LAB 3

Exercice 1: Multiplication de matrices

1- Écrivez l'algorithme qui permet de calculer le produit de 2 matrices A et B :

```
Debut
for i=1 to n do
   for j=1 to p do
          C[i][j] =0
   end if
end if
for i = 1 to n do
    for j = 1 to p do
        for k = 1 to m do
          C[i][j] = C[i][j] + A[i][k] × B[k][j]
        end for
   end for
end for
for i=1 to n do
   for j=1 to p do
          printf(C[i][j] );
   end if
endif
fin
```

Explication

L'algorithme prend en entrée deux matrices A de dimensions N×M et B de dimensions M×P.

On initialise une matrice C de dimensions N×P à des valeurs nulles.

Le calcul du produit matriciel est réalisé en parcourant les lignes de A, les colonnes de B, et en effectuant une somme pondérée par les éléments correspondants.

2- Calculer la complexité temporelle théorique de ce programme en termes de n, m et p

```
1- Debut
2- for i=1 to n do  → n
3-    for j=1 to p do  → p
4-      C[i][j] =0  →1
5-    end if
6- end if
7- for i = 1 to n do →n
8-    for j = 1 to p do →p
9-      for k = 1 to m do →m
10-         C[i][j] = C[i][j] + A[i][k] × B[k][j]  →3
11-      end for
12-     end for
13-  end for
14-  for i=1 to n do → n
15-      for j=1 to p do →p
16-         printf(C[i][j] );→1
17-      end for
18-  endfor
19-  fin
```

$f(n) = (n \times p) + (n \times p \times m \times 3) + (n \times p) = 2(n \times p) + 3(n \times p \times m)$

D'où $f(n) \in O(n \times p \times m)$

<u>Dans le cas où n=m=p (cas des matrices carrées), donner la nouvelle formulation de la complexité.</u>

Si n=m=p alors $f(n) = 2n^2 + 3n^3$

D'où la complexité de O(n^3)

3- Calculez l'espace mémoire nécessaire pour exécuter ce problème.

L'espace mémoire est principalement occupé par les trois matrices A, B et C. Chaque matrice nécessite n^2unités de mémoire, soit un total de $3n^2$unités de mémoire.

Les variables i,j et k sont des entiers et nécessitent un espace mémoire constant =3

Et on a 10 instructions

Donc la complexité spéciale=$3n^2$+3+10=$3n^2$+13

4- Écrivez le programme C correspondant et mesurez les temps d'exécution T du produit de deux matrices carrées (nxn) pour un échantillon de données de la variable n et représentez les résultats sous forme de tableau

Remarque : Tous les matrices crées sont de type dynamique en utilisant des double pointeur int **Matrice

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

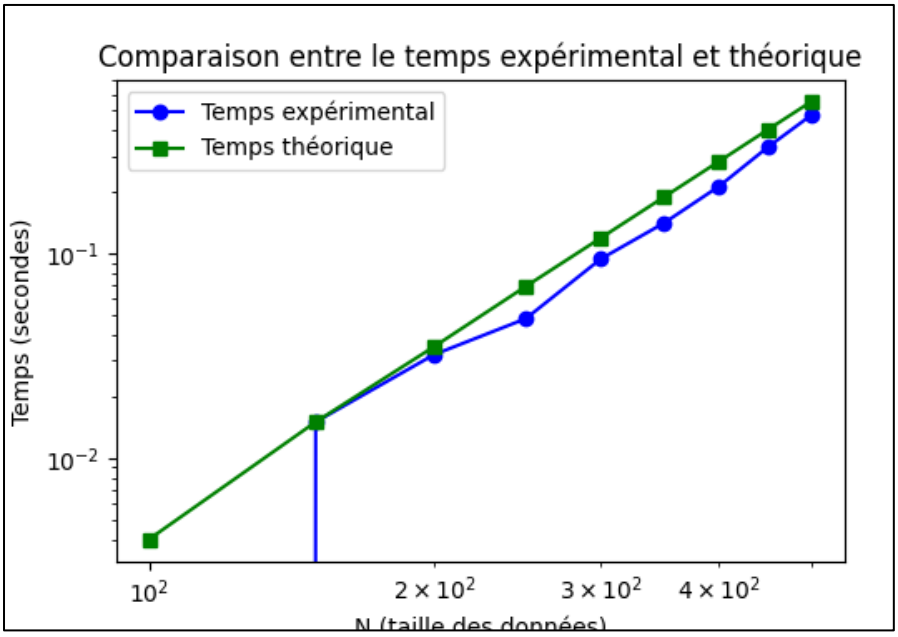
n	100	150	200	250	300	350	400	450	500
T exp	0.000	0.015	0.032	0.048	0.094	0.141	0.213	0.330	0.473
T theo	0.004	0.015	0.035	0.069	0.119	0.189	0.282	0.402	0.551

