Introduction à l'algorithmique : notes de cours

Etienne Renault

5 octobre 2016

Table des matières

1 Introduction générale **4**
- 1.1 Notion d'algorithme . . . 4
- 1.2 Première approche pour comparer des algorithmes . . . 5
- 1.3 Random Access Machine . . . 8
- 1.4 Introduction aux notations asymptotiques . . . 10
- 1.5 Évaluation des coûts . . . 12

2 Étude de la complexité **14**
- 2.1 Notations de Landau . . . 14
- 2.2 Théorème général . . . 14
- 2.3 Preuve par récurrence . . . 14

3 Les tris **16**
- 3.1 Tri par sélection . . . 17
- 3.2 Tri par insertion . . . 18
- 3.3 Tri fusion . . . 19
- 3.4 Tri par tas . . . 22
- 3.5 Tri rapide . . . 25
  - 3.5.1 Complexité . . . 26
  - 3.5.2 Variantes . . . 26
- 3.6 Choix du pivot & calcul de médiane . . . 27
- 3.7 Comparaison des algorithmes de tri basés sur les comparaisons 32
- 3.8 Tri par dénombrement . . . 34
- 3.9 Tri par base . . . 35
- 3.10 Tri par paquets . . . 37

4 Les structures de données **39**
- 4.1 Étude des structures « classiques » . . . 39
- 4.2 Les arbres rouge-noir . . . 41
- 4.3 Tableaux dynamiques . . . 43
- 4.4 Tables de Hachages . . . 43

4.4.1 Résolution par chaînage 45
4.4.2 Résolution par coalescence/sondage linéaire 46
4.4.3 Fonctions de hachage 46

Remarques préliminaires

Ce document, bien qu'incomplet, présente les principales notions abordées dans le cours d'algorithmique dispensé aux *apprentis* (expert et ingénieurs) de l'EPITA. De plus, ce document étant en cours d'écriture, il est amené à être modifié régulièrement.

Ce présent document peut être trouvé ici :
`https://www.lrde.epita.fr/~renault/teaching/algo/cours.pdf`

Et est largement inspiré du document (en anglais) :
`https://www.lrde.epita.fr/~adl/ens/algo.pdf`

Les travaux dirigés associés à ce cours peuvent être trouvés ici :
`https://www.lrde.epita.fr/~renault/teaching/algo/tds.pdf`

Les travaux dirigés associés au séminaire ALGO se situent ici :
`https://www.lrde.epita.fr/~adl/ens/tutalgo.pdf`

Chapitre 1

Introduction générale

1.1 Notion d'algorithme

Le nom algorithme provient du mathématicien Al Khowarizmi qui vivait à Bagdad au IX-ième siècle et qui a rédigé ce qui est considéré comme le premier manuscrit d'algèbre. Son objectif était de spécifier des schémas de calcul sous la forme de suites finies d'opérations élémentaires obéissant à un enchaînement déterminé. Plus formellement :

Définition 1.1 *Un algorithme est un ensemble d'opérations de calcul élémentaires qui sont organisées (via des structures de contrôle) de manière à transformer un ensemble de valeurs d'entrées (input) dans un ensemble de valeurs de sortie (output).*

Les opérations de calcul élémentaire sont :
- les opération arithmétiques : $+$, $\times$, $/$, $-$, ...
- les opérations de lecture ou d'affectation de variables
- ...

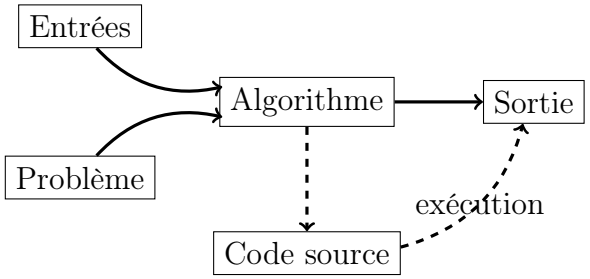
Les structures de contrôle sont :
- les tests
- les boucles : for, while, ...
- les comparaisons
- ...

Exemples.

1. Multiplication
   - **input** a et b deux entiers
   - **output** $a * b$

2. Plus Grand Diviseur Commun
   - **input** a et b deux entiers
   - **output** pgcd(a, b)
3. Test de primalité
   - **input** a un entier
   - **output** $\top$ si a est premier, $\bot$ sinon

Un algorithme est indépendant d'un langage de programmation : il est généralement écrit en pseudo-code. Ce style d'écriture permet de rendre indépendant la technique de résolution d'un problème du choix technique (langage de programmation) utilisé pour résoudre ce problème.



1.2 Première approche pour comparer des algorithmes

Pour un problème donné, il existe de nombreux algorithmes mais tous n'ont pas la même efficacité. Par exemple, le calcul de n factoriel peut être envisagé selon (au moins !) deux grands axes. Rappel :

$$n! = \prod_{1 \leq i \leq n} i = 1 \times 2 \times \ldots \times n$$

- Dans une approche itérative une simple boucle permet de calculer le résultat souhaité.

```
1 Function fact-iter(Integer n)
2 |   res ← 1
3 |   for i ← 1 to n do
4 |   |   res ← res × i
5 |   end
6 |   return res
```

- Dans une approche récursive la suite des appels récursifs construit la solution progressivement.

```
1 Function fact-rec(Integer n)
2 |  if n = 1 then
3 |  |  return 1
4 |  end
5 |  return n × fact-iter(n-1)
```

Pour savoir lequel de ces deux algorithmes utiliser, il faut être capable de mesurer les performances. Une première idée peut être d'exécuter le code sur une machine et de mesurer le temps d'exécution.

	Intel Core i7 2.90 GHz	Intel Xeon @ 2.66 GHz
100 appels à Fact-iter(5000)	6.0 sec	8.2 sec
100 appels à Fact-rec(5000)	5.8 sec	8.3 sec

Au vu des mesures ci-dessus, les deux algorithmes semblent avoir la même efficacité mais mesurer le temps d'exécution d'un programme est compliqué car :

- les machines exécutent en parallèle plusieurs processus qui vont interférer dans leurs exécutions (invalidation des caches, commutation des processus, ...)
- les performances peuvent dépendre de la mémoire vive disponible
- les performances peuvent dépendre de la fréquence du CPU
- les performances peuvent dépendre du degré d'optimisation réalisé par un compilateur

Nous avons donc besoin d'une mesure objective qui ne soit pas sensible à ces variations pour mesurer l'efficacité d'un algorithme. Il existe de nombreuses définitions formelles qui ont toutes pour objectif de pouvoir raisonner sur les algorithmes, de les composer, de les prouver, et de calculer leurs coûts d'exécution en temps et en mémoire. Par exemple :

- les fonctions récursives
- le λ-calcul
- les machines de Turing
- les Random Access Machine

Toutes ces définitions sont équivalentes. Tout ce qui est calculable intuitivement l'est aussi par les formalismes précédents. Il existe cependant des problèmes non-calculables (indécidables) pour lesquels il n'existe pas d'algorithme. Il s'agit typiquement du problème de l'arrêt :

- **input** Un Algorithme A et une entrée I
- **output** A termine-t-il pour I ?

Démonstration (problème de l'arrêt). Un problème P est donc dit décidable ssi il existe un algorithme A qui résout P. Cependant certains problèmes sont indécidable, i.e. il n'existe pas d'algorithmes pour résoudre ces problèmes. La preuve de ce résultat repose sur un argument diagonal et a été présenté par Turing en 1936.

Supposons qu'il existe un programme HALT, dont l'objectif est de lancer un programme passé en argument et de retourner $\top$ (vrai) ssi celui-ci termine. Ce programme peut être vu comme un shell et exprimé de la manière suivante :

```
1 Function HALT(Program p, String input)
2 |  if p(input) terminates then
3 |  |  return ⊤
4 |  end
5 |  return ⊥
```

À partir de ce programme, on peut donc construire le programme DIAG, dont le comportement va être l'inverse de HALT. Ce programme peut être exprimé de la manière suivante :

```
1 Function DIAG(Program p, String input)
2 |  if HALT(p, input) then
3 |  |  while ⊤ do
4 |  |  |  // Loop forever
5 |  |  end
6 |  end
```

Supposons maintenant que tous les problèmes soient décidables. Pour un programme donné, deux cas se présentent :

- si HALT retourne $\top$, alors DIAG va boucler infiniment
- sinon HALT ne termine pas.

Si l'on suppose l'existence d'un programme HALT, alors on peut facilement s'arranger pour construire un programme qui va l'empêcher de terminer et donc de savoir si un programme est décidable ou non. On obtient donc une contradiction par l'argument diagonal.

Nota. *Cette démonstration peut aussi être vue comme le paradoxe du menteur qui aurait été inventé par Eubulide, un adversaire d'Aristote. Si une personne dit "Je mens", deux cas se présentent :*

- Soit la déclaration est vraie et la personne mens

- *Soit cette déclaration est fausse, la personne ne dit pas la vérité ce qui conduit à une contradiction.*

1.3 Random Access Machine

De nombreux modèles de calculs existent qui sont tous équivalents aux machines de Turing. Ici nous nous intéressons particulièrement aux machines RAM qui constituent le modèle mémoire le plus classiquement utilisé.

Définition 1.2 *Une RAM (Random Access Machine) est constituée de registres, de deux bandes et d'un programme. Une des bande est une bande d'entrée sur laquelle la machine lit ses données et l'autre bande constitue la bande de sortie sur laquelle la machine écrit ses résultats. Le programme est quant à lui une suite finie d'instructions.*

Les registres sont des cases mémoire qui contiennent des entiers. Deux registres sont distingués des autres :
- le compteur ordinal : qui contient le numéro de la prochaine instruction à exécuter
- l'accumulateur qui est un registre spécial contenant des entiers

A l'exception faite de ces deux registres, on dispose d'une infinité de registres indicés par un entier naturel. Par exemple, le registre numéro n est dénoté par r_n.

Afin de manipuler les registres on dispose de deux opérations : `load` et `store`. L'instruction `load` permet de stocker une valeur dans l'accumulateur. On peut s'en servir de trois manières différentes :
- `load` $\#n$: met la valeur n dans l'accumulateur, il s'agit de l'adressage absolu ;
- `load` n : met la valeur stockée dans le registre n dans l'accumulateur, il s'agit de l'adressage direct ;
- `load` (n) : si p est la valeur de r_n, alors l'accumulateur reçoit le contenu de r_p. Il s'agit de l'adressage indirect.

L'instruction `store` quant à elle stocke le contenu de l'accumulateur dans un registre. Il existe la aussi plusieurs adressages possibles.
- `store` n : met la valeur stockée dans l'accumulateur dans le registre n (adressage direct) ;
- `store` (n) : si p est la valeur de r_n, alors stocke le contenu de l'accumulateur dans r_p (adressage indirect).

Dans les machines RAM, il existe deux opérateurs arithmétiques :

- `incr` : qui incrémente la valeur de l'accumulateur ;
- `decr` : qui décrémente la valeur de l'accumulateur si celle si est positive et la laisse nulle sinon.

Trois structures de contrôle sont à disposition :
- `stop` : qui provoque l'arrêt de la machine ;
- `jump` ℓ : qui permet de sauter à la ligne ℓ : cela revient à inscrire ℓ dans le registre contenant le compteur ordinal ;
- `jumpz` ℓ : qui effectue le saut seulement si l'accumulateur contient zéro.

