Université de Sciences et de la Technologie Houari Boumediene

Faculté Informatique

Rapport Projet 3

TP 3 Algorithmique et complexité

Le produit matriciel et la recherche d'une sous-matrice

Mahmoud Bacha rabah sami G_02

Saadallah Abderahim G_03

M1 RSD 2023/2024

Environnement de travail :

- Langage : C , gcc 6.3.0
- CPU: i5 8365U 3.9 GHZ max frequency
- Ram: 8 GB
- OS: windows 11
- Logical: VScode

Tous les matrices crees sont de type dynamique en utilisant des double pointeur int **Matrice

Le produit matriciel :

Implémentation de l'algorithme :

```
int** Produit_matrice_carre(int** mat1, int** mat2, int N) {

    int** result = (int**)malloc(N * sizeof(int*));
                                    // creer les pointeur sur les ligne

    for (int i = 0; i < N; i++) {
        result[i] = (int*)malloc(N * sizeof(int));
    }                             //creer sur les column



    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {

            result[i][j] = 0;
            for (int k = 0; k < N; k++) {              // le produit
                result[i][j] += mat1[i][k] * mat2[k][j];
            }
        }
    }

    return result;
}
```

Explication:

Matrice result = matrice A * matrice B

- L'algorithme prend comme argument les deux matrices a multiplié , dans notre cas les matrices sont carré donc on aura pas besoin de vérifier la condition sur le nombre de ligne et colonne
- On commence par créer la matrice carrée résultat en utilisant une structure dynamique.
- Pour chaque case de la matrice résultat , on l'initialise a 0 puis on faire un parcours ligne de la matrice A et un parcours de colonne de la matrice B , en sommant les deux valeurs.

Complexité :

- Dans le cas général :
  Resultat (N,M) = A (N,P) * B (P,M)
  On a 3 boucles imbriqués a parcourir :
  1ere boucle : N itération
  2eme boucle : M itération
  3eme boucle : P itération

  Donc le nombre d'itération est : $N * M * P$

  D'où la complexité est cubique $\boldsymbol{O(N^3)}$

- Dans le cas où les deux matrices sont carré , le nombre d'itération est N^3

Espace mémoire :

Espace mémoire nécessaire est équivalent a la somme de taille des Matrice A et B et Résultat de produit

Resultat (N,M) = A (N,P) * B (P,M)

- Dans le cas général :
  matrice A : $N * P\ cases\ memoires$
  matrice B : $P * M\ cases\ memoires$
  matrice résultat : $N * M\ cases\ memoire$

  chaque case est de type int , taille de int dans le langage c'est 4 octets

  donc espace mémoire = $4(N * P + P * M + N * M)$
- Dans le cas de matrice carrés de taille N :

  espace mémoire = $12N^2$

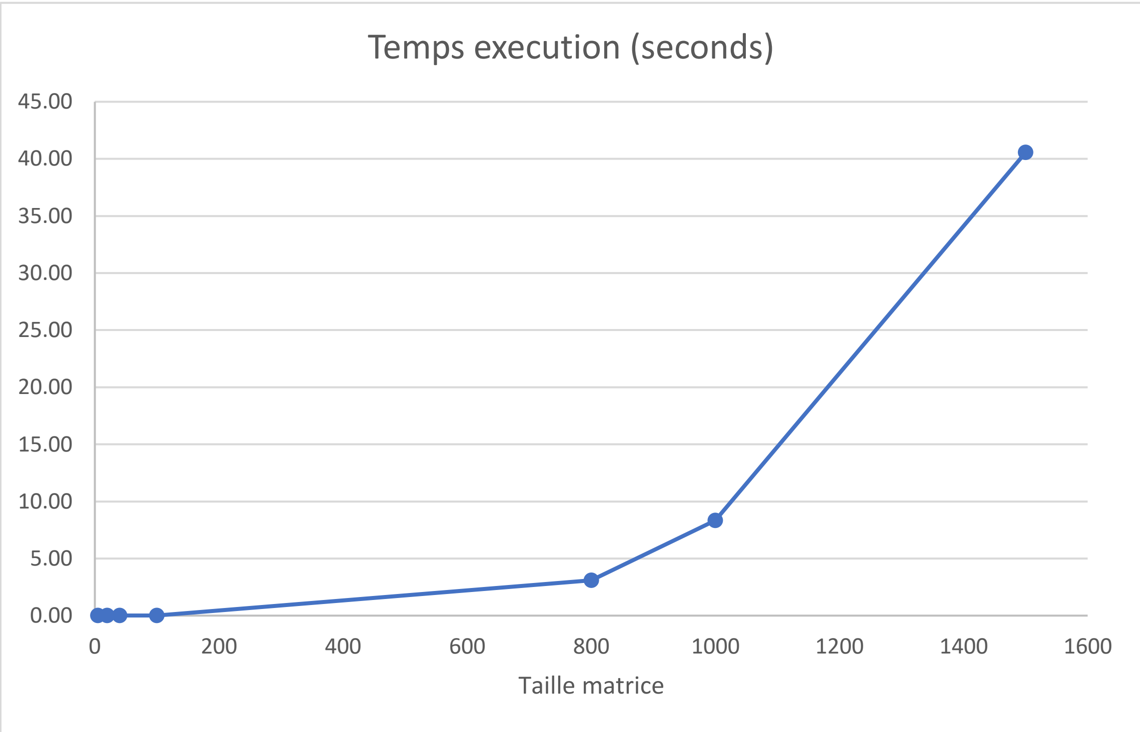
Etude expérimentale :

