Project1 Performance Measurement (MSS)

Author Names:
Group-02
柴子炜
章含挺

Date: 2018-10-08

CONTENTS

Chapter 1: Introduction

Problem description

MSS, the Maximum Submatrix Sum problem, is the two-dimensional expansion of the Maximum Subsequence Sum problem. We are given an $N \times N$ integer matrix $(a_{ij})N \times N$, and shall 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 $1 \leq j \leq n \leq N$. We are asked to complete the O(N^6) and O(N^4) versions of algorithms. Besides, we are supposed to compare the complexity of different algorithms by analyzing and testing.

For this topic, our main task is to test the performance of the three algorithms for solving the Maximum Submatrix Sum problem. In order to analyze the time complexity and space complexity, it is necessary to select a reasonable number of iterations and test data range.

In the process of testing the performance of the algorithm, we can visually see the impact of algorithm complexity on the performance of the algorithm, and analyze the advantages and disadvantages of the algorithm.

Our expected goals

To improve efficiency, we have designed an O(N^3) algorithm. Then we design a test program to get the running time of the three functions. Since the function runs so quickly that it takes less than a tick to finish, we set iterations(K) and repeat the function for K times to obtain a more precise duration for a single run of the function.

After working out this project, our group will be more familiar with some C standard library, *<time.h>*, for example. And we'll have a deeper understanding of the space and time efficiency of an algorithm. Meanwhile, we gain experience in report writing and teamwork.

Chapter 2: Algorithm Specification

In this chapter we'll show our main ideas and steps by pseudo-code.

Algorithm 1 Maximum Submatrix Sum

```
procedure MaxSubMatrixSum (M : n × n matrix)
max := 0
for i := 1 to n
    for j := i to n
        for k := 1 to n
            for l := k to n
                sum := 0
                for a := i to j
                    for b := k to l
                        sum := sum + M_ab
                if sum > max
                    then max = sum
return max
{max is the maximum submatrix sum of matrix M}
```

Algorithm 2 Maximum Submatrix Sum

```
procedure MaxSubMatrixSum (M : n × n matrix)
max := 0
for i := 1 to n
    set all elements in temp as 0
    for j := i to n
        for k := 1 to n
            temp[ k ] := temp[ k ] + M_jk
        for k := 1 to n
            sum := 0
            for l := k to n
                sum := sum + temp[ l ]
                if sum > max
                    then max := sum
return max
{max is the maximum submatrix sum of matrix M}
```

Algorithm 3 Maximum Submatrix Sum

procedure $MaxSubMatrixSum$ ($\mathbf{M} : n \times n\ matrix$)
$max := 0$
for $i := 1$ **to** n
    set *all elements in temp as* 0
    for $j := i$ **to** n
        for $k := 1$ **to** n
            $temp[\ k\ ] := temp[\ k\ ] + M_{jk}$
        $sum := 0$
        for $k := 1$ **to** n
            $sum := sum + temp[\ k\ ]$
                if $sum > max$
                    then $max := sum$
return max
$\{max\ is\ the\ maximum\ submatrix\ sum\ of\ matrix\ \mathbf{M}\}$

Chapter 3: Testing Results

In this chapter we will show the different time to run those three programs with different data sizes N = 5, 10, 30, 50, 80, 100. As for iterations(K), we change the value of K to fit the change of N because the total time grows rapidly as N increases. Then we find the best fitting function of the three algorithms.

The accuracy test

We have set some special cases to test the potential bugs such as memory overrun. After testing, we confirm that the three programs all work well.