Enfin, deux instruction permettent d'effectuer des entrées sorties :
- `read` : permet de lire le premier entier sur la bande de lecture, le positionne dans l'accumulateur et positionne la tête de lecture sur l'entier suivant.
- `write` : permet d'écrire un entier stocké sur dans l'accumulateur, sur la bande de sortie et de déplacer celle-ci d'un entier.

Différences avec les microprocesseurs. Les machines RAM que nous venons de présenter sont proches des microprocesseurs mais possèdent néanmoins des différences notables :
- Le nombre de registres d'une machine RAM est illimité tandis que les microprocesseurs ne peuvent accéder qu'à une mémoire limitée. De même, dans une machine RAM un registre peut contenir n'importe quelle valeur tandis qu'un microprocesseur ne peux y stocker qu'un nombre limité de bits.
- Le jeux d'instruction présenté ici est bien plus limité que le nombre d'instruction offert par le microprocesseurs actuels. D'une certaine manière, la machine RAM présentée ici se rapproche plus des architectures RISC qui ont un jeu restreint d'instructions. En particulier, les opérations arithmétiques ne sont pas présentes mais facilement simulables.

Exemple calcul somme entiers. Le programme ci-dessous permet de calculer la somme de deux entiers sur une machine RAM.

```
0   read
1   store 1
2   read
3   store 2
4   load 1
5   jumpz 12
6   decr
7   store 1
8   load  2
9   incr
10  store  2
11  jump 5
12  load 2
13  write
14  stop
```

Remarque. Dans la pratique, nous considérons une machine RAM plus évoluée que l'on peut manipuler non pas avec du code assembleur mais du pseudo-code. Dans un tel schéma, nous considérerons que les opérations arithmétiques sont faites en temps constant, conformément aux instruction offertes par les processeurs actuels. De même, nous considérerons que l'exécution des structures de contrôles de base sont aussi faites en temps constant.

1.4 Introduction aux notations asymptotiques

La machine RAM permet donc de raisonner sur un modèle simple et où les accès mémoires sont directs et non affectés par la présence de caches. On peut donc maintenant évaluer de manière plus fine l'évolution d'un programme.

L'exemple suivant montre un programme simple qui calcule la somme des éléments d'un tableau.

```
1 Function sum-tab(Integer[] tab, Integer n)
2 |   sum ← 0
3 |   for i ← 0 to n do
4 |   |   sum ← sum + tab[i]
5 |   end
6 |   return sum
```

À partir de cet exemple on peut affecter des coûts d'exécution à chaque ligne. Ainsi, la ligne ℓ_1 aura pour coût c_1, la ligne ℓ_2 aura pour coût c_2, et ainsi de suite. On observe alors que les lignes ℓ_1, ℓ_2, et ℓ_6 ne seront exécutées

qu'une unique fois, tandis que les autres lignes seront exécutées un nombre de fois dépendant de n, la taille du tableau. On peut donc en approximer la complexité de cet algorithme de la manière suivante :

$$T_{sum-tab} = 1 \times c_1 + 1 \times c_2 + n \times c_3 + n \times c_4 + n \times c_5 + 1 \times c_6$$

En substituant toutes les constantes on obtient alors :

$$T_{sum-tab} = K_1 \times n + K_2$$

Avec cette substitution, il apparaît que la complexité de l'algorithme est linéaire par rapport à n, i.e. par rapport à la taille du problème en entrée. Lorsqu'on calcule la complexité des algorithmes, c'est cette variation en fonction de la taille du problème qui nous intéresse. On ne regarde que le comportement à l'asymptote.

De plus, si pour un problème P de taille n nous avons le choix entre plusieurs algorithmes il est évident qu'il vaut mieux choisir celui qui à le meilleur comportement asymptotique. Par exemple, si un algorithme est quadratique en fonction de n tandis qu'un autre est linéaire, nous favoriserons l'utilisation de celui qui est linéaire.

Rappels. Pour un problème P de taille n, si un algorithme résout P en :
- K opérations (avec K constante) alors l'algorithme est dit de temps *constant*
- $log(n)$ opérations alors l'algorithme est dit de temps *logarithmique*
- n^2 opérations alors l'algorithme est dit de temps *quadratique*
- 2^n opérations alors l'algorithme est dit de temps *exponentiel*

Supposons que l'on possède un ordinateur qui effectue 10^6 opérations par seconde. La table suivante permet d'évaluer le temps pris par un programme en fonction de sa complexité (exprimée en fonction de la taille n du problème en entrée).

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
10^2	6.6 μs	0.1 ms	0.6 ms	10 ms	1 s	$4 \cdot 10^6$ ans
10^3	9.9 μs	1 ms	10 ms	1 s	16.6 min	
10^4	13.3 μs	10 ms	0.1 s	1.6 min	11.6 j	
10^5	16.6 μs	0.1 s	1.6 s	2.7 h	317 ans	
10^6	19.9 μs	1 s	19.9 s	11.6 j	10^6 ans	
10^7	23.3 μs	10 s	3.9 min	3.17 ans		
10^8	26.6 μs	1.6 min	44.3 min	317 ans		
10^9	29.9 μs	16.6min	8.3 h	31709 ans		

1.5 Évaluation des coûts

Dans les exemples précédents, il était possible de déterminer le nombre exact d'opérations dans chaque boucles. Chaque opération étant associée à un coût spécifique, il était alors simple de déduire le coût général de l'algorithme à l'asymptote. Néanmoins, la combinaisons d'opérateurs de conditions (*if*, *while*, ...) peuvent rapidement rendre ce calcul très compliqué : la simple présence d'une instruction de condition peut faire varier considérablement la complexité d'un algorithme entre des données de tailles n ou de taille $n+1$.

Pour pouvoir comparer de manière équitable deux algorithmes résolvant un même problème, il devient nécessaire de raffiner la notion de coût.

Coût pire-cas. Pour un algorithme donné, nous souhaitons être capable de déterminer le coût dans le cas le plus défavorable. Ce coût est par définition le maximum des coûts sur toutes les données de taille n, i.e.

$$C(n) = max(c(x)), \forall x \in 0 \leq x \leq n$$

Avec cette définition nous pourrons évaluer la complexité de deux algorithmes dans les cas qui leurs sont le plus défavorable (même si ce cas n'est pas le même pour les deux algorithmes).

Coût meilleur-cas. Pour un algorithme donné, nous souhaitons aussi être capable de déterminer le coût dans le cas le favorable. Ce coût est par définition le minimum des coûts sur toutes les données de taille n, i.e.

$$C(n) = min(c(x)), \forall x \in 0 \leq x \leq n$$

Coût moyen. Dans certaines situations, le pire-cas ne se produit que très rarement ; nous sommes alors plus intéressé par connaître le coût moyen d'un algorithme. Une formulation correcte de ce coût suppose que l'on connaisse une distribution des probabilités sur les données de taille n. Si l'on note $p(x)$ la probabilité d'apparition de la donnée x, alors le coût moyen $\gamma(n)$ d'un algorithme sur des données de tailles n, peut être donné par :

$$\gamma(n) = \sum_{0 \leq x \leq n} p(x).c(x)$$

Dans la pratique, on suppose souvent que cette répartition est uniforme, i.e. $p(x) = \frac{1}{n}$. Il vient alors :

$$\gamma(n) = \frac{1}{n}. \sum_{0 \leq x \leq n} c(x)$$

Lorsque l'on travaille sur des structures des données l'on parle souvent de **coût amorti**. Ce coût peut être vu comme la compensation d'un surcoût ponctuel (pire-cas) par toutes les autres opérations qui sont généralement sur-évaluées. On obtient alors une approximation plus simple a obtenir que lors du calcul du coût moyen.

Chapitre 2

Étude de la complexité

Ce chapitre a deux objectifs principaux. Le premier est d'introduire un ensemble de notations permettant de raisonner sur la complexité des algorithmes. Le deuxième objectif affiché de ce chapitre est de fournir des techniques pour déterminer la complexité des algorithmes.

2.1 Notations de Landau

todo.

2.2 Théorème général

todo.

2.3 Preuve par récurrence

Lorsqu'on a une intuition de la complexité d'un algorithme on peut faire une preuve par induction pour prouver l'hypothèse initiale. L'utilisation de l'induction permet de prouver qu'un algorithme est minoré ou majoré (ou bien les deux) par une fonction donnée. Cette technique, à l'instar du théorème général, peut toujours être appliquée. Le seul inconvénient de cette technique est qu'il faut être capable de *deviner* la complexité finale.

Exemple. Montrons que si $T(n) = 2 \times T(\frac{n}{4}) + \Theta(1)$, alors $T(n) = O(n)$.

Hypothèse. Pour tout m tel que $m < n$, on a $T(m) \leq c \times m$, pour une valeur de c donnée qui peut être aussi grande que l'on veut.

Récurrence. On veut montrer que $T(n) \leq c \times n$.

$$T(n) = 2 \times T(\frac{n}{4}) + \Theta(1)$$

D'après l'hypothèse de récurrence, on a $T(\frac{n}{4}) \leq c \times \frac{n}{4}$. On a donc :

$$\begin{aligned} T(n) &\leq 2 \times c \times \frac{n}{4} + \Theta(1) \\ &\leq c \times \frac{n}{2} + \Theta(1) \\ &\leq (c \times n) - (\frac{c \times n}{2} - \Theta(1)) \end{aligned}$$

Or $(\frac{c \times n}{2} - \Theta(1))$ est positif si c est assez grand, on peut alors le négliger. On a donc montré que que $T(n)$ est majoré par n, i.e. $T(n) = O(n)$.

Exemple. Montrons que si $X(n) = 2 \times X(\frac{n}{2}) + \Theta(n)$, alors $X(n) = \Omega(n)$.

Hypothèse. Pour tout m tel que $m < n$, on a $X(m) \geq c \times m$, pour une valeur de c donnée qui peut être aussi grande que l'on veut.

Récurrence. On veut montrer que $X(n) \geq c \times n$.

$$\begin{aligned} X(n) &= X(\frac{n}{2}) + \Theta(n) \\ &\geq (c \times \frac{n}{2}) + \Theta(n) \\ &\geq (c \times n - c \times \frac{n}{2}) + \Theta(n) \\ &\geq (c \times n) - (c \times \frac{n}{2} - \Theta(n)) \end{aligned}$$

Or $(c \times \frac{n}{2} - \Theta(n))$ est positif si c est assez grand, donc négligeable. On a donc montré que que $X(n)$ est minoré par n, i.e. $X(n) = \Omega(n)$.

Chapitre 3

Les tris

La manipulation de grandes quantités de données nécessite souvent de pouvoir les trier pour en extraire les informations pertinentes de manière efficace. Pour pouvoir trier un ensemble de données il suffit de définir une relation d'ordre R sur ces données. En fonction de la taille et du type de données que l'on manipule, deux critères permettent de caractériser un algorithme de tri.

