# Performance Measurement

Fundamental of Data Structure

Project 1

**Yizhou Chen** **Jinze Wu** **Zheming Xu**

**2018-10-12**

# Chapter 1 Introduction

As we all know, when it comes to solving a problem, we can always come up with different algorithms. But different algorithms can have the different performance while solving the same problem. So, we need to compare different algorithms in the same environment, to know which is better and how to improve the efficiency.

Our project is to provide three different algorithms, for the **MAXIMUM SUBMATRIX SUM PROBLEM.**

**MAXIMUM SUBMATRIX SUM PROBLEM**

Given an $N \times N$ matrix $(a_{ij})_{N \times N}$ of positive and negative integers, find the maximum value of $\sum_{k=i}^{m} \sum_{l=j}^{n} a_{kl}$ for all $1 \leq i \leq m \leq N$ and $1 \leq j \leq n \leq N$. The maximum submatrix sum will be 0 if all the integers are negative.

**Example:**

For matrix $\begin{bmatrix} 0 & -2 & -7 & 0 \\ 9 & 2 & -6 & 2 \\ -4 & 1 & -4 & 1 \\ -1 & 8 & 0 & -2 \end{bmatrix}$ , the maximum submatrix is $\begin{bmatrix} 9 & 2 \\ -4 & 1 \\ -1 & 8 \end{bmatrix}$ and has the sum of 15.

We first finish the three algorithms.

- Algorithm 1 exhaustively try calculate all the possible sums to find maximum.
- Algorithm 2 uses a prefix sum matrix instead of traverse to calculate each sum.
- Algorithm 3 uses dynamic programming instead of exhaustive try.

Then we finish the test program and test the algorithms in case of n=5,10,30,50,80,100 to measure the running time, using C’ s standard library time.h. We draw the output into tables and use visible analysis to analyze the data and compare the algorithms.

Finally, through analysis, we found that the time complexity of algorithm 1 is O(n^6), algorithm 2 is O(n^4) and algorithm 3 is O(n^3), while algorithm 2 and algorithm 3 need a little more space.

# Chapter 2 Algorithm Specification

## 2.1 Main Program Sketch

### 2.1.1 specification

The whole project consists of a main function, a test function and three algorithms of the MAXIMUM SUBMATRIX SUM PROBLEM. In the main program, we input N,M,K, each stands for the size of the matrix, the number of the algorithm to test and the test iterations. And then, we call the TestTime function to test the total time consumption of each algorithm. Then we output the answer.

### 2.1.2 Pseudo code

---

**Main Function** The main function

**function** main( )
    **input:** N,M,K
    **for** i←1 to N **do**
        **for** j←1 to N **do**
            a[i][j]=random(-16383,16384)
    **if** M=1 **then** TestTime($(a_{ij})_{N\times N}$, Algorithm1,N,K)
    **if** M=2 **then** TestTime($(a_{ij})_{N\times N}$, Algorithm2,N,K)
    **if** M=3 **then** TestTime($(a_{ij})_{N\times N}$, Algorithm3,N,K)
**end function**

---

**Test Function** The test function

**function** TestTime( $(a_{ij})_{N\times N}$, Algorithm,N,K)
    start←StartTime
    **for** i←1 to K **do** Run Algotrithm
    stop←StopTime
    **output:** TotalTime←StopTime-StartTime
**end function**

## 2.2 Algorithm 1

### 2.2.1 Specification:

We exhaustingly try every sub-matrix by enumerating all the possible upper left corners and the lower right corners. And for each sub-matrix, we traverse each element and sum it up every time, and compare each time to find the maximum.

