Université de sciences et technologies Houari Boumedian

Faculté d'Informatique

TP N°3 : Complexité polynomiale

Algorithmique avancé et complexité
Prof: dr.Moussaoui

Etudiants:

- **Belmouloud Mustapha Abdellah** : partie test
- **Merazi Wail :** partie Redaction
- **Ben chaban Razika :** partie generation des données
- **Tabti Ikram :** partie code

1 Execrice 1

1.1 Generation des données

la generation des données a été faite en utilisant Malloc et la fonction rand(), il faut creer 3 matrice de taiile n donné par l'utilisateur:

```
printf("donnez la taille de la matrice: ");
scanf("%d", &n);

int **m1 = malloc(n * sizeof(int*));
for (int i = 0; i < n; i++) {
    m1[i] = malloc(n * sizeof(int));
    for (int j = 0; j< n; j++){
        m1[i][j] = rand() % (n*10 +1);
    }

}

int **m2 = malloc(n * sizeof(int*));
for (int i = 0; i < n; i++) {
    m2[i] = malloc(n * sizeof(int));
    for (int j = 0; j< n; j++){
        m2[i][j] = rand() % (n*10 +1);
    }
}

int **res = malloc(n * sizeof(int*));
for (int i = 0; i < n; i++) {
    res[i] = malloc(n * sizeof(int));
}
```

1.2 creation de la function de produit de deux matrice taille n

```
void matrixmulti(int n, int **m1, int **m2, int **res){

    for (int i = 0; i < n; i++){
        for (int j = 0; j < n; j++){
            res[i][j] = 0;
            for (int k = 0; k < n; k++){
                res[i][j] = res[i][j] + m1[i][k] * m2[k][j];
            }
        }
    }

}
```

1.3 La Partie Tests

les tests ont été faits sure un laptop i3 11eme generation, 2 cores @3Ghz et 8GB de ram avec un system d'operation linux Zorinos (base ubuntu)

1.3.1 la fonction utilisée pour mesurer le temps d'execution

la fonction utilisé pour mesurer le temps:

```
double now() {
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return ts.tv_sec + ts.tv_nsec * 1e-9;
}

double t0 = now();
matrixmulti(n, m1, m2, res);
double t1 = now();

printf("Execution time: %.6f seconds\n", t1 - t0);
fprintf(ftime, "%d -> %.6f \n", n, t1-t0);
```

1.3.2 les tableaux resultants

n	100	200	300	400	500
temps en seconds	0.008934	0.062806	0.191228	0.432845	0.823493

Table 1: Execution times for n = 100 to 500

n	600	700	800	900	1000
temps en seconds	1.531348	2.504993	3.869664	5.350861	8.966937

Table 2: Execution times for n = 600 to 1000

n	2000	3000	4000	5000	6000
temps en seconds	97.818775	334.602784	818.551430	1516.392213	2640.094264

Table 3: Execution times for n = 2000 to 6000

n	7000	8000	9000	10000
temps en seconds	4707.665077	7652.726984	12274.188779	15930.494307

Table 4: Execution times for n = 7000 to 10000

1.3.3 le graph resultants

generé par python et matplotlib

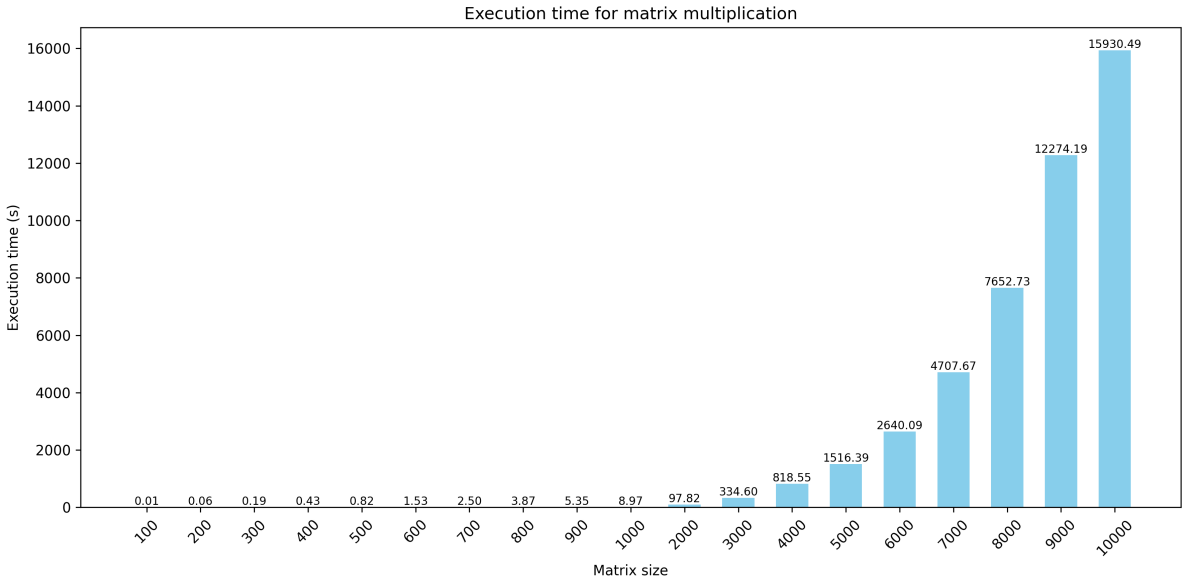

Figure 1: temps d'execution pour des differentes tailles

1.3.4 Remarques sur les Resultats

On remarque que les temps d'exécution se développent de manière exponentielle avec la taille de la matrice.

