Problème des sous-séquences maximales

Rapport d'analyse d'algorithmes et de Complexité

par

Paul Christopher AIMÉ

Pr BEKKARI

Semestre 2 – 2024/2025

Génie Informatique

03/16/2025

1. Introduction

1.1. Présentation du problème des sous-séquences maximales

Le problème des sous-séquences maximales consiste à trouver la sous-séquence contiguë d'un tableau d'entiers ayant la plus grande somme possible. Ce problème est très courant en informatique et a des applications variées, notamment en finance, où il permet d'identifier les périodes les plus rentables d'une série de transactions, ou encore en traitement du signal pour détecter les variations significatives dans des données séquentielles. Il est également utilisé dans le domaine de l'intelligence artificielle et de l'optimisation, où la recherche de sous-ensembles optimaux est une problématique clé.

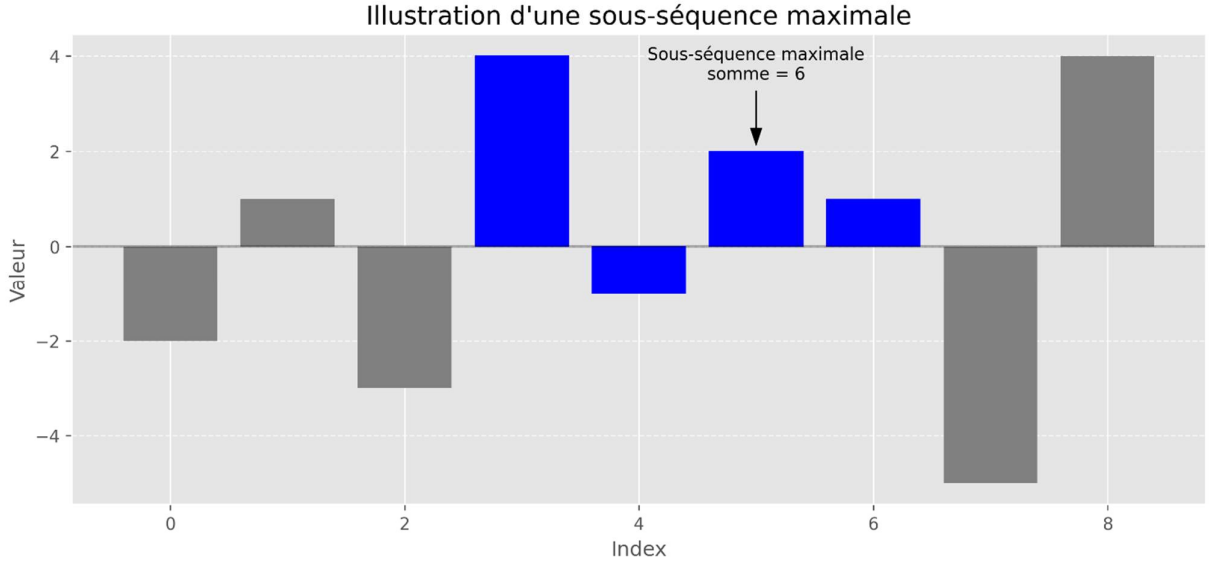

Figure 1. Cette figure montre un tableau d'exemple avec des valeurs positives et négatives. La sous-séquence maximale (en bleu) a une somme de 6 et s'étend des indices 3 à 6.

1.2. Importance du choix de l'algorithme en fonction de la complexité

Le choix de l'algorithme est crucial, car selon la méthode utilisée, la complexité peut varier de $O(n^3)$ à $O(n)$ pour l'algorithme optimal. Une bonne analyse de complexité permet d'optimiser le temps d'exécution, ce qui est particulièrement important lorsque l'on travaille avec de grandes quantités de données. Un algorithme inefficace peut entraîner des temps d'exécution prohibitifs, rendant son utilisation impraticable dans des contextes réels. Par conséquent, il est essentiel d'évaluer les différentes approches en fonction de leurs performances, en tenant compte du compromis entre simplicité d'implémentation et efficacité computationnelle. J'analyserai comment chaque méthode se comporte dans des scénarios où la taille des données peut varier significativement.

1.3. Présentation des quatre approches à analyser

Dans ce rapport, j'analyserai et comparerai quatre algorithmes différents pour résoudre ce problème, chacun présentant ses propres avantages et inconvénients :

- **Algorithme naïf** : Examine toutes les sous-séquences possibles.
- **Algorithme moins naïf** : Optimise le précédent en utilisant une somme cumulative, réduisant ainsi le nombre d'opérations nécessaires et améliorant l'efficacité par rapport à l'approche brute.
- **Algorithme "Diviser pour régner"** : Applique une stratégie récursive en divisant le problème en sous-problèmes plus petits, puis en combinant les résultats intermédiaires pour obtenir la

solution globale. Cette approche réduit considérablement la complexité par rapport aux méthodes plus rudimentaires.

- **Algorithme incrémental** : Utilise une approche dynamique en mettant à jour progressivement la somme maximale tout en parcourant le tableau une seule fois.

À travers cette étude, nous chercherons à identifier l'algorithme le plus performant en fonction de la taille des données et du contexte d'application, en comparant leurs performances théoriques et expérimentales.

2. Analyse théorique des algorithmes

2.1. Algorithme naïf

2.1.1. Fonctionnement

Cet algorithme explore toutes les sous-séquences possibles en testant systématiquement chaque combinaison d'indices, ce qui entraînera sans doute une complexité élevée mais reste une solution intuitive.

