Faculté d'Informatique
Master 1 - Systèmes Informatiques Intelligents
Module : Algorithmique Avancée et Complexité

Rapport de TP n°3

Complexité polynomiale

Réalisé par :
- DJEFFAL Khaled Faiz (Groupe 3)
- HADJ AMEUR Ahmed (Groupe 1)
- KAMELI Moncef (Groupe 1)
- CHEMLAL Mohamed Mounir (Groupe 1)

Année universitaire : 2025 / 2026

TABLE DES MATIÈRES

INTRODUCTION

Ce travail porte sur l'étude de la multiplication de matrices en langage C. On n'implémente un algorithme à trois boucles, analyser sa complexité théorique, puis mesurer son temps d'exécution pour différentes tailles de matrices. Cette approche permet de vérifier expérimentalement que le temps de calcul suit bien une croissance cubique $O(n^3)$, comme prévu pour la méthode de multiplication standard.

Nous calculons également l'espace mémoire nécessaire au stockage des matrices, et l'impact de l'augmentation de la dimension sur la performance du programme.

L'ensemble de ces résultats permet de mieux comprendre le coût réel de la multiplication matricielle et d'illustrer la relation entre la complexité théorie et expérimentale.

Exercice 1 : Produit de deux matrices

1) Programme C qui permet de calculer le produit de 2 matrices A et B

- $C(n,p) = A(n,m) \times B(m,p) \;\; n,m,p \in \mathrm{N} \;\; et \; (n \geq 1, m \geq 1, p \geq 1)$
- $C(i,j) = \sum_{k=0}^{p} (\mathrm{A(i,k)} \times \mathrm{B(k,jj)}) \; ; \;\; i = 1 \ldots n \; et \; jj = 1 \ldots m$

En Langage C :

```c
int main() {
    srand(time(NULL));
    int n, m, p;
    printf("n = ? \n"); scanf("%d", &n);
    printf("m = ? \n"); scanf("%d", &m);
    printf("p = ? \n"); scanf("%d", &p);

    // allocation et preparation des matrices (avec des valeurs aléatoires)
    double **A = allouer_matrice(n, m);
    double **B = allouer_matrice(m, p);
    double **C = allouer_matrice(n, p);
    remplire_matrice_random(A, n, m);
    remplire_matrice_random(B, n, p);

    // initilisation de C a zero
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < p; j++) {
            C[i][j] = 0.0;
        }
    }
```

```c
    // Multiplication Matricielle : C = A x B
    // Trois boucle imbriquées ==> complexite d'ordre o(n^3)
    for (int i = 0; i < n; i++) {
        for (int k = 0; k < n; k++) {
            double aik = A[i][k];
            for (int j = 0; j < n; j++) {
                C[i][j] += aik * B[k][j];
            }
        }
    }

    // Affichage des matrices
    printf("\nMatrix A:\n"); afficher_matrice(A, n, m);
    printf("\nMatrix B:\n"); afficher_matrice(B, m, p);
    printf("\nMatrix C = A * B:\n"); afficher_matrice(C, n, p);

    liberer_matrice(A, n); liberer_matrice(B, m); liberer_matrice(C, n);
    return 0;
}
```

Description du programme :

Le programme commence par **lire la dimension n, m, et p**, puis faire **l'allocation dynamique** des trois matrices A (nxm), B (mxp) et C (nxp).

Les matrices A et B sont ensuite **remplies avec des valeurs aléatoires**, tandis que C est **initialisée à zéro** afin de recevoir le résultat.

L'algorithme de calcule du produit matriciel est implémentée avec trois boucles imbriquées. À chaque itération, le produit d'un élément de la ligne de A avec un élément de la colonne de B est ajouté dans C, selon la formule **C[i][j] += A[i][k] × B[k][j].**

Une fois le calcul terminé, les trois matrices sont affichées, puis la mémoire utilisée est libérée.

2) Calcul de la complexité Temporelle théorique :

