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 :

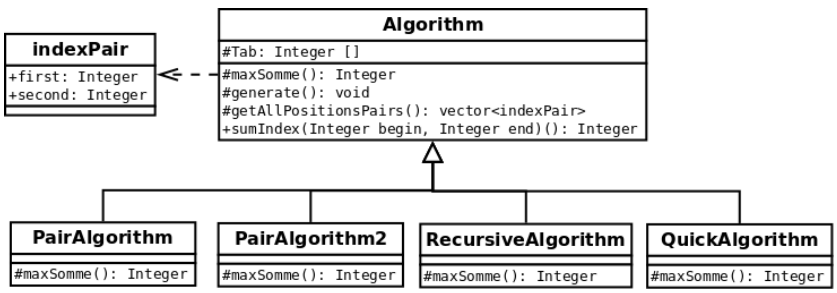

Figure 1 – Diagramme UML du programme

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 $O(n)$.

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 $O(n^2)$ et pour chaque résultat obtenu on effectue une méthode en $O(1)$ On peut donc aisément en déduire que la complexité finale de cet algorithme est en $O(n^2)$

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 (secondes
0,01 // 2	0,01// 3
0,01// 4	0,01// 5
0,01// 6	0,01// 7
0,01// 8	0,01// 9
0,01// 10	0,01// 20
0,01// 40	0,01// 60
0,01// 80	0,01// 100
0,01// 200	0,013// 400
0,091// 600	0,264// 800
0,62// 1000	1,18// 1500
3,981// 2000	9,474// 2500
18,58// 3000	32,221// 3500
51,193// 4000	76,61// 6000
PAR// height	

2.4 Analyse