# Solution Série 1

Andrey Martinez Cruz

## Table des matières

# 1 Glossaire et notation

L'analyse des algorithmes se fera en deux temps : une non simplifié et une simplifié.

- Non simplifié : par non simplifié, cela veut dire que la complexité d'une ligne spécifique ne sera pas réduite directement à sa classe de complexité. Par exemple, si une ligne d'un algorithme est exécutée en temps constant, mais on n'exécute $n+1$ fois cette ligne, alors on dira seulement que cette ligne $\mathcal{O}(n+1)$. On ne simplifira pas que $n+1 \in \mathcal{O}(n)$.
- Simplifié : Ici on simplifie tout directement à sa classe de fonction approprié. Par exemple, si une ligne s'exécute $n+1$ fois et qu'une exécution de cette ligne est en temps constant, alors on simplifira qu'elle s'exécute en $\mathcal{O}(n)$.

Note : dans les deux cas, il y aura simplification de l'algorithme seulement lorsqu'on prend la somme de tous les temps d'exécution de chaque ligne qui sont exécuté dans l'algorithme (appel de fonction, comparaison, affectation et etc...). La complexité de certaines fonctions sont et de ce qu'elles font sont plus détaillée dans la section annexe.

En ce qui concerne les algorithmes et les tableaux, on posera qu'ici, la première case d'un tableau se retoruve à l'indice 1 et sa dernière case à l'indice $n$ pour un tableau de taille $n$.

Pour les exercises 1 et 2 pour la partie analyser la complexité de l'algorithme, les opérations qui permet de déterminer la complexité ou le nombre de fois que ces opérations sont effectuées seront en rouge.

# 2 Exercise 1

Pour compredre comment résoudre le problème, il faut se rappeler comment faire des additions élémentaires (en base 10 d'abord).

Prenons les deux nombres suivants 999 et 888 dont on veut connaître la somme :

$$\begin{array}{r} {\scriptstyle 11} \\ 999 \\ +\ 888 \\ \hline 1887 \end{array}$$

Remarque que les 1 au dessus sont les retenus à faire. En base 2 cela s'applique aussi excepté que la retenue se fait à partie du moment qu'on a au moins l'opération $(1)_2 + (1)_2$ à faire.

Exemple : soit $A = 7 = (111)_2$ et $B = 15 = (1111)_2$, alors $C$, la somme de ces deux nombres est le suivant :

$$\begin{array}{r} {\scriptstyle 111} \\ 0111 \\ +\ 1111 \\ \hline 10110 \end{array}$$

On peut vérifier qu'on a le bon résultat $(0001\ 0110)_2 = (16)_{16} = (22)_{10}$ ce qui est bien le résultat attendu $7 + 15 = 22$.

## 2.1 Question A

La première variante de l'algorithme est si le bit le moins significatif se retrouve à la première case du tableau :

1. **Fonction** AdditionBin(A,B)
2. $n \leftarrow Taille(B)$
3. $C \leftarrow CreeTab(n+1)$
4. $retenue \leftarrow 0$
5. Pour $i \leftarrow 1$ haut $n$ faire
6. $C[i] \leftarrow (A[i] + B[i] + retenue) \mod 2$
7. $retenue \leftarrow (A[i] + B[i] + retenue) \div 2$
8. Fin pour

9. $C[n+1] \leftarrow retenue$
10. *Renvoyer* $C$
11. Fin Fonction

La deuxième variante de l'algorithme est si le bit le plus significatif se retrouve à la première cas du tableau :

1. **Fonction** AdditionBin(A,B)
2. $n \leftarrow Taille(B)$
3. $C \leftarrow CreeTab(n+1)$
4. $retenue \leftarrow 0$
5. Pour $i \leftarrow n+1$ bas 2 faire
6. $C[i] \leftarrow (A[i-1] + B[i-1] + retenue) \mod 2$
7. $retenue \leftarrow (A[i-1] + B[i-1] + retenue) \div 2$
8. Fin pour
9. $C[1] \leftarrow retenue$
10. *Renvoyer* $C$
11. Fin Fonction

## 2.2 Question B

Pour l'analyse de la complexité, on a pris la première variante, mais la première et deuxième variante auront la même complexité.