1.4 Repondre sur les questions theorique

1.4.1 Complexité theorique

on a trois bouble imbriqué et n=m=p donc c'est o(n^3)

1.4.2 memoire necessaire

taille d'une seul matrice = n^2, on a 3 matrice de meme taille don 3*n^2, la taille d'une seul case et de 4 octets (sur les system 64 bits) donc la taille finale est de $\mathbf{12n^2}$

1.4.3 les resultats theorque sont ils en concordance avec les resultats pratiques?

oui les resultats theoriques et pratique sont en concordance l'evolution est clairement exponontielle ce qui est exprimé par es resultats theoriques

2 Exercise 2

2.1 Generation des données

l'idée principale est de creer une matrice grande la remplir soit avec des elements triés ou aleatoires, puis remplir la petite matrice avec des elements deja existant dans la grande matrice, pour avoir des resultats coherents, la petite matrice a étée placé dans une place pré-connue.

```
int n1, n2, tr;
printf("donnez la taille de grande matrice: ");
scanf("%d", &n1);
printf("donnez la taille de la petitle matrice: ");
scanf("%d", &n2);
if (n2>n1/2){
    printf("n2 dit etre inferieur que n1/2");
    exit(1);
}

printf("appuiyer 1 pour trié et 2 pour non trié: ");
scanf("%d", &tr);
if (tr != 1 && tr != 2){
    printf("vous devez apuyer 1 ou 2");
    exit(1);
}

//creating the data
int **mb = malloc(n1 * sizeof(int*));
for (int i = 0; i < n1; i++) {
    mb[i] = malloc(n1 * sizeof(int));
    for (int j = 0; j< n1; j++){
        if(tr==2) mb[i][j] = rand() % (100);
        else {
            if(j==0) mb[i][j] = rand() % 10;
            else mb[i][j] =  mb[i][j-1] + rand() % 10;
        }
    }
}

int **ms = malloc(n2 * sizeof(int*));
for (int i = 0; i < n2; i++) {
    ms[i] = malloc(n2 * sizeof(int));
    for (int j = 0; j< n2; j++){
        ms[i][j] = mb[n1/2 + i][n1/2 + j];
    }
}
```

2.2 creations des functions sousMat1 et sousMat2

il s'agit de definir la condition d'arret a n1-n2 (taille de grande matrice - taille de petite matrice) et utiliser le premier element de la petite matrice comme un indice de recherche, pour sousMat2, on a utilisé la recherche Dichotomique sur les lignes de la matrice

```
void sousMat1(int **mb, int **ms, int n1, int n2){

    int found = 0; int skip = 0;
    for (int i = 0; i < n1-n2; i++){
        for (int j = 0; j < n1-n2; j++){
            if (mb[i][j] == ms[0][0]){
                found = 1;
                skip = 0;
                for (int k = 0; k < n2; k++){
                    for (int kk = 0; kk < n2; kk++){
                        if (mb[i+k][j+kk] != ms[k][kk]){
                            skip = 1;
                            found = 0;
                            break;
                        }
                    }
                    if (skip == 1) break;
                }
            }
            if(found == 1 ){
                printf("MATRIX FOUND\n");
                return;
            }
        }
    }
    if (found == 0) printf("matrix NOT found\n");
}

void sousMat2(int **mb, int **ms, int n1, int n2){

    int found = 0; int skip = 0; int start; int finish;
    for (int i = 0; i < n1-n2; i++){
        if(mb[i][n1/2] > ms[0][0]){
            start = 0; finish = n1/2 - 1;
        }else if (mb[i][n1/2] < ms[0][0]){
            start = n1/2 + 1;
            finish = n1-n2;
        }else if(mb[i][n1/2] == ms[0][0]){
            found = 1;
            skip = 0;
            for (int k = 0; k < n2; k++){
                for (int kk = 0; kk < n2; kk++){
                    if (mb[i+k][n1/2 + kk] != ms[k][kk]){
                        skip = 1;
                        found = 0;
                        break;
```

```
                    }
                }
                if (skip == 1) break;
            }
        }
        if(found == 1 ){
            printf("MATRIX FOUND\n");
            return;
        }
        else{
            for (int j = start; j < finish; j++){
                if (mb[i][j] == ms[0][0]){
                    found = 1;
                    skip = 0;
                    for (int k = 0; k < n2; k++){
                        for (int kk = 0; kk < n2; kk++){
                            if (mb[i+k][j+kk] != ms[k][kk]){
                                skip = 1;
                                found = 0;
                                break;
                            }
                        }
                        if (skip == 1) break;
                    }
                }
            }
        }
        if(found == 1 ){
            printf("MATRIX FOUND\n");
            return;
        }
    }
    if (found == 0) printf("matrix NOT found\n");
}
```