Définition 3.1 *Un algorithme de tri est dit stable s'il garde l'ordre relatif des élément égaux pour la relation d'ordre R.*

Exemple. Supposons que nous souhaitions trier | 1 | 5 | 3 | 3 | 2 | avec la relation $\leq$, t.q. défini classiquement sur les entiers. Il apparaît clairement que deux résultats sont alors possibles : | 1 | 2 | 3 | 3 | 5 | ou | 1 | 2 | 3 | 3 | 5 |. Les deux résultats sont corrects puisqu'ils trient les données en respectant la relation $\leq$. Néanmoins, on souhaite parfois trier les données en conservant l'ordre original. Si cet ordre est conservé (comme c'est le cas pour le premier tableau) le tri est dit *stable* sinon il est dit *instable* (second tableau).

Définition 3.2 *Un algorithme de tri est en place s'il ne nécessite qu'un petit nombre de variables et modifie directement la structure sur laquelle il est en train de travailler.*

Remarque. L'utilisation de tris *en place* est particulièrement intéressante lorsque l'on manipule de très grosses quantités de données et que l'espace mémoire est limité. Néanmoins, les tris non en-place ont aussi leur intérêts et nous verrons dans ce chapitre qu'ils peuvent majoritairement être adaptés pour être réalisés *en place*.

Connaissant ces deux critères, le reste du chapitre s'attardera à étudier la complexité des différents algorithmes de tri existant.

3.1 Tri par sélection

Le tri par sélection est un tri naïf pour trier un tableau T de taille n. Il peut être vu comme la succession de trois étapes. :

1. on commence par chercher le minimum (en accord avec la relation d'ordre) dans le tableau $T[0..n-1]$
2. on échange la position de cet élément avec l'élément en position 0
3. on répète ces 2 opérations sur le tableau privé de sa première case.

Dans cet algorithme, lorsque l'on traite la case i, tout le tableau $T[0..i[$ est trié et les valeurs $T[i..n-1]$ sont plus grande que celles dans $T[0..i[$. Le tableau $T[i..n-1]$ n'est donc pas trié. On a donc a tout moment :

Valeurs plus petites que $T[i]$ et triées	i	Valeurs plus grandes que $T[i]$ et non triées

Cet algorithme de tri peut être exprimé avec le pseudo-code suivant :

```
1 Function selectionSort(Integer[] tab, Integer n)
2     for i ← 0 to n-2 do
3         minInd ← i
4         for j ← i+1 to n − 1 do
5             if T[j] < T[minInd] then
6                 minInd ← j
7         T[minInd] ↔ T[i]
```

Ce tri est en place puisqu'il modifie directement les données du tableau sur lequel il s'exécute. Cependant, ce tri n'est pas en stable. En effet, si l'on considère le tableau | 4 | 4 | 1 |, alors la première (et unique) itération produira le tableau | 1 | 4 | 4 |.

Intéressons nous maintenant à calculer le coût temporel pire-cas de cet algorithme. La complexité de cet algorithme vient du nombre de fois où les lignes 5 et 6 vont être exécutées. Ces opérations étant en temps constant, il vient que la complexité de l'algorithme peut être exprimée par :

$$\sum_{0\leq i\leq n-2}\ (\sum_{i+1\leq j\leq n-1} 1) = \sum_{0\leq i\leq n-2} n-i-1 = \frac{(n)\times(n-1)}{2}$$

Cet algorithme est en $\Theta(n^2)$ pour le cas général. Comme cet algorithme effectue toujours la même chose quelque soit les données, il est aussi en $\Theta(n^2)$ dans les pires, moyens et meilleurs cas.

3.2 Tri par insertion

Cet algorithme, bien qu'il puisse à première vue sembler très proche de l'algorithme précédent, fonctionne de manière totalement différente. Néanmoins, il se découpe lui aussi en trois grandes étapes :

1. On commence par supposer que le tableau constitué uniquement de la première case est trié
2. On considère ensuite le second élément, et on l'insère à la bonne position dans le tableau composé uniquement des deux premières cases
3. On recommence en supposant que les deux premières cases sont triées

```
1 Function selectionSort(Integer[] tab, Integer n)
2     for i ←1 to n-2 do
3         key ← T[i]
4         j ← i − 1
5         while j ≥ 0 ∧ T[j] > key do
6             T[j + 1] ← T[j]
7             j ← j -1
8         T[j + 1] ← key
```

À tout moment dans cet algorithme, lorsque l'on est sur une case i, le tableau $T[0..i[$ est trié tandis que $T[i..n-1]$ ne l'est pas. On a donc, à tout moment, l'invariant suivant :

Valeurs triées	i	Valeurs non triées

On peut aussi noter que ce tri est en place et stable. En effet, il modifie directement les données sur lesquelles il travaille, et comme chaque valeur est insérée l'une après l'autre, l'ordre des éléments égaux est respecté.

Étudions maintenant la complexité de cet algorithme. Clairement, celle-ci dépend du nombre de fois ou va être exécuté les lignes ℓ_6 et ℓ_7. On distingue donc trois cas :

1. **Meilleur cas.** Les lignes ℓ_6 et ℓ_7 ne sont jamais exécutées (cas où le tableau est trié). Le nombre d'opérations peut être représenté par un polynôme du premier ordre, i.e. $\mathrm{T}_{insertSort}(n) = \Omega(n)$
2. **Pire cas.** Les lignes ℓ_6 et ℓ_7 sont exécutées i fois pour chaque itération de la première boucle. Il vient :
   $\mathrm{T}_{insertSort}(n) = \sum_{1 \leq i \leq n-1} i = \frac{n \times (n-1)}{2} = \mathrm{O}(n^2)$
3. **Cas moyen.** On suppose une distribution uniforme, i.e pour la moitié des itérations ($\frac{n}{2}$), les lignes ℓ_6 et ℓ_7 ne sont pas exécutées ; et pour les autres itérations elles sont exécutées i fois. On exécute le pire cas une fois sur 2, i.e $\frac{n \times (n-1)}{4}$. On a donc $\mathrm{T}_{insertSort}(n) = \Theta(n^2)$.

Note. Avec les notations de notations de Landau on perd la notion que le cas moyen est deux fois meilleur que le pire cas.

3.3 Tri fusion

Les deux algorithmes précédents peuvent être vus comme des algorithmes qui prennent les éléments un par un et les placent à la *bonne position*. L'algorithme présenté ici fonctionne d'une manière totalement différente et exploite l'idée suivante : "*si l'on possède deux liste triées L_1 et L_2, il est facile de construire une liste L qui soit triée et représente la fusion de L_1 et L_2*". En effet, pour construire cette liste L il suffit d'avancer progressivement dans les deux listes pour construire la liste résultante.

Bien que cet algorithme soit plus facile a expliquer sur des données représentées sous forme de listes, nous l'expliquons ici sur des tableaux pour avoir un cadre de comparaison homogène entre les différents algorithmes. Le tri fusion peut être découpé en trois grandes étapes :

1. Si le tableau ne possède qu'une seule valeur, alors celui-ci est trié. On renvoie donc ce tableau.
2. Sinon, on découpe le tableau en deux parties égales (à plus ou moins un élément près) et l'on effectue un appel récursif sur chacune des partie
3. Au retour des appels récursifs, les tableaux renvoyés sont triés. Il suffit alors de les fusionner.

Cet algorithme fait clairement partie de la famille des algorithmes diviser pour régner. En effet, cette famille d'algorithme part du principe que si un problème de taille n est trop compliqué à résoudre il suffit de considérer deux sous-problèmes de $\frac{n}{2}$ (*diviser*) pour ensuite construire la solution résultante à partir des solutions partielle (*régner*).

Une des étape clef du tri fusion est la fusion de deux tableaux. L'algorithme ci-dessous suppose que $tab[i..j-1]$ est trié, et que $tab[j..k-1]$ est lui aussi trié et retourne la concaténation des deux sous-tableaux.

```
1  Function merge(Integer[] tab, Integer i, Integer j, Integer k)
2      ℓ ← i
3      r ← j
4      for b ← i to k − 1 do
5          if r = k ∨ (ℓ < j ∧ tab[ℓ] ≤ tab[r] ) then
6              B[b] ← tab[ℓ]
7              ℓ ← ℓ + 1
8          else
9              B[b] ← tab[r]
10             r ← r + 1
11     B[i..k − 1] ← tab[i..k − 1]
```

Cette procédure n'est clairement pas en place puisqu'elle nécessite un tableau B dans lequel va être stocké le tableau final avant d'être recopié dans le tableau original. De plus, les lignes 6–7 et 9–10 sont les lignes les plus exécutées de l'algorithme. Elles sont en $O(k-i)$, cette procédure est donc en $O(n)$, en posant $n = k - i$.

Intéressons nous maintenant au cœur de l'algorithme de tri, tel que décrit ci dessous.

```
1  Function mergeSort(Integer[] tab, Integer i, Integer k)
2      if k − i > 1 then
3          j ← ⌊(i+k)/2⌋
4          mergeSort(tab, i, j)
5          mergeSort(tab, j, k)
6          merge(tab, i, j, k)
```

Cet algorithme est donc récursif comme on peut le voir aux lignes 4–5. De plus l'appel de la ligne 6 à la fonction merge montre que cet algorithme ne peut pas être en place. En revanche cet algorithme est stable puisque pour deux valeurs données égales deux cas se posent :

1. soit les deux éléments sont dans le même sous tableau, alors les appels récursifs vont finir par ramener au cas deux
2. soit les éléments sont dans des tableaux distincts. Pendant le merge (lignes 5–6), la priorité (lorsque l'on tombe sur des éléments égaux) est toujours donné à l'élément dans le tableau de gauche. Cela permet d'assurer que dans le tableau final les égaux apparaîtront dans l'ordre dans lequel ils apparaissent dans le tableau original.

Calculons maintenant la complexité du mergeSort. On a vu plus haut que la procédure merge était en $\Theta(n)$. Il vient pour n > 1,

$$T_{mergeSort}(n) = T_{mergeSort}(\lfloor \frac{n}{2} \rfloor) + T_{mergeSort}(\lceil \frac{n}{2} \rceil) + \Theta(n)$$

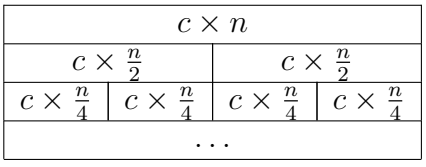
Supposons que n soit une puissance de deux, i.e. $n = 2^m$, alors :

$$\begin{aligned}
T_{mergeSort}(2^m) &= 2 \times T_{mergeSort}(\frac{2^m}{2}) + \Theta(2^m) && (3.1)\\
&= 2 \times T_{mergeSort}(2^{m-1}) + \Theta(2^m) && (3.2)\\
&= 2 \times T_{mergeSort}(2^{m-1}) + c \times 2^m && (3.3)\\
&= 2 \times (2 \times T_{mergeSort}(2^{m-2}) + c \times 2^m) + c \times 2^m && (3.4)\\
&= \dots && (3.5)\\
&= 2^i \times T_{mergeSort}(2^{m-i}) + i \times c \times 2^m && (3.6)
\end{aligned}$$

Quand $i = m$, $2^{m-i} = 1$ et $T_{mergeSort}(2^m) = \Theta(1)$. Il vient alors :

$$\begin{aligned}
T_{mergeSort}(2^m) &= 2^m \times \Theta(1) + \log_2(2^m) \times c^m && (3.7)\\
&= \Theta(n) + \log_2(n) \times c \times n && (3.8)\\
&= \Theta(n \times log_2(n)) && (3.9)
\end{aligned}$$

La complexité du mergeSort est donc en $\Theta(n \times log_2(n))$. Un autre moyen d'anticiper cette complexité est de schématiser ce qu'il se passe a chaque étapes :



On voit qu'il y a au plus $\log_2(n)$ lignes sur lesquelles on effectue des opérations en $\Theta(n)$. Enfin, à l'aide du théorème général cette complexité peut facilement être retrouvée.