1. **Fonction** AdditionBin(A,B)
2. $n \leftarrow Taille(B)$ $(\mathcal{O}(n))$
3. $C \leftarrow CreeTab(n)$ $(\mathcal{O}(n))$
4. $retenue \leftarrow 0$ $(\mathcal{O}(1))$
5. Pour $i \leftarrow 1$ haut $n$ faire $(n+1$ fois$)$
6. $C[i] \leftarrow (A[i] + B[i] + retenue) \mod 2$ $(n$ fois$)$
7. $retenue \leftarrow (A[i] + B[i] + retenue) \div 2$ $(n$ fois$)$
8. Fin pour
9. *Renvoyer* $C$ $(\mathcal{O}(1))$
10. Fin Fonction

Voici des détails sur la complexité des différentes lignes :

1. Ligne 2 ($c_1$) et 3 ($c_2$) : ces deux fonctions nécessitent de parcourir tout le tableau ($\mathcal{O}(n)$). Voir dans annexe pour plus de détails.

2. Ligne 4 ($c_3$) : l'affectation d'une valeur à une variable se fait en temps constant ($\mathcal{O}(1)$).
3. Ligne 5 ($c_4$) : Outre l'initialisation de $i$, la boucle pour nécessite de comparer à chaque fois la valeur de $n$ pour savoir s'il faut encore iterer dans la boucle ou non. Il faut ajouter à celà qu'il y a une comparaison supplémentaire à faire qui sera celle de l'arrêt de la boucle. Et $f(n) = n + 1 \in \mathcal{O}(n)$.
4. Ligne 6 ($c_5$) et 7 ($c_6$) : L'accès à un tableau se fait en temps constant ($\mathcal{O}(1)$) et les opérations élémentaire effectués se font aussi en temps constant (dans ce contexte). Cependant, parce que ces lignes se font exécutés $n$ fois dans la boucle, leur complexité tombera en temps linéaire. ($\mathcal{O}(n)$).
5. Ligne 9 ($c_7$) : Renvoyer une variable ou valeur sans faire aucune opération se fait en temps constant. ($\mathcal{O}(1)$).

La complexité peut être représentée de la manière suivante :

$$f(n) = c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 \tag{1}$$

Complexité non simplifié :

$$f(n) = n + n + 1 + (n + 1) + n + n + 1 = 5n + 3 \in \mathcal{O}(n) \tag{2}$$

Complexité simplifié :

$$f(n) = n + n + 1 + n + n + n + 1 = 5n + 2 \in \mathcal{O}(n) \tag{3}$$

# 3 Exercise 2

Pour comprendre comment faire une multiplication élémentaire en binaire, on doit revoir ce que ça fait du côté en base 10.

Prenons les nombres suivants : 5130 et 6120. La multiplication de celle-ci est la suivante :

$$\begin{array}{rr} & {\scriptstyle 1} \\ & 5130 \\ \times & 6120 \\ \hline & 0000 \\ + & 101600 \\ + & 513000 \\ + & 30780000 \\ \hline & 31395600 \end{array}$$

Note : la retenue 1 est seulement applicable lorsque qu'on ait rendu à faire $1 \times 6$.

On remarque que pour chaque multiplication d'un chifre, avant de multiplier chaque chiffre, on rajoute un certain nombre de zéros (avant de faire la multiplication) qui correspond à la position du chiffre dans le deuxième facteur moins 1.

Exemple avec $A = (111)_2$ et $B = (111)_2$.

$$\begin{array}{rr} & 111 \\ \times & 111 \\ \hline & 111 \\ + & 1110 \\ + & 11100 \\ \hline & 110001 \end{array}$$

Or, $A = (7)_{10}$ et $B = (7)_{10}$ et $7 \times 7 = 49 = (110001)_2$.

## 3.1 Question A

Pour comprendre le nombre d'éléments qu'au moins $C$ doit contenir, il faut re-

voir quelle est la position du bit le plus significatif dans $B$. Ce qu'on remarque c'est que pour chaque puissance de 2 pour $B$, le prochain nombre pour la prochaine addition partielle du résultat contient un certain nombre de zéros qui est équivalent à $i$ la puissance de 2 en cours de traitement moins 1. Par exemple, le nombre $(1110)_2$ le bit le plus significatif est dans la position $2^3$ et en prenant l'exposant, on obtient le nombre de zéro qu'il faut ajouter au début du nombre avant de commencer à multiplier le premier facteur.

$C$ contient $n$ chiffres découlant de $A$ et $B$ plus $i-1$ chiffres qui représente le nombre de chiffres supplémentaire que doit contenir $C$ où $i$ représente la position du bit le plus significatif dans $B$. À cela, il faut rajouter un chiffre de plus, car il est possible d'avoir une retenue pour l'addition finale. Donc, $C$ doit contenir le nombre de chiffre suivants :

$$n + (n-1) + 1 = 2n \tag{4}$$

Donc, $C$ contient $2n$ cases dans son tableau.

