                 if Choice could lead to better Solution then
10:                    TEST(i + 1)
11:                Undo Choice
```

FIGURE 5.5 – Pseudo-code de la méthode de branch-and-bound

Chapitre 6

Programmation dynamique

6.1 Bilan de mi-parcours

Nous avons vu les 5 étapes de conception d'un algorithme en programmation dynamique :

ETAPES DE CONCEPTION – PROGRAMMATION DYNAMIQUE

1. Déterminer les "choix" pour construire une solution partielle
2. Définir les sous-problèmes
3. Relier les solutions des sous-problèmes entre elles (relation de récurrence)
4. Concevoir l'algorithme proprement dit : soit à partir de la formule de récurrence (algorithme récursif avec mémoïsation), soit en établissant le graphe de dépendance (algorithme de bas en haut)
5. Résoudre le problème initial (fonction principale)

6.1.1 Types de sous-problèmes

Nous allons voir qu'on a des types de sous-problèmes qui reviennent souvent et qu'on peut classifier. Remarquons déjà qu'en général, un problème traité en programmation dynamique peut se modéliser via un tableau comme ça l'a toujours été jusqu'à présent. Considérons un tableau $T[0 : n-1]$. Il y aura comme possibilité de sous-problèmes :

Type	Description	# sous-problèmes
Suffixes	$T[i : n-1]$ pour tout i	n
Préfixes	$T[0 : i]$ pour tout i	n
Sous-tableaux	$T[i : j]$ pour tous $i \leq j$	$\sim n(n+1) = \Theta(n^2)$

TABLE 6.1 – Types de sous-problèmes à traiter les de la résolution d'un problème par une technique de programmation dynamique.

6.2 Exemples avancés

6.2.1 Parenthésage

Le problème de parenthésage concerne la multiplication matricielle, et plus précisément lorsque nous sommes amenés à évaluer une multiplication d'une chaîne de matrices, soit

$$M = A_1 \cdot A_2 \cdot ... \cdot A_{n-1} \cdot A_n$$

et que nous cherchons à **établir le parenthésage** qui rende la multiplication plus rapide. Il s'agit en gros de l'endroit où séparer deux termes d'une paire de parenthèses dans le calcul de M (et de même pour la séparation en deux termes de chaque terme). On peut en effet montrer en prenant l'exemple

$$M = A_1 \cdot A_2 \cdot A_3$$

où

- A_1 est de taille $n \times 1$ (matrice colonne)
- A_2 est de taille $1 \times n$ (matrice ligne)
- A_3 est de taille $n \times 1$ (matrice colonne)

qu'il existe un parenthésage optimal. En effet, si nous considérons le parenthésage $(A_1 \cdot A_2) \cdot A_3$, le coût du calcul de $M_1 = A_1 \cdot A_2$ vaut $n \cdot 1 \cdot n = n^2$ et celui de $M = M_1 \cdot A_3$ vaudra $n^2 n \cdot n \cdot 1 = n^2$. Le coût de la multiplication en chaîne de ces 3 matrices sera en $\Theta(n^2)$.

Si nous choisissons en revanche l'option qui consiste à calculer d'abord $A_2 \cdot A_3$, alors nous arriverons à diminuer le coût total. En effet : le coût pour $M_2 = A_2 \cdot A_3$ (scalaire de taille $1 \times \underline{1}$) est de $1 \cdot n \cdot 1$ et celui de $A_1 \cdot M_2$ sera de $n \cdot 1 \cdot \underline{1} = \Theta(n)$ ce qui est meilleur que $\Theta(n^2)$!

Les 5 étapes

Nous allons maintenant décrire le problème de parenthésage général et l'explorer par la méthode de programmation dynamique, en commençant d'abord par l'analyser selon les 5 étapes.