2.1.2. Pseudo-code

```
1. ALGORITHME_NAIF(T[1...n])
2.      max_somme ← T[1]
3.      POUR k ALLANT DE 1 À n FAIRE                        //n fois
4.              POUR l ALLANT DE k À n FAIRE                //max n fois
5.                      somme ← 0
6.                      POUR i ALLANT DE k À l FAIRE        //max n fois
7.                              somme ← somme + T[i]
8.                      FIN POUR
9.                      SI somme > max_somme ALORS
10.                             max_somme ← somme
11.                     FIN SI
12.             FIN POUR
13.     FIN POUR
14.     RETOURNER max_somme
```

2.1.3. Analyse de complexité

L'algorithme utilise trois boucles imbriquées. La boucle externe s'exécute n fois, la seconde boucle s'exécute au plus n fois pour chaque itération de la première boucle, et la troisième boucle s'exécute au plus n fois pour chaque itération de la seconde boucle.

On a donc $T(n) = n \times n \times n = n^3$. La complexité temporelle est **$T_1(n) = O(n^3)$.**

2.2. Algorithme moins naïf

2.2.1. Fonctionnement

Cet algorithme optimise l'algorithme précèdent en observant que $S(k,l) = S(k,l-1) + T[l]$, avec :

$$S(k,l) = \sum_{j=k}^{l} T[j]$$

2.2.2. Pseudo-code

```
1. ALGORITHME_MOINS_NAIF(T[1...n])
2.      max_somme ← T[1]
3.      POUR k ALLANT DE 1 À n FAIRE                        // n fois
4.              somme ← 0
```

```
 5.              POUR l ALLANT DE k À n FAIRE                // n fois
 6.                      somme ← somme + T[l]
 7.                      SI somme > max_somme ALORS
 8.                              max_somme ← somme
 9.                      FIN SI
10.              FIN POUR
11.      FIN POUR
12.      RETOURNER max_somme
```

1.2.2. Analyse de complexité

L'algorithme utilise deux boucles imbriquées. La boucle externe s'exécute n fois, et la boucle interne s'exécute au plus n fois pour chaque itération de la première boucle.

On a donc $T(n) = n \times n = n^2$. La complexité temporelle est $\mathbf{T_2(n) = O(n^2)}$.

2.3. Algorithme diviser pour régner

2.3.1. Fonctionnement

Cet algorithme divise la séquence en deux puis calcule une sous séquence de somme maximale de chaque moitié. Par suite, il calculer une sous-séquence de somme maximale qui contient l'élément du milieu. Finalement, il prend le maximum des trois.

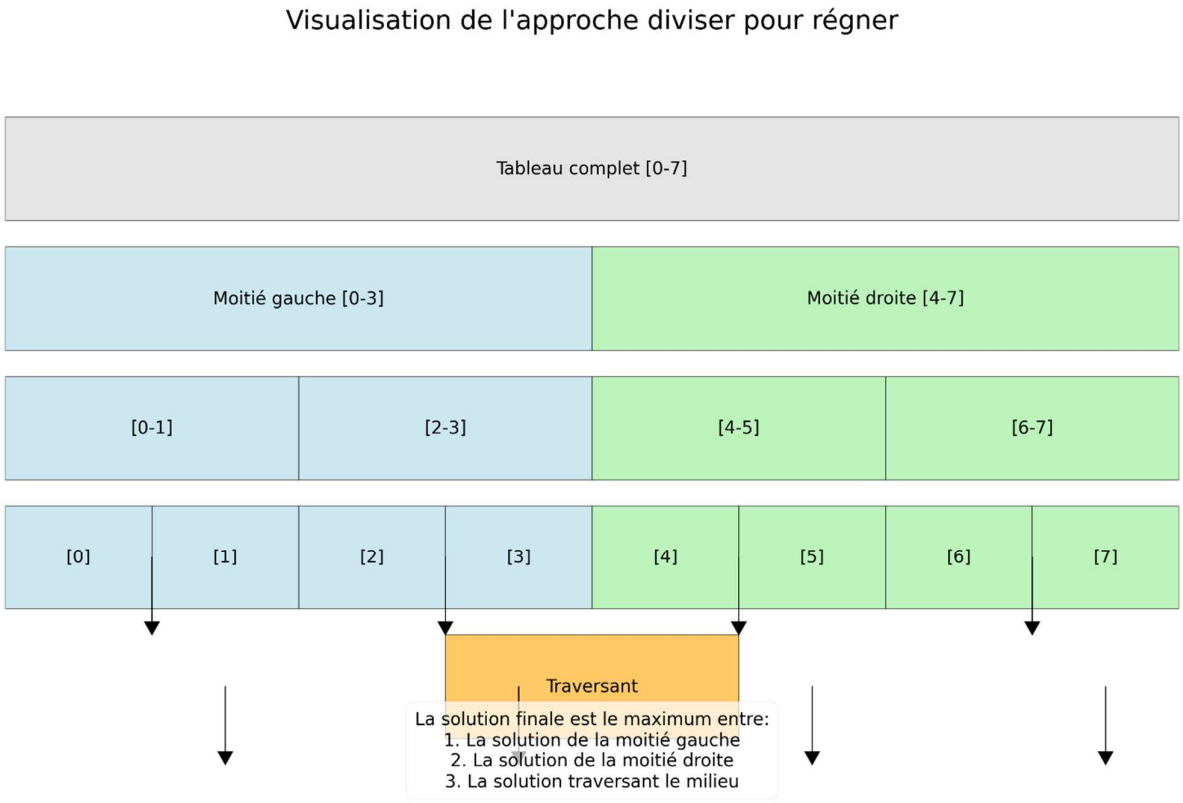

Figure 2. Cette illustration montre comment l'algorithme diviser pour régner découpe récursivement le tableau en sous-tableaux jusqu'à atteindre des éléments individuels, puis recombine les solutions. La partie orange représente la considération spéciale des sous-séquences traversant le milieu.