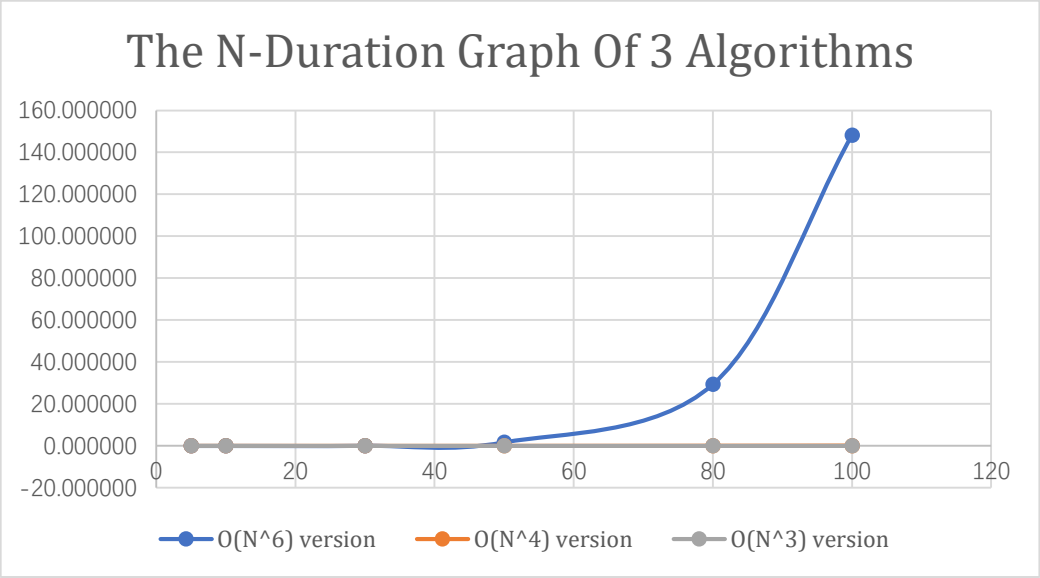
Program	O(N^6) Version		O(N^4) Version		O(N^3) Version	
Case	N=10000 K=1	N=1 K=1	N=10000 K=1	N=1 K=1	N=10000 K=1	N=1 K=1
Result	Invalid Input	Invalid Input	Invalid Input	Invalid Input	Invalid Input	Invalid Input
Comment	PASS	PASS	PASS	PASS	PASS	PASS

The time complexity test

	N	5	10	30	50	80	100
O(N^6) version	Iterations(K)	100000	10000	100	1	1	1
	Ticks	5797	3838	8508	1705	29351	148232
	Total Time(sec)	5.797	3.838	8.508	1.705	29.351	148.232
	Duration(sec)	0.000058	0.000384	0.085080	1.705	29.351000	148.232
O(N^4) version	Iterations(K)	100000	10000	100	100	100	100
	Ticks	5402	2574	1911	1227	5705	12096
	Total Time(sec)	5.402	2.574	1.911	1.227	5.705	12.096
	Duration(sec)	0.000054	0.000257	0.001911	0.01227	0.0575	0.12096
O(N^3) version	Iterations(K)	100000	10000	1000	1000	100	100
	Ticks	5471	2196	1344	5843	1504	2845
	Total Time(sec)	5.471	2.196	1.344	5.843	1.504	2.845
	Duration(sec)	0.000055	0.00022	0.001344	0.005843	0.01504	0.02845