1. **Choix.**
   Par "choix", nous entendons la chose suivante. Nous voyons le problème global :

   > $M = A_1 \cdot A_2 \cdot ... \cdot A_{n-1} \cdot A_n$ dont le coût est à optimiser en calculant au préalable deux termes : c'est-à-dire effectuer le produit
   >
   > $$M = (A_1 \cdot A_2 \cdot ... \cdot A_j) \cdot (A_{j+1} \cdot A_{j+2} \cdot ... \cdot A_n)$$

   Le choix suivant consiste à optimiser le calcul des deux termes par la même méthode :

   $$((A_1 \cdot ... \cdot A_{i-1}) \cdot (A_i \cdot ... \cdot A_j)) \quad \cdot \quad ((A_{j+1} \cdot ... \cdot A_k) \cdot (A_{k+1} \cdot ... \cdot A_n))$$

2. **Sous-problèmes.**
   Les sous-problèmes du premier choix sont consistués de suffixes et de préfixes. A partir du deuxième choix, on se retrouve avec **tous les sous-tableaux**

$$A_i \cdot ... \cdot A_j \qquad \forall 1 \leq i \leq j \leq n \ .$$

   On a donc $\Theta(n^2)$ sous-problèmes (cf. table **??**). Et pour chacun de ces problèmes, on a $j - i \leq n$ choix possibles.

3. **Récurrence.**
   Si nous notons $P_{i:j}$ le coût du **parenthésage optimal** (et non du produit matriciel !) pour le problème

$$A_i \cdot ... \cdot A_j = (A_i \cdot ... \cdot A_k) \cdot (A_{k+1} \cdot ... \cdot A_j)$$

   On obtient aisément la récurrence suivante :

$$P_{i:j} = \min_{i \leq k \leq j} \left\{ \text{Cost}(i,j,k) + P_{i:k} + P_{k+1:j} \right\} \qquad \forall 1 \leq i \leq j \leq n$$

   où cette fois $\text{Cost}(i,j,k)$ est le coût du produit matriciel, qui dépend des dimensions de A_i, A_k, A_j.

   Le cas de base correspond à un produit avec une seule matrice, ce qui prend un temps constant, d'où

$$P_{i:i} = 0 \qquad \forall i \ .$$

   Par sous-problème, on a toujours $j - i$ choix possibles, d'où

$$\boxed{\text{Coût/sous-pb.} \quad O(j-i) = O(n)} \ .$$

4. **Graphe de dépendance.**
   Enfin, si l'on trace le graphe de dépendance pour dérécursifier l'algorithme, on voit les dépendances suivantes, en simplifiant le graphe sur la figure **??**.

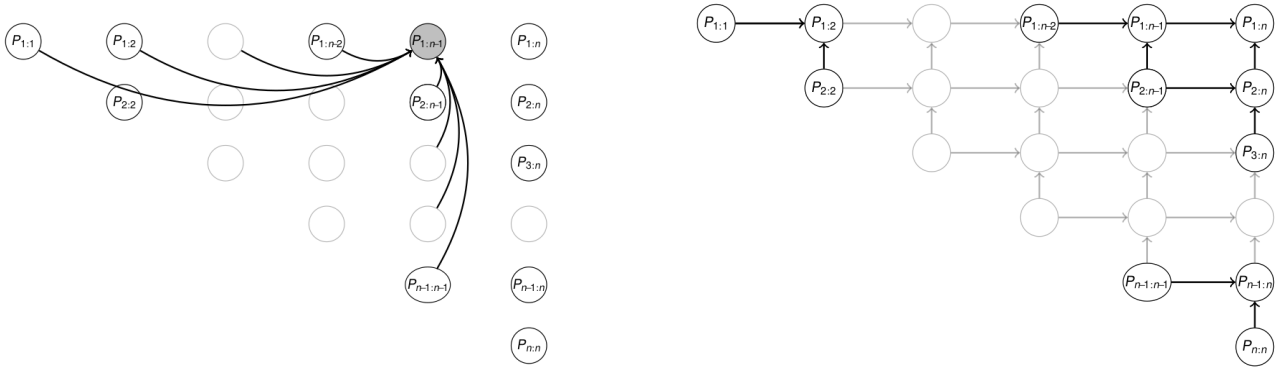


FIGURE 6.1 – Graphe de dépendance avec une version simplifiée.

5. **Problème initial.**
   Pour retourner au problème initial, qui est $P_{1:n}$, on a $\Theta(n^2)$ sous-problèmes, $O(n)$ pour le coût d'un sous-problème et un coût constant pour le problème initial, ce qui revient à un coût de $O(n^3)$ pour le coût total de la solution du problème de parenthésage optimal.

