Étude et Comparaison d'Algorithmes

Thibault BEZIERS LA FOSSE

Table des matières

1 Introduction

Ce projet en six séances à pour objectif d'implémenter des algorithmes, afin de calculer leurs complexités et comparer les performances. Quatre algorithmes sont à notre disposition, ce compte rendu les traitera dans l'ordre donné. Seront expliqués :

L'implémentation
La complexité théorique
Les résultats d'exécutions
L'analyse des résultats
Le comparatif des résultats théoriques et expérimentaux

1.1 Objectif

Les quatre algorithmes à implémenter consistent à trouver la suite de termes du tableau dont la somme est maximum. Les éléments du tableau sont choisis de manière aléatoire, et nous exécuterons nos algorithmes sur des tableaux de tailles variables, en commençant par des tableaux de moins d'une dizaine d'éléments, à des tableaux qui en contiennent des dizaines de milliers.

1.2 Outils utilisés

Le langage de programmation à utiliser pour ce projet étant libre, j'ai choisi de m'orienter vers le C++. D'abord pour une raison pratique, l'université nous fait régulièrement utiliser ce langage. Ensuite pour des raisons de performances, car C++ reste un langage assez efficace pour effectuer des opérations comme celles de ce projet. Le compilateur utilisé est G++, avec la librairie C++11. Effectivement, G++ est disponible à l'université, et la librairie C++11 offre des méthodes efficaces, et surtout pratiques pour manipuler des tableaux.

1.3 Structure du programme

Quatre algorithmes étant à implémenter, j'ai choisi de créer une arborescence de classes sur un seul niveau. En haut, une classe abstraite implémentant les principales méthodes nécessaires à l'exécution des algorithmes, et en bas les algorithmes qui redéfinissent la méthode de calcul de la méthode MaxSomme :

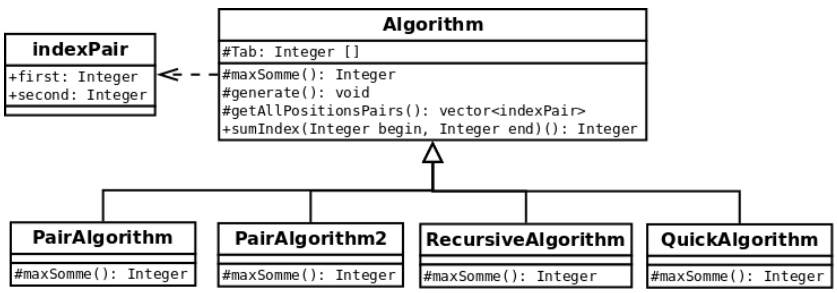


Figure 1 – Diagramme UML du programme

Les algorithmes implémentés doivent calculer la différence entre deux cases du tableau. J'ai décidé de créer une structure très simple, appelée IndexPair, contenant deux entiers : first et second, contenant chacun une cellule du tableau :

```
struct {
    int first;
    int second;
}
```

1.4 Méthodes principales

Voici les méthodes, ainsi que leurs complexités respectives, qui sont implémentées dans la classe mère pour être ensuite réutilisées dans tous les algorithmes.

1.4.1 Méthode : generate(Int n) : void

Cette méthode remplit le tableau avec des nombres aléatoires entre -1000 et 1000. Le paramètre n correspond au nombre de cases désirées pour le tableau, et l'algorithme est très simple :

```
Pour i allant de 1 à n:
    tab[i] = random(-1000, 1000)
FinPour
```

La complexité de la fonction random étant constante, cette méthode a une complexité en $\theta(n)$.

1.4.2 Méthode : getAllPositionPairs()

Cette méthode calcule toutes les paires possibles en fonction de la taille du tableau. Par exemple pour un tableau [1, 2, 3, 4], elle renverra un tableau qui contient les paires [1,2], [1, 3], [1, 4], [2, 3], [2, 4], [3, 4] L'algorithme est le suivant :

```
Paires = []                          Coût: 1
Pour i allant de 1 à n-1:            Coût: Voir ci-dessous
    Pour j allant de i+1 à n:        Coût: Voir ci-dessous
        Paires.ajouter([i, j])       coût: 1
    FinPour
FinPour
```

Cet algorithme est assez simple à implémenter, et nécessaire pour plusieurs des algorithmes de MaxSomme. Détail du calcul de sa complexité théorique :

Boucle 1 : effectuée $n-1$ fois

Boucle 2 : effectuée $(n-2)+(n-3)+(n-4)+...+n-(n-1)$ fois soit $1+2+...+n-2$

Ainsi on a la somme des entiers de 1 à n-2 qui peut être calculée ainsi :

$$\frac{(n-3)(n-2)}{2}$$

Sans détailler le calcul, on voit immédiatement qu'on a un produit de n, on peut donc en déduire une complexité en $O(n^2)$.