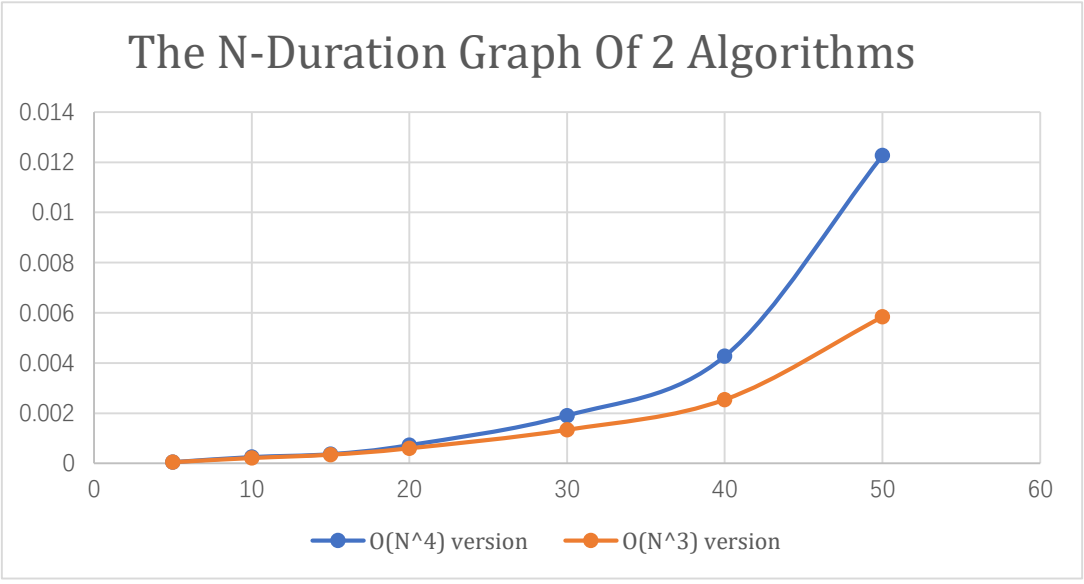
Run time plots

We use duration(sec) as y-axis and data size N as x-axis to plot the three functions in the same figure as following.



The N-Duration graph of the 3 algorithms(N<100)

As we can see, the duration of O(N^6) algorithm grows much faster than other two algorithms when N>50. Under the situation, the duration of O(N^4) algorithm and O(N^3) algorithm can be considered almost the same. Therefore, we let N<50 and discard the O(n^6) curve to plot a more specific partial figure and compare the time efficiency of the remaining two algorithms.

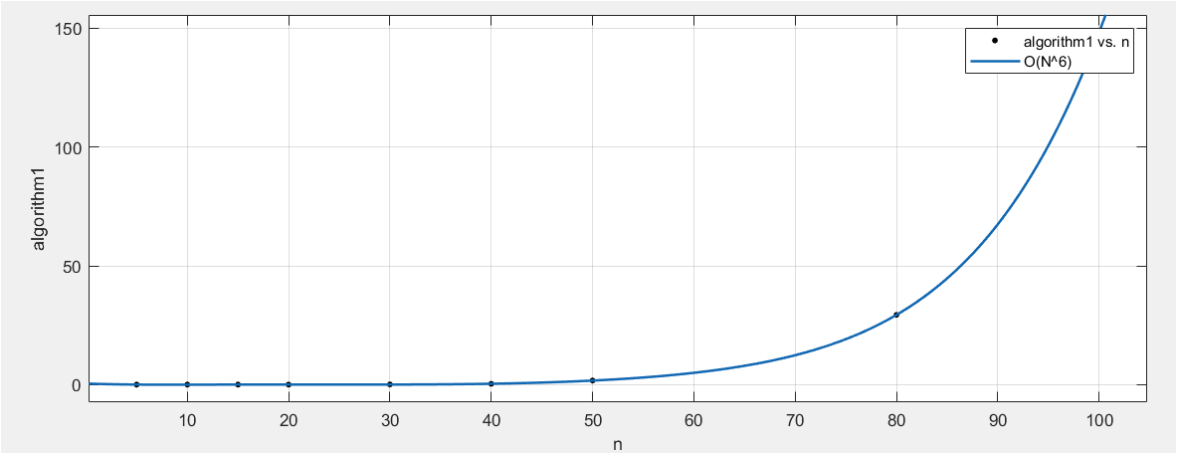


The N-Duration graph of the 2 algorithms(N<50)

According to the figures above, it's obvious that O(N^6) and O(N^4) algorithms take far more time. Meanwhile, all three curves look like polynomial curves. Next we will use Curve Fitting Tool to do function fitting.

Function fitting

The O(N^6) algorithm:



The figure of fitting function

Linear model Poly6:
f(x) = p1*x^6 + p2*x^5 + p3*x^4 + p4*x^3 + p5*x^2 + p6*x + p7