On utilise une fonction afin de créer des matrices de taille N^2 ayant des valeurs random.

L'échantillon de taille de tableaux est [5 ,20 ,40 ,100 ,800 ,1000, 1500]

On calcul le temps avant et après l'exécution de la fonction du produit afin de calculer le temps d'exécution pour chaque taille

Taille matrices	Temps execution
5	0.00000
20	0.00000
40	0.00100
100	0.00300
800	3.09800
1000	8.33800
1500	40.55500



Remarque :

- Selon le graph , pour des tailles petites , le temps de calculs est très petite par rapport a la puissance de la machine
- La croissance de ce graphe ressemble a la croissance du graphe de la fonction $x \longmapsto x^3$ donc les résultat des calculs sont compatibles avec les résultats théorique

La recherche d'une sous-matrice :

Implémentation de l'algorithme 1 :

```
int SousMat1( int N, int M, int A, int B , int **grandmat, int **petitmat) {
    int i, j, x, y;

    if(A > N || B > M){
        printf("La matrice B est plus grande que A \n");
        return 0;
    }


    for (i = 0; i <= N - A; i++) {

        for (j = 0; j <= M - B; j++) {

            int trouve = 1; // si on trouve une valeure different on met trouve a 
0 et on sort de boucle

            for (x = 0; x < A; x++) {

                for (y = 0; y < B; y++) {

                    if (grandmat[i + x][j + y] != petitmat[x][y]) {
                        trouve = 0; // Submatrix doesn't match at this position
                        break;
                    }
                }
                if (!trouve) {
                    break;
                }
            }
            if (trouve) {
                printf("Soumatrice trouvee a ligne =  %d , colonne =  %d\n", i, 
j);

                return 1;
            }
        }
    }
    printf("Soumatrice n'existe pas \n");
    return 0;
}
```

Explication sousmat1 :

- L'algorithme recherche l'existence de **petitmat(A,B)** dans **grandmat(N,M)**
- On commence d'abord par le test sur la taille des deux matrices

- Comme on voit il y'a 4 boucles imbriqué
  Les deux premières **boucles extérieures** sont pour parcourir les cases de la grande matrice
  on utilise les indice **i** et **j** pour le parcours de lignes et colonne respectivement

  <u>Remarque :</u>
  On arrête le parcours a N-A et M-B car sinon la sou matrice recherché va dépasser l'extrémité de la grande matrice

N-A N

0	5	4	2	3	5	3
5	5	3	3	4	2	1
1	1	0	3	2	5	0
2	4	0	5	5	4	4
3	4	1	3	4	0	4
0	2	3	5	2	1	3
5	5	2	0	1	3	5