To implement this algorithm, we have to use 6 for-loops;

Time complexity: O(N^6)  Space complexity: O(1)

### 2.2.2 Pseudo code:

**Algorithm 1** The algorithm 1

```
function MaxSubMatrixSum( (a_ij)_N×N, N )
    maximum←0
    for row1←1 to N do
        for column1←1 to N do
            for row2←1 to N do
                for column2←1 to N do
                    sum←0
                    for i←row1 to row2 do
                        for j←column1 to column2 do
                            sum←sum+a[i][j]
                            if sum>maximum then maximum←sum
    return maximum
end function
```

## 2.3 Algorithm 2

### 2.3.1 Specification:

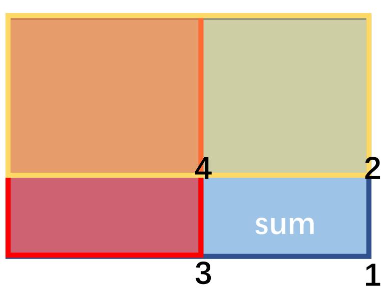
We still enumerate all the possible upper left corners and the lower right corners to find the maximum. Instead of traverse every time to get the sum, we calculate the sum of each sub-matrix by a prefix sum matrix of the initial matrix. That is, we create a new matrix called PreSum in which **PreSum[i][j]**= $\sum_{k=1}^{i}\sum_{l=1}^{j} a_{kl}$ .

**Example:**

Initial matrix a= $\begin{bmatrix} 0 & -2 & -7 & 0 \\ 9 & 2 & -6 & 2 \\ -4 & 1 & -4 & 1 \\ -1 & 8 & 0 & -2 \end{bmatrix}$ Prefix sum matrix PreSum= $\begin{bmatrix} 0 & -2 & -9 & -9 \\ 9 & 9 & -4 & -2 \\ 5 & 6 & -11 & -8 \\ 4 & 13 & -4 & -3 \end{bmatrix}$

Then we use including excluding principal to calculate PreSum and sum of sub-matrix.

**including excluding principal usage explanation:**




**Sum=PreSum1-PreSum3-PreSum2+PreSum4**

Calculate the PreSum matrix:

**PreSum[i][j]=PreSum[i-1][j]+PreSum[i][j-1]-PreSum[i-1][j-1]+a[i][j]**

Calculate the sum of each sub-matrix:

$\sum_{k=i1}^{i2}\sum_{l=j1}^{j2} a_{kl}$=**PreSum[i2][j2]-PreSum[i2][j1-1]-PreSum[i1-1][j2]+PreSum[i1-1][j1-1]**

To implement this algorithm, we have to use 2 for-loops to calculate PreSum in advance, and 4 for-loops to enumerate all the sub-matrixes.

Time complexity: O(N^4)  Space complexity: O(1)

### 2.3.2 Pseudo code:

---

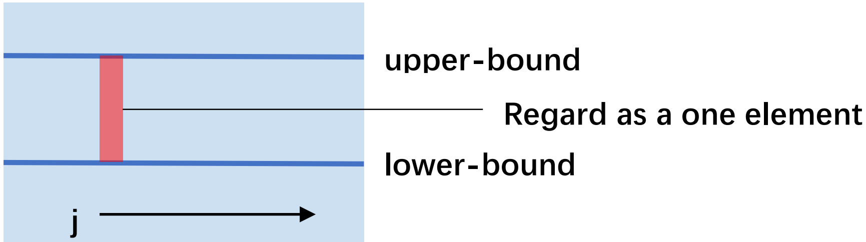
**Algorithm 2** The algorithm 2

```
function MaxSubMatrixSum( (a_ij)_{N×N}, N )
    for i←0 to n do
        for j←0 to n do
            PreSum[i][j]←0
    for i←1 to n do
        for j←1 to n do
            PreSum[i][j]=PreSum[i-1][j]+PreSum[i][j-1]-PreSum[i-1][j-1]+a[i][j]
    maximum←0
    for row1←1 to n do
        for column1←1 to n do
            for row2←1 to n do
                for column2←1 to n do
                    sum←∑_{k=row1}^{row2} ∑_{l=column1}^{colum2} a_kl  calculated directly using Presum
                    if sum>maximum then maximum←sum
    return maximum
end function
```

## 2.4 Algorithm 3

### 2.4.1 Specification:

Instead exhaustive try, we choose dynamic programming. We enumerate the upper-bounds and the lower-bounds of the sub-matrixes, and make **element[j]=**$\sum_{i=upper\_bound}^{lower\_bound} a_{ij}$ and we calculate each element using PreSum matrix within O(1) time. Then we have just simplified the problem to the MAX SUBSEQUENCE SUM problem.