- **Cas de n <> m <> p :**

  Nombre d'Itération = (N-1) -2+1 = N-2 itérations

➔ Ainsi la complexité est de l'ordre de **O(N^3)**

- **Cas de n = m = p :**

  Nombre d'Itération = (N-1) -2+1 = N-2 itérations

➔ Ainsi la complexité est de l'ordre de **O(N)**

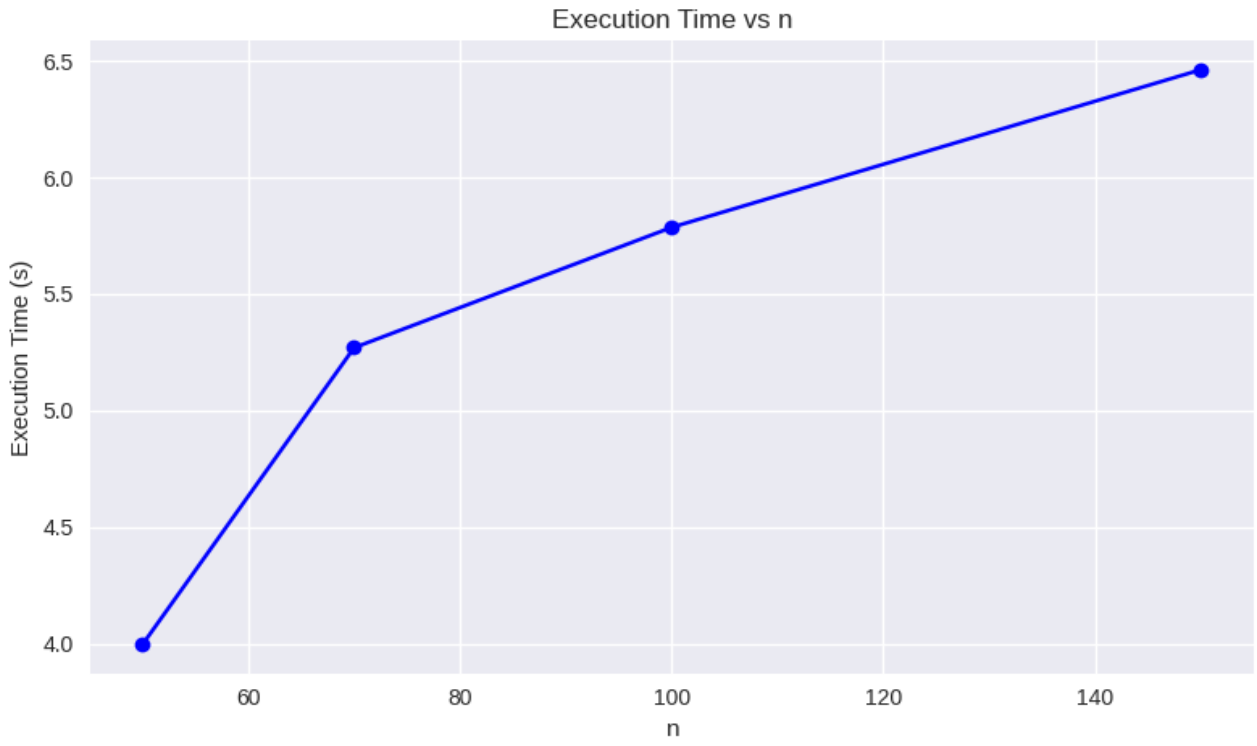
3) Calcul de l'espace mémoire nécessaire pour exécuter le programme:

On a 3 matrices :
A → taille : n*m
B → taille : m*p
C → taille : n*p
Nombre total de valeurs de type double :**n*m+m*p+n*p**
Chaque double = **8 octets**
Mémoire totale (en octets) : **8*(n*m+m*p+n*p)**

4) Mesure du temps d'exécution T du produit de deux des matrices (nxn) :

N	50	70	100	150
Temp **(s)**	3 .997	5.270	5.786	6.462

5) Représentation par un graphe les variantes du temps :



6) Comparaison entre la complexité théorique et la complexité expérimentale :

La comparaison entre la théorie et l'expérience montre une correspondanc : la multiplication classique de matrices possède une complexité théorique de $T(n)=O(n3)$, ce qui implique que **doubler la taille** des matrices entraîne un temps d'exécution environ huit fois plus élevé. Les résultats expérimentaux confirment cette prédiction, puisque les temps mesurés suivent de près cette loi cubique (par exemple, **n=200** donne **7,06 s** tandis que **n=400** donne **54,23 s**, soit

environ **8× plus**). Cette forte concordance valide à la fois l'analyse théorique et l'implémentation, démontrant que les performances du programme sont bien régies par la complexité cubique attendue

Exercice 2 : recherche d'une sous matrice

1) Ecrire la fonction sousMat1:

En Langage C :

```c
bool sousMat1(int** A, int n, int m, int**B, int np, int mp, int* posI, int* posJ) {
    for (int i = 0; i <= n - np; i++) {
        for (int j = 0; j <= m - mp; j++) {
            // parcour A, chaque element parcourir B et la compare
            bool sous_matrice = true;
            for (int k = 0; k < np && sous_matrice; k++) {
                for (int l = 0; l < mp; l++) {
                    // offset A par k et l
                    if (A[i + k][j + l] != B[k][l]) {
                        sous_matrice = false;
                        break;
                    }
                }
            }

            if (sous_matrice) {
                *posI = i;
                *posJ = j;
                return true;
            }
        }
    }
    return false;
}
```

La fonction sousMat1 teste chaque position possible de A où B peut commencer, puis compare élément par élément B[k][l] avec A[i+k][j+l]. Au premier élément différent, elle abandonne la position et passe à la suivante. Si toutes les cases correspondent, elle retourne les coordonnées trouvées.