1.4.3 Méthode : sumIndex(Int a, Int b) : int

Cette méthode est simple, elle prend deux entiers en paramètre, qui sont chacun un indice du tableau, et calcule la somme des cases du tableau entre ces deux indices. Cet algorithme est récursif :

```
Si debut == fin:
    Retourner fin
Sinon
    Retourner tab[debut] + sumIndex(debut+1, fin)
Fin
```

On a une complexité en $\omega(1)$ si on désire la somme entre les deux mêmes cellules, et en $O(n)$ pour une somme du début à la fin du tableau.

2 Algorithme 1 : PairAlgorithm

2.1 Implémentation

Cet algorithme est facile à implémenter et repose surtout sur les principales méthodes de la classe mère *Algorithm* :

```
paires = getAllPositionPairs()
max = NaN
pour chaque paire p:
    courant = sumIndex(p.first, p.second)
    si max == NaN ou courant < max:
        max = courant
    fin si
fin pour
retourner max
```

L'implémentation en $C++$ est visible dans le programme ci-joint.

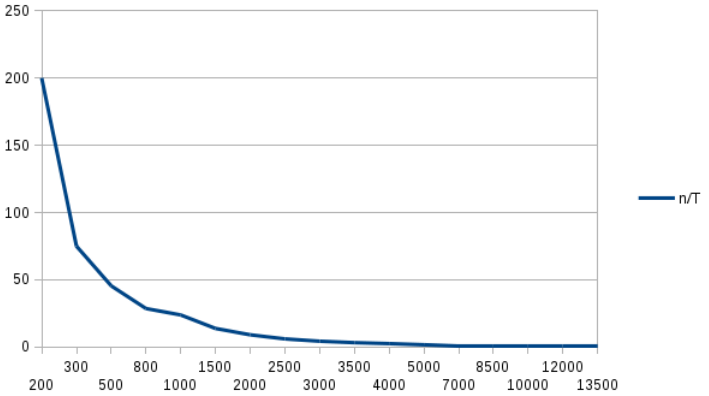
2.2 Complexité théorique

La complexité de cet algorithme repose sur la complexité des méthodes utilisées. On utilise la méthode *getPositionPairs* qui est en $\theta(n^2)$ et pour chaque résultat obtenu on effectue une méthode en $O(n)$: *SumIndex*. On peut donc en déduire que la complexité finale de cet algorithme est en $O(n^3)$

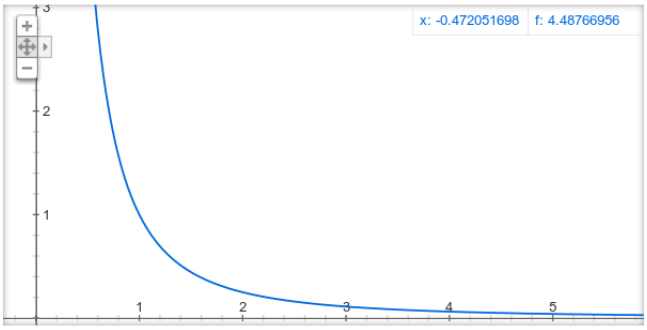
2.3 Résultats d'exécutions

L'algorithme a été exécuté sur des tableaux de taille variable, de 1 à 4000 éléments, en augmentant progressivement le pas. On obtient les résultats suivants :

Taille	Temps (millisecondes)
100	0
200	1
300	4
500	11
800	28
1000	42
1500	109
2000	221
2500	417
3000	703
3500	1084
4000	1579
5000	3023
7000	8172
8500	14289
10000	23105
12000	39960
13500	56736
14000	PAR



Voici la courbe obtenue en calculant le rapport n/t. L'augmentation des données n'étant pas régulière, cette courbe est bien plus représentative de l'évolution du temps d'exécution en fonction de la taille des données. On la compare à la courbe de $f(x) = x/x^3$.



Elles ont effectivement, à peu prêt, la même allure.