Then for each pair of upper-bound and lower-bound, we make **s[j] the maximal sum of the element sequence ending with element[j].**

**DP equation :**

$$\mathbf{s[j] = max\{s[j-1] + element[j],\ element[j]\} = max\{s[j-1], 0\} + element[j].}$$

$$\mathbf{maximum = max\{s[j], 0\}\quad 1 \le j \le n.}$$

To implement this algorithm, we have to use 2+1 for-loops to find the maximum.

Time complexity: O(N^3)  Space complexity: O(1)

## 2.4.2 Pseudo code:

---

**Algorithm 3** The algorithm 3

```
function MaxSubMatrixSum( (a_ij)_{N×N}, N )
    for i←0 to n do
        for j←0 to n do  PreSum[i][j]←0
    for i←1 to n do
        for j←1 to n do
            PreSum[i][j]=PreSum[i-1][j]+PreSum[i][j-1]-PreSum[i-1][j-1]+a[i][j]
    maximum←0
    for row1←1 to n do
        for row2←row1 to n do
            sum←0
            for j←1 to n do
                sum←sum+element(j) calculated directly using Presum
                if sum<0 then sum←0
                if sum>maximum then maximum←sum
    return maximum
end function
```

# Chapter 3 Testing Results

## 3.1 Purpose

our purpose is to substantiate the correctness of each algorithm’s theoretical time complexity and compare their efficiency.

## 3.2 Preparation

To quantify the time complexity of the three algorithms, we put each of them into the C standard library time.h and used the function rand() to create a random matrix. We chose to print out ticks(ticks=stop-start) and the duration for each data recording. And determine K when the running time is too short.

## 3.3 Test Data

| | N | 5 | 10 | 30 | 50 | 80 | 100 |
|---|---|---|---|---|---|---|---|
| o(N^6) | Iterations(K) | 1000000 | 10000 | 10 | 1 | 1 | 1 |
| | Ticks | 4200 | 1412 | 699 | 1257 | 19486 | 73245 |
| | Total time(sec) | 4.2 | 1.412 | 0.699 | 1.257 | 19.486 | 73.245 |
| | Duration(sec) | 0.0000042 | 0.0001412 | 0.0699 | 1.257 | 19.486 | 73.245 |
| o(N^4) | Iterations(K) | 1000000 | 100000 | 1000 | 100 | 10 | 1 |
| | Ticks | 2149 | 2400 | 1572 | 1183 | 726 | 173 |
| | Total time(sec) | 2.149 | 2.4 | 1.572 | 1.183 | 0.726 | 0.173 |
| | Duration(sec) | 0.000002149 | 0.000024 | 0.001572 | 0.01183 | 0.0726 | 0.173 |
| O(N^3) | Iterations(K) | 1000000 | 100000 | 10000 | 1000 | 1000 | 1000 |
| | Ticks | 921 | 555 | 1416 | 582 | 2303 | 4147 |
| | Total time(sec) | 0.921 | 0.555 | 1.416 | 0.582 | 2.303 | 4.147 |
| | Duration(sec) | 0.000000921 | 0.00000555 | 0.0001416 | 0.000582 | 0.002303 | 0.004147 |

# Chapter 4 Analysis and Comments

## 4.1 Time complexity

### 4.1.1 correctness analysis

**Pre-Analysis for time complexity**

As for Algorithm1, there were 2 loops for input and 6 loops for adding the matrix data to the sum and comparison, so the time complexity was O(N^6). Other O(1) statements were ignored. As for Algorithm2, there were 2 loops for assigning the matrix, 2 loops for assigning the 2D array(PreSum), and 4 loops for adding data to the sum and comparison, so the time complexity was O(N^4).As for Algorithm3, there were 3 loops at most in adding the data to the sum, so the time complexity was O(N^3).