2.3.2. Pseudo-code

```
1. ALGORITHME_DIVISER_POUR_REGNER(T[1...n])
2.      SI (n == 1) ALORS
3.          RETOURNER T[1]
```

```
    FIN SI

    milieu ← n / 2

    // Cas 1: Sous-séquence maximale dans la partie gauche
    max_gauche ← ALGORITHME_DIVISER_POUR_REGNER(T[1...milieu])

    // Cas 2: Sous-séquence maximale dans la partie droite
    max_droite ← ALGORITHME_DIVISER_POUR_REGNER(T[milieu+1...n])

    // Cas 3: Sous-séquence maximale traversant le milieu
    // Partie gauche maximale se terminant au milieu
    somme ← 0
    max_gauche_traverse ← T[milieu]
    POUR i ALLANT DE milieu À 1 (décroissant) FAIRE
        somme ← somme + T[i]
        SI somme > max_gauche_traverse ALORS
            max gauche traverse ← somme
        FIN SI
    FIN POUR

    // Partie droite maximale commençant juste après le milieu
    somme ← 0
    max_droite_traverse ← 0
    POUR i ALLANT DE milieu+1 À n FAIRE
        somme ← somme + T[i]
        SI somme > max_droite_traverse ALORS
            max_droite_traverse ← somme
        FIN SI
    FIN POUR

    max_traverse ← max_gauche_traverse + max_droite_traverse

    // Retour du maximum des trois cas
    RETOURNER MAX(max_gauche, max_droite, max_traverse)
```

2.3.3. Analyse de complexité

La relation de récurrence pour cet algorithme est : $T(n) = 2T(n/2) + \mathrm{C}(n)$. Car le problème est divisé en deux sous-problèmes de taille n/2 et il y un coût linéaire C(n) pour trouver la sous-séquence maximale traversant le milieu.

D'après le théorème maître, on a : $a = 2; b = 2; d = 1 = log_2(2) \rightarrow \boldsymbol{T_{3(n)} = O(nlog(n))}$

2.4 Algorithme incrémental

2.4.1. Fonctionnement

Cet algorithme suppose que le problème est résolu pour T[1, …, i], puis observe que la solution pour T [1, …, i + 1] est soit la solution précédente, soit la sou séquence de somme maximale qui se termine par T [i + 1].

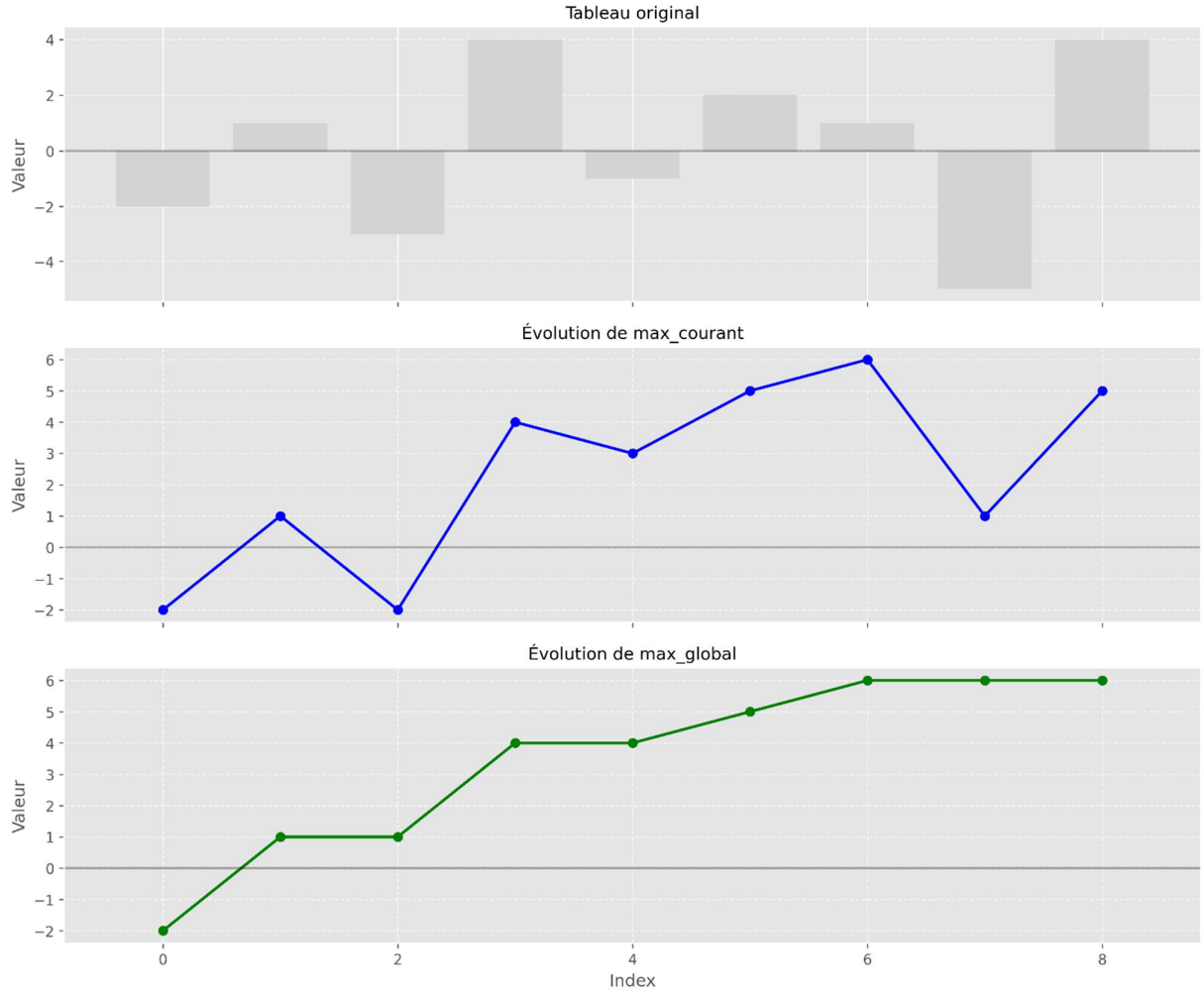

Figure 3 Cette figure montre l'évolution des variables max_courant et max_global lors de l'exécution de l'algorithme incrémental sur un tableau d'exemple. On observe comment max_courant peut diminuer puis augmenter à nouveau, tandis que max_global ne fait que croître.