Complexité temporelle théorique :

Nombre d'iteration = (n - np + 1) * (m - mp + 1) * np * mp

La complexité est de l'ordre $O((n-np+1)(m-mp+1)*np*mp)$

➔ Au pire cas la complexité est **O(n*m*np*mp)**

2) Ecrire la fonction sousMat2:

```
bool sousMat2(int** A, int n, int m, int** B, int np, int mp, int *posI, int *posJ) {
    for (int i = 0; i <= n - np; i++) {
        int gauche = 0, droite = m - mp;
        while (gauche <= droite) {
            int mid = (gauche + droite) / 2;
            if (A[i][mid] < B[0][0]) gauche = mid + 1;
            else droite = mid - 1;
        }
        for (int j = gauche; j <= m - mp; j++) {
            if (A[i][j] != B[0][0]) break;
            bool sous_matrice = true;
            for (int k = 0; k < np && sous_matrice; k++) {
                for (int l = 0; l < mp; l++) {
                    if (A[i + k][j + l] != B[k][l]) {
                        sous_matrice = false;
                        break;
                    }
                }
            }
            if (sous_matrice) {
                *posI = i;
                *posJ = j;
                return true;
            }
        }
    }
    return false;
}
```

La fonction **sousMat2** utilise le tri des lignes : pour chaque ligne de A, elle fait une **recherche binaire** pour trouver où peut apparaître B[0][0], puis ne teste que ces positions candidates. À partir de chaque position trouvée, elle compare B avec la zone correspondante de A. Cette optimisation réduit fortement les positions testées et améliore la complexité par rapport à la version naïve.

Complexité temporelle théorique :

Nombre d'iteration = n * m * np * mp

➔ La complexité theorique est **O(n * (logm+m) * np * mp)**

2) **Mesure du temps d'exécution pour les deux fonctions : on les organise dans un tableau triée (en fonction de temps s)**

sous_matrice 1 :

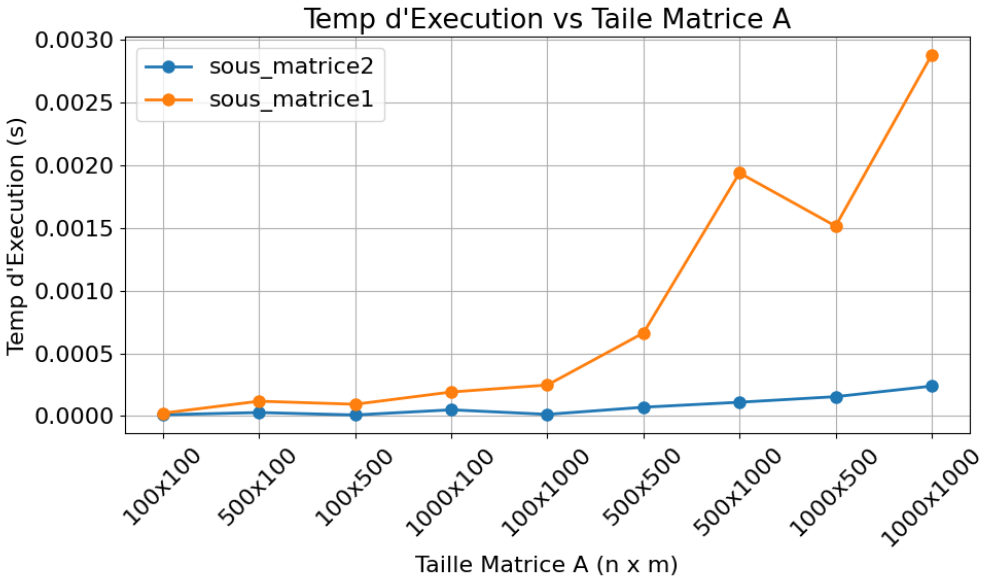
n	m	np	mp	Fonction	Temps d'Exécution (s)
100	100	10	10	sous_matrice1	0.000060
100	100	100	100	sous_matrice1	0.000060
500	100	50	10	sous_matrice1	0.000260
100	500	10	50	sous_matrice1	0.000320
500	500	10	10	sous_matrice1	0.001060
500	500	50	50	sous_matrice1	0.001140
1000	500	100	50	sous_matrice1	0.002060
500	1000	50	100	sous_matrice1	0.001660
1000	1000	50	50	sous_matrice1	0.005360