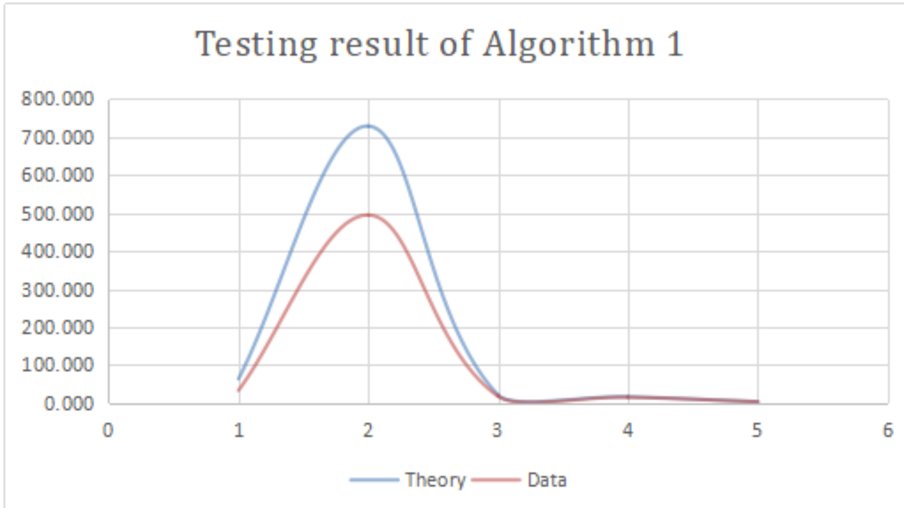
**Data processing**

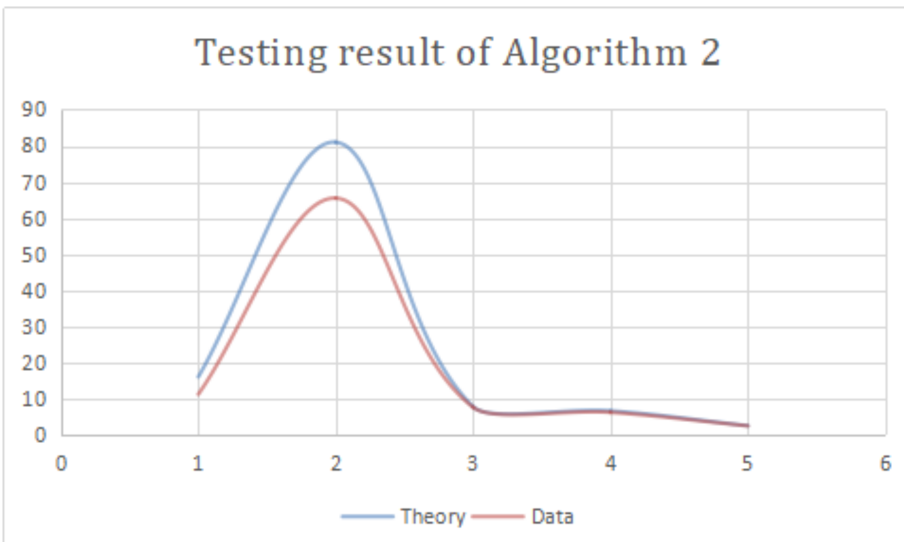
We chose to compare the actual and theoretical running time growth rate.We expected that if the algorithm' s time complexity is O(n^e)(e=3,4,6),then the ratil of time consumption for case n=N1 and n=N2 should be approximately (N1/N2)^e.

