Performance Measurement (MSS)

Date: 2023-10-08

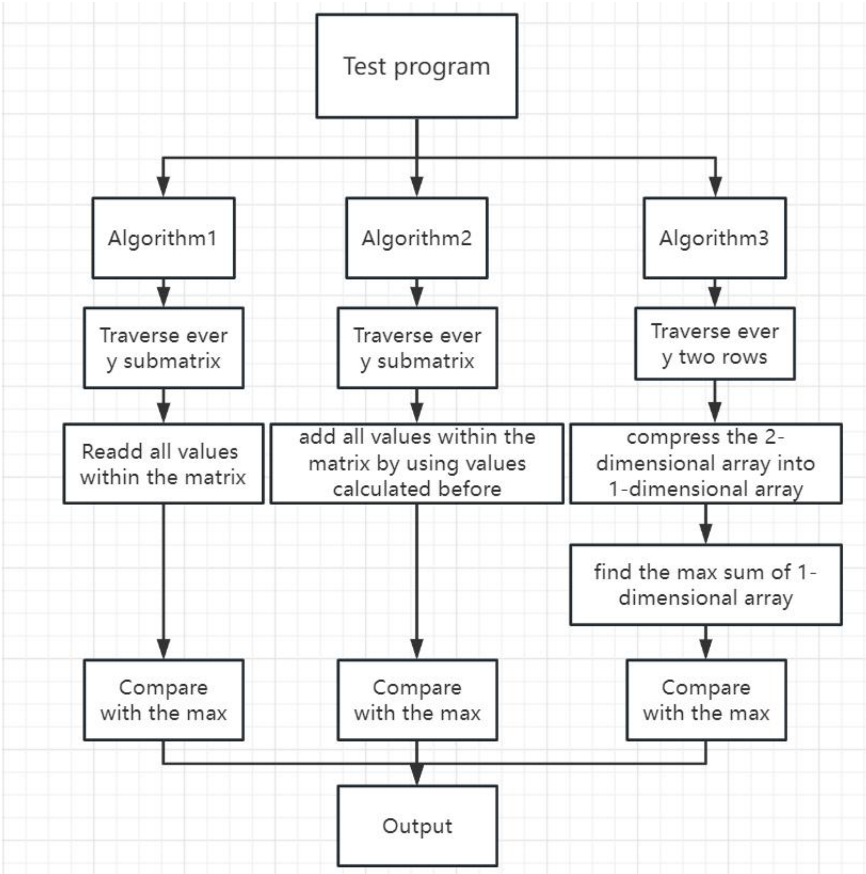
Chapter 1: Introduction

Problem description: For a given $N \times N$ integer matrix, find the maximum sum of its submatrix. For the convenience of algorithm, the maximum sum is 0 if all the integers are negative.

Chapter 2: Algorithm Specification

Main program : test the algorithm by function the algorithm and counting the ticks.

Sketch:



Algorithm 1: the simplest algorithm $O(N^6)$ which computes every possible submatrix sum and find the maximum number.

Pseudo-code of algorithm 1

Declare integer variables ThisSum, MaxSum
Set MaxSum to 0
For i from 0 to N-1
    For j from 0 to N-1
        For k from i to N-1
            For l from j to N-1
                Set ThisSum to 0;
                For m from i to k

```
                    For n from j to l
                        Add A[m][n] to ThisSum;
                    End For
                End For
                If ThisSum is greater than MaxSum
                    Set MaxSum to ThisSum;
                End If
            End For
        End For
    End For
End For
```

Algorithm 2: the improved algorithm $O(N^4)$ which computes possible submatrix sum by adding new part of the next submatrix.