2.3 les Tests

2.3.1 les temps d'executions

elements triés:

Matrix size	Small matrix size	Execution time (s)
100	10	0.000033
200	20	0.000074
300	30	0.000159
400	40	0.000244
500	50	0.000365
600	60	0.000539

Table 5: Ordered search: Matrix size 100–600

Matrix size	Small matrix size	Execution time (s)
700	70	0.000670
800	80	0.000933
900	90	0.001072
1000	100	0.001137
2000	200	0.003670
3000	300	0.008517

Table 6: Ordered search: Matrix size 700–3000

Matrix size	Small matrix size	Execution time (s)
4000	400	0.014253
5000	500	0.025372
6000	600	0.039839
7000	700	0.049066
8000	800	0.067164
9000	900	0.080761

Table 7: Ordered search: Matrix size 4000–9000

Matrix size	Small matrix size	Execution time (s)
10000	1000	0.099204

Table 8: Ordered search: Matrix size 10000

elements non triés:

Matrix size	Small matrix size	Execution time (s)
100	10	0.000060
200	20	0.000138
300	30	0.000300
400	40	0.000681
500	50	0.000771
600	60	0.001065

Table 9: Unordered search: Matrix size 100–600

Matrix size	Small matrix size	Execution time (s)
700	70	0.001399
800	80	0.001705
900	90	0.002236
1000	100	0.002565
2000	200	0.008599
3000	300	0.019162

Table 10: Unordered search: Matrix size 700–3000

Matrix size	Small matrix size	Execution time (s)
4000	400	0.033946
5000	500	0.053714
6000	600	0.082268
7000	700	0.108207
8000	800	0.135675
9000	900	0.172304

Table 11: Unordered search: Matrix size 4000–9000

Matrix size	Small matrix size	Execution time (s)
10000	1000	0.213147

Table 12: Unordered search: Matrix size 10000

2.3.2 le graph resultant

generé par python et mathplotlib

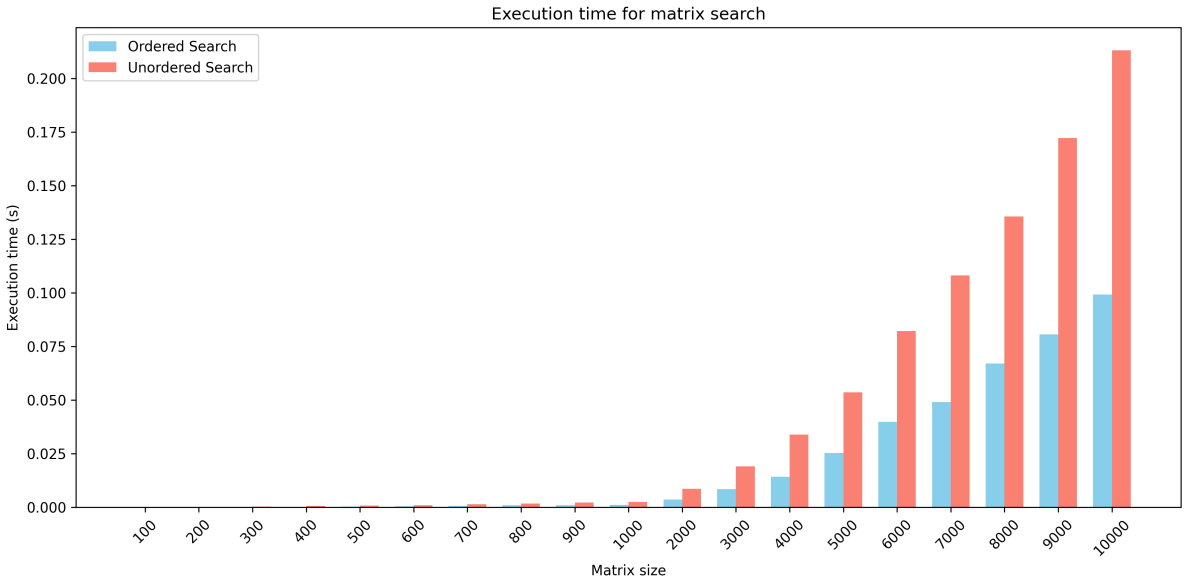

Figure 2: temps d'execution pour des differentes tailles

2.3.3 remarques sur les resultats

on remarque que les temps se devolopent d'une façon exponontielle, mais avec une courbe plus douce que celle de le produit matricielle, on remarque aussi que les temps sont trops petits, car les operations de comparaisons en c sur les processeurs moderne sont trés rapids, mais la courbe affirme que la complixité se devolope d'une manniere logique

2.4 Repondre sur les questions theoriques

aux pire cas, la complexité de sousMat1 et sousMat2 est toujour o(n^2) car on a (n1-n2)^2 iterations pour sousMat1 et (n1/2-n2)^2 iterations pour sousMat2, ce qui est coherents avec les resultats pratiques.