## 3.2 Question B

Comme pour la première question, il y aura deux variantes : bit le moins significatif jusqu'au bit le plus significatif et la deuxième variante qui est l'inverse.

Première variante :

```
1.  Fonction MultiplierBin(A,B)
2.      Resultat ← CreeTab(Taille(B))
3.      decalage ← 0
4.      Pour i ← 1 haut n faire
5.          Temp ← Initialiser(n + decalage)
6.          Pour j ← 1 haut n faire
7.              Temp[j + decalage] ← A[j] × B[decalage]
8.          Fin pour
9.          Res ← AdditionnerBin(Temp, Res)
10.         decalage ← decalage + 1
11.     Fin Pour
12.     Renvoyer Resultat
13. Fin Fonction
```

Deuxième variante :

```
1.  Fonction MultiplierBin(A,B)
```

2. $Resultat \leftarrow CreeTab(Taille(B))$
3. $decalage \leftarrow 0$
4. Pour $i \leftarrow n$ bas 1 faire
5. $Temp \leftarrow Initialiser(n + decalage)$
6. Pour $j \leftarrow n$ bas 1 faire
7. $Temp[j - decalage] \leftarrow A[j] \times B[decalage]$
8. Fin pour
9. $Res \leftarrow AdditionnerBin(Temp, Res)$
10. $decalage \leftarrow decalage + 1$
11. Fin Pour
12. Renvoyer Resultat
13. Fin Fonction

### 3.3 Question C

L'analyse est basée sur la première approche, mais la complexité pour la deuxième variante restera le même.

1. **Fonction** MultiplierBin(A,B)
2. $Resultat \leftarrow CreeTab(Taille(B))$ $(\mathcal{O}(n))$
3. $decalage \leftarrow 0$ $(\mathcal{O}(1))$
4. Pour $i \leftarrow 1$ haut $n$ faire $(n + 1$ fois$)$
5. $Temp \leftarrow Initialiser(n + decalage)$ $(n$ fois$)$
6. Pour $j \leftarrow 1$ haut $n$ faire $(n + 1$ fois$)$
7. $Temp[j + decalage] \leftarrow A[j] \times B[decalage]$ $(n^2$ fois$)$
8. Fin pour
9. $Res \leftarrow AdditionnerBin(Temp, Res)$ $(n$ fois$)$
10. $decalage \leftarrow decalage + 1$ $(n$ fois$)$
11. Fin Pour
12. Renvoyer Resultat $(\mathcal{O}(1))$
13. Fin Fonction

Voici des détails sur la complexité des différentes lignes :

1. Ligne 2 ($c_1$) : cette fonction s'exécute en temps linéaire. Voir dans annexe pour plus de détails.
2. Ligne 3 ($c_2$) : l'affectation d'une valeur à une variable se fait en temps constant ($\mathcal{O}(1)$).

3. Ligne 4 ($c_3$) et 6 ($c_5$) : Outre l'initialisation de $i$ ou de $j$, la boucle pour nécessite de comparer à chaque fois la valeur de $n$ pour savoir s'il faut encore iterer dans la boucle ou non. Il faut ajouter à celà qu'il y a une comparaison supplémentaire à faire qui sera celle de l'arrêt de la boucle. Et $f(n) = n + 1 \in \mathcal{O}(n)$. En revanche, pour la ligne 6, on fait la boucle $n$ fois dû que c'est une boucle interne à la première boucle qui s'exécute $n$ fois. Donc, le nombre de comparaison fait pour la ligne 6 se fait $(n + 1) \times n = n^2 + n \in \mathcal{O}(n^2)$ fois.
4. Ligne 5 ($c_4$) : La fonction $Initialiser$ s'exécute en temps linéaire ($\mathcal{O}(n)$), mais parce qu'on appel $n$ fois cette fonction dû au nombre d'itération nécessaire à exécuter dans la boucle, la complexité devient quadratique ($\mathcal{O}(n^2)$.
5. Ligne 7 ($c_6$) : cette ligne revient à dire que pour un chiffre dans $B$ précisé par la variable *decalage*, on multiplie chaque chiffre de $A$ et on insére chaque résultat dans la case approprié de $Temp$. Dû que $A$ est de taille $n$, l'instruction se fait $n$ fois et on rèpete cela pour chaque chiffre de $B$ qui est aussi de taille $n$. Finalement, on exécute cette ligne $n^2$ fois ce qui donne une complexité en temps quadratique. ($\mathcal{O}(n^2)$).
6. Ligne 9 ($c_7$) : la fonction $AdditionnerBin$ se fait en temps linéaire (revoir la complexité dans l'exercise 1) et parce qu'on l'exécute $n$ fois, la complexité de cette ligne se fait en temps quadratique ($\mathcal{O}(n^2)$).
7. Ligne 10 ($c_8$) : L'affectation d'une valeur à une variable ce fait en temps constant, mais parce qu'on exécute cette opération $n$ fois, cette instruction s'exécute en temps linéaire. ($\mathcal{O}(n)$).
8. Ligne 12 ($c_9$) : Renvoyer une variable ou valeur sans faire aucune opération se fait en temps constant. ($\mathcal{O}(1)$).