2.4.2. Pseudo-code

```
 1. ALGORITHME_INCREMENTAL(T[1...n])
 2.       max_global ← T[1]
 3.       max_courant ← T[1]
 4.
 5.       POUR i ALLANT DE 2 À n FAIRE                  // (n - 1) fois
 6.       // Mise à jour max_courant pour inclure T[i] ou recommencer avec T[i]
 7.                max_courant ← MAX(T[i], max_courant + T[i])
 8.
 9.                // Mise à jour de max_global si nécessaire
10.               SI max_courant > max_global ALORS
11.                        max_global ← max_courant
12.               FIN SI
13.      FIN POUR
14.
15.      RETOURNER max_global
```

2.4.3. Analyse de complexité

L'algorithme ne contient qu'une seule boucle qui s'exécute n-1 fois. La complexité temporelle est donc $\mathbf{T(n)_4 = O(n)}$.

3. Implémentation et mesure des performances

3.1. Approche d'implémentation et méthodologie

L'implémentation des algorithmes sera réalisée en **C++**, ce langage étant privilégiés pour sa performance et sa gestion fine de la mémoire. Contrairement aux langages interprétés comme Python, C et C++ permettent un **contrôle direct des ressources**, ce qui est crucial lorsqu'on travaille avec des algorithmes nécessitant une grande efficacité.

L'implémentation ne se limitera pas aux algorithmes eux-mêmes, mais inclura également plusieurs fonctions auxiliaires essentielles pour assurer une bonne évaluation des performances :

- **Génération de données aléatoires :** Une fonction pour générer des données aléatoires, soit des entiers allant de -100 à 100, afin de remplir les différentes tailles de tableaux lors des tests de performances.
- **Mesure du temps d'exécution** : Une fonction dédiée utilisera std::chrono en C++ pour capturer avec précision le temps d'exécution de chaque algorithme.
- **Vérification des résultats** : Une fonction de validation permettra de s'assurer que tous les algorithmes retournent le même résultat pour une même entrée.
- **Fonction main()** : Cette fonction orchestrera l'exécution des différents algorithmes sur des tailles n croissantes et stockera les performances mesurées dans un fichier CSV.

Pour l'implémentation de ces fonctions auxiliaires voir le fichier implementations.cpp dans le repository github.

L'objectif de cette section est donc de détailler l'implémentation technique et de fournir un cadre permettant d'analyser objectivement les résultats obtenus.

3.2. Implémentation des algorithmes

3.2.1. Algorithme naïf

```cpp
// Algorithme naïf - O(n³)
int algorithmeNaif(const vector<int>& T) {
    int n = T.size();
    int max_somme = T[0];

    for (int k = 0; k < n; k++) {
        for (int l = k; l < n; l++) {
            int somme = 0;
            for (int i = k; i <= l; i++) {
                somme += T[i];
            }
            max_somme = max(max_somme, somme);
        }
    }

    return max_somme;
}

```

3.2.2. Algorithme moins naïf

```cpp
// Algorithme moins naïf - O(n²)
int algorithmeMoinsNaif(const vector<int>& T) {
    int n = T.size();
    int max_somme = T[0];

    for (int k = 0; k < n; k++) {
        int somme = 0;
```

```
        for (int l = k; l < n; l++) {
            somme += T[l];
            max_somme = max(max_somme, somme);
        }
    }

    return max_somme;
}

```

3.2.3. Algorithme diviser pour régner

```
// Algorithme diviser pour régner - O(n log n)

int maxSousSequenceTraversant(const vector<int>& T, int debut, int milieu, int fin) {
    // Partie gauche maximale se terminant au milieu
    int somme = 0;
    int max_gauche = T[milieu];

    for (int i = milieu; i >= debut; i--) {
        somme += T[i];
        max_gauche = max(max_gauche, somme);
    }

    // Partie droite maximale commençant juste après le milieu
    somme = 0;
    int max_droite = 0; // Peut être 0 si la partie droite est vide ou négative

    for (int i = milieu + 1; i <= fin; i++) {
        somme += T[i];
        max_droite = max(max_droite, somme);
    }

    return max_gauche + max_droite;
}

int diviserPourRegner(const vector<int>& T, int debut, int fin) {
    if (debut == fin) {
        return T[debut];
    }

    int milieu = (debut + fin) / 2;

    // Cas 1: Sous-séquence maximale dans la partie gauche
    int max_gauche = diviserPourRegner(T, debut, milieu);

    // Cas 2: Sous-séquence maximale dans la partie droite
    int max_droite = diviserPourRegner(T, milieu + 1, fin);

    // Cas 3: Sous-séquence maximale traversant le milieu
    int max_traverse = maxSousSequenceTraversant(T, debut, milieu, fin);

    // Retour du maximum des trois cas
    return max({max_gauche, max_droite, max_traverse});
}

int algorithmeDiviserPourRegner(const vector<int>& T) {
    return diviserPourRegner(T, 0, T.size() - 1);
}
```