3.4 Tri par tas

Cet algorithme fonctionne différemment de tous les autres algorithmes puisqu'il considère le tableau à trier comme un tas. Un tas-max est un arbre binaire quasiment complet (i.e. un arbre binaire tel que tous les niveaux à l'exception du dernier doivent être remplis, et dans lequel le dernier niveau doit être rempli de gauche à droite) qui satisfait les propriétés la *contrainte de tas* suivante : tous les fils d'un nœd doivent avoir une valeur plus grande ou égale à la valeur de ce noeud. Un tas min peut alors être défini symétriquement.

Un arbre binaire quasiment complet peut facilement être représenté par un tableau. L'accès aux fils d'un nœud i peut alors se faire de la façon suivante :

— fils gauche : $2 \times i + 1$, si $2 \times i + 1 < n$
— fils droit : $2 \times i$, si $2 \times i < n$

De plus pour un arbre binaire quasiment complet à n nœuds, alors on sait qu'il y a exactement $\lfloor \frac{n}{2} \rfloor$ nœuds internes et $\lceil \frac{n}{2} \rceil$ feuilles. Une dernière propriété des tas-max (resp. tas-min) est que la valeur maximale (resp. minimale) se situe à la racine de l'arbre, i.e. dans la première case du tableau. L'extraction de cette valeur se fait donc en temps constant.

Le tri par par peut être découpé en 2 grandes étapes :

1. Construire le tas à partir du tableau que l'on souhaite trier.
2. Utiliser le tas pour trier le tableau. Pour cela il suffit d'échanger la racine du tas (qui est la plus grande valeur) avec la dernière cellule du tableau pour que la plus grande valeur soit définitivement rangée.

Les deux étapes ci-dessus nécessitent de pouvoir manipuler des tas tout en s'assurant qu'ils en respectent bien toutes les propriétés. Pour cela, on a besoin d'une fonction heapify qui prend en paramètre un nœud i tel que les fils gauche et droite de ce nœud respectent les propriétés de tas. À partir de cela, heapify construit un nouveau tas contenant le nœud i et ses fils.

La fonction heapify prend donc en paramètre un tableau de taille m et l'indice i du nœud que l'on souhaite traiter. Comme le fils gauche et le fils droit de i respectent les propriétés d'un tas et que $tab[i] \geq tab[left]$ et que $tab[i] \geq tab[right]$ alors le sous arbre commençant au nœud i est un tas. Dans

le cas contraire, on doit échanger deux valeurs et vérifier que le sous arbre impacté respecte toujours la propriété d'un tas.

```
1  Function heapify(Integer[] tab, Integer m, Integer i)
2      left ← 2 × i + 1
3      right ← 2 × i
4      if left < m ∧ tab[left] > tab[i] then
5          g ← left
6      else
7          g ← i
8      if right < m ∧ tab[right] > tab[g] then
9          g ← right
10     if g ≠ i then
11         tab[i] ↔ tab[j]
12         heapify(tab, m, g)
```

À partir de cette fonction on peut donc écrire la méthode permettant de construire un tas à partir d'un tableau passé en entrée. L'algorithme, ci-dessous, est relativement simple puisqu'il commence au milieu du tableau et construit successivement le tas. Notons que l'on commence à $\lfloor \frac{n}{2} \rfloor - 1$ afin de s'assurer que les fils d'un nœd respectent bien la propriété de tas.

```
1  Function buildHeap(Integer[] tab, Integer n)
2      for i from ⌊n/2⌋ − 1 down to 0 do
3          heapify(tab, i, n)
```

Avant de regarder plus en avant comment on peut réaliser une procédure de tri avec ces deux méthodes, regardons leur complexité.

Le meilleur cas pour heapify, est quand $g = i$: dans ce cas aucun appel récursif ne sera nécessaire à la ligne 12. Dans le meilleur cas cet algorithme est en temps constant. Dans le cas où un appel récursif est requis, dans le pire cas, le sous arbre gauche peut avoir deux fois plus de nœuds que le sous arbre droit. En effet, il s'agit du cas où le fils gauche du nœud i possède un *niveau* de plus de le fils droit de i. Si s représente le nombre de nœuds dans l'arbre, alors le fils gauche peut avoir jusqu'à $\lceil \frac{2s-1}{3} \rceil = \frac{2s-1}{3} + \mathrm{O}(1)$ nœuds On peut alors exprimer la complexité de heapify de la manière suivante :

$$T_{heapify}(s) \leq T_{heapify}(\frac{2s-1}{3} + \mathrm{O}(1)) + \mathrm{O}(1)$$

Comme on peut le voir, il ne s'agit pas réellement du théorème général puisque l'on fait ici une sur-approximation. Cependant en posant $U(s) =$

$U(\frac{2s-1}{3} + \mathrm{O}(1)) + \mathrm{O}(1)$, on peut appliquer le théorème général et déduire que $U(s) = \Theta(\log(s))$. Il vient $T_{heapify}(s) = \mathrm{O}(\log(s))$.

On peut maintenant calculer la complexité de la fonction buildheap. Clairement la complexité vient du nombre de fois où la ligne 3 va être exécutée. Dans le pire cas, l'on travaille sur un sous arbre de taille n (cas où i vaut zéro) et l'appel à heapify est en $\mathrm{O}(\log(n))$. Bien que ce coût soit cependant moindre dans les autres itérations, on peut exprimer la complexité de la manière suivante :

$$T_{buildheap}(n) = \Theta(n) + \lfloor \frac{n}{2} \rfloor \times \mathrm{O}(\log(n) = \mathrm{O}(n \times \log(n))$$

Il s'agit d'une borne supérieure pour notre algorithme. Néanmoins on peut être plus précis en ne surapproximant pas tous les appels à heapify :

$$T_{buildheap}(n) = \Theta(n) + \sum_{1 \leq h \leq \lfloor \log n \rfloor} S(h, n) \times T_{heapify}(h)$$

En posant : $S(h, n)$ le nombre de sous arbres de taille h. En exprimant les choses de la façon précédente, on peut facilement déduire que $S(h, n) \leq \frac{n}{2^h}$. Il vient alors :

$$\sum_{1 \leq h \leq \lfloor \log n \rfloor} S(h, n) \times T_{heapify}(h) \leq \sum_{1 \leq h \leq \lfloor \log n \rfloor} \frac{n \times \mathrm{O}(h)}{2^h} = \mathrm{O}(n)$$

En effet, après avoir sorti le n, la suite converge et on peut la réduire à $\mathrm{O}(1)$. Avec cette expression plus fine, on a pu établir une complexité plus précise pour buildheap.

À partir d'un tas-max, il est facile de trier un tableau. En effet, l'élément maximum du tableau se trouve à la racine de l'arbre, il suffit de l'échanger avec l'élément en dernière position dans le tableau. L'élément est ainsi à sa place mais le tas privé de sa dernière case ne respecte pas les propriétés d'un tas. Il est alors nécessaire d'appeler buildheap sur le tableau privé de la dernière case. Il suffit ensuite de recommencer jusqu'à ce que le tas à considérer soit vide.

```
1 Function heapSort(Integer[] tab, Integer n)
2     buildHeap(tab, n)
3     for i from n − 1 down to 0 do
4         tab[0] ↔ tab[i]
5         heapify(tab, 0, i)
```

Il vient :

$$T_{heapsort}(n) = \Theta(n) + n \times \mathrm{O}(\log(n)) = \mathrm{O}(n \times \log(n))$$

De plus, l'algorithme heapsort est en place puisqu'il ne nécessite qu'une quantité constante de mémoire supplémentaire. En revanche cet algorithme n'est pas stable puisque prendre le maximum pour le placer en fin de tableau va faire varier l'ordre des éléments égaux.

Note. Lorsque l'on passe un tableau trié à la fonction heapsort, l'algorithme va commencer par mélanger ce tableau pour construire la tas, puis va ensuite le réordonner correctement. L'algorithme smoothsort proposé par E.W. Dijkstra propose de résoudre se problème en se basant non pas sur des arbres binaires mais sur des arbres de Léonardo.

3.5 Tri rapide

L'algorithme de tri rapide a été proposé par Sir Charles Antony Richard Hoare en 1960. Cet algorithme peut être décomposé en deux étapes :

1. le partitionnement du tableau de telle façon que tous les éléments à gauche d'un élément spécifique sont plus petits que cet élément et tous ceux à droite sont plus grands.
2. L'appel récursif sur chacun des sous tableaux précédents (gauche et droite).

La première étape de cet algorithme consiste donc à partitionner le tableau. Une façon de faire cela est de prend une valeur x (le *pivot*) appartenant au tableau et de l'utiliser pour savoir si les éléments appartiennent à la parie gauche ou bien à la partie droite. De manière générale, cette technique consiste à avance deux indices i et j jusqu'à ce que $tab[i] > x \wedge tab[j] < x$: il suffit alors d'inverser les deux cases et de recommencer jusqu'à ce que i et

j se rencontrent.

```
1  Function partition(Integer[] tab, Integer ℓ, Integer r)
2      x ← A[ℓ]
3      i ← ℓ − 1
4      j ← r
5      while ⊤ do
6          do
7              i ← i + 1
8          until tab[i] ≥ x
9          do
10             j ← j − 1
11         until tab[j] ≤ x
12         if j ≤ i then
13             return i + (i = ℓ)
14         tab[i] ↔ tab[j]
```

La seconde étape de cet algorithme effectue tout d'abord un appel à la fonction partition pour positionner de manière définitive le pivot. Cet appel permet en plus de « ranger » toutes les valeurs plus petites que le pivot à gauche de celui-ci et toutes les valeurs plus grandes à droite de celui-ci. Il suffit alors d'appeler récursivement la fonction quicksort sur la partie gauche et sur la partie droite pour obtenir un tableau trié.

```
1  Function quickSort(Integer[] tab, Integer ℓ, Integer r)
2      if r − ℓ = 1 then
3          p ← partition(A, ℓ, r)
4          quickSort(A, ℓ, p)
5          quickSort(A, p, r)
```

3.5.1 Complexité

todo.

3.5.2 Variantes

todo.

3.6 Choix du pivot & calcul de médiane

Dans l'algorithme précédent, la valeur prise pour partionner le tableau était choisie de manière arbitraire comme le premier élément de ce tableau (ligne 2 de l'algorithme partition). Cette valeur, appelée *pivot*, peut en réalité être choisie parmi toutes les cases du tableau (ou du sous-tableau) considéré. Le choix d'un bon pivot permet un meilleur partitionnement du tableau. Deux grandes stratégies peuvent être envisagées :

1. Choix arbitraire : on choisit arbitrairement n'importe quel élément du tableau (premier élément, dernier élément, élément aléatoire, ...)
2. Choix sélectif : on sélectionne l'élément qui a le plus de « chances » de partionner correctement le tableau. Il est évident que le meilleur pivot est l'élément qui possède la valeur médiane, puisqu'il va scinder le tableau en deux partie égales.