La complexité de cette fonction peut être représentée de la manière suivante :

$$f(n) = c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_8 + c_9 \tag{5}$$

Complexité non simplifié :

$$f(n) = n+1+(n+1)+n^2+(n^2+n)+n^2+n^2+n+1 = 4n^2+4n+3 \in \mathcal{O}(n^2) \tag{6}$$

Complexité simplifié :

$$f(n) = n + 1 + n + n^2 + n^2 + n^2 + n^2 + n + 1 = 4n^2 + 3n + 2 \in \mathcal{O}(n^2) \tag{7}$$

# 4 Exercise 3

## 4.1 Force brute

Pour trouver la plus grande valeur de $n$ telle que $100n^2 > 2^n$, on peut essayer toutes les combinaisons possibles et prendre la dernière valeur de $n$ qui respecte l'équation si haut.

| n | Algo 1 | Algo 2 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 100 | 2 |
| 2 | 400 | 4 |
| 3 | 900 | 8 |
| 4 | 1600 | 16 |
| 5 | 2500 | 32 |
| 6 | 3600 | 64 |
| 7 | 4900 | 128 |
| 8 | 6400 | 256 |
| 9 | 8100 | 512 |
| 10 | 10000 | 1024 |
| 11 | 12100 | 2048 |
| 12 | 14400 | 4096 |
| 13 | 16900 | 8192 |
| 14 | 19600 | 16384 |
| 15 | 22500 | 32768 |

## 4.2 Approche "fouille binaire"

Plutôt que d'essayer toute les combinaisons jusqu'à trouver la première valeur qui enfreint l'inégalité précédemment citée, on peut essayer de deviner le nombre en faisant une approche comme si c'était une recherche binaire. Par exemple, si $n = 1$, alors l'algo #1 donne 100 et l'algo #2 et si on prends $n = 20$, alors l'algo #1 donne 40000 et l'algo #2 donne 1048576. Donc, on sait que la valeur à rechercher et entre ses deux bornes. On regarde si le milieu est le prochain élément fait que pour $n$, l'inéquation est respecté et que pour $n+1$, l'inéquation n'est plus respecté. Or, quand $n = 10$ et $n = 11$, l'inéquation est toujours respecté et donc, on avance la borne inférieur de 1. Et répéte le processus jusqu'à qu'on trouve le dernier $n$ telle que $100n^2 > 2^n$ est vraie et son prochain terme qui rend l'inéquation fausse. À la fin, on obtiendra 14 comme résultat valide.

En bref, voici le processus appliquée : soit $i$ et $j$ telle que

1. $i = 1$

2. $j = c$ où $c$ est une valuer telle que $100n \leq 2^n$

regarde l'élément au centre des deux bornes et le prochain élément du milieu et voir si le milieu calculé de la même façon qu'une fouille binaire $\lfloor \frac{i+j}{2} \rfloor$ respecte l'inéquation suivante $100m > 2^n$ et si l'élément après le milieu respecte l'inéquation de l'indice $j$. Si ce n'est pas le cas avancer la borne inférieur de 1 et refaire le même processus jusqu'à la condition citée précédemment soit vraie.

# 5 Exercise 4

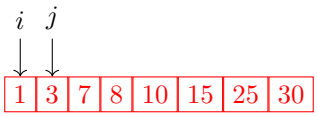
## 5.1 Approche naive (inefficace)

L'approche naive consiste à vérifier pour chaque élément dans le tableau s'il présent seulement une seule fois dans le tableau. C'est à dire pour chaque élément, il faut parcourir tout le tableau chaque fois pour tous les éléments. Donc, si on parcours un tableau avec $n$ éléments pour chaque élément, alors la complexité de l'approche naive se fait en $\mathcal{O}(n^2)$.