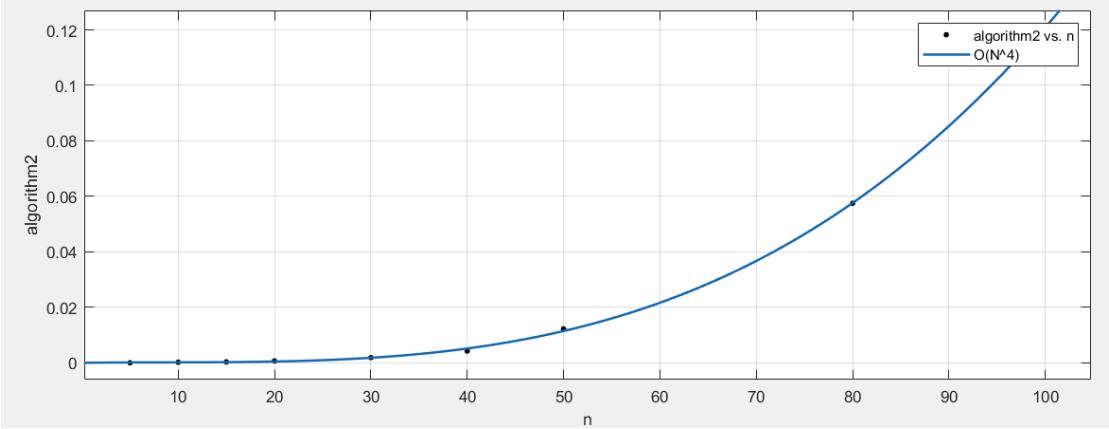
Coefficients (with 95% confidence bounds):

p1 =	1.316e-09	(1.494e-10, 2.482e-09)
p2 =	-2.717e-07	(-5.996e-07, 5.616e-08)
p3 =	2.324e-05	(-1.127e-05, 5.775e-05)
p4 =	-0.0009455	(-0.002649, 0.0007581)
p5 =	0.01864	(-0.02195, 0.05924)
p6 =	-0.1622	(-0.5889, 0.2645)
p7 =	0.4613	(-1.015, 1.937)

Goodness of fit:
SSE: 0.008144
R-square: 1
Adjusted R-square: 1
RMSE: 0.06381

The data of fitting function

The O(N^4) algorithm:



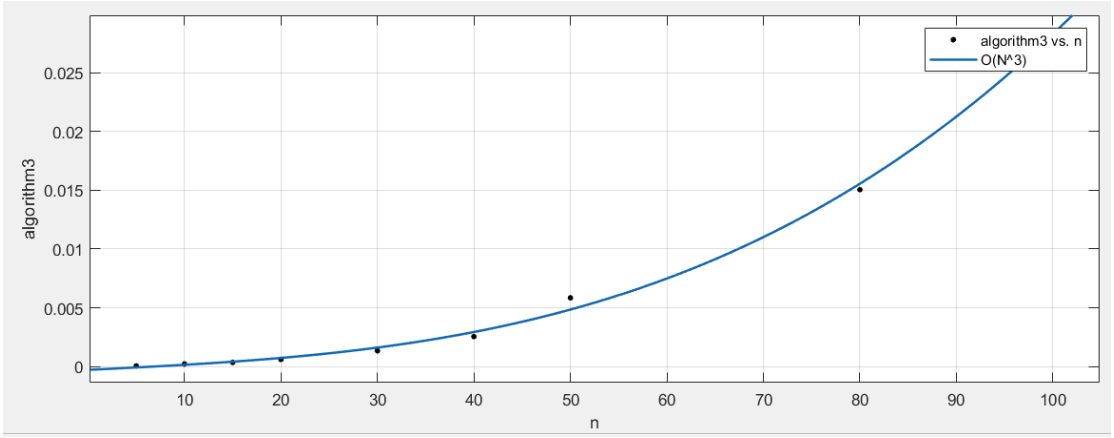
The figure of fitting function

Linear model Poly4:
f(x) = p1*x^4 + p2*x^3 + p3*x^2 + p4*x + p5
Coefficients (with 95% confidence bounds):
p1 = 4.054e-11 (-1.154e-09, 1.235e-09)
p2 = 1.522e-07 (-9.902e-08, 4.033e-07)
p3 = -3.936e-06 (-2.128e-05, 1.341e-05)
p4 = 4.02e-05 (-0.0003977, 0.0004781)
p5 = 5.616e-05 (-0.003072, 0.003184)

Goodness of fit:
SSE: 1.565e-06
R-square: 0.9999
Adjusted R-square: 0.9998
RMSE: 0.0006254