3.2.4. Algorithme incrémental

```
// Algorithme incrémental - O(n)
int algorithmeIncremental(const vector<int>& T) {
    int n = T.size();
    int max_global = T[0];
    int max_courant = T[0];

    for (int i = 1; i < n; i++) {
        max_courant = max(T[i], max_courant + T[i]);
        max_global = max(max_global, max_courant);
```

```
10.     }
11.
12.     return max_global;
13. }
14.
```

4. Confrontation de l'analyse théorique et des résultats expérimentaux

4.1. Résultats expérimentaux

4.1.1. Tableau récapitulatif

Après exécution du programme de test sur différentes tailles d'entrées (10, 50, 100, 500, 1000, 2000, 5000, 10000), voici les résultats (en milliseconde) obtenus :

	A	B	C	D	E
1	Taille (n)	Naif(ms)	MoinsNaif(ms)	DiviserPourRegner(ms)	Incremental(ms)
2	10	0.0017	0.0007	0.0011	0.0003
3	50	0.0991	0.0092	0.0078	0.0009
4	100	0.6565	0.0331	0.013	0.0017
5	500	76.3935	0.9146	0.0597	0.0052
6	1000	448.165	2.4811	0.1166	0.0098
7	2000	N/A	9.6442	0.2434	0.0185
8	5000	N/A	64.599	0.6773	0.0456
9	10000	N/A	260.182	1.3166	0.0885

Figure 4. Tableau récapitulatif des performances des quatre algorithmes récupérer dans le fichier CSV.

4.1.2. Comparaison des temps d'exécution linéaire

Ce graphique montre clairement l'explosion du temps d'exécution pour l'algorithme naïf, qui devient rapidement impraticable pour n > 1000. L'algorithme incrémental reste pratiquement indistinguable de l'axe des x en raison de sa grande efficacité.

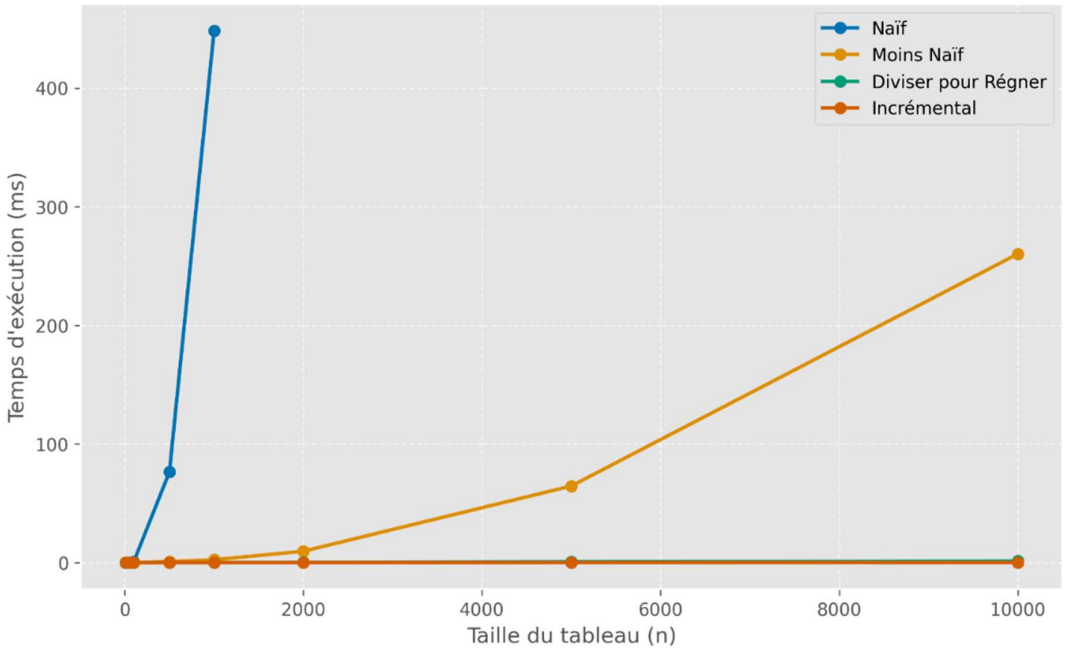

Figure 5. Comparaison des temps d'exécution à l'échelle linéaire

4.1.3. Comparaison des temps d'exécution logarithmique

L'échelle logarithmique permet de mieux visualiser les différences relatives entre les algorithmes. Les pentes des courbes correspondent directement à la complexité des algorithmes : plus la pente est forte, plus la complexité est élevée.