## 5.2 Approche 1

Une approche possible est de en premier triée le tableau avec un algorithme de tri qui se fait au plus en $\mathcal{O}(n \log n)$ (tri fusion ou tri par monceau) et de vérifier si le prochain élément de l'élément courant qu'on est équivalent, alors il y a présence de doublon.

Par exemple si $i = 1$ et que $j = i + 1$, alors dans le tableau $T$ suivant après le tri :




Dans ce cas-ci, si l'élément à l'indice $i$ et l'élément à l'indice $j$ sont différents, on avance $i$ de 1 et on refait la même chose jusqu'à que soit $i$ est égale au dernier index représentant le dernier élément du tableau (renvoyer vrai) ou soit que l'élément à l'indice $i$ et l'élément à l'indice $j$ sont égaux (renvoyer faux).

Voici l'algorithme suivant :

1. **Fonction** EstSansDoublon (T)
2. $n \leftarrow$ Taille($T$)
3. TrierTableau($T$)
4. Pour $i = 1$ haut $n - 1$ faire
5. Si $T[i] = T[i + 1]$ alors
6. Renvoyer Faux
7. Fin Si
8. Fin pour
9. Renvoyer Vrai

Temps d'exécution :

- Ligne 2 ($c_1$) : en temps linéaire ($\mathcal{O}(n)$)
- Ligne 3 ($c_2$) : si le tri utilisé est un tri fusion ou un tri par monceau, la complexité est en $\mathcal{O}(n \log n)$.
- Ligne 4 ($c_3$) : il y a $n$ comparaisons à faire entre $i$ et $n-1$ pour savoir s'il faut continuer la boucle ou s'arrêter. ($\mathcal{O}(n)$).
- Ligne 5 ($c_4$) : la comparaison se fait en temps constant, mais dans le pire des cas, on fait cette comparaison $n-1$ fois, et cette ligne est en $\mathcal{O}(n)$.
- Ligne 9 ($c_5$) : Temps constant ($\mathcal{O}(1)$)

La complexitée de cette fonction peut être représentée par la somme de tous les temps d'exécutions qui fûrent analyser de la façon suivante :

$$f(n) = c_1 + c_2 + c_3 + c_4 + c_5 \tag{8}$$

Complexité non simplifié :

$$f(n) = n + (n \log n) + n + (n-1) + 1 = n \log n + 3n \in \mathcal{O}(n \log n) \tag{9}$$

Complexité simplifié :

$$f(n) = n + (n \log n) + n + n + 1 = n \log n + 3n + 1 \in \mathcal{O}(n \log n) \tag{10}$$

## 5.3 Approche 2

Une seconde approche possible est d'utiliser un arbre binaire de recherche (AVL ou rouge-noir) est de vérifier si l'élément courant qu'on vérifie est déjà dans l'arbre. Si ce n'est pas le cas, on insére cette élément et on passe au suivant jusqu'à avoir fait tous les éléments (pas de doublon) ou soit qu'un élément est déjà présent dans l'arbre (contient un doublon).

Donc, on peut formuler cette approche avec l'algorithme suivant :

```
1. Fonction EstSansDoublonArbre (T)
2.     n ← Taille(T)
3.     abr ← arbreVide()
4.     Pour i = 1 haut n faire
5.         Si existe(abr,T[i]) alors
6.             Renvoyer Faux
7.         Sinon
8.             inserer(abr,T[i])
```

9. Fin Si
10. Fin pour
11. Renvoyer Vrai
12. Fin Fonction

La complexité de certaines lignes dans cet algorithme sont les suivantes :

- Ligne 2 ($c_1$) : temps linéaire $\mathcal{O}(n)$
- Ligne 3 ($c_2$) : en temps constant $\mathcal{O}(1)$
- Ligne 4 ($c_3$) : il y a $n+1$ comparaisons à faire entre $i$ et $n$ pour savoir s'il faut conteinuer la boucle ou s'arrêter. ($\mathcal{O}(n)$).
- Ligne 5 ($c_4$) : on appel dans le pire des cas $n$ fois cette fonction et parce qu'il est executé en temps logarithmique, la complexité est en $\mathcal{O}(n \log n)$.
- Ligne 8 ($c_5$) : se fait exécutée $p$ fois où $1 \leq p \leq n$. Dans le pire des cas, $p = n$. Et donc, parce que la procédure est exécutée en temps logarithmique, dans le pire des cas, cette ligne se fait en $\mathcal{O}(n \log n)$.
- Ligne 11 ($c_6$) : en temps constant $\mathcal{O}(1)$