The data of fitting function

The O(N^3) algorithm:



The figure of fitting function

Linear model Poly3:
f(x) = p1*x^3 + p2*x^2 + p3*x + p4
Coefficients (with 95% confidence bounds):
p1 = 2.418e-08 (-9.683e-10, 4.933e-08)
p2 = 3.314e-08 (-3.84e-06, 3.906e-06)
p3 = 4.048e-05 (-0.0001215, 0.0002025)
p4 = -0.000281 (-0.002009, 0.001447)

Goodness of fit:
SSE: 1.547e-06
R-square: 0.9979
Adjusted R-square: 0.9967
RMSE: 0.0005563

The data of fitting function

Chapter 4: Analysis and Comments

Time complexity analysis

Algorithm 1:

At the outer loop we scan all the rows twice to set the top row and the bottom row. At the inner loop we scan all the columns twice to set the top column and the bottom column. Then we scan each submatrix to calculate the sum. In the worst case, it will scan N^6 times and scan once will take constant time. Hence, the time complexity is O(N^6).

```
14      for( TopRow = 0; TopRow < N; TopRow++ )  /* set the top row */
15          for( BottomRow = TopRow; BottomRow < N; BottomRow++)  /* set the bottom row by outer loop */
16              for( LeftCol = 0; LeftCol < N; LeftCol++)   /* set the left col */
17                  for( RightCol = LeftCol; RightCol < N; RightCol++)  /* set the right col by the inner loop */
18                  {
19                      ThisSum = 0;    /* the maximum submatrix sum is 0 if all the integers are negative */
20                      for( i = TopRow; i <= BottomRow; i++)
21                          for( j = LeftCol; j <= RightCol; j++)
22                              ThisSum += A[i][j];     /* calculate the matrix sum with boundary rows and columns */
23
24                      if( ThisSum > MaxSum )
25                          MaxSum = ThisSum;   /* compare ThisSum with MaxSum and if ThisSum > MaxSum,update MaxSum */
26                  }
```

According to the code, we have T(n) = 2*n^4+n^6 = O(N^6)

Algorithm 2:

At the outer loop we scan all the rows twice to set the top row and the bottom row. At the inner loop, firstly we scan all the columns to calculate the sum of elements between top row and bottom row. Then we scan all the submatrix. In the worst case, it will scan N^4 times and scan once will take constant time. Hence, the time complexity is O(N^4).

```
45      for( TopRow = 0; TopRow < N; TopRow++ )
46      {
47          memset( temp, 0, sizeof(temp)); /*Initialize all elements of temp as 0 */
48          for( BottomRow = TopRow; BottomRow < N; BottomRow++)
49          {
50              /* calculate the sum of elements  between TopRow to BottomRow for every column i */
51              for( i = 0; i < N; i++)
52                  temp[i] += A[BottomRow][i];
53
54              for( i = 0; i < N; i++)
55              {
56                  ThisSum = 0;    /* the maximum submatrix sum is 0 if all the integers are negative */
57                  for( j = i; j < N; j++ )
58                  {
59                      ThisSum += temp[j];
60
61                      if( ThisSum > MaxSum )
62                          MaxSum = ThisSum;    /* compare ThisSum with MaxSum and if ThisSum > MaxSum,update MaxSum */
63                  }
64              }
65
66          }
67      }
```

According to the code, we have T(n) = n+2*n^3+2*n^4 = O(N^4)

Algorithm 3:

This algorithm is similar to algorithm 2. The difference lies in scanning the submatrix and calculate the sum. Algorithm 3 uses just a single layer loop rather than a double layer loop, thus saving time and improving the efficiency. In the worst case, it will scan N^3 times and scan once will take constant time. Hence, the time complexity is O(N^3).

```
90      for( TopRow = 0; TopRow < N; TopRow++ )
91      {
92          memset( temp, 0, sizeof(temp)); /*Initialize all elements of temp as 0 */
93          for( BottomRow = TopRow; BottomRow < N; BottomRow++)
94          {
95              /* calculate the sum of elements  between TopRow to BottomRow for every column i */
96              for( i = 0; i < N; i++)
97                  temp[i] += A[BottomRow][i];
98
99              ThisSum = 0;    /* the maximum submatrix sum is 0 if all the integers are negative */
100             for( i = 0; i < N; i++)
101             {
102                 ThisSum += temp[i];
103
104                 if( ThisSum > MaxSum )
105                     MaxSum = ThisSum;
106                 else if ( ThisSum < 0 )
107                     ThisSum = 0;    /* compare ThisSum with MaxSum and if ThisSum > MaxSum,update MaxSum */
108             }
109         }
110     }
```