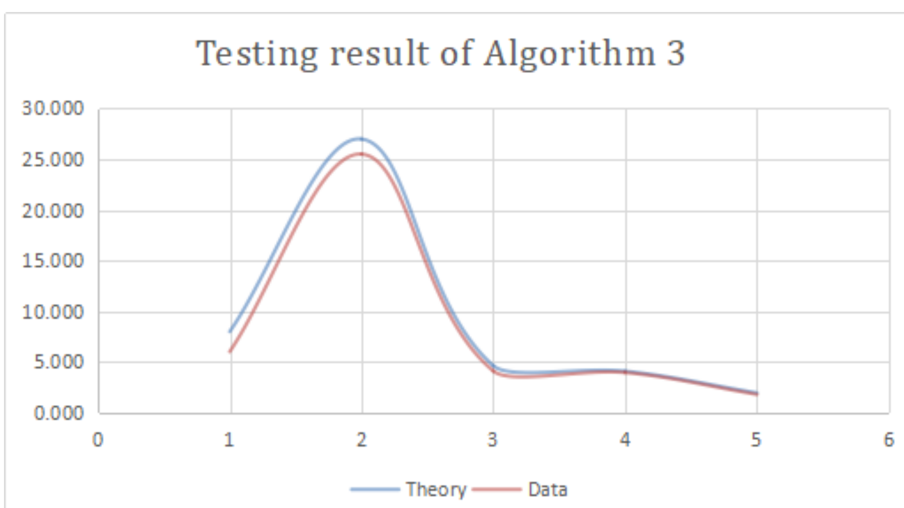
So we record 5 theoretical data(in the theory row) whose general formal is $(N_{latter} / N_{former})^{Exponent}$ (exponent can be 3,4,6), and at the same time we calculate 5 actual data whose general formal is $[Duration(N_{latter}) / Duration(N_{former})]^{Exponent}$ . And we number them in turn(in the first row, eg:1,2,3...). Then we made number-theoretical data function image and number-actual data function image in the same coordinate to compare their tendency.

**Data and Plots**

| | Number | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| O(N^6) | Theory | 64.000 | 729.000 | 21.433 | 16.777 | 3.815 |
| | Data | 33.619 | 495.042 | 17.983 | 15.502 | 3.759 |
| O(N^4) | Theory | 16.000 | 81.000 | 7.716 | 6.554 | 2.441 |
| | Data | 11.168 | 65.500 | 7.525 | 6.137 | 2.383 |
| O(N^3) | Theory | 8.000 | 27.000 | 4.630 | 4.096 | 1.953 |
| | Data | 6.026 | 25.514 | 4.110 | 3.957 | 1.801 |


Testing result of Algorithm 1
800.000
700.000
600.000
500.000
400.000
300.000
200.000
100.000
0.000
0
1
2
3
4
5
6
Theory
Data


Testing result of Algorithm 2
90
80
70
60
50
40
30
20
10
0
0
1
2
3
4
5
6
Theory
Data