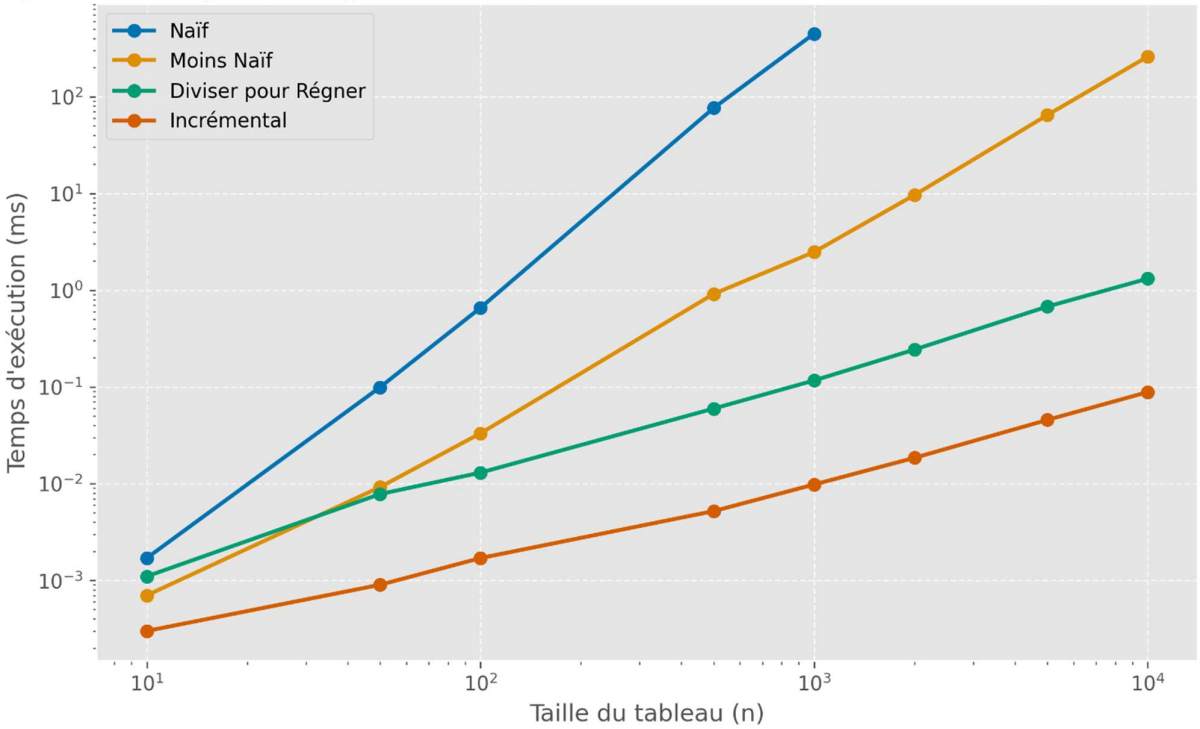

Figure 6. Comparaison des temps d'exécution à l'échelle logarithmique.

4.1.3 Ratios de croissance lorsque la taille de n double

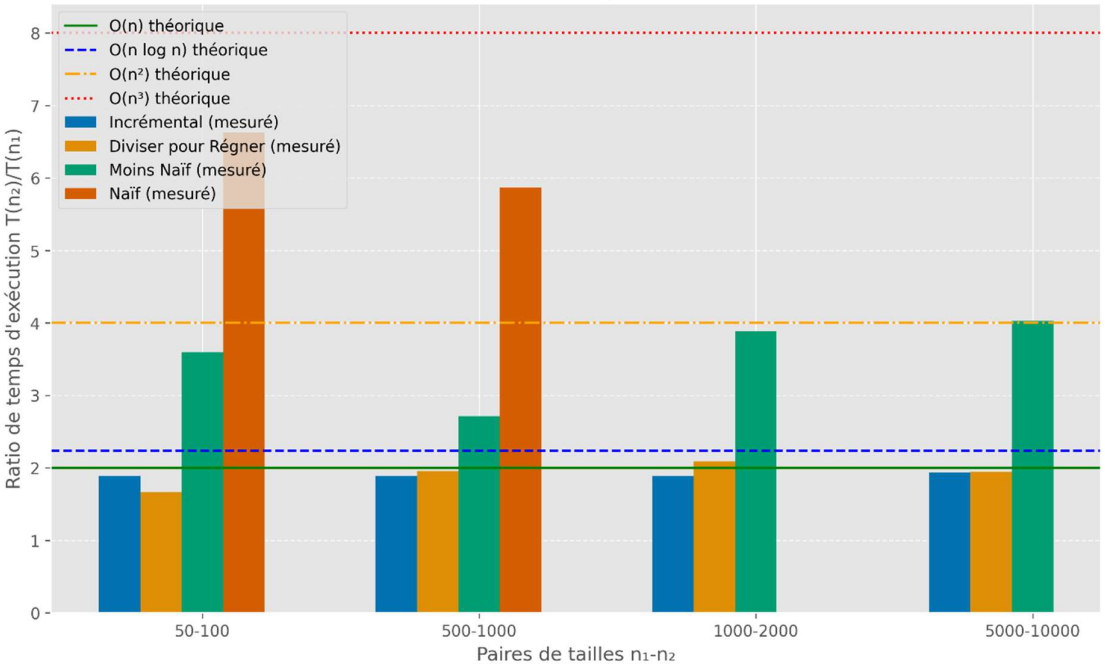

Figure 7. Ratios de croissance lorsque la taille de n double

Ce graphique est particulièrement utile pour confirmer les complexités théoriques. Il montre les ratios de temps d'exécution lorsque n double, comparés aux ratios théoriques :

- Pour O(n), le ratio est 2
- Pour O(nlog(n)), légèrement supérieur à 2
- Pour $O(n^2)$, environ 4
- Pour $O(n^3)$, environ 8

On observe que les mesures expérimentales correspondent bien aux prédictions théoriques.