Bien que le choix sélectif permette un partionnement optimal, il n'est en pratique par envisageable car trop coûteux. En effet, choisir la valeur médiane parmi n valeurs requiert $n-1$ comparaisons ce qui augmenterait considérablement la complexité du quicksort. Généralement, ce choix sélectif est réalisé sur un sous ensemble de t valeurs (avec t fixé) ce qui permet de n'impacter que faiblement la complexité de quicksort tout en améliorant le partionnement pour éviter que l'algorithme tombe dans son pire cas à chaque appel récursif, i.e. une partition de taille 1 et une de taille $n-1$. Même si dans la pratique les implémentations de quickstort posent $t = 3$ et n'ont donc besoin d'aucun algorithme pour trouver la médiane, le calcul de celle-ci peut être fait de plusieurs manières. La première consiste à trier les t valeurs avec un algorithme en $t \times \log_2(t)$, puis de prendre la valeur en position $\frac{t}{2}$. Néanmoins, d'autres approches moins coûteuses existent. Toutes s'apparentent au problème de la selection de la k^{e} plus petite valeur parmi n valeurs. Ce problème est aussi connu sous le nom de *sélection de la médiane*.

Quickselect

Cet algorithme a été conçu par Sir C.A.R. Hoare et fonctionne sur le même principe que le quicksort. Une première étape consiste à choisir un pivot de manière aléatoire, puis de partionner les éléments du tableau selon ce pivot. La deuxième étape, contrairement à ce qui est fait dans le quicksort, n'effectue une récursion que sur une des seule des deux partitions, i.e. celle qui va contenir la k^{e} valeur recherchée. L'algorithme se termine alors lorsque

le sous-tableau considéré n'est composé que d'une unique case.

```
1  Function quickSelect(Integer[] tab, Integer ℓ, Integer r, Integer k)
2      if r − ℓ = 1 then
3          return tab[ℓ]
4      p ← partition(A, ℓ, r)
5      t ← p − ℓ
6      if k ≤ t then
7          return quickSelect(A, ℓ, p, k)
8      else
9          return quickSelect(A, p, r, t − k)
10 Function partition(Integer[] tab, Integer ℓ, Integer r)
11     x ← A[random(ℓ, r)]
12     i ← ℓ − 1
13     j ← r
14     while ⊤ do
15         do
16             i ← i + 1
17         until tab[i] ≥ x
18         do
19             j ← j − 1
20         until tab[j] ≤ x
21         if j ≤ i then
22             return i + (i = ℓ)
23         tab[i] ↔ tab[j]
```

Dans le pire cas, l'appel à partition est en $\Theta(n)$ et le tableau est découpé en deux sous parties dont l'une des tailles est $n-1$. Il vient :

$$T_{quickSelect}(n) = \Theta(n) + T_{quickSelect}(n-1)$$

On ne peut donc pas appliquer le théorème général ici. On résout par substitutions :

$$\begin{aligned} T_{quickSelect}(n) &= c \times n + T_{quickSelect}(n-1) \\ &= c \times n + (c \times n - 1) + T_{quickSelect}(n-2) \\ &= \dots \\ &= (c \times \sum_{2 \leq i \leq n} i) + \Theta(1) \\ &= \Theta(n^2) \end{aligned}$$

Dans le meilleur cas, l'appel à partition est en $\Theta(n)$ et le tableau est découpé en deux sous parties dont l'une des tailles est 1. Il vient :

$$\begin{aligned} T_{quickSelect}(n) &= \Theta(n) + T_{quickSelect}(1) \\ &= \Theta(n) \end{aligned}$$

Dans le cas moyen, en supposant une répartition équitable des valeurs, on obtient :

$$\begin{aligned} T_{quickSelect}(n) &\leq \Theta(n) + \frac{1}{n-1} \times \sum_{1 \leq t \leq n-1} max(T_{quickSelect}(t), T_{quickSelect}(n-t)) \\ &\leq \Theta(n) + \frac{1}{n-1} \times \sum_{1 \leq t \leq n-1} T_{quickSelect}(max(t, n-t)) \\ &\leq \Theta(n) + \frac{2}{n-1} \times \sum_{\lfloor \frac{n}{2} \rfloor \leq t \leq n-1} T_{quickSelect}(t) \end{aligned}$$

Cette équation de récursion ne peut clairement pas être résolue via l'utilisation du théorème général. On suppose que le cas moyen est en $O(n)$ et l'on montre cela par induction :

Hypothèse. Pour tout m tel que $m < n$, on a $T_{quickSelect}(m) \leq c \times m$, pour une valeur de c donnée qui peut être aussi grande que l'on veut. Montrons tout d'abord : $\sum_{\lfloor \frac{n}{2} \rfloor \leq i \leq n-1} i = \frac{\lceil \frac{n}{2} \rceil \times (\lfloor \frac{n}{2} \rfloor + n - 1)}{2} \leq \frac{3}{8} \times n^2$. On distingue deux cas :

1. Si n est pair.

$$\sum_{\lfloor \frac{n}{2} \rfloor \leq i \leq n-1} i = \frac{3 \times n - 2}{2} \times \frac{n}{4} = \frac{3 \times n^2 - 2 \times n}{8} \leq \frac{3}{8} \times n^2$$

2. Si n est impair.

$$\sum_{\lfloor \frac{n}{2} \rfloor \leq i \leq n-1} i = \frac{3 \times n - 3}{2} \times \frac{n+1}{4} = \frac{3 \times n^2 - 3 \times n}{8} \leq \frac{3}{8} \times n^2$$

Récurrence. On souhaite montrer que $T_{quickSelect}(n) = O(n)$ dans le cas moyen. On a par hypothèse de récurrence $\forall t < n-1$, $T_{quickSelect}(t) \leq c \times t$. On a donc

$$
\begin{aligned}
T_{quickSelect}(n) &\leq \Theta(n) + \frac{2}{n-1} \times \sum_{\lfloor \frac{n}{2} \rfloor \leq t \leq n-1} T_{quickSelect}(t) \\
&\leq \Theta(n) + \frac{2}{n-1} \times c \times \sum_{\lfloor \frac{n}{2} \rfloor \leq t \leq n-1} (t) \\
&\leq \Theta(n) + \frac{2}{n-1} \times \frac{3}{8} \times c \times n^2 \\
&\leq \Theta(n) + \frac{6 \times c \times n^2}{8 \times (n-1)} \\
&\leq \Theta(n) + \frac{6 \times c \times (n \times (n-1) + (n-1) + 1)}{8 \times (n-1)} \\
&\leq \Theta(n) + \frac{6 \times c \times n \times (n-1)}{8 \times (n-1)} + \frac{6 \times c \times ((n-1))}{8 \times (n-1)} + \frac{6 \times c}{8 \times (n-1)} \\
&\leq \Theta(n) + \frac{6 \times c \times n}{8} + \frac{6 \times c}{8} + \frac{6 \times c}{8 \times (n-1)} \\
&\leq c \times n - \frac{1}{4} \times c \times n + \Theta(n)
\end{aligned}
$$

Or $(\Theta(n) - \frac{1}{4} \times c \times n)$ est négligeable si c est assez grand. On a donc montré que dans le cas moyen $T_{quickSelect}(n) = O(n)$.

Linearselect

L'algorithme de la section précédente est non-déterministe car il choisit le pivot de manière aléatoire. De plus, ce choix aléatoire le conduit à être en $\Theta(n^2)$ dans le pire cas. Il a longtemps été considéré qu'on ne pouvait pas construire d'algorithme linéaire dans le pire cas qui soit en plus déterministe. C'est en essayant de prouver cette théorie en 1972 que Blum, Floyd, Pratt, Rivest et Tarjan ont découvert qu'elle était fausse. Ils ont alors proposé un algorithme linéaire dans le pire cas comme réponse à ce problème. Cet algorithme peut être découpé en cinq grandes étapes :

1. Diviser toutes les valeurs en groupes de 5 valeurs. On obtient alors $\lceil \frac{n}{5} \rceil$ groupes.
2. Calculer la médiane de chaque groupe. Cette opération peut être faite en un temps constant sur un groupe de 5 valeurs.
3. Calculer la médiane de toutes les médianes en s'appelant récursivement.

4. En fonction du rang que l'on cherche on peut supprimer un quart du tableau. Cette opération est linéaire dans le nombre d'éléments.
5. On récurse sur ce qu'il reste, i.e. sur les $\frac{3}{4}$ de tableau restant.

Bien que cet algorithme soit simple sur son concept, il nécessite de grosses manipulations du tableau en entrée. Nous ne fournirons donc pas le code ici. Nous pouvons néanmoins déduire facilement la complexité de cette fonction :

$$\begin{aligned} T_{linearSelect} &= \Theta(n) + \Theta(1) + T(\lceil \frac{n}{5} \rceil) + \Theta(n) + T(\frac{3 \times n}{4}) \\ &= \Theta(n) + T(\lceil \frac{n}{5} \rceil) + T(\frac{3 \times n}{4}) \end{aligned}$$

Nous ne pouvons clairement pas appliquer ici le théorème général. On effectue donc une preuve par induction pour montrer $T_{linearSelect}(n) = \Theta(n)$. Il suffit cependant de montrer que $T_{linearSelect}(n) = O(n)$ car d'après l'équation de récurrence il est clair que $T_{linearSelect}(n) = \Omega(n)$.

Hypothèse. Pour tout m tel que $m < n$, on a $T_{linearSelect}(m) \leq c \times m$, pour une valeur de c donnée qui peut être aussi grande que l'on veut

Récurrence. On souhaite montrer que $T_{linearSelect}(n) = O(n)$. On a par hypothèse de récurrence $\forall t < n - 1$, $T_{linearSelect}(t) \leq c \times t$. On a donc :

$$\begin{aligned} T_{linearSelect} &= \Theta(n) + T(\lceil \frac{n}{5} \rceil) + T(\frac{3 \times n}{4}) \\ &\leq \Theta(n) + \frac{c \times n}{5} + \frac{3 \times c \times n}{4} \\ &\leq \Theta(n) + \frac{19 \times c \times n}{20} \\ &\leq \Theta(n) + c \times n - \frac{1}{20} \times c \times n \end{aligned}$$

Or $(\Theta(n) - \frac{1}{20} \times c \times n)$ est négligeable si c est assez grand. On a donc montré que $T_{linearSelect}(n) = O(n)$. Comme $T_{linearSelect}(n) = O(n)$ et $T_{linearSelect}(n) = \Omega(n)$, il vient $T_{linearSelect}(n) = \Theta(n)$.

Remarque. Il est important de diviser le tableau en groupes de 5 minimum. En effet si on divise en groupe de 3, on obtient :

$$\begin{aligned} T_{linearSelect} &= \Theta(n) + T(\lceil \frac{n}{3} \rceil) + T(\frac{3 \times n}{4}) \\ &\leq \Theta(n) + \frac{c \times n}{3} + \frac{3 \times c \times n}{4} \\ &\leq \Theta(n) + \frac{13 \times c \times n}{12} \end{aligned}$$

On ne peut alors pas trouver de c tel que $(\Theta(n)+\frac{1}{12}\times c\times n)$ soit négligeable. Si on divise par groupes de 3 on ne peut alors pas déterminer la complexité de l'algorithme.