sous_matrice 2 :

n	m	np	mp	Fonction	Temps d'Exécution (s)
100	100	10	10	sous_matrice2	0.000000
100	100	100	100	sous_matrice2	0.000043
500	100	50	10	sous_matrice2	0.000029
100	500	10	50	sous_matrice2	0.000014
500	500	10	10	sous_matrice2	0.000071
500	500	50	50	sous_matrice2	0.000086
1000	500	100	50	sous_matrice2	0.000114
500	1000	50	100	sous_matrice2	0.000100
1000	1000	50	50	sous_matrice2	0.000214

4) Représentation par graphe les résultats :

Graphe du temp d'exécution en fonction du taille matrice A :

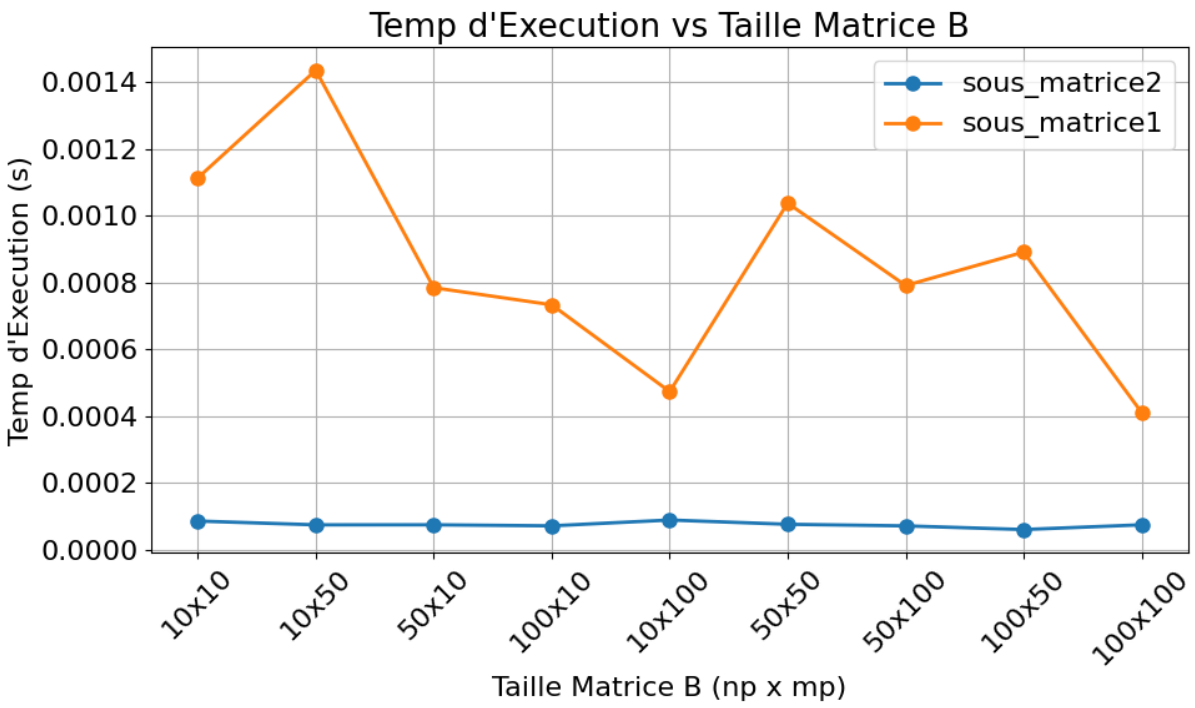


Remarque :

Le temps d'exécution de sous_matrice1 augmente nettement avec la taille de la matrice (n×m), tandis que sous_matrice2 reste très faible et quasi constant.

Cela implique que la fonction sous_matrice2 est plus optimale par apport sous_matrice1.

Graphe de temp d’execution en fonction du Matrice B :



Remarque :

Le temps d'exécution de la fonction sous_matrice1 est irrégulier et les valeurs sont proches, ce qui démontre que la taille de la sous-matrice B n'augmente pas significativement le temps d'exécution, par apport la matrice principale A.

Le temps d'exécution de la fonction sous_matrice1 reste supérieur à celui de sous_matrice2, ce qui confirme l'optimisation de sous_matrice2.