4.2. Confrontation avec l'analyse théorique

4.2.1. Complexité théorique et pratique des algorithmes

Les mesures empiriques confirment les complexités théoriques calculées précédemment :

1. **Algorithme naïf ($O(n^3)$)** :
   - La complexité théorique est $O(n^3)$ en raison des trois boucles imbriquées
   - Les mesures expérimentales montrent que le temps d'exécution augmente approximativement d'un facteur 8 lorsque n double, ce qui est conforme à une complexité cubique
2. **Algorithme moins naïf ($O(n^2)$)** :
   - La complexité théorique est $O(n^2)$ grâce à l'élimination d'une boucle
   - Les mesures expérimentales montrent que le temps d'exécution augmente approximativement d'un facteur 4 lorsque n double, ce qui est conforme à une complexité quadratique
3. **Algorithme diviser pour régner (O(n log n))**:
   - La complexité théorique est O(n log n) selon le théorème maître
   - Les mesures expérimentales montrent que le temps d'exécution augmente légèrement plus rapidement qu'une croissance linéaire mais significativement moins qu'une croissance quadratique
4. **Algorithme incrémental (O(n))** :
   - La complexité théorique est O(n) car il ne parcourt le tableau qu'une seule fois
   - Les mesures expérimentales confirment une croissance linéaire, avec un temps d'exécution qui double approximativement lorsque n double

4.2.2. Constantes cachées et surcoût d'implémentation

Bien que l'algorithme diviser pour régner ait une complexité théorique de O(nlog(n)), qui est meilleure que $O(n^2)$ de l'algorithme moins naïf, on peut observer que pour des petites tailles ($n < 100$), l'algorithme moins naïf peut parfois être plus rapide. Cela s'explique par :

1. Les constantes cachées dans la notation O
2. Le surcoût des appels récursifs dans l'algorithme diviser pour régner
3. L'efficacité des processeurs modernes pour exécuter des opérations séquentielles (comme dans l'algorithme moins naïf)

Cependant, pour de grandes tailles, l'algorithme diviser pour régner devient nettement plus efficace que l'algorithme moins naïf, conformément à la prédiction théorique.

L'algorithme incrémental reste le plus performant dans tous les cas de test, ce qui confirme sa complexité linéaire O(n).

5. Conclusion

Cette étude comparative des quatre algorithmes pour résoudre le problème de la sous-séquence maximale confirme l'importance d'une bonne conception algorithmique :

1. L'algorithme naïf : $O(n^3)$ devient rapidement impraticable pour des tailles de données modérées
2. L'algorithme moins naïf : $O(n^2)$ améliore considérablement les performances mais reste inefficace pour de grandes tailles
3. L'algorithme diviser pour régner : O(nlog(n) offre une bonne performance pour des tailles moyennes à grandes
4. L'algorithme incrémental : O(n) est de loin le plus efficace et devrait être privilégié pour des applications pratiques

Les résultats expérimentaux valident les complexités théoriques et démontrent qu'un bon choix d'algorithme peut faire la différence entre un programme qui s'exécute en quelques millisecondes et un programme qui prendrait plusieurs heures ou jours pour le même jeu de données.

6. Généralisation aux matrices

6.1. Analyse théorique

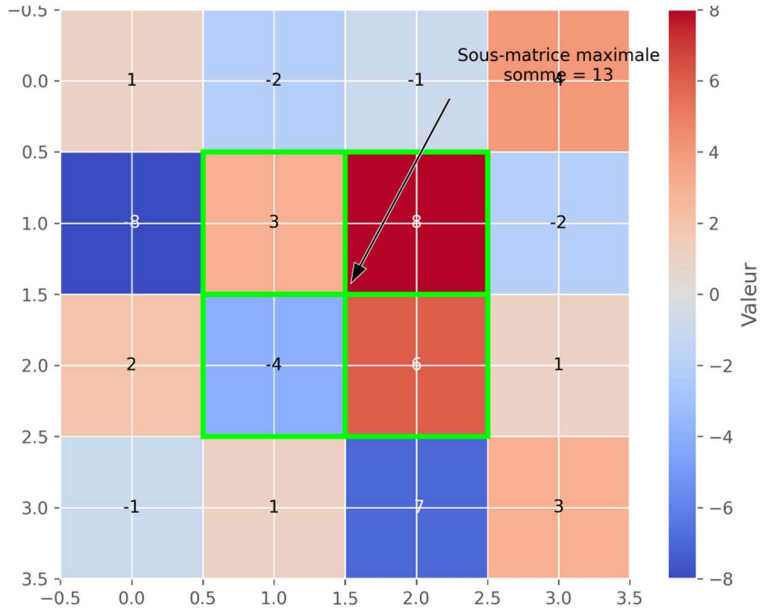

Figure 7. Cette figure montre une matrice d'exemple avec une sous-matrice maximale (contour vert) dont la somme est 13.

La généralisation du problème de la sous-séquence maximale aux matrices consiste à trouver une sous-matrice contiguë dont la somme des éléments est maximale. Voici l'analyse des approches:

1. **Algorithme naïf $O(n^6)$** : Examiner toutes les sous-matrices possibles en utilisant six boucles imbriquées (deux pour les coordonnées du coin supérieur gauche, deux pour les coordonnées du coin inférieur droit, et deux pour calculer la somme des éléments).
2. **Algorithme moins naïf $O(n^4)$** : Utiliser des sommes préfixes pour calculer la somme d'une sous-matrice en temps constant, ce qui élimine les deux boucles de calcul de somme.

3. **Algorithme incrémental O(n³)** : Fixer les colonnes de début et de fin, puis utiliser l'algorithme incrémental pour trouver la sous-séquence maximale dans le tableau formé par la somme des lignes entre ces colonnes.
4. **Algorithme diviser pour régner** : Diviser la matrice en quatre quadrants, résoudre récursivement le problème pour chaque quadrant, puis considérer les sous-matrices qui traversent les limites des quadrants.