2.4 Analyse

En observant les résultats d'exécution, on peut clairement voir que le temps augmente très rapidement à partir de $n = 100$. On ignore les valeur antérieure car la vitesse d'exécution de l'algorithme est instantanée sur des données inférieures. En observant la courbe on remarque que la complexité temporelle correspond à la croissance de la fonction $f(x) = x^3$. Ainsi on peut affirmer que la complexité théorique calculée correspond à celle obtenue avec l'analyse de l'algorithme : $\theta(n^3)$.

3 Algorithme 2 : PairAlgorithm2

3.1 Implémentation

Comme l'indique le sujet, cet algorithme fonctionne en deux temps. Dans un premier temps on remplit un tableau SOMME parallèle au tableau TAB qui, pour chaque cellule de TAB contient la somme depuis la première cellule, jusqu'à la cellule courante. Par exemple :

TAB								
Cellule :	1	2	3	4	5	6	7	8
Valeur :	3	4	8	9	2	1	4	5

Somme								
Cellule :	1	2	3	4	5	6	7	8
Valeur :	3	7	15	24	26	27	31	36

De cette manière on peut aisément connaître la Somme entre deux indices du tableau en soustrayant la seconde somme à la première. Par exemple, pour connaître la somme des nombres entre la cellule 3 et la cellule 6 de Tab on peut soit faire :
$8 + 9 + 2 = 19$ avec le tableau Tab comme avec l'algorithme *PairAlgorithm*.

Ou alors :
$26 - 7 = 19$ avec le tableau Somme.
On a donc comme méthodes :

3.1.1 CalculateSommeTab

```
Somme = []
current = 0
pour i allant de 1 à taille de Tab:
    current += Tab[i]
    Somme[i] = current
finPour
retourner Somme
```

L'algorithme consiste à faire un parcours du tableau, et à sommer les cellules précédentes à la courante. La complexité est en $\theta(n)$, puisque l'on ne fait qu'un unique parcours du tableau.

3.1.2 MaxSomme

```
Somme = CalculateSommeTab()
Paires = getAllPositionPairs()
Max = NaN
Pour chaque paire p de Paires:
    si p.first = 1
        courant = Somme[p.second]
    sinon
        courant = Somme[p.second] - Somme[p.first]
    finSi
    si courant > max || max = NaN
        max = courant
    finSi
FinPour
```

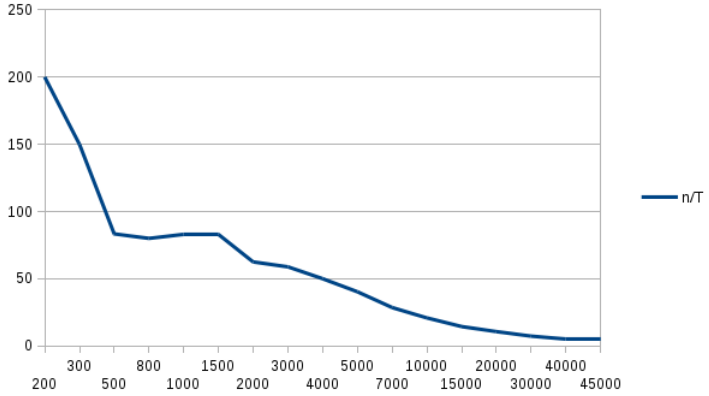
3.2 Complexité Théorique

CalculateSommeTab a une complexité en $\theta(n)$. On retrouve ici la méthode *getAllPositionPairs* dont on avait fixé la complexité à $\theta(n^2)$. Et pour chaque paires obtenues on effectue une opération en $\theta(1)$. On peut donc déterminer une complexité en $\theta(n^2)$ pour cet algorithme. On gagne un ordre de grandeur par rapport à l'algorithme précédent, qui devrait se ressentir lors des résultats d'exécution.

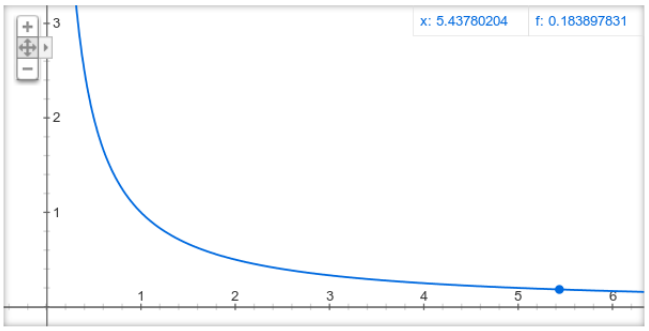
3.3 Résultats d'exécution

On exécute le programme sur des tableaux de taille de plus en plus grande. On s'arrête lorsque le programme est trop lent, ou la mémoire de l'ordinateur ne suffit pas. On obtient ainsi les résultats suivants :