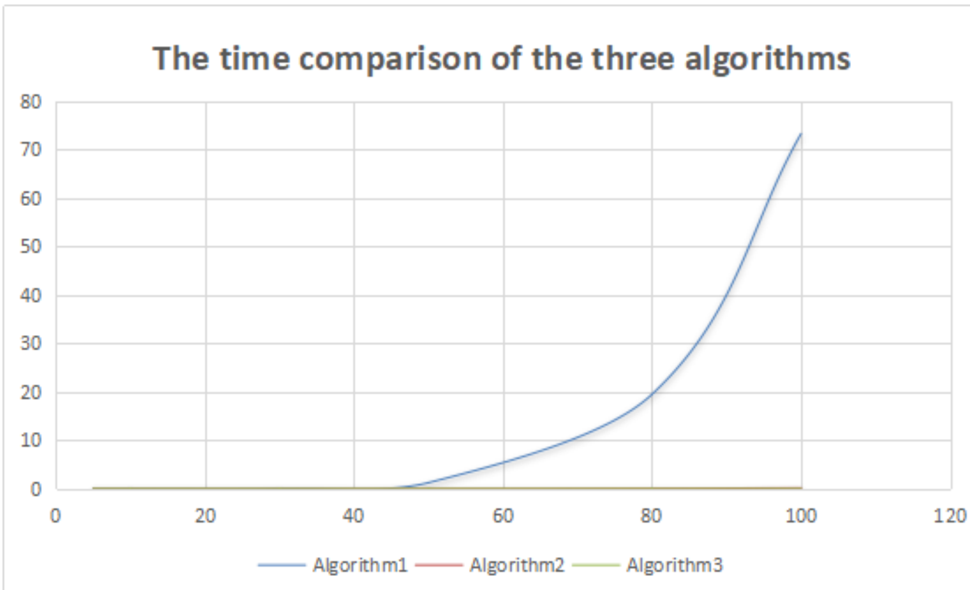

Testing result of Algorithm 3
30.000
25.000
20.000
15.000
10.000
5.000
0.000
0
1
2
3
4
5
6
Theory
Data

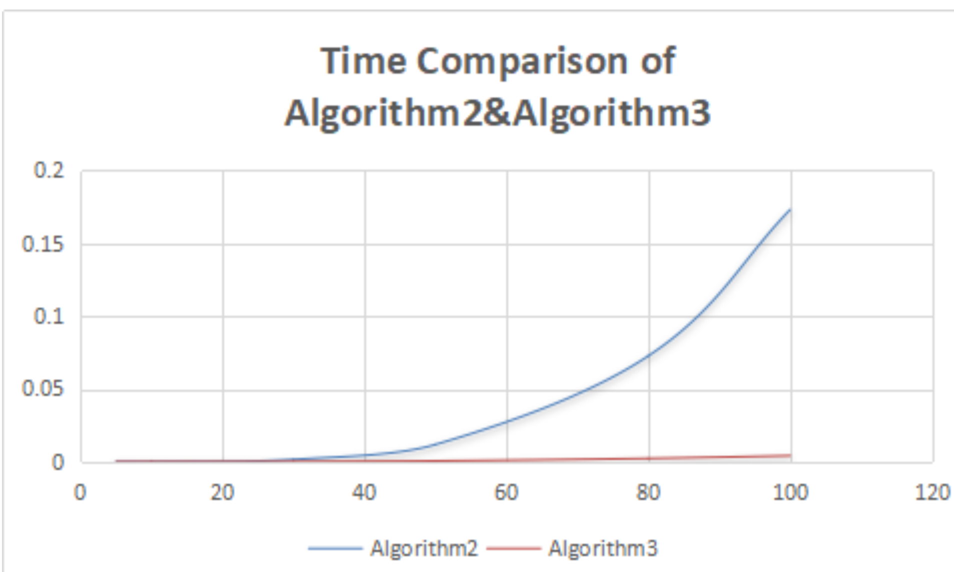
### 4.1.2 Time consumption comparison

**Data processing**

We made n the x-axis and time consumption(/s) the y-axis to draw the first plot for all three algorithms. Since in the first plot the Algorithm 2 and Algorithm 3 are too close, we draw a second plot to see the tendency.

**Plots**



### 4.1.3 Conclusions-Pass

After measuring and comparing the function, we got some conclusions.

(1) To a specific N, each test showed a different running time but of the same order of magnitude.

(2) As N grew, the running time of Algorithm2 grew far slower compared to that of Algorithm1, but Algorithm3 ran fastest. This feature was apparent when N got larger.

(3) Algorithm1 grew slow when N was smaller than 50 but grew sharply fast when beyond 50.

(4) The three algorithm showed the similar growing tendency when N was smaller than 50.

(5) In the three visualization of correctness tests, we found as N grew larger, the theoretical data matched higher the actual data.