According to the code, we have T(n) = n+3*n^3 = O(N^3)

Comments

To deal with the Maximum Submatrix Sum Problem, our team have designed three different algorithms: the O(N^6) version, the O(N^4) version, and the O(N^3) version as bonus. Next, we will evaluate the three algorithms from different aspects.

Accuracy

All three algorithms can deal the problem and give correct answers.

Readability

The source codes have a lot of comments to help reading. Besides, the use of Macro substitution makes it easier to understand the whole program. Furthermore, the test code, algorithm code and the header file are all separate, which facilitates quick browsing and finding.

Robustness

All three algorithms can deal with invalid inputs and some special cases. No bugs found yet.

Space complexity

In the program we use pointer to pointer rather than a single static array. So the space complexity of Algorithm 1 is O(1). In Algorithm 2 and Algorithm 3 we use an extra N array to store the ColumnSum, so the space complexity of both is O(N).

Time complexity

We have O(N^6) algorithm, O(N^4) algorithm, and O(N^3) algorithm as bonus. When N<50, all three algorithms are useful. But when N increases, it's apparent that O(N^6) algorithm is not applicable to this problem. In the test we set the maximum value of N as 100. In the fact, when N = 1000, O(N^3) algorithm can still solve the problem but O(N^4) algorithm is not applicable.

In a word, the code is clear and concise. The program is reliable and effective. Particularly, the O(N^3) algorithm(bonus) is elegant and efficient. Of course, there are still some parts that can be improved. We will work hard to find and improve.

Appendix: Source Code (in C)

MaxSubMatrix.h

```
#ifndef MAXSUBMATRIX_H
#define MAXSUBMATRIX_H

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

int MaxSubMatrixSum_A1( const int **A, int N ); /* Algorithm 1 with the time
complexity of O(N^6) */
int MaxSubMatrixSum_A2( const int **A, int N ); /* Algorithm 2 with the time
complexity of O(N^4) */
int MaxSubMatrixSum_Bonus( const int **A, int N );  /* Algorithm 3 with the
time complexity of O(N^3) */

#endif
```