La complexitée de cette fonction peut être représentée par la somme de tous les temps d'exécutions qui fûrent analyser de la façon suivante :

$$f(n) = c_1 + c_2 + c_3 + c_4 + c_5 + c_6 \tag{11}$$

Complexité non simplifié :

$$f(n) = n+1+(n+1)+(n\log n)+(n\log n)+1 = 2n\log n+2n+3 \in \mathcal{O}(n\log n) \tag{12}$$

Complexité simplifié :

$$f(n) = n+1+n+(n\log n)+(n\log n)+1 = 2n\log n+2n+2 \in \mathcal{O}(n\log n) \tag{13}$$

## 5.4 Approche 3

On reprend la même approche que l'approche 2 sauf qu'on utilise à la place une table de hachage qui nous permet de vérifier et d'insérer un élément en temps constant si la fonction de hachage est bonne.

Algorithme :

1. **Fonction** EstSansDoublonHachage (T)
2. $n \leftarrow$ Taille($T$)
3. $set \leftarrow$ initTabHash($n$)
4. Pour $i = 1$ haut $n$ faire

5. Si existeHash($set$,$T[i]$) alors
6. Renvoyer Faux
7. Sinon
8. insererHash($set$,$T[i]$)
9. Fin Si
10. Fin pour
11. Renvoyer Vrai
12. Fin Fonction

Voici quelques détails sur certaines lignes :

- Ligne 2 ($c_1$) et 3 ($c_2$) : temps linéaire $\mathcal{O}(n)$
- Ligne 4 ($c_3$) : il y a $n+1$ comparaisons à faire entre $i$ et $n$ pour savoir s'il faut continuer la boucle ou s'arrêter. ($\mathcal{O}(n)$).
- Ligne 5 ($c_4$) : on appel dans le pire des cas $n$ fois cette fonction et parce que dans le cas d'une table de hachage cette fonction est exécuté en temps constant si on assume une bonne fonction de hachage, la complexité de cette ligne est en $\mathcal{O}(n)$.
- Ligne 8 ($c_5$) : se fait exécutée $p$ fois où $1 \leq p \leq n$. Dans le pire des cas, $p = n$. Et donc,parce que cette procédure est exécutée en temp constant pour une table de hachage, dans le pire des cas, cette ligne se fait en $\mathcal{O}(n)$.
- Ligne 11 ($c_6$) : en temps constant $\mathcal{O}(1)$

La complexité de cette fonction peut être représentée par la somme de tous les temps d'exécutions qui fûrent analyser de la façon suivante :

$$f(n) = c_1 + c_2 + c_3 + c_4 + c_5 + c_6 \tag{14}$$

Non simplifié :

$$f(n) = n + n + (n+1) + n + n + 1 = 5n + 2 \in \mathcal{O}(n) \tag{15}$$

Simplifié :

$$f(n) = n + n + n + n + n + 1 = 5n + 1 \in \mathcal{O}(n) \tag{16}$$

La complexité de cet algorithme se fait dans le pire des cas est la suivante :

$$f(n) = 4n + 1 \in \mathcal{O}(n) \tag{17}$$

# 6 Exercise 5

## 6.1 Approche naive (inneffícace)

L'approche naive consistera de parcourir pour chaque case du tableau un élément autre que celui de la case courante un autre nombre qui ferais qu'additionner ces deux nombres donnerai la somme recherché. Dans le pire des cas, il faudrait regarder pour tous les éléments du tableau s'il existe un élément autre que celui de la dernière case qui ferais que la sommation des deux nombres donnerai la sommation recherché. Donc, si la taille du tableau est de $n$ éléments et qu'il faut parcourir $n$ fois ce tableau (1 pour chaque case du tableau) alors la complexité de cette approche serait en $\mathcal{O}(n^2)$.

## 6.2 Approche 1

Cette approche consiste à d'abord trier le tableau (avec les mêmes algortihmes de tris proposés à l'exercise 4) et ensuite on crée deux variables qui serviront de bornes inférieurs et supérieurs. Ensuite, on fait la procédure suivante :

1. Voir si la somme des éléments pointées par les deux bornes donnent la somme recherchée.
2. Si la somme n'est pas celle recherché, avancer la borne inférieur de 1 si la somme des éléments des deux bornes est inférieur à la somme recherché sinon dans le cas contraire, faire reculer la borne supérieur de 1.
3. Refaire les deux étapes jusqu'à trouvé la somme recherché ou jusqu'à que les deux bornes se croisent et si c'est le cas, renvoyer faux.