Exemple. Considérons le tableau $tab = \{7, 8, 9, 14, 2, 3, 10, 11, 1, 13, 4, 12, 5, 6\}$ dont on souhaite extraire la le ième élément (ou la médiane). Nous présentons ici un déroulement schématique des grandes étapes de l'algorithme.

1. Découpage en groupes de cinq éléments :
   7 2 13
   8 3 4
   9 10 12
   14 11 5
   15 1 6
2. Trouver la médiane de chaque groupe et réordonner chaque groupe
   7 1 4
   8 2 5
   9 3 6
   14 10 12
   15 11 13
3. Calculer la médiane des médiannes récursivement :
   1 4 7
   2 5 8
   3 6 9
   10 12 14
   11 13 15

   Il apparaît alors clairement que l'on peut supprimer un quart du tableau. En effet, toutes les valeurs vertes sont inférieures ou égales à la médiane tandis que les valeurs bleues sont supérieurs.

3.7 Comparaison des algorithmes de tri basés sur les comparaisons

Les tableaux ci-dessous résument les différentes complexité calculées sur les tris vu précédemment. Il est important de noter que ces tableaux présentent aussi les complexités spatiales jusqu'alors ignorées. On voit que pour le QuickSort par exemple la complexité spatiale est de $\Theta(n)$ ce qui correspond aux différents appels récursifs de l'algorithme. On note en revanche que le coût mémoire de HeapSort n'est que de $\Theta(1)$: comme il s'agit d'une

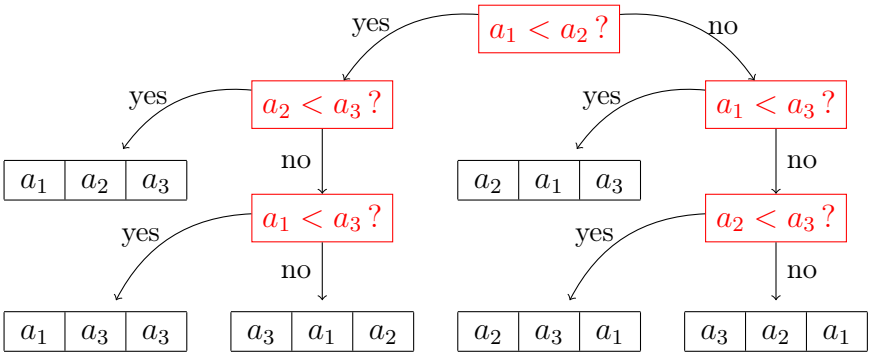
récursion terminale, celle-ci peut aisément être supprimée et remplacée par une boucle.

	Cas général	Pire	Moyen	Meilleur
SelectionSort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
InsertionSort	$O(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
MergeSort	$\Theta(n\log(n))$	$\Theta(n\log(n))$	$\Theta(n\log(n)$	$\Theta(n\log(n)$
HeapSort	$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n)$	$O(n\log(n)$
QuickSort	$O(n^2)$	$O(n^2)$	$O(n\log(n)$	$O(n\log(n)$
QuickSort (rec petit)	$O(n^2)$	$O(n^2)$	$O(n\log(n)$	$O(n\log(n)$
IntroSort	$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n)$	$O(n\log(n)$

	Memoire	En place	Stable
SelectionSort	$\Theta(1)$	✓	×
InsertionSort	$\Theta(1)$	✓	✓
MergeSort	$\Theta(n)$	×	✓
HeapSort	$\Theta(1)$	✓	×
QuickSort	$\Theta(n)$	✓	×
QuickSort (rec petit)	$\Theta(\log(n))$	✓	×
IntroSort	$\Theta(\log(n))$	✓	×

Au vu de ces résultats, il est légitime de se demander si l'on peut construire un algorithme de tri basé sur les comparaisons dont le pire cas soit mieux que $\Omega(n \times \log(n))$.

Les algorithmes de tri peuvent être représentés par un arbre binaire (ou de décision). Les nœuds internes représentent alors la comparaison de deux éléments : le fils gauche désigne une réponse négative et le fils droit une réponse positive. Les feuilles représentent alors les permutations à effectuer pour obtenir un tableau trié.



Comme il existe $n!$ permutations possibles pour un tableau de taille n, l'arbre binaire possède $n!$ feuilles et sa hauteur est donc d'au moins $\lceil \log(n!) \rceil$. La plus courte branche a donc au moins $n-1$ comparaison et les algorithmes de tris basés sur les comparaisons sont donc en $\Omega(n \times \log(n))$.

3.8 Tri par dénombrement

Dans la section précédente nous avons vu qu'il n'était pas possible de construire un algorithme de tri basé sur les comparaisons qui soit linéaire. Il est cependant possible de construire des algorithmes de tri qui soient linéaires en ne les basant pas sur des comparaisons. L'algorithme proposé par Harold H. Seward [1] est basé sur le calcul d'histogramme ce qui lui permet d'être linéaire. Il peut être découpé en 5 grandes étapes :

1. Recherche du minimum et du maximum dans le tableau considéré. Cette étape est linéaire.
2. À partir du minimum et du maximum, on peut créer une fonction $key(x)$ qui, pour un élément x du tableau, retourne un entier appartenant à $[0, M[$ où M est la différence entre le le minimum et le maximum.

   Si le minimum est positif ou nul, il est simple de poser directement M comme le maximum du tableau. Sinon la fonction key doit décaler tous les éléments de la valeur minimum.
3. On construit ensuite l'histogramme H des valeurs du tableau.
4. H est ensuite modifié de façon à connaître, pour une valeur x le nombre d'éléments qui lui sont petits ou égaux.
5. On peut alors construire le tableau trié en s'aidant des informations calculées.

```
1  Function countingSort(Integer[] tab, Integer n, Integer[] out,
2                        Integer M, Function key(x) → x ∈ [0, M[)
3      for i = 0 to M − 1 do
4          H[i] ← 0
5      for i = 0 to n − 1 do
6          H[key(tab[i])] ← 1 + H[key(tab[i])]
7      for i = 1 to M − 1 do
8          H[i] ← H[i] + H[i − 1]
9      for i = n − 1 down to 0 do
10         H[key(tab[i])] ← H[key(tab[i])] − 1
11         out[H[key(tab[i])]] ← tab[i]
```

1. Cet algorithme (ainsi que le tri par base de la section suivante) date en réalité de 1887 avec les travaux sur les tabulatrices de Herman Hollerith (qui fondera Tabulating Machine Co qui devint IBM en 1917).

Comme l'algorithme n'utilise que des boucles simples il est facile d'en analyser la complexité. Il vient

$$\begin{aligned} T_{countingSort}(n, M) &= 2 \times (\sum_{0 \leq i \leq M-1} 1) + 2 \times (\sum_{0 \leq i \leq n-1} 1) \\ &= \Theta(n) + \Theta(M) \end{aligned}$$

Si $M = O(n)$ alors il vient $T_{countingSort}(n, M) = \Theta(n)$

Remarque. Notons que cet algorithme n'est clairement pas en place puisqu'il utilise deux tableau auxiliaires H et *out*. Néanmoins comme la dernière boucle est effectuée de manière décroissante cet algorithme est stable.

3.9 Tri par base

Le tri par base (aussi connu sous le nom *radix sort*) est plus une technique qu'un algorithme de tri a proprement parler. Tel que conçu originalement, l'algorithme vise à trier des nombres en considérant seulement un chiffre (ou un certain nombre de bits) à la fois. On distingue donc deux grande catégories de tris par base :

1. Least Significant Bits (LSB) : on considère les nombres à trier en parcourant les chiffres de gauche à droite, i.e. en partant des bits les moins significatifs.
2. Most Significant Bits (MSB) : on considère les nombres à trier en parcourant les chiffres de droite à gauche, i.e. en partant des bits les plus significatifs.

Lors d'un tri LSB (respectivement MSB) il suffit donc de commencer par trier les valeurs en regardant seulement le chiffre le plus a droite (resp. plus à gauche), puis seulement les chiffre en avant dernière position à droite (resp. avant seconde position depuis la gauche) et ainsi de suite. À la fin de la dernière itération, toutes les valeurs sont triées.

Remarques. Dans cet algorithme il est important de considérer le *zero-padding*, i.e considérer tous les nombres comme étant exprimés sur le même nombre de bits. De plus, bien que cet algorithme soit présenté sur les nombres, il est facile de l'adapter à pour la comparaison d'objets plus complexes tels que les chaînes de caractère qui peuvent être vues comme des entiers.

Exemple. Nous allons maintenant appliquer un *radix sort LSB* pour trier des codes postaux. Ce tri est aussi connu comme tri postal. Considérons $tab = \{1234, 9504, 9712, 7512, 3201\}$ l'ensemble des codes postaux que l'on souhaite trier.

En suivant la technique du tri par base on peut donc commencer par trier les valeurs en ne considérant que le chiffre des unité, puis que celui des dizaines, puis que celui des centaines pour finir par celui des milliers. Ces différentes étapes sont décrites ci-dessous :

1. Le tableau à trier

   1234
   9504
   9712
   7512
   3201

2. Tri selon le chiffre des unités :

   3201
   9712
   7512
   1234
   9504

3. Tri selon le chiffre des dizaines :

   3201
   9504
   9712
   7512
   1234

4. Tri selon le chiffre des centaines :

   3201
   1234
   9504
   7512
   9712

5. Tri selon le chiffre des milliers :

   1234
   3201
   7512
   9504
   9712

Notons que ce tri fonctionne correctement essentiellement grâce au caractère stable de chaque étape de tri. Nous pouvons donc exprimer cet algorithme de la manière suivante :

```
1 Function postalSort(Integer[] tab, Integer n)
2     Integer[n] out
3     countingSort(tab, n, out, 10, x ↦ x%10)
4     countingSort(out, n, tab, 10, x ↦ ⌊x/10⌋%10)
5     countingSort(tab, n, out, 10, x ↦ ⌊x/100⌋%10)
6     countingSort(out, n, tab, 10, x ↦ ⌊x/1000⌋%10)
7     tab ↔ out
```

Il vient alors : $T_{postalSort} = \Theta(n)$ si $n > 10$. Dans le cas général, un tri par base sur des mots de taille maximale w a donc une complexité de $\Theta(w \times n)$.