MaxSubMatrix.c

```
#include "MaxSubMatrix.h"

/**
 * @brief  Algorithm similar to Algorithm 1
 * enumerate every SubMatrix and compute its sum
 * @param A secondary point to memory space stores matrix
 * @param N Matrix Rank
 * @return MaxSum MaxSubMatrixSum
 */

int MaxSubMatrixSum_A1( const int **A, int N )
{
    A =(int (*)[N])A; /* cast int** into int[N]* */

    int i, j;
    int ThisSum, MaxSum, TopRow, BottomRow, LeftCol, RightCol;
    MaxSum = 0; /* variable to store the MaxSubMatrixSum */

    for( TopRow = 0; TopRow < N; TopRow++ )  /* set the top row */
        for( BottomRow = TopRow; BottomRow < N; BottomRow++)  /* set the
bottom row by outer loop */
            for( LeftCol = 0; LeftCol < N; LeftCol++)   /* set the left col */
                for( RightCol = LeftCol; RightCol < N; RightCol++)  /* set the
right col by the inner loop */
                {
                    ThisSum = 0;    /* the maximum submatrix sum is 0 if all
the integers are negative */
                    for( i = TopRow; i <= BottomRow; i++)
```

```
                        for( j = LeftCol; j <= RightCol; j++)
                            ThisSum += A[i][j];     /* calculate the matrix
sum with boundary rows and columns */

                    if( ThisSum > MaxSum )
                        MaxSum = ThisSum;    /* compare ThisSum with MaxSum and
if ThisSum > MaxSum,update MaxSum */
                }
    return MaxSum;
}


/**
 * @brief  Use algorithm similar to Algorithm 2
 * Calculate the sum of elements  between TopRow to BottomRow for every column
 * and store in an array, thus compressing 2D array in 1D array
 * then use algorithm 2 to calculate the MaxSubsequenceSum
 * @param A secondary point to memory space stores matrix
 * @param N Matrix Rank
 * @return MaxSum MaxSubMatrixSum
 */

int MaxSubMatrixSum_A2( const int **A, int N )
{
    A =(int (*)[N])A; /* cast int** into int[N]* */

    int i, j;
    int ThisSum, MaxSum;
    int TopRow, BottomRow;
    int temp[N];
    MaxSum = 0; /* variable to store the MaxSubMatrixSum */

    for( TopRow = 0; TopRow < N; TopRow++ )
    {
        memset( temp, 0, sizeof(temp)); /*Initialize all elements of temp as 0
*/
        for( BottomRow = TopRow; BottomRow < N; BottomRow++)
        {
            /* calculate the sum of elements  between TopRow and BottomRow for
every column i */
            for( i = 0; i < N; i++)
                temp[i] += A[BottomRow][i];

            for( i = 0; i < N; i++)
            {
                ThisSum = 0;    /* the maximum submatrix sum is 0 if all the
integers are negative */
                for( j = i; j < N; j++ )
                {
                    ThisSum += temp[j];
```

```
                    if( ThisSum > MaxSum )
                        MaxSum = ThisSum;     /* compare ThisSum with MaxSum
and if ThisSum > MaxSum,update MaxSum */
                }
            }

        }
    }

    return MaxSum;
}



/**
 * @brief  Use algorithm similar to Algorithm 4
 * Calculate the sum of elements  between TopRow to BottomRow for every column
 * and store in an array, thus compressing 2D array into 1D array
 * then use algorithm 4 to calculate the MaxSubsequenceSum
 * @param A secondary point to memory space stores matrix
 * @param N Matrix Rank
 * @return MaxSum MaxSubMatrixSum
 */

int MaxSubMatrixSum_Bonus( const int **A, int N )
{
    A =(int (*)[N])A; /* cast int** into int[N]* */

    int i;
    int ThisSum, MaxSum;
    int TopRow, BottomRow;
    int temp[N];
    MaxSum = 0; /* variable to store the final output */

    for( TopRow = 0; TopRow < N; TopRow++ )
    {
        memset( temp, 0, sizeof(temp)); /*Initialize all elements of temp as 0
*/
        for( BottomRow = TopRow; BottomRow < N; BottomRow++)
        {
            /* calculate the sum of elements  between TopRow to BottomRow for
every column i */
            for( i = 0; i < N; i++)
                temp[i] += A[BottomRow][i];

            ThisSum = 0;     /* the maximum submatrix sum is 0 if all the
integers are negative */
            for( i = 0; i < N; i++)
            {
                ThisSum += temp[i];
```

```
                if( ThisSum > MaxSum )
                    MaxSum = ThisSum;
                else if ( ThisSum < 0 )
                    ThisSum = 0;    /* compare ThisSum with MaxSum and if
ThisSum > MaxSum,update MaxSum */
            }
        }
    }

    return MaxSum;
}
```