6.2.2 Plus longue sous-séquence commune

Par sous-séquence, nous entendons "partie d'un mot dont tous les caractères se trouvent dans le même ordre dans le mot". Le problème de la plus longue sous-séquence commune est le suivant.

On se donne deux chaînes de caractères et on cherche une chaîne de caractère qui soit à la fois une sous-séquence du premier et du deuxième. On recherche plus précisément la **plus longue**. La solution n'est pas forcément unique! Le problème a l'air abstrait mais a une application typique, par exemple la comparaison de chaînes ADN.

Résolution en programmation dynamique – les 5 étapes

Si on commence à l'étape 2, on identifie les sous-problèmes : des suffixes, préfixes, ou des sous-tableaux. La différence ici est qu'on a deux tableaux, s et t. Pour deux chaînes, on va **supposer dans un premier temps que les sous-problèmes sont représentés par des suffixes** (on va le montrer). Alors on a les suffixes suivants pour les premiers sous-problèmes :

$$s[i : n-1] \qquad t[j : n-1]$$

Le nombre de sous-problèmes est $\Theta(n^2)$, vu qu'il faut considérer tous les couples de suffixes. Notons L_{ij} le sous-problème correspondant.

1. **Choix.**
   On part de deux chaînes complètes et on essaye de former la sous-séquence commune la plus longue. Le premier choix concerne le premier caractère. On va encore supposer qu'on regarde des suffixes, c'est beaucoup plus facile.

   Premier choix – premier caractère.
   - ⋆ Si $s[i] = t[j]$, on a intérêt à prendre ce caractère dans la PLSC.
   - ⋆ Si $s[i] \neq t[j]$, soit $s[i]$ n'est pas dans la PLSC, soit $t[j]$ n'est pas dans la PLSC (soit aucun des deux n'y est mais c'est à l'étape suivante qu'on aura supprimé les deux).

   Le nombre de choix possibles pour chaque sous-problèmes est 1 ou 2, selon si on doit garder les deux ou supprimer l'un des deux. Le coût est donc dans tous les cas **constant**, d'où $\Theta(1)$.

2. **Sous-problèmes.** Déjà été traité. On remarque par ailleurs aisément que les sous-problèmes sont des suffixes, parce qu'on avance progressivement dans les chaînes, comme on peut le voir sur la figure **??**.
3. **Récurrence.**
   Soit L_{ij} la longueur de la PLSC pour $s[i : n-1]$ et $t[j : n-1]$. Si $s[i] = t[j]$, alors

   $$L_{ij} = 1 + L_{i+1\ j+1}$$

   , où $+1$ correspond à l'ajout de la lettre commune.

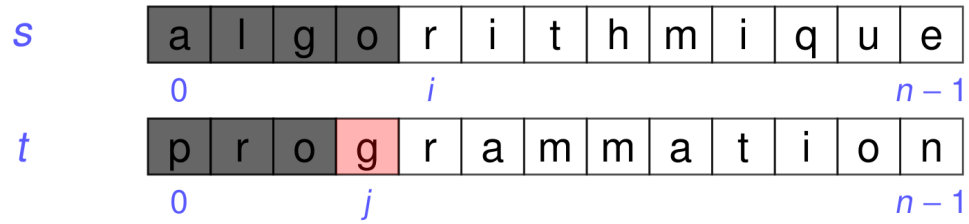


FIGURE 6.2 – Représentation du problème de recherche de la plus longue séquence commune à deux chaînes de caractères s et t.

Si $s[i] \neq t[j]$, alors les deux cas sont possibles : $\begin{cases} L_{i+1\,j} \\ L_{i\,j+1} \end{cases}$ et celui qui sera pris est celui qui maximise la longueur de la PLSC.

$$L_{ij} = \begin{cases} 1 + L_{i+1\,j+1} & \text{si } s[i] = t[j] \\ \max\{L_{i+1\,j}\,, \quad L_{i\,j+1}\} & \text{sinon} \end{cases}$$

Le cas de base correspond au cas où un des deux indices est n, quand une des deux chaînes est vide.

$$\text{Cas de base :} \qquad L_{in} = L_{nj} = 0 \quad \forall\, i\,, i$$

Et les coûts par sous-problèmes sont constants parce que nous considérons que tout a déjà été mémoïsé.