En bref, voici l'algorithme :

1. **Fonction** TrouverSomme(T,x)
2. $i \leftarrow 1$
3. $j \leftarrow Taille(T)$
4. $TrierTableau(T)$
5. Tant que $i < j$ faire
6. Si $T[i] + T[j] < x$ alors
7. $i \leftarrow i + 1$
8. Sinon Si $T[i] + T[j] > x$ alors
9. $j \leftarrow j - 1$
10. Sinon
11. Renvoyer Vrai
12. Fin Tant que
13. Renvoyer Faux

14. Fin Fonction

Voici des détails de la complexité temporelle de certaines lignes dans cet algorithme :

- Ligne 2 ($c_1$) : en temps constant $\mathcal{O}(1)$.
- Ligne 3 ($c_2$) : en temps linéaire $\mathcal{O}(n)$.
- Ligne 4 ($c_3$) : trier un tableau avec un tri fusion ou un tri par monceau $\mathcal{O}(n \log n)$.
- Ligne 5 ($c_4$) : il y a $n+1$ comparaisons à faire entre $i$ et $j$ pour savoir s'il faut continuer la boucle ou s'arrêter. ($\mathcal{O}(n)$).
- Ligne 6 ($c_5$) : en temps linéaire $\mathcal{O}(n)$.
- Ligne 7 ($c_6$) : $p_1$ fois où $1 \leq p_1 \leq n$
- Ligne 8 ($c_7$) : $l$ fois où $1 \leq l \leq n$
- Ligne 9 ($c_8$) : $p_2$ fois où $1 \leq p_2 \leq n$.
- Ligne 13 ($c_9$) : en temps constant $\mathcal{O}(1)$

Le pire des cas peut se produire de plusieurs façons et celle proposé et celle où la moitié du temps on incrémente la borne inférieur et l'autre moitié du temps on décrémente la borne supérieur. Donc, $p_1 = l = p_2 = \frac{n}{2} \in \mathcal{O}(n)$.

$$f(n) = c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_8 + c_9 \tag{18}$$

Complexité non simplifié :

$$f(n) = 1+n+(n\log n)+n+n+\frac{n}{2}+\frac{n}{2}+\frac{n}{2}+1 = n\log n+\frac{9n}{2}+2 \in \mathcal{O}(n\log n) \tag{19}$$

Complexité simplifié

$$f(n) = 1+n+(n\log n)+n+n+n+n+n+1 = n\log n+6n+2 \in \mathcal{O}(n\log n) \tag{20}$$

## 6.3 Approche 2

Cette approche consiste à utiliser un arbre binaire de recherche balancé (rouge-noir ou AVL) dont les éléments seront la différence entre la somme recherché et le nombre qu'on traite couramment si celui-ci n'est pas présent dans déjà dans l'arbre.

1. **Fonction** TrouverSomme(T,x)
2. $n \leftarrow Taille(T)$
3. $abr \leftarrow arbreVide()$
4. Pour $i \leftarrow 1$ haut $n$ faire
5. Si existe($abr$,$T[i]$) alors

6. renvoyer Vrai
7. Sinon
8. inserer($abr$,$x - T[i]$)
9. Fin Si
10. Fin Pour
11. Renvoyer Faux
12. Fin Fonction

Voici des détails de la complexité temporelle de certaines lignes dans cet algorithme :

- Ligne 2 ($c_1$) : en temps linéaire $\mathcal{O}(n)$.
- Ligne 3 ($c_2$) : en temps constant $\mathcal{O}(1)$.
- Ligne 4 ($c_3$) : il y a $n+1$ comparaisons à faire entre $i$ et $n$ pour savoir s'il faut continuer la boucle ou s'arrêter. ($\mathcal{O}(n)$).
- Ligne 5 ($c_4$) : en exécute $n$ fois une procédure qui se fait en temps logarithmique et donc est exécuté en $\mathcal{O}(n \log n)$.
- Ligne 8 ($c_5$) : $p$ fois où $1 \leq p \leq n$. Or, dans le pire ces cas, cette ligne se fait exécutée $n$ fois et parce que cette procédure est exécutée en temps logarithmique, la complexité de cette ligne est en $\mathcal{O}(n \log n)$.
- Ligne 11 ($c_6$) : en temps constant $\mathcal{O}(1)$.