3.10 Tri par paquets

Les algorithmes de tri précédents ne permettent pas de travailler facilement sur des nombres réels. L'algorithme présenté ici s'attaque à ce problème en triant les éléments par paquets (*buckets*). L'idée générale de cet algorithme est d'essayer de partionner l'ensemble des n valeurs à trier dans n partitions différentes. Pour cela, il suffit de découper l'intervalle des valeurs à trier en n paquets, puis d'insérer chaque élément dans le paquet correspondant. Chaque paquet peut ensuite être trié indépendamment des autres, et il suffit de combiner tous le paquets pour obtenir un tableau trié. L'algorithme peut donc être décomposé en 5 grandes étapes :

1. Trouver l'intervalle $[a, b[$ qui contient toutes les valeurs de *tab*
2. Initialisation d'un tableau de buckets, où chaque bucket est représenté par une liste vide. Pour un bucket k on sait que l'intervalle associé est : $[a + k \times \frac{b-a}{n}, a + (k+1) \times \frac{b-a}{n}[$
3. Parcours du tableau original et insertion de chaque élément dans le bon bucket
4. Tri de chaque bucket indépendamment
5. Reconstruction du tableau en fusionnant tous les buckets

```
1 Function bucketSort(Integer[] tab, Integer n)
2     Buckets ← new array[n] of empty lists
3     for i = 0 to n − 1 do
4         Insert tab[i] in Bucket[⌊n × (tab[i]−a)/(b−a)⌋]
5     for i = 0 to n − 1 do
6         Sort Bucket[i] using insertSort
7     return Concatenate Bucket[0], Bucket[1] ... Bucket[n-1]
```

Dans le meilleur cas, chaque paquet ne contient qu'un unique élément, i.e. $\forall i \in [0, n[, |Bucket[i]| = 1$. Il vient alors $T_{bucketSort} = \Theta(n)$.

Dans le pire cas, un paquet possède tous les éléments et tous les autres paquets sont vides, i.e. $\exists i$ t.q. $|Bucket[i]| = n$ et $\forall j \neq i \ |Bucket[i]| = 0$. Dans ce cas, il vient :

$$\begin{aligned} T_{bucketSort} &= \Theta(n) + \Theta(n^2) + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

Chapitre 4

Les structures de données

4.1 Étude des structures « classiques »

Tous les algorithmes du chapitre précédents ont été présentés sur des tableaux. Dans le cas des tris, cette structure est particulièrement adaptée car elle permet d'accéder aux éléments en temps constant. Dans ces algorithmes, l'utilisation d'une autre structure de donnée peut avoir un impact significatif sur la complexité de l'algorithme.

Lorsque l'on écrit un algorithme il faut donc être capable d'identifier la structure de donnée qui est le mieux adaptée au problème auquel l'on s'attaque. Lorsque l'on dispose d'une structure de donnée S, plusieurs opérations sont envisageables :

1. $Access(S, p) \rightarrow v$: retourne l'élément en position p de la structure S.
2. $Search(S, v) \rightarrow p$: cherche si l'élément v est présent dans la structure S. Si l'élément est trouvé son indexe est retourné, sinon la valeur -1 est retournée.
3. $Insert(S, v)$: effectue l'insertion de l'élément v dans la structure S.
4. $Delete(S, p)$: effectue la suppression de l'élément en position p dans la structure S.
5. $Min(S) \rightarrow p$: retourne la position de l'élément ayant la valeur minimale dans S.
6. $Max(S) \rightarrow p$: retourne la position de l'élément ayant la valeur maximale dans S.
7. $Succ(S, p) \rightarrow p$: retourne la position du successeur de l'élément d'indice p en respectant l'ordre des éléments de S

8. $Pred(S, p) \rightarrow p$: retourne la position du prédécesseur de l'élément d'indice p en respectant l'ordre des éléments de S

Au vu de cette interface, on peut évaluer le coût de chacune de ces opérations sur les structures de données classiques, i.e. les tableaux, les listes simplement chaînées et les listes doublement chaînées.

	Tableau		Liste		Liste DC	
Opération	non-trié	trié	non-triée	triée	non-triée	triée
$Access(S, p)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
$Search(S, v)$	$\Theta(n)$	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
$Insert(S, v)$	$\Theta(1)$	$O(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
$Delete(S, p)$	$\Theta(1)$	$O(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
$Min(S)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
$Max(S)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
$Pred(S, p)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
$Succ(S, P)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$

Remarques diverses

1. Il existe des listes particulières pour lesquelles la recherche est en $\Theta(\log(n))$: il s'agit de skiplists
2. L'insertion dans un tableau trié demande une première opération en $\Theta(\log(n))$ pour trouver la position à laquelle insérer l'élément, puis une opération en $O(n)$ pour déplacer tous les autres éléments.
3. La suppression d'un élément d'une liste étant donné sa position (un pointeur) peut être faite en temps constant uniquement si l'on peut copier les valeurs, i.e. copier le successeur de l'élément à supprimer dans l'élément à supprimer et ensuite supprimer ce successeur

À partir de ces informations on peut choisir la structure de donnée la plus adaptée pour un algorithme. Par exemple si un algorithme a besoin de faire n^2 insertions, n recherches et n suppression, on peut calculer le coût d'utiliser un tableau trié ou non trié.

	tableau non-tiré	tableau trié
n^2 insertions	$\Theta(n^2)$	$O(n^3)$
n recherches	$O(n^2)$	$O(n \log(n))$
n suppression	$\Theta(n^2)$	$O(n^3)$
Total	$\Theta(n^2)$	$O(n^3)$

Dans le cas ci-dessus il apparaît clairement que l'utilisation d'un tableau non-trié est plus intéressante en termes de complexité que l'utilisation d'un tableau non-trié. Dans le cas suivant, en revanche, l'utilisation d'un tableau trié semble plus avantageuse :

	tableau non-tiré	tableau trié
n insertions	$\Theta(n)$	$O(n^2)$
n^2 recherches	$O(n^3)$	$O(n^2 \log(n))$
n suppression	$\Theta(n)$	$O(n^2)$
Total	$\Theta(n^3)$	$O(n^2 \log(n))$

4.2 Les arbres rouge-noir

Les arbres rouge-noir (ou bicolores) sont des arbres binaires de recherche particuliers sur lesquelles toutes les opérations » classiques « peuvent être faites en $O(\log(n))$. Cette structure de donnée a été inventée en 1972 par Rudolf Bayer. Un tel arbre peut être défini suivant quatre points :

1. Il s'agit d'un arbre binaire de recherche avec toutes les propriétés qui y sont associées.
2. La racine de l'arbre est noire
3. Le(s) fils d'un nœud rouge est(sont) noir(s)
4. La hauteur *noire* de l'arbre est constante. Ainsi, le ratio entre la plus petite et la plus grande branche est au maximum de 2, i.e. la plus grande branche est au pire deux fois plus grande que la plus petite branche.

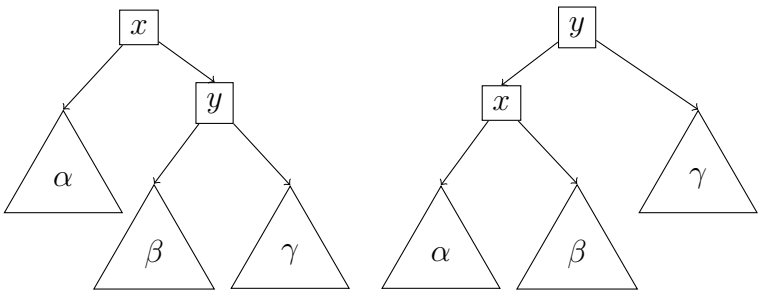


FIGURE 4.1 – Ordre à préserver $\alpha \leq x \leq \beta \leq y \leq \gamma$. De gauche à droite $Rot(y)$. De droite à gauche $Rot(x)$.

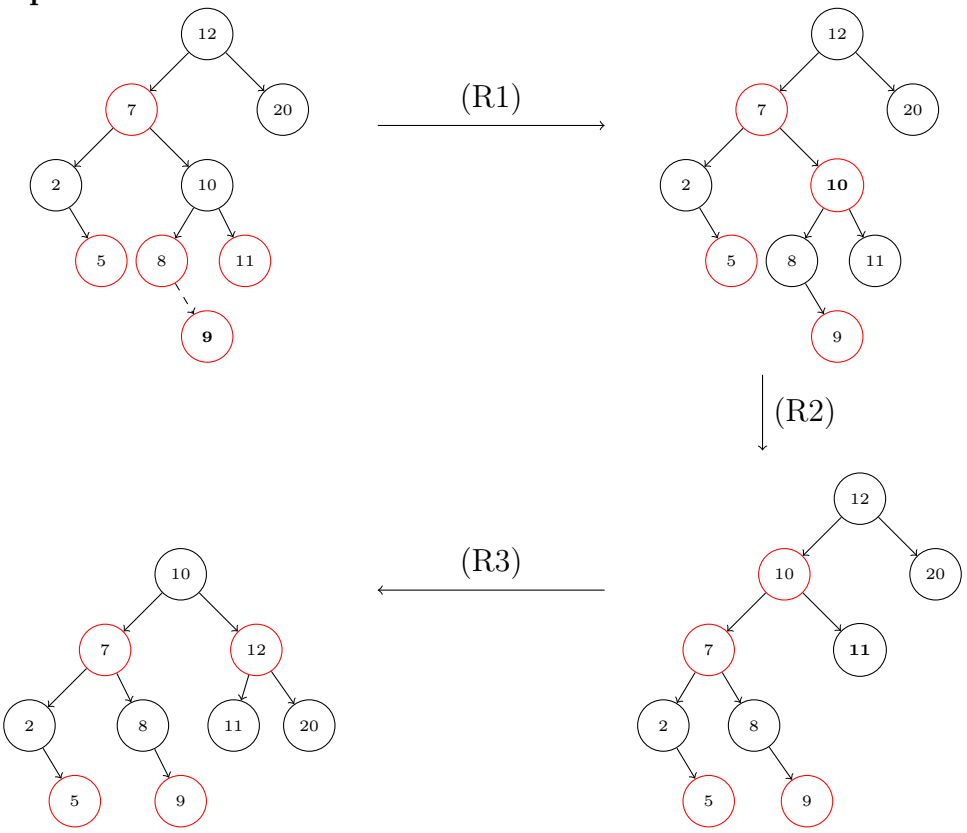
L'inconvénient sur ce type d'arbre réside dans la mise en place des procédures d'insertions/suppression. Toutes les deux sont basées sur le principe

de rotation propre aux arbres binaires. L'exemple ci-dessus montre deux rotations possibles pour un arbre binaire de recherche.

À partir de cette rotation on peut facilement construire la procédure d'insertion dans les arbres rouge-noir. Il suffit en effet seulement d'insérer le nouveau nœud (de couleur rouge) dans l'arbre puis de corriger l'arbre jusqu'à ce que toutes les propriété soient satisfaites. Pour cela il suffit d'appliquer les règles suivantes :

1. Si le père et l'oncle sont rouges, alors on peut les repeindre en noir et inverser la couleur du grand père.
2. Si le père et l'oncle ont des couleurs différentes **et** que le père et le grand père ne sont pas alignés alors on effectue une rotation
3. Si le père et l'oncle ont des couleurs différentes **et** que le fils, le père et le grand père ne sont pas alignés alors on échange les couleurs du père et du grand père après avoir fait une rotation.
4. Si le noeud courant est rouge et racine de l'arbre, alors il est repeint en noir.

Exemple.



Ecrire la procédure minimum/maximum Ecrire la procédure de recherche dans l'arbre Ecrire la procédure de recherche de deux noeuds successifs rouge Ecrire la procédure qui vérifie que l'arbre vérifie la propriété 4 Ecrire la procédure rot(left)/rot(right)

4.3 Tableaux dynamiques

Dans les algorithmes de tri vus à la section précédente, les tableaux manipulés étaient de taille fixe. Néanmoins, dans certains cas, la taille n'est pas connue à l'avance. On a alors besoin de manipuler des tableaux dynamiques, aussi appelés vecteurs.