4. **Graphe de dépendance.**
   Nous pouvons dresser pour cela un tableau bidimensionnel. La question est "qu'est-ce qui dépend de quoi ?". Nous savons que L_{ij} dépend de $L_{i+1\,j+1}$, $L_{i+1\,j}$ et $L_{i\,j+1}$, mais tous les éléments ont cette même structure de dépendance. Il faut enfin construire l'ordre topologique en regardant d'où partent les flèches : il s'agit d'en bas à droite,c'est de là que doit partir la première flèche. Attention : plusieurs ordres topologiques peuvent marcher, bien entendu !

5. **Problème initial.**
   C'est lorsque les deux chaînes sont complètes : L_{00}. La détermination du coût total revient donc à prendre le nombre de sous-problèmes, le coût par sous-problème, et le coût additionnel pour le problème initial (une fois tous les sous-problèmes calculés, le problème initial est lui-même un des sous-problèmes et la valeur se trouve déjà en table de mémoïsation) : $\Theta(n^2) + \Theta(1) + \Theta(1) = \Theta(n^2)$. Remarque : si on essayait ça en recherche exhaustive on aurait un coût exponentiel.

6.3 Démultiplication des sous-problèmes

Comme on l'a déjà vu, il est très typique que l'input d'un problème soit un tableau. On va voir ici comment choisir les bons sous-tableaux lorsque l'input du problème est un tableau.

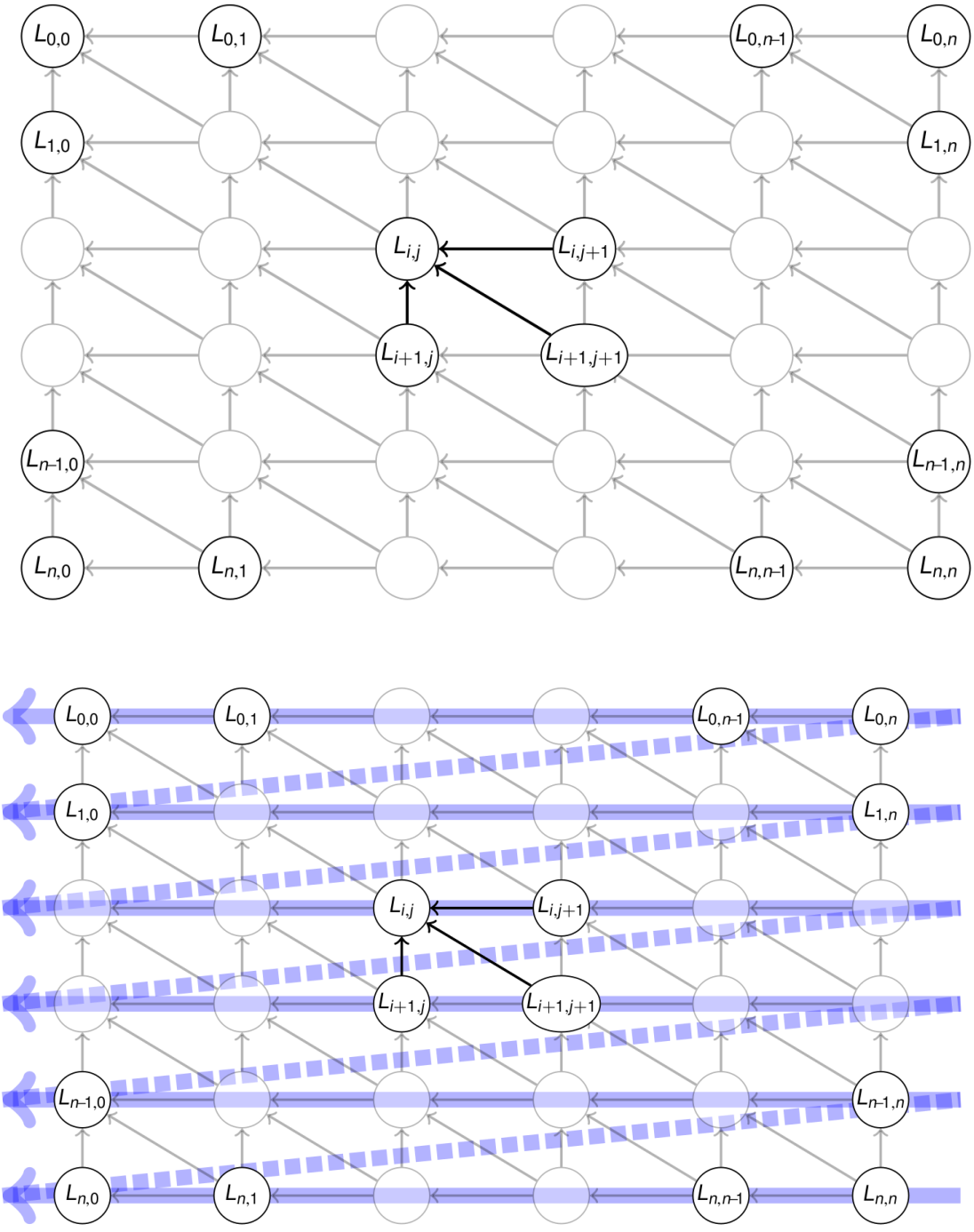


FIGURE 6.3 – Graphe de dépendance et ordre(s) topologiques du problème de PLSC.