A

1	0	3
4	0	5
4	1	3

- Pour chaque case de la grande matrice :
  - on va d'abord initialiser une valeur Boolean a 1
    ensuite renter dans deux boucle pour parcourir la petite matrice et tester si les deux cases sont diffèrent pour mettre le Boolean a 0 et passer a la case suivante en utilisant le **break**.
  - si les cases sont égaux alors on continue le parcours de la petite matrice
  - si le parcours de la petite matrice est terminé avec le Boolean = 1 alors tous les cases sont égaux et donc la matrice est trouvé.
  - si le parcours de la grande matrice est terminé avec Boolean = 0 alors la sous matrices n'est pas trouvé

Le pire cas :

Dans cet algorithme **, le pire cas est lorsque la sousmatrice recherché est la dernière sommatrice de la grande matrice** , car si la matrice n'existe pas , on fera moins de parcours de la sousmatrice a cause de l'utilisation de **break** !

Exemple de pire cas :

0	5	4	2	3	5
5	5	3	3	4	2
1	1	0	3	2	5
2	4	0	5	5	4
3	4	1	3	4	0
0	2	3	5	2	1

5	5	4
3	4	0
5	2	1

En testant sur une matrice A de taille (2500 ,2500) , on obtient le resultat suivant qui valide le choix du pire cas

```
temps quand soumat existe pas : 0.056000 Souma
temps quand soumat est la derniere  : 0.112000
```

Complexité soumat1 :

1ere boucle : N-A iteration
2eme boucle : M-B iterations

3eme boucle : A iterations
4eme boucle : B iterations

nombre iterations totale : $(\mathrm{N}-\mathrm{A})(\mathrm{M}-\mathrm{B})(\mathrm{A})(\mathrm{B})$

Mais N >> A et M >> B donc nombre itération = N * M

complexité : $\boldsymbol{O}(\ \boldsymbol{N^2}\ )$

Dans notre cas , les deux matrices sont carrés
le nombre d'itérations est : $(N-A)^2 * A^2 \approx \ N^2$

Implemenation de l'algorithme 2 :

```
int SousMat2(int N, int M, int A, int B, int **grandmat, int **petitmat) {

    if (A > N || B > M) {
        printf("La matrice B est plus grande que A \n");
        return 0;
    }

    int i, j, x, y;
    for (i = 0; i <= N - A; i++) {

        for (j = 0; j <= M - B; j++) {

            int trouve = 1;

            for (x = 0; x < A; x++) {

                // tester seulement la premiere case de chaque ligne
                // si vrai , on parcours la petite matrice , sinon on passe a la
prochaine ligne

                if (grandmat[i + x][j] != petitmat[x][0]) {
                    trouve = 0;
                    break;
                }

                //  parcourir la petite matrice si la premiere case est trouvé

                for (y = 1; y < B; y++) {
                    if (grandmat[i + x][j + y] != petitmat[x][y]) {
                        trouve = 0;
                        break;
                    }
                }

                if (!trouve) {
                    break;
                }
            }

            if (trouve) {
                printf("Soumatrice trouvee a ligne =  %d , colonne =  %d\n", i, j);
                return 1;
            }
        }
    }

    printf("Soumatrice n'existe pas \n");
    return 0;
}
```

Explication sousmat2 :

L'amélioration par rapport au premier algorithme est que :

Dans l'algorithme 1 : On va tester pour chaque case la petite-matrice si elle existe dans la grande matrice
Dans l'algorithme 2 : On ne teste que la première valeur de chaque ligne de la petite matrice dans la grande matrice
si on trouve , alors on rentrer dans la dernière boucle pour parcourir toute la petite matrice
sinon on pace a la ligne suivante

Complexité sousmat2 :

nombre iterations totale : $(\mathrm{N}-\mathrm{A})(\mathrm{M}-\mathrm{B})(\mathrm{A})(\mathrm{B})$

complexité : $\boldsymbol{O}(\boldsymbol{N}^{2})$

Dans notre cas , les deux matrices sont carrés
le nombre d'itérations est : $(N-A)^2 * A^2 \approx N^2$

Etude expérimental :