Pseudo-code of algorithm 2

```
Declare integer variables MaxSum and integer 2-dimensional arrays sum[N][N]
For i from 0 to N-1
    For j from 0 to N-1
        For k from i to N-1
            For l from j to N-1
                If k equals i and l equals j
                    Set sum[k][l] to A[k][l]
                Else If k equals i
                    Set sum[k][l] to sum[k][l-1] + A[k][l]
                Else If l equals j
                    Set sum[k][l] to sum[k-1][l] + A[k][l]
                Else
                    Set sum[k][l] to sum[k-1][l] + sum[k][l-1] - sum[k-1][l-1] + A[k][l]
                If sum[k][l] is greater than MaxSum
                    Set MaxSum to sum[k][l]
                End If
            End For
        End For
    End For
End For
```

Algorithm 3: the advanced algorithm $O(N^3)$ which converts the submatrix into a one-dimensional array and solve it using solution in book algorithm 4(namely $O(N)$)

Pseudo-code of algorithm 3

```
Declare integer variables MaxSum, ThisSum
For i from 0 to N-1
    For j from i to N-1
        Set ThisSum to 0
        For l from 0 to N-1
            For k from i to j
                Add A[l][k] to ThisSum
            End For
            If ThisSum is greater than MaxSum
```

```
                    Set MaxSum to sum[k][l]
                Else if ThisSum is less than 0
                    Set ThisSum to 0
                End If
            End For
        End For
End For
```

Chapter 3: Testing Results

Table of test cases:

	N	5	10	30	50	80	100
$O(N^6)$ version	Iterations (K)	300000	10000	100	10	5	1
	Ticks	1346	1379	5505	9157	57216	42079
	Total Time (sec)	1.346	1.379	5.505	9.157	57.216	42.079
	Duration (sec)	0.000004	0.000138	0.05505	0.9157	11.4432	42.079
$O(N^4)$ version	Iterations (K)	10000000	1000000	50000	5000	500	100
	Ticks	12095	15983	51694	37928	23740	11240
	Total Time (sec)	12.095	15.983	51.694	37.928	23.740	11.240
	Duration (sec)	0.000001	0.000016	0.001034	0.007586	0.047480	0.112400
$O(N^3)$ version	Iterations (K)	50000000	5000000	100000	5000	2000	1000
	Ticks	14009	17949	18946	6670	16378	18587
	Total Time (sec)	14.009	17.949	18.946	6.67	16.378	18.587
	Duration (sec)	0.00000028	0.0000036	0.0001895	0.001334	0.008189	0.018587

The purpose of table: compare the performances of the above three functions by measuring the duration

The expected result: the time consumed by the three functions decreases in order of algorithm complexity,and the difference becomes larger as N increases.

The actual behavior of program: as expected,the duration-times figure is in Chapter 4,which perfectly fits the expected result

The current status: pass

Chapter 4: Analysis and Comments

Algorithm 1:

Time Complexity is $O(N^6)$.

Analysis of time complexity of algorithm 1
1.The loop of 'i' and 'j' runs for N times. 2.The loop inside,namely 'k' and 'l' ,iterate from 'i' and 'j' respectively to N-1 ,so the total number of iterations of the innermost loop is approximately $\sum_{\substack{i=0\\j=0}}^{N-1}$ [(N-i-1) * (N-j-1)],which could be small but could also be of size N. 3.The innermost loop, namely 'm' and 'n' ,iterate from 'i' and 'j' respectively to 'k' and 'l',so the total number of iterations of the innermost loop is approximately $\sum$ [(k-i) * (l-j)],which could be small but could also be of size N. Thus the total is $O(N^2 \cdot N^2 \cdot N^2) = O(N^6)$.

Space Complexity is $O(1)$

Analysis of space complexity of algorithm 1
1.'ThisSum','MaxSum', 'i', 'j', 'k', 'l', 'm', and 'n' are integer variables and have a constant space requirement $O(1)$. Hence, the overall space complexity is $O(1)$

Algorithm 2:

Time Complexity is $O(N^4)$.

Analysis of time complexity of algorithm 2
1.The loop of 'i' and 'j' runs for N times. 2.The loop inside,namely 'k' and 'l' ,iterate from 'i' and 'j' respectively to N-1 ,so the

total number of iterations of the innermost loop is approximately $\sum_{\substack{i=0 \\ j=0}}^{N-1}$ [(N-i-1) * (N-j-1)],which could be small but could also be of size N.