6.3.1 Approche générale

L'approche générale consistait à considérer les suffixes, les préfixes ou les sous-tableaux. Cette approche ne marche pas toujours, parce qu'en construisant une solution candidate partielle, on se retrouve en effet avec un sous-problème similaire au problème initial, **mais avec des paramètres différents**. L'input est encore un tableau mais un (ou plusieurs) paramètre a changé.

Nous allons alors **définir un sous-problème par valeur possible des paramètres** : il s'agit de la **démultiplication des sous-problèmes**.

> Démultiplication des sous-problèmes : un sous-problème par valeur possible des paramètres

6.3.2 Exemples

- Pour le problème de pavage, comme on fait deux choix différents à chaque étape, on se retrouve à chaque étape avec deux versions du sous-problème : la version avec le côté gauche bleu et l'autre avec rouge.
- Variante du problème du sac : chaque objet a un volume et une valeur. Et soit le problème d'optimisation de la **valeur du sac** et non le volume. On va l'étudier en détail.

Variante du problème du sac

On peut dire qu'à chaque choix, on se retrouve avec un suffixe, mais le volume dans le sac est plus petit ! Donc on voudra maximiser la valeur du sac mais avec un volume disponible plus petit : le **volume disponible** qui est le paramètre. On démultiplie donc les sous-problèmes selon v le volume disponible.

Après démultiplication, nous avons donc les sous-problèmes suivants, notés K_{iv} :

- Pour chaque paramètre v, on a les n suffixes car $i = 0 \longrightarrow n-1$
- Le paramètre v varie de 0 à V : $0 \leq v \leq V$.

$\Rightarrow$ # sous-problèmes : $\Theta(n \cdot V)$.

La **troisième étape** consiste à évaluer K_{iv} pour obtenir une récurrence : faire le meilleur choix possible à chaque étape. Pour atteindre K_{iv}, il faut soit prendre i soit non, et résoudre le sous-problème restant de manière optimale. Si on prend i, alors la valeur du sac augmente de $\text{Val}[i]$ et le volume diminue de $\text{Vol}[i]$.

$$K_{iv} = \begin{cases} \max\{K_{i+1\,v}\,, \quad K_{i+1\,v-\text{Vol}[i]} + \text{Val}[i]\} & \text{Si } \text{Vol}[i] \leq v \\ K_{i+1\,v} & \text{Sinon} \end{cases}$$

Le cas de base correspond à un cas K_{nv} qui est nul pour tout volume restant parce qu'il n'y a plus aucun objet à considérer.

Le coût par sous-problème est constant parce qu'on consière que tout a été mémoïsé.

La **quatrième étape** est le graphe de dépendance : K_{iv} dépend de $K_{i+1\ v}$ et des $Ki+1\ w$ où les w sont des volumes inférieurs à v. On commence donc à la première ligne du graphe et puis on remonte : voilà l'ordre topologique. Sur la première ligne on peut calculer tous les éléments indépendamment, on peut donc proposer un ordre topologique qui calcule de gauche à droite en remontant à chaque ligne.