$$f(n) \times \Delta t = T(n)$$

$$\Delta t = \frac{T(n)}{f(n)}$$

$$\Delta t = \frac{T(n)}{2n^2+3n^3} = \frac{0.015}{2(150)^2+3(150)^3} = 1.47 * 10^{-9}$$

5- Représenter par un graphique les variations du temps T en fonction des valeurs de n.



5- Comparer la complexité théorique et la complexité expérimentale. Y a-t-il une concordanceentre le modèle théorique et les mesures expérimentales ?

Les deux mesures ($Texp\ et\ Ttheo$) augmentent de manière cubique, comme prévu par le modèle théorique O(n^3).
Cela confirme que la croissance expérimentale suit bien la complexité théorique.

Exercice 2 : Recherche d'une sous-matrice

1- En supposant que les éléments de A et B ne sont pas triés, écrire une fonction subMat1 qui recherche B dans A. Évaluer sa complexité temporelle théorique.

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

if( N_prime > N || M_prime  > M){
printf("La matrice B est plus grande que A \n"); return 0;
}
for (i = 0; i <= N - N_prime; 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 < N_prime; x++) {
       for (y = 0; y < M_prime; y++) {
        if (A[i + x][j + y] != B[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

L'algorithme recherche l'existence de B(N_prime,M_prime) dans A(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 matrice A

On utilise les indice i et j pour le parcours de lignes et colonne respectivement

On arrête le parcours a N-N_prime et M-M_prime car sinon la sou matrice recherché va dépasser

l'extrémité de la matrice A

Évaluer sa complexité temporelle théorique.

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

if( N_prime > N || M_prime  > M){
printf("La matrice B est plus grande que A \n"); return 0;
}
for (i = 0; i <= N - N_prime; 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 < N_prime; x++) {
       for (y = 0; y < M_prime; y++) {
        if (A [i + x][j + y] != B [x][y]) {
        trouve = 0; // Submatrix doesn't match at this
position
     break;
    }
S
   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;
}
```

1ere boucle: $N - N\_prime$ iteration
2eme boucle: $M - M\_prime$ iterations
3eme boucle: N_prime iterations
4eme boucle: M_prime iterations

Nombre itérations totale : $(N - N\_prime)(M - M\_prime)(N\_prime)(M\_prime)$

Complexité temporelle :

$$O((n - n_{prime}) * (m - m_{prime}) * n_{prime} * m_{prime})$$

2- En supposant que chacune des lignes de A et B est triée par ordre croissant (voir figure), écrivez une fonction subMat2 non naïve de complexité minimale pour trouver B dans A
Évaluez sa complexité temporelle théorique.

```
int SousMat2(int N, int M, int N_prime, int M_prime, int **A, int **B) {
if (N_prime > N || M_prime > M) {
printf("La matrice B est plus grande que A \n"); return 0;
}
int i, j, x, y;
for (i = 0; i <= N -N_prime; i++) {
for (j = 0; j <= M - M_prime; j++) {
int trouve = 1;
for (x = 0; x < N_prime; x++) {
// tester seulement la premiere case de chaque ligne
// si vrai , on parcours la matrice B , sinon on passe
if (A[i + x][j] != B[x][0]) { trouve = 0;
break;
}
//  parcourir la matrice B si la premiere case est trouvé
for (y = 1; y < B; y++) {
if (A[i + x][j + y] != B[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 :