main.c

```
#include "MaxSubMatrix.h"
#define true (1)
#define TestDataSize (100000)  /* the scale of the test data */void
CreateRandSample( void );  /* function used to generate test data */

int main(void){

    int i, j, k, choice;
    int n; /* size of matrix */
    int iter; /* iteration number */
    int **A; /* pointer to pointer to matrix */

    clock_t start, stop;
    double total; /* records the total runtime */
    int res;
    char cont; /* judge whether to continue main loop of the test program*/

    FILE *fp;

    CreateRandSample(); /* generate test data stored in test.dat */

    /* open the test data file */
    if( (fp = fopen( "test.dat","r")) == NULL )
    {
        printf( "Open File Failed\n");
        exit(1);
    }

    /* main loop */
    while( true )
    {
        /* select the algorithm to test */
        printf("Please select the algorithm you want to test:\n");
        printf( "1.Algorithm 1 O(N^6)\n2.Algorithm 2 O(N^4)\n3.Algorithm 3
O(N^3)  (Bonus)\n");
        printf( "Your Choice: ");
```

```
        while( true )
        {
            scanf( "%d", &choice);
            if( choice <=3 && choice >=1 ) /* Judge the validity of input */
                break;
            printf( "Invalid Input\nYour Choice: ");

        }
        /* get the matrix size(N) */
        printf( "Please Input the value of N: ");
        while( true )
        {
            scanf( "%d", &n);
            if( n >= 2  && n <= 100 ) /* Judge the validity of matrix size [2-
100] */
                break;
            printf( "Invalid Input\nPlease Input the value of N: ");

        }
        /* get the number of iteration to run */
        printf( "Please Input the numbers of iteration(K): ");
        while( true )
        {
            scanf("%d", &iter);
            if( iter >= 1 ) /* Judge the validity of iteration number */
                break;
            printf( "Invalid Input\nPlease Input the numbers of iteration(K):
");

        }


        /* allocate memory block to store matrix */
        A = (int**)malloc(sizeof(int*)*n);
        /*Judge whether sufficient memory available*/
        if( A == NULL )
        {
            printf( "Out of Space!");
            exit(1);
        }
        /* allocate memory block to store each row of the matrix */
        for( i = 0; i < n; i++)
        {
            A[i] = (int*)malloc(sizeof(int)*n);
            /*Judge whether sufficient memory available*/
            if( A[i] == NULL )
            {
                printf( "Out of Space!");
                exit(1);
            }
        }
```

```
        start = clock(); /* records the ticks at the beginning */

        /* repeat for the times of iteration number */
        for( k = 0; k < iter; k++ ){

            /* read the test data */
            for( i = 0; i < n; i++ )
                for( j = 0;j<n;j++ )
                    fscanf( fp, "%d", &A[i][j]);
            /* choose the algorithm to run */
            switch(choice){
                case 1:
                    res = MaxSubMatrixSum_A1( A, n ); /* Algorithm 1 with the
time complexity of O(N^6) */
                    break;
                case 2:
                    res = MaxSubMatrixSum_A2( A, n ); /* Algorithm 2 with the
time complexity of O(N^4) */
                    break;
                case 3:
                    res = MaxSubMatrixSum_Bonus( A, n ); /* Algorithm better
than the above two with time complexity of O(N^3) */
                    break;
                default:
                    break;
            }
        }

        free( A );

        stop = clock(); /* records the ticks at the end */
        total = ((double)(stop - start))/CLK_TCK; /* get the total runtime
(CLK_TCK = ticks per second) */


        /* Ticks: the number of elapsed ticks during the runtime *
         * Total time : total runtime of repeating the chosen algorithm for
iter times
         * Duration time : average time of the chosen algorithm
         */
        printf(
"============================================================\n");
        printf( "| Iterations(K) | Ticks | Total Time(sec) | Duration(sec)
|\n");
        printf( "|%15d|%7ld|%17lf|%15f|\n", iter, stop-start, total,
total/iter);
        printf(
"============================================================\n");

        /* Input the value of cont */
```

```
        getchar();
        printf( "Continue?(Y/N)");
        scanf( "%c", &cont);
        if( cont != 'Y' && cont != 'y' )
            break;

    }

    return 0;

}


void CreateRandSample( void )
{
    int i;
    FILE *fp = NULL;

    srand((unsigned)time(NULL)); /* set the seed for the random number
generator */

    /* Create the test data file test.dat */
    if( (fp = fopen( "test.dat", "w")) == NULL )
    {
        printf("Create File Failed\n");
        exit(1);
    }

    /* generate 100000 random number from -20 to 20 and write to test.dat */
    for( i = 0; i < TestDataSize; i++)
        fprintf( fp, "%d ", rand()%40-20 );

    fclose(fp);
    return;

}
```

Declaration

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

Duty Assignments:

Programmer : 柴子炜

Tester: 章含挺

Report writer: 章含挺/柴子炜