La complexité de cette algorithme peut être décrite de la façon suivante :

$$f(n) = c_1 + c_2 + c_3 + c_4 + c_5 + c_6 \tag{21}$$

Complexité non simplifié :

$$f(n) = n+1+(n+1)+(n\log n)+(n\log n)+1 = 2n\log n+2n+3 \in \mathcal{O}(n\log n) \tag{22}$$

Complecité simplifié :

$$f(n) = n+1+n+(n\log n)+(n\log n)+1 = 2n\log n+2n+2 \in \mathcal{O}(n\log n) \tag{23}$$

## 6.4 Approche 3

Cette approche utilise la même tactique que la deuxième suaf qu'on utilise une table de hachage à la place.

1. **Fonction** TrouverSomme(T,x)
2. $n \leftarrow Taille(T)$
3. $hashSet \leftarrow initTabHash(n)$
4. Pour $i \leftarrow 1$ haut $n$ faire

5. Si existeHash($hashSet$,$T[i]$) alors
6. renvoyer Vrai
7. Sinon
8. insererHash($hashSet$,$x - T[i]$)
9. Fin Si
10. Fin Pour
11. Renvoyer Faux
12. Fin Fonction

Voici des détails de la complexité temporelle de certaines lignes dans cet algorithme :

- Ligne 2 ($c_1$) et 3 ($c_2$) : en temps linéaire $\mathcal{O}(n)$.
- Ligne 4 ($c_3$) : il y a $n + 1$ comparaisons à faire entre 1 et $n$ pour savoir s'il faut continuer la boucle ou s'arrêter. ($\mathcal{O}(n)$).
- Ligne 5 ($c_4$) : en exécute $n$ fois une procédure qui se fait en temps exécutée en temps constant (si on utilise une bonne fonction de hachage) et donc est exécuté en $\mathcal{O}(n)$.
- Ligne 8 ($c_5$) : $p$ fois où $1 \leq p \leq n$. Or, dans le pire ces cas, cette ligne se fait exécutée $n$ fois et parce que cette procédure est exécutée en temps constant (si on utilise une bonne fonction de hachage), la complexité de cette ligne est en $\mathcal{O}(n)$.
- Ligne 11 ($c_6$) : en temps constant ($\mathcal{O}(1)$).

La complexité de cet algorithme peut être décrite de la façon suivante :

$$f(n) = c_1 + c_2 + c_3 + c_4 + c_5 + c_6 \tag{24}$$

Complexité sans simplification :

$$f(n) = n + n + (n + 1) + n + n + 1 = 5n + 2 \in \mathcal{O}(n) \tag{25}$$

Complexité avec simplification :

$$f(n) = n + n + n + n + n + 1 = 5n + 1 \in \mathcal{O}(n). \tag{26}$$

# 7 Annexe

Voici la complexité de certaines fonctions utilisés :

| Fonctions/procédure | Complexité temporelle |
|---|---|
| Taille | $\mathcal{O}(n)$ |
| CreeTab | $\mathcal{O}(n)$ |
| arbreVide | $\mathcal{O}(1)$ |
| existe | $\mathcal{O}(\log n)$ |
| inserer | $\mathcal{O}(\log n)$ |
| initTabHash | $\mathcal{O}(n)$ |
| existeHash | $\mathcal{O}(1)^*$ |
| insererHash | $\mathcal{O}(1)^*$ |

Description des fonctions :

1. Taille : prend en paramètre un tableau et renvoit la taille du tableau mis en paramètre.
2. CreeTab : Crée un tableau vide dont chaque case sont initilialisées à 0
3. arbreVide : Retourne un arbre vide
4. existe : prend en paramètre un arbre binaire de recherche et un élément et vérifie si l'élément est dans l'arbre.
5. inserer : prend en paramètre un arbre binaire de recherche et un élément et insére l'élément dans l'arbre
6. initTabHash : prend un paramètre un nombre et retourne une table de hachage vide qui possède une taille correspondant au paramètre. (On suppose que la fonction de hachage ne générera aucune ou presque jamais de collisions)
7. existeHash : même fonction que la fonction existe excepté qu'elle s'attend à une table de hachage en paramètre
8. insererHash : même fonction que la fonction inserer excepté qu'elle s'attend à une tablea de hachage que plutôt à un arbre. (On suppose que la fonction de hachage ne générera aucune ou presque jamais de collisions)