L'amélioration par rapport au premier algorithme est que :

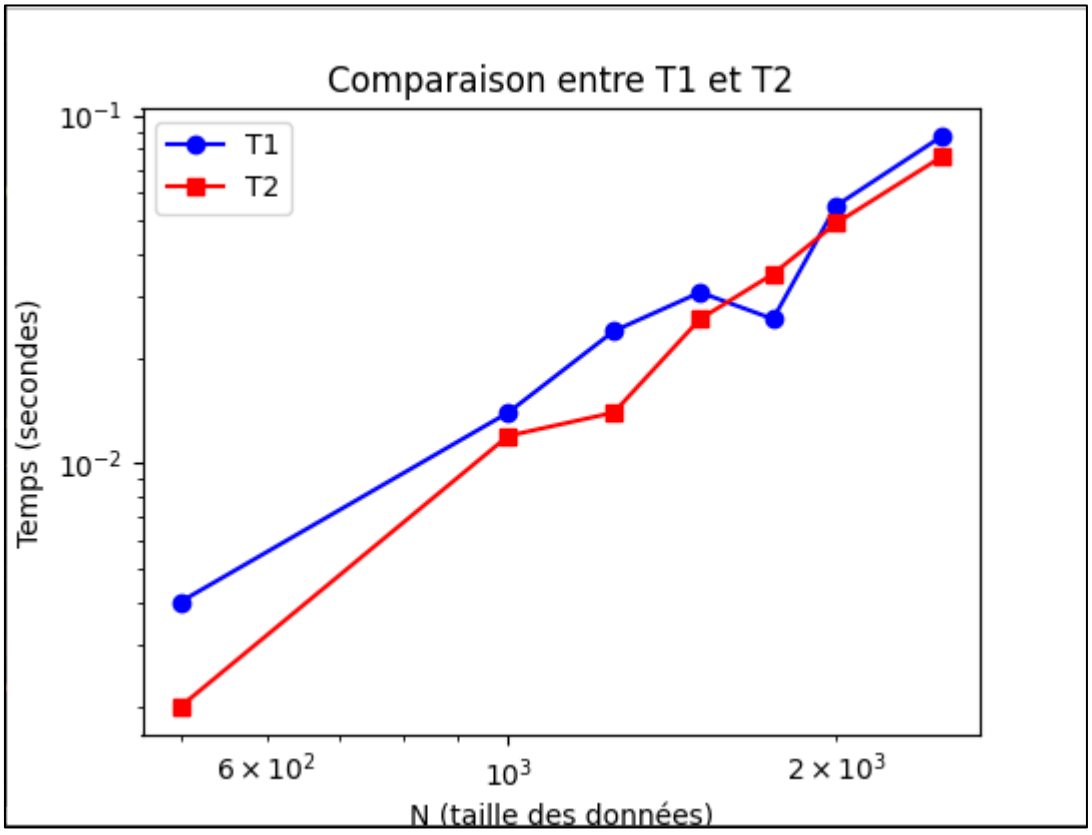
Dans l'algorithme 1 : On va tester pour chaque case matrice B si elle existe dans matrice A
Dans l'algorithme 2 : On ne teste que la première valeur de chaque ligne de la matrice B dans la matrice A
Si on trouve, alors on rentrer dans la dernière boucle pour parcourir toute la matrice B sinon on pace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

Complexité temporelle :

$$O((n - n_{prime}) * (m - m_{prime}) * n_{prime} * m_{prime})$$

3- Mesurer les temps d'exécution en faisant varier n, m puis n', m' et représenter les résultats sous forme d'un tableau pour les fonctions subMat1 et subMat2.

Taille de la matrice	500	1000	1250	1500	1750	2000	2500
T1	0.004	0.014	0.024	0.031	0.026	0.055	0.087
T2	0.002	0.012	0.014	0.026	0.035	0.049	0.076



Le graph montre bien que l'algorithme 2(T2) est une optimisation de l'algorithme 1(T1), en diminuant le nombre d'itération que l'algorithme fait dans la recherche des éléments de la petite matrice