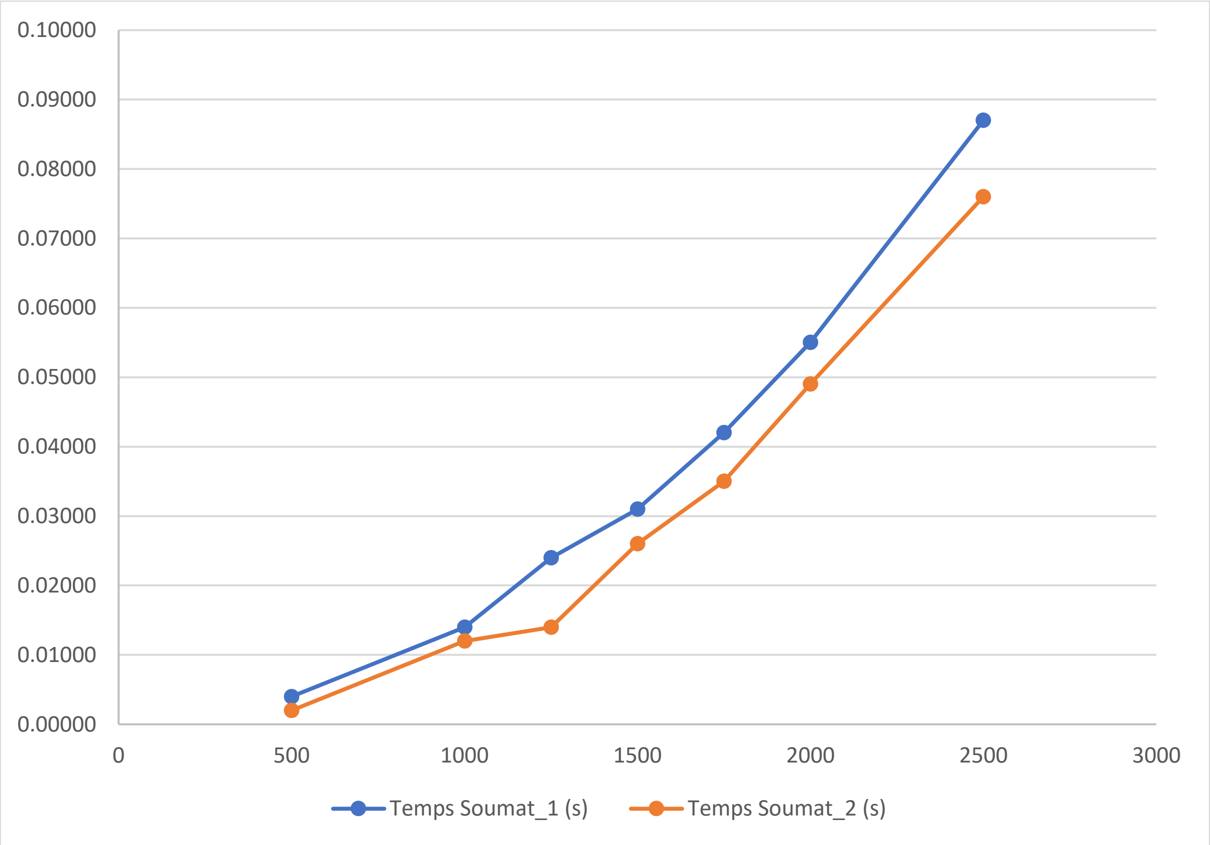
N est la taille de tableau , elle prend les valeurs : 500 , 1000 , 1250 , 1500 , 1750 , 2000 , 2500

On utilise une fonction pour créer la grande matrice de taille (N,N) ayant des valeurs random

On utilise une fonction qui affecte a la petite matrice (100,100) la dernière sousmatrice de cette taille de la grande matrice

Pour chaque valeur de N , on compare le temps d'exécution des deux fonctions

Taille matrice	Temps Soumat_1 (s)	Temps Soumat_2 (s)
500	0.00400	0.00200
1000	0.01400	0.01200
1250	0.02400	0.01400
1500	0.03100	0.02600
1750	0.04200	0.03500
2000	0.05500	0.04900
2500	0.08700	0.07600



Remarque :

Le graph montre bien que l'algorithme 2 est une optimisation de l'algorithme 1 , en diminuant le nombre d'iteration que l'algorithme fait dans la recherche des elements de la petite matrice

Les deux algorithmes sont d'ordre $\boldsymbol{O}(\boldsymbol{N}^{\boldsymbol{2}})$, il converge de la même façon mais avec une petite optimisation dans le nombre d'itération .

END 😳