Thus the total is $O(N^2 \cdot N^2) = O(N^4)$.

Space Complexity is $O(N^2)$

Analysis of space complexity of algorithm 2
1. 'MaxSum', 'i', 'j', 'k', 'l' are integer variables and have a constant space requirement $O(1)$. 2. The store matrix 'sum' requires $O(N^2)$ space to store the elements. Hence, the overall space complexity is $O(N^2)$

Algorithm 3：

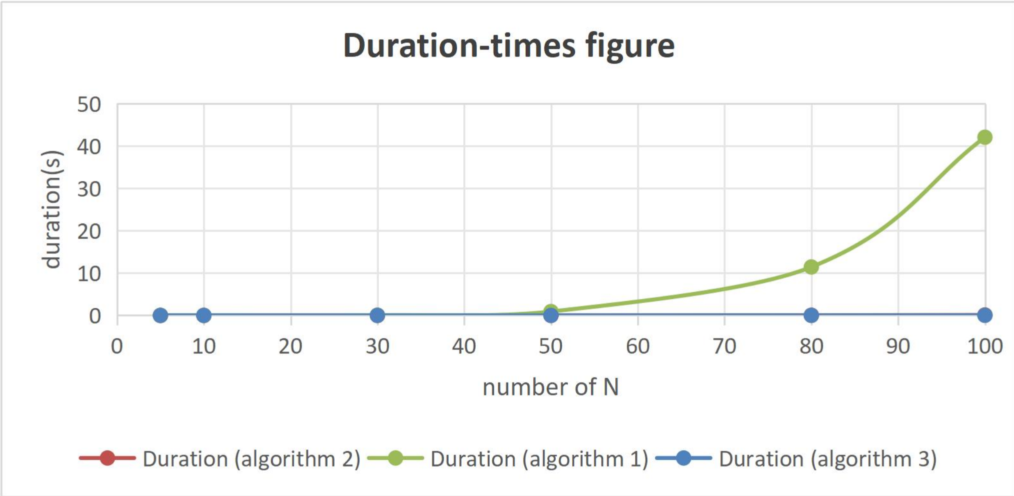
Time Complexity is $O(N^3)$.

Analysis of time complexity of algorithm 3
1.The loop of 'i', 'j' and ‘l’ runs for N times,and ‘l’ and 'j' are parallel. 2.The loop inside,namely 'k' ,iterate from 0 to N-1 ,which could be $O(N)$ Thus the total is $O(N^2 \cdot N) = O(N^3)$.

Space Complexity is $O(N)$

Analysis of space complexity of algorithm 3
1. 'ThisSum', 'MaxSum', 'i', 'j', 'k' and 'l',are integer variables and have a constant space requirement $O(1)$. 2. The store matrix 'B' requires $O(N)$ space to store the elements. Hence, the overall space complexity is $O(N)$