Ces tableaux dynamiques fonctionnent de la manière suivante. On alloue un tableau d'une taille donnée fixée à l'avance. Lors d'une insertion, si le nombre d'élément maximal pouvant être stocké dans le tableau est atteint, on augmente la taille du tableau et l'on recopie les valeurs. Il vient donc que toutes les opérations, à l'exception de l'insertion, ont le même coût que pour les tableaux classiques.

Plusieurs stratégies peuvent être envisagées pour la réallocation :

1. Initialement le tableau est d'une case. Chaque insertion provoque une réallocation. Le coût d'une insertion est alors $O(n)$.
2. L'insertion utilise une réallocation pour ajouter 1000 entiers. L'insertion coûte donc $O(n)$ lorsqu'il y a réallocation et $O(1)$ sinon. La complexité amortie est donc de $\frac{O(n)+999\times O(1)}{1000} = O(n)$.
3. L'insertion appelle une réallocation qui va doubler la taille du tableau. L'insertion coûte donc $O(n)$ lorsqu'il y a réallocation et $O(1)$ sinon. $\frac{O(n)+(n-1)\times\Theta(1)}{n} = \Theta(1)$.

La troisième stratégie permet donc d'avoir une insertion qui a un coût similaire a l'insertion dans un tableau de taille fixe.

4.4 Tables de Hachages

Il existe de nombreux problèmes manipulant des ensembles dynamiques dans lesquels les seules opérations réalisées sont l'insertion, la suppression et la recherche. On cherche donc a mettre en place une structure efficace pour ce type de problème.

Principe des tables de hachages. Chaque élément à stocker est associé à une clef. Il suffit alors de rechercher les informations via cette clef. De manière plus générale, une table de hachage peut être vue comme l'association d'un ensemble de données appelées *clefs* auxquelles on associe des données appelées *valeurs*.

— La table de hachage peut être vue comme un tableau qui va stocker toutes les *valeurs*

— Pour insérer une *valeur* dans la table de hachage, on commence par regarder la *clef* associée. Cette clef est ensuite passée à une fonction appelée *fonction de hachage*. L'objectif de cette fonction est de retourner un indice dans le tableau où l'on va pouvoir stocker la *valeur*.

Une fonction de hachage est dite parfaite si elle ne produit jamais de collisions, i.e. s'il n'existe pas deux clefs qui puissent avoir la même valeur de hachage.

Exemple. Considérons l'ensemble des clefs suivantes K = { "january", "february", "march", "april", "may", "june", "july", "august", "september", "october", "november", "december" }. On peut constuire une fonction de hachage parfaite $h : w \to (w[1] + w[2])\%32$. On a donc :

$$\begin{aligned}
h(\text{"}january\text{"}) &= (97 + 110)\%32 = 15 \\
h(\text{"}february\text{"}) &= (98 + 101)\%32 = 7 \\
h(\text{"}march\text{"}) &= (97 + 114)\%32 = 19 \\
h(\text{"}april\text{"}) &= (112 + 114)\%32 = 2 \\
h(\text{"}may\text{"}) &= (97 + 121)\%32 = 26 \\
h(\text{"}june\text{"}) &= (117 + 110)\%32 = 3 \\
h(\text{"}july\text{"}) &= (117 + 108)\%32 = 1 \\
h(\text{"}august\text{"}) &= (117 + 103)\%32 = 28 \\
h(\text{"}september\text{"}) &= (101 + 112)\%32 = 21 \\
h(\text{"}october\text{"}) &= (99 + 116)\%32 = 23 \\
h(\text{"}november\text{"}) &= (111 + 118)\%32 = 5 \\
h(\text{"}december\text{"}) &= (101 + 99)\%32 = 8
\end{aligned}$$

Avec cette fonction de hachage, il suffit simplement d'un tableau de 32 valeurs pour pouvoir stocker toutes les clefs et donc toutes les valeurs que l'on souhaite y associer.

Remarque. Il est très difficile de trouver une fonction de hachage parfaite puisqu'il s'agit de trouver une fonction qui soit injective entre l'ensemble des clefs K et l'ensemble des valeurs V. En effet, le nombre de fonctions de K dans V est $|V|^{|K|}$, et le nombre de fonction injectives de K dans V est $|V|^{\underline{|K|}}$.

4.4.1 Résolution par chaînage

Dans l'exemple de la section précédente, l'ensemble des clefs était connu à l'avance ce qui a permis de construire une fonction de hachage parfaite. Lorsque l'on n'arrive pas à trouver (ou que l'on ne peut pas trouver) une fonction de hachage parfaite, il peut avoir plusieurs éléments qui ont la même valeur de hachage. Dans ce cas, il est nécessaire de gérer les collisions.

L'idée d'une résolution par chaînage de chaîner les éléments en collision. Le tableau représentant la table de hachage contient alors des listes initialement vides. Lors d'une insertion, l'élément est alors inséré dans la liste correspondant au bucket auquel il est associé. *Note : le bucket correspond à la case retournée par la fonction de hachage pour un élément donné.*

En moyenne la taille d'une liste est $\frac{nb_valeurs}{nb_cases}$. On a donc les complexités suivantes pour chaque opérations :

- Suppression : $O(1 + \frac{nb_valeurs}{nb_cases})$
- Recherche : $O(1 + \frac{nb_valeurs}{nb_cases})$
- Insertion : $O(1 + \frac{nb_valeurs}{nb_cases})$. L'insertion peut aussi être en $O(1)$ si l'on ne vérifie pas la présence d'un élément avant l'insertion.

L'objectif est donc d'avoir une table qui soit dimensionné de manière à ce que $\frac{nb_valeurs}{nb_cases}$ tende vers zéro. En effet, dans ce cas, toutes les opérations sont en $O(1)$. Pour réaliser cela, il suffit de détecter lorsque $\frac{nb_valeurs}{nb_cases}$ devient trop grand. Dans ce cas là, comme pour les tableaux dynamiques, on double la taille du tableau et on hache a nouveau toutes les valeurs.

Avantages	Inconvénients
Insertion facile	Une allocation à chaque insertion
Recherche facile	Chaque couple (clef, valeur) doit en plus stocker un pointeur

4.4.2 Résolution par coalescence/sondage linéaire

Lorsque l'on veut éviter d'avoir une allocation pour chaque insertion on peut directement stocker les valeurs dans le tableau utilisé pour représenter la table de hachage. Deux techniques existent alors, la résolution par coalescence et celle par sondage linéaire.

Résolution par sondage linéaire. Dans cette technique, lors de l'insertion, on commence par calculer la valeur hachée de la clef. On regarde alors si il y a déjà un élément à cette indice dans le tableau. Si il n'en existe pas on insère l'élément, sinon on regarde si la case suivante est libre et ainsi de suite. Cette technique est efficace lorsque l'on effectue une recherche fructueuse mais peut se révéler particulièrement inefficace lorsque la recherche est infructueuse. De plus, lors d'une suppression on est obligé d'introduire une valeur particulière pour ne pas rendre incorrect les futures recherches.

Résolution par coalescence. Cette technique se rapproche de la technique précédente dans le sens où lors d'une insertion, si la case est déjà occupée, il va être nécessaire de trouver une nouvelle case libre. En revanche plutôt que d'effectuer une recherche linéaire, on va rechercher une nouvelle case libre en faisant un nouveau hachage par exemple. Une fois cette nouvelle case trouvée, on peut insérer le nouvel élément. L'inconvénient de cette technique est que chaque élément doit avoir un pointeur sur son successeur pour faciliter les recherches.

4.4.3 Fonctions de hachage

Dans les sections précédentes nous avons vu que de nombreuses stratégies existent pour gérer les conflits. Cependant, ces techniques ne sont déclenchées que lorsque la fonction de hachage produit des collisions. Il est donc essentiel d'avoir une bonne fonction de hachage, i.e. une fonction :

1. qui se calcule rapidement en $O(1)$
2. qui minimise les collisions autant que possible

Il existe plusieurs techniques pour construire une fonction de hachage :

1. **Hachage par extraction.** Dans ce type de fonction, la valeur de hachage d'une clef est obtenue en récupérant des bits précis sur la clef. Par exemple si $key = 0010011101100110$, et que la fonction ne regarde que les bits 1, 3, 5 et 7, on obtient $h(key) = (0011)_b = 3$

L'inconvénient de ce type de fonction est que la valeur de hachage ne dépend pas de l'intégralité de la clef. Une bonne fonction de hachage fait intervenir tous les bits de la clef.

2. **Hachage par division.** Dans ce type de fonction, on obtient la valeur de hachage en faisant $h(key) = key \mod m$, avec m premier et éloigné d'une puissance de 2. Cette technique offre l'avantage d'avoir une bonne répartition mais multiplie plus facilement les collisions et le choix de m est critique. Elle est donc généralement combinée avec les autres techniques.
3. **Hachage par multiplication.** Dans ce type de fonction, on obtient la valeur de hachage en appliquant $h(key) = \lfloor m \times (key \times A - \lfloor x \times A \rfloor) \rfloor$, avec $A \in]0, 1[$ et m quelconque. Dans cette technique, comme m n'est pas critique on peut utiliser la taille de la table pour avoir une bonne répartition. De plus, Knuth a montré que pour $A = \frac{\sqrt{5}-1}{2}$ la fonction obtenue est plutôt bonne.
4. **Hachage par Multiplication, Addition et Division (MAD).** Dans ce type de fonction, on obtient la valeur de hachage en appliquant $h(key) = (a \times key + b) \mod m$, avec $a > 0$ et $b > 0$. Dans cette technique il est important que $a \neq m$ pour que toutes les valeurs ne soient pas égales à b. Cette technique peut offrir de bon résultats mais le choix des constantes peut être compliqué à trouver.
5. **Hachage par compression logique.** Dans ce type de fonction la clef est transformée en utilisant les opérations de logique booléenne, i.e. $h(key) = key \; op \; m$, où $op \in \{\wedge, \vee, \oplus\}$. L'utilisation du OU logique ($\vee$) permet d'obtenir un nombre plus grand que la clef, ce qui va favoriser les collisions en fin de tableau. L'utilisation du ET logique ($\wedge$) permet d'obtenir un nombre plus petit que la clef, ce qui va favoriser les collisions en début de tableau. L'utilisation du OU exclusif ($\oplus$) évite les problèmes rencontrés avec le $\wedge$ ou le $\vee$, et fournit une répartition uniforme des valeurs. Néanmoins, il y a plus de collisions dues aux permutations.
6. **Hachage par compression polynomiale.** Dans ce type de fonction, la clef est découpée en blocks de bits de taille fixée, i.e. $key = a_0 a_1 a_{...} a_{n-1}$. On évalue ensuite le polynome : $P(z) = a_0 \times z^0 + a_1 \times z^1 + \ldots + a_{n-1} \times z^{n-1}$. Ce type de hachage offre une très bonne compression pour les chaînes de caractère et il a été montré qu'en prenant $z = 33$ on a au plus 6 collision pour un ensemble de 50000 mots anglais.

De manière générale, la construction d'une bonne fonction de hachage est un problème très compliqué et dépendant du problème que l'on souhaite traiter.