## 4.2 Space complexity

The space complexity of an algorithm only considers the size of storage space allocated to local variables during the operation. As for Algorithm1, there were 9 storage spaces for integer variables (i1, i2, i3, j1, j2, j3, n, Thissum, Maxsum), 1 space for address unit(a) and 202 spaces for data in the array. So space complexity is O(1). Algorithm2 allocated 2 spaces for address unit(a, PreSum), 9 for integer variables and 404 for data in two arrays. Space complexity is O(1). The space complexcity of Algorithm3 is O(1) as well. But in the same order, Algorithm3 took more space.

## 4.3 Comments

(1) The measuring and comparing result basically matched the theoretical anticipation.
(2) To each N, multiple measurements were required for a relatively accurate result.
(3) When N was small, the impact of some other useless statements which caused more running time was apparent, but that impact got smaller when N got larger. This could well explain foundings(5).
(4) Through the test, to an algorithm time complexity and space complexity had mutual impacts. One who purchased a better time complexity might result in a relatively worse space complexity. So was the opposite situation. Therefore, we need to take both two factors into account to create the most efficient algorithm.

# Appendix

## Appendix A Source code

### A.1 header

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* three alogrithm of different time complexity*/

/*
 *Description£/This procedure is used to find the maximum sub-matrix within O(n^6)
 *Parameter[in] a  the pointer array of the initial matrix
 *Parameter[in] n  the size of the square matrix
 *return maximum  the maximal sum of sub-matrix
 */
long int GetMaxInOn6(int (*a)[102],int n);

/*
 *Description£/This procedure is used to find the maximum sub-matrix within O(n^4)
 *Parameter[in] a  the pointer array of the initial matrix
 *Parameter[in] n  the size of the square matrix
 *return maximum  the maximal sum of sub-matrix
 */
long int GetMaxInOn4(int (*a)[102],int n);

/*
 *Description£/This procedure is used to find the maximum sub-matrix within O(n^3)
 *Parameter[in] a  the pointer array of the initial matrix
 *Parameter[in] n  the size of the square matrix
 *return maximum  the maximal sum of sub-matrix
 */
long int GetMaxInOn3(int (*a)[102],int n);

/* testing function */

/*
 *Description: This procedure is used to test the time of each algorithm
 *Parameter[in] a  the pointer array of the initial matrix
 *Parameter[in] Algorithm  the pointer of the test algorithm
 *Parameter[in] n  the size of the square matrix
 *Parameter[in] k  the run times of the algorithm for testing
 *output stop-start the total ticks
 *output duration  algorithm comsuption time
*/
void TestTime(int (*a)[102],long int (*Algorithm)(int(*)[102],int),int n,int k);

```

# A.2 main function

```c
int main(){

    int n,m,k,i,j;
    int a[102][102];

    /* n stands for the size of matrix
     * m stands for the test algorithm 1/2/3
     * k stands for the iteration */
    scanf("%d %d %d",&n,&m,&k);

    /* generate a random matrix for testing*/
    srand((unsigned)time(NULL));
    for (i=0;i<=n;i++)
        for (j=0;j<=n;j++) a[i][j]=0;
    for (i=1;i<=n;i++)
        for (j=1;j<=n;j++) a[i][j]=rand()-16383;    //make sure there the range is [-16363,16384]

    /* test each algorithm */
    if (m==1) TestTime(a,GetMaxInOn6,n,k);
    if (m==2) TestTime(a,GetMaxInOn4,n,k);
    if (m==3) TestTime(a,GetMaxInOn3,n,k);

    return 0;
}
```

# A.3 test function

```c
void TestTime(int (*a)[102],long int (*Algorithm)(int(*)[102],int),int n,int k) {
    clock_t start,stop;
    double TotalTime;
    int i;
    long int ans;
    start=clock();  //start time
    for(i=1;i<=k;i++) ans=(*Algorithm)(a,n);
    stop=clock();   //stop time

    printf("%ld\n",stop-start);
    TotalTime=((double)(stop-start))/CLK_TCK;
    printf("%lf\n",TotalTime);
}

```

# A.4 Algorithm 1