Taille	Temps(millisecondes)
100	0
200	1
300	2
500	6
800	10
1000	12
1500	18
2000	32
3000	51
4000	80
5000	124
7000	245
10000	479
15000	1038
20000	1870
30000	4098
40000	7450
45000	8989
50000	DM



Voici la courbe obtenue en calculant le rapport n/t. On la compare à la courbe de $f(x) = x/x^2$.



Elles ont presque la même allure.

3.4 Analyse

La première chose que l'on puisse dire, c'est ce que cet algorithme est nettement plus efficace que le précédent. Effectivement, on s'arrête parce que la mémoire de l'ordinateur ne suffit plus. Sans ça le temps d'exécution pour un tableau de taille $n = 45000$ reste tout à fait correct. On boit bien que le temps d'exécution augmente de plus en plus vite en fonction de la taille des données, on a bien une correspondance avec la complexité théorique calculée : $\theta(n^2)$.

4 Algorithme 3 : Recursive Algorithm

4.1 Implémentation

4.2 Complexité Théorique

4.3 Résultats d'exécution

4.4 Analyse

5 Algorithme 4 : Quick Algorithm

5.1 Implémentation

Cet algorithme est tout petit en taille, et très efficace. C'est pour ça qu'il porte le nom de *Quick Algorithm*. Son implémentation tient en quelques lignes, n'utilise aucune méthodes complexe :

```
a=0
b=0
i=1
Tant que i <= n faire:
    b = max(b+T[i], 0)
    a = max(a, b)
    i++
Fin TantQue
Retourner a
```

Il est beaucoup plus simple à comprendre que le précédent : On fait un unique parcours du tableau, et tant que la somme des cellules successives est croissante on la conserve dans la variable b, et si cette valeur dépasse le maximum, on écrase le maximum avec cette nouvelle valeur.

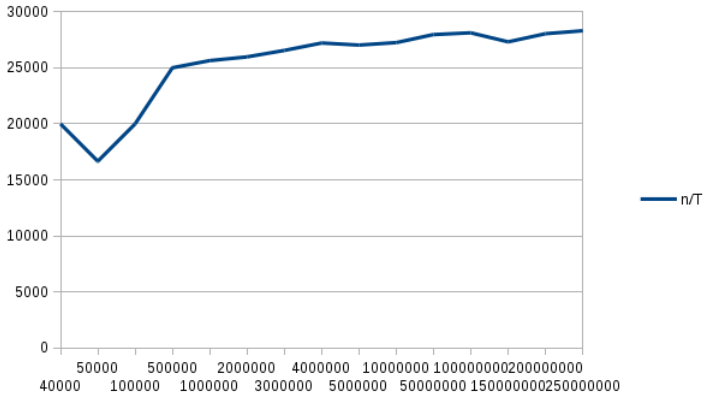
5.2 Complexité Théorique

Cet algorithme n'effectue qu'un seul parcours du tableau, et pour chaque itération, n'effectue que des opérations en temps constant : deux additions, et deux maximums. La complexité obtenue est donc de $\theta(n)$. Le coût en espace aussi est maximum, puisque l'on ne stocke que le tableau lui même, et 3 variables entières (32 bits chacune), le coût en mémoire est donc aussi excellent.

5.3 Résultats d'exécution

Voici les résultats d'exécution de l'algorithme 4. On commence avec des données de taille $n = 10000$, puisque avec des données de taille inférieure, les temps d'exécution restent figés à 0.

n	t(ms)
10000	0
20000	1
40000	2
50000	3
100000	5
500000	20
1000000	39
2000000	77
3000000	113
4000000	147
5000000	185
10000000	367
50000000	1788
100000000	3556
150000000	5491
200000000	7132
250000000	8832
300000000	DM



Voici la courbe obtenue en calculant le rapport n/t. On remarque que au début, les valeurs sont assez variables. C'est dû au peu de précision des données avec le chronomètre. Néanmoins à partir de $n = 5000000$ on remarque que la courbe est presque constante.

5.4 Analyse

La courbe presque constante au dessus appuie la complexité théorique que nous avons calculé. On peut donc affirmer que la complexité de cet algorithme est en $\theta(n)$.