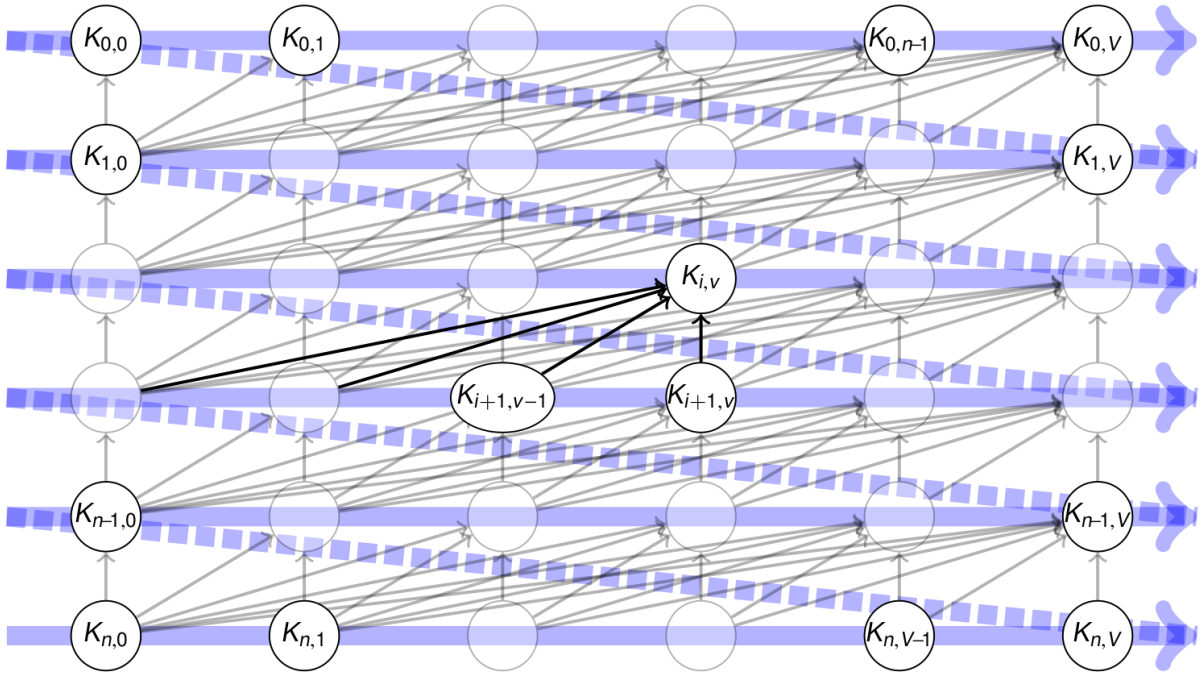


FIGURE 6.4 – Graphe de dépendance pour la variante du problème de sac à dos, abordée par la programmation dynamique.

La **cinquième étape** concerne le problème initial : il faut résoudre tous les problèmes puis calculer la valeur pour le problème initial. Mais si tout a déjà été calculé, le coût additionnel est un temps constant, donc, avec $\Theta(n \cdot V)$ sous-problèmes et $\Theta(1)$ comme coût pour chaque sous-problème, nous avons un coût total de $\Theta(n \cdot V)$.

C'est une complexité **pseudo-polynomiale** parce qu'il y a un autre paramètre apparaît, et que ce facteur est la **valeur maximale des entiers définissant le problème**. C'est mieux qu'une complexité exponentielle mais c'est moins bon qu'une complexité polynomiale.

6.4 Algorithmes gloutons

Nous allons faire un retour en arrière et discuter des algorithmes gloutons. Contrairement à la programmation dynamique, on n'arrive pas ici à une solution optimale, mais une complexité souvent plus petite.

6.4.1 Comparaison glouton – progDyn

En programmation dynamique, à chaque étape il faut choisir entre plusieurs possibilités. On s'assure de prendre le choix qui **mène à une solution optimale**. On a pour ça besoin de la récursion pour aller voir plus loin.

Avec un algorithme glouton, face à un choix, on choisit directement la possibilité qui **semble être la meilleure**. Ils sont souvent simples et rapides, mais ne mènent pas toujours) une solution optimale. Alors, si on préfère proditer du gain en temps de calcul et qu'une solution sous-optimale est acceptable, alors on **peut utiliser un algorithme glouton**. Une autre possiblite d'utiliser un algorithme glouton est si on arrive à prouver qu'on peut arriver à une solution optimale.

6.4.2 Exemple du problème du sac-à-dos