```c
long int GetMaxInOn6(int (*a)[102],int n) {
    int i1,i2,i3,j1,j2,j3;
    long int ThisSum,MaxSum=0;

    ThisSum=0;
    for(i1=1;i1<=n;i1++)
        for(j1=1;j1<=n;j1++)  //enumerate the upper-left corners (i1,j1)
            for(i2=i1;i2<=n;i2++)
                for(j2=j1;j2<=n;j2++) {  //enumerate the lower-right corners (i2,j2)
                    ThisSum=0;
                    for(i3=i1;i3<=i2;i3++)
                        for(j3=j1;j3<=j2;j3++) ThisSum=ThisSum+a[i3][j3]; //traverse the sub-matrix to get the sum
                    if(ThisSum>MaxSum) MaxSum=ThisSum;

                }
    return MaxSum;
}

```

# A.5 Algorithm 2

```
long int GetMaxInOn4(int (*a)[102],int n) {
    int i,j,i1,j1,i2,j2;
    long int Sum,MaxSum;
    long int PreSum[102][102];

    MaxSum=0;

    /* initialize the prefix sum matrix */
    for(i=0;i<=n;i++)
        for(j=0;j<=n;j++)
            PreSum[i][j]=0;

    /* calculate the prefix sum matrix */
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            PreSum[i][j]=PreSum[i-1][j]+PreSum[i][j-1]-PreSum[i-1][j-1]+a[i][j];

    /* find the maximum */
    for(i1=1;i1<=n;i1++)
        for(j1=1;j1<=n;j1++) //enumerate the upper-left corners (i1,j1)
            for(i2=i1;i2<=n;i2++)
                for(j2=j1;j2<=n;j2++){  //enumerate the lower-right corners (i2,j2)
                    Sum=PreSum[i2][j2]-PreSum[i1-1][j2]-PreSum[i2][j1-1]+PreSum[i1-1][j1-1];  //calculate the sum
                    if(Sum>MaxSum) MaxSum=Sum;
                }
    return MaxSum;
}
```

# A.6 Algorithm 3

```
long int GetMaxInOn3(int (*a)[102],int n) {
    int i,j,iu,il;
    long int Sum,MaxSum;
    long int PreSum[102][102];

    /* initialize the prefix sum matrix */
    for(i=0;i<=n;i++)
        for(j=0;j<=n;j++)
            PreSum[i][j]=0;

    /* calculate the prefix sum matrix */
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            PreSum[i][j]=PreSum[i-1][j]+PreSum[i][j-1]-PreSum[i-1][j-1]+a[i][j];

    MaxSum=0;
    for (iu=1;iu<=n;iu++) //enumerate the upper-bounds
        for (il=iu;il<=n;il++) { //enumerate the lower-bounds
            Sum=0;
            /* dynamic prgramming */
            for (j=1;j<=n;j++) {
                if (Sum<0) Sum=0;  // DP equation: sum[j]=max{sum[j-1],0}+element[j]
                Sum+=PreSum[il][j]-PreSum[il][j-1]-PreSum[iu-1][j]+PreSum[iu-1][j-1];
                if (Sum>MaxSum) MaxSum=Sum;
            }
        }
    return MaxSum;
}
```

## Appendix B Declaration

We hereby declare that all the work done in this project titled "Performance Measurement" is of our independent effort as a group.

# Duty Assignments

**Programmer: Jinze Wu**

**Tester: Zheming Xu**

**Reporter: Yizhou Chen**

**Bonus:**

**Programmer: Yizhou Chen**

**Tester: Jinze Wu**

**Reporter: Yizhou Chen**

**Since our tester had withdrawn from the course , so the revise for the report paper mainly including the chapter 3 and 4 was done by the programmer and the reporter.**