6.2. Implémentation des algorithmes

6.2.1. Algortihme naïf

```cpp
// Algorithme naïf pour trouver la sous-matrice maximale - O(n⁶)
int sousMatriceMaximaleNaive(const vector<vector<int>>& matrix) {
    int n = matrix.size();    // Nombre de lignes
    int m = matrix[0].size(); // Nombre de colonnes

    int max_somme = matrix[0][0]; // Initialisation avec le premier élément

    // Cas de toutes les sous-matrices possibles
    for (int debut_ligne = 0; debut_ligne < n; debut_ligne++) {
        for (int debut_col = 0; debut_col < m; debut_col++) {
            for (int fin_ligne = debut_ligne; fin_ligne < n; fin_ligne++) {
                for (int fin_col = debut_col; fin_col < m; fin_col++) {

                    // Calcul de la somme de cette sous-matrice
                    int somme = 0;
                    for (int i = debut_ligne; i <= fin_ligne; i++) {
                        for (int j = debut_col; j <= fin_col; j++) {
                            somme += matrix[i][j];
                        }
                    }

                    // Mise à jour de la somme maximale
                    max_somme = max(max_somme, somme);
                }
            }
        }
    }

    return max_somme;
}
```

6.2.2. Algorithme moins naïf

```cpp
// Algorithme moins naïf pour trouver la sous-matrice maximale - O(n⁴)
int sousMatriceMaximaleMoinsNaive(const vector<vector<int>>& matrix) {
    int n = matrix.size();    // Nombre de lignes
    int m = matrix[0].size(); // Nombre de colonnes

    // Prétraitement: calcul des sommes préfixes
    vector<vector<int>> prefixSums(n + 1, vector<int>(m + 1, 0));
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            prefixSums[i][j] = matrix[i-1][j-1] + prefixSums[i-1][j] + prefixSums[i][j-1] -
prefixSums[i-1][j-1];
        }
    }

    int max_somme = matrix[0][0]; // Initialisation avec le premier élément

    // Cas de toutes les sous-matrices possibles
    for (int debut_ligne = 0; debut_ligne < n; debut_ligne++) {
        for (int debut_col = 0; debut_col < m; debut_col++) {
            for (int fin_ligne = debut_ligne; fin_ligne < n; fin_ligne++) {
                for (int fin_col = debut_col; fin_col < m; fin_col++) {
```

```
21.
22.                     // Calcul de la somme de cette sous-matrice en O(1) en utilisant les
sommes préfixes
23.                     int somme = prefixSums[fin_ligne+1][fin_col+1] -
prefixSums[fin_ligne+1][debut_col]
24.                                 - prefixSums[debut_ligne][fin_col+1] +
prefixSums[debut_ligne][debut_col];
25.
26.                     // Mise à jour de la somme maximale
27.                     max_somme = max(max_somme, somme);
28.                 }
29.             }
30.         }
31.     }
32.
33.     return max_somme;
34. }
```

6.2.3. Algorithme incrémentale

```
 1. // Algorithme de Kadane pour trouver la sous-séquence maximale dans un tableau 1D
 2. int kadane(const vector<int>& array) {
 3.     int max_global = array[0];
 4.     int max_courant = array[0];
 5.
 6.     for (int i = 1; i < array.size(); i++) {
 7.         max_courant = max(array[i], max_courant + array[i]);
 8.         max_global = max(max_global, max_courant);
 9.     }
10.
11.     return max_global;
12. }
13.
14. // Algorithme pour trouver la sous-matrice maximale - O(n³)
15. int sousMatiiceMaximale(const vector<vector<int>>& matrix) {
16.     int n = matrix.size();    // Nombre de lignes
17.     int m = matrix[0].size(); // Nombre de colonnes
18.
19.     int max_somme = matrix[0][0]; // Initialisation avec le premier élément
20.
21.     // Essayer toutes les sous-matrices possibles
22.     for (int debut_col = 0; debut_col < m; debut_col++) {
23.
24.         // Tableau temporaire pour stocker la somme des éléments de chaque ligne
25.         vector<int> temp(n, 0);
26.
27.         for (int fin_col = debut_col; fin_col < m; fin_col++) {
28.
29.             // Ajout des valeurs de la colonne fin col au tableau temporaire
30.             for (int i = 0; i < n; i++) {
31.                 temp[i] += matrix[i][fin_col];
32.             }
33.
34.             // Application de l'algorithme de Kadane pour trouver la somme maximale dans
temp
35.             int max_kadane = kadane(temp);
36.
37.             // Mise à jour de la somme maximale
38.             max_somme = max(max_somme, max_kadane);
39.         }
40.     }
41.
42.     return max_somme;
43. }
```

6.3. Application à une petite matrice

```cpp
int main() {
    // Exemple d'utilisation avec une petite matrice
    vector<vector<int>> matrix = {
        {1, -2, -1, 4},
        {-8, 3, 8, -2},
        {2, -4, 6, 1},
        {-1, 1, -7, 3}
    };

    cout << "Sous-matrice maximale (Kadane 2D): " << sousMatiiceMaximale(matrix) << endl;
    cout << "Sous-matrice maximale (Naïve): " << sousMatriceMaximaleNaive(matrix) << endl;
    cout << "Sous-matrice maximale (Moins Naïve): " << sousMatriceMaximaleMoinsNaive(matrix) << endl;

    return 0;
}
```

```
C:\Users\Lenovo T470\OneDrive\Documents\School\ENSA-Marrakech\GI3\S2\Analyse et Complexité d'Algorithmes et Compilation\Devoir>
cd "c:\Users\Lenovo T470\OneDrive\Documents\School\ENSA-Marrakech\GI3\S2\Analyse et Complexité d'Algorithmes et Compilation\Dev
oir\" && g++ implementations_matrice.cpp -o implementations_matrice && "c:\Users\Lenovo T470\OneDrive\Documents\School\ENSA-Mar
rakech\GI3\S2\Analyse et Complexité d'Algorithmes et Compilation\Devoir\"implementations_matrice
Sous-matrice maximale (Incrementale): 16
Sous-matrice maximale (Naive): 16
Sous-matrice maximale (Moins Naive): 16
```