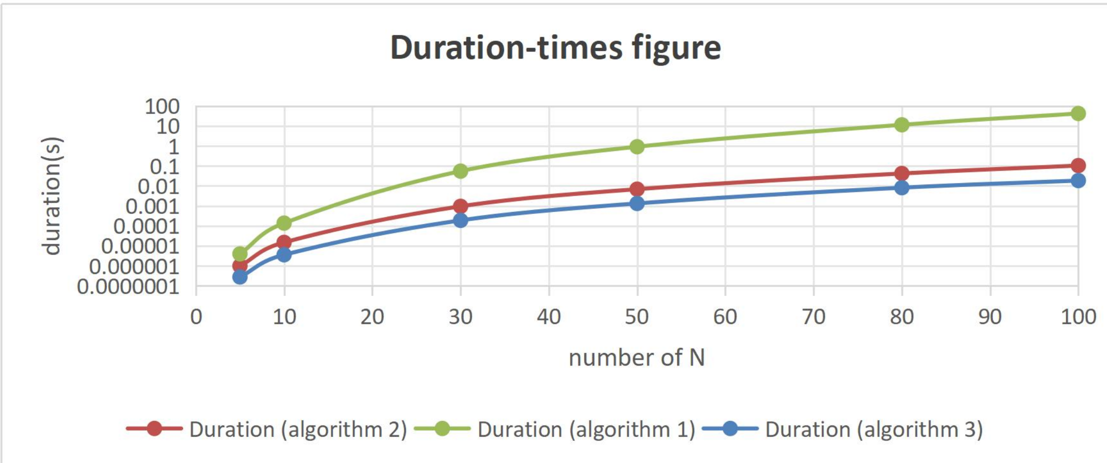
Figure1



Comment:Algorithm2 and 3 reduce the process of adding part,so they are much faster than Algorithm1.

Due to significant differences in the runtime of different functions, in order to significantly represent the trend in a graph, we logarithmically processed the y-axis,and the figure is as follows.

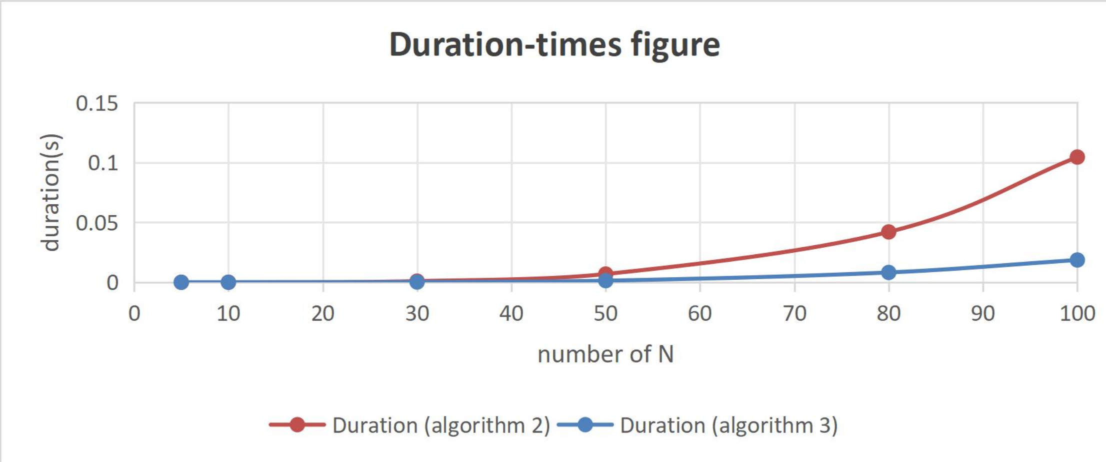
Figure2



Comment:Algorithm2 and 3 have proximity time complexity,and Algorithm1 has big difference with 2 and 3.

In order to significantly represent the difference between algorithm 2 and 3, we could make figure as follows.

Figure3



Comment:Algorithm3 reduce the process of finding part,so it is faster than Algorithm2.

Further possible improvements: we can possibly improve it by segement tree to reduce the time complexity to $O(N^2 * logN)$.(Indeterminacy)

Appendix: Source Code (in C)

Declarations and Definitions

```
/*Import required libraries*/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
/* define processor time (ticks) */
clock_t start, stop;
/* define the interation(K) and number of N */
const int N = 4;
const int K = 1;
/*Declear needed functions*/
int MaxSubSum1( int (*A)[N], int N );
int MaxSubSum2( int (*A)[N], int N );
int MaxSubSum3( int (*A)[N], int N );
```

Test program

```
int main ()
{
	/* define the 2-dimensional array and max */
	 int A[N][N],max1 = 0;
	 /* records the duration and runtime of a function */
	double duration , times;
	/* get the Random number */
	 srand(time(0));
	 /* generate the array elements randomly */
	 for (int i = 0; i < N; i++)
```

```
        {
                for (int j = 0; j < N; j++)
                {
                        /* Assuming the generated random number range is between -50 and 50
*/
                            A[i][j] = rand()%100-50;
                            /* output the generated array */
                            printf("%d ",A[i][j]);
                }
                /* change the row */
                printf("\n");
}
        /* records the ticks at the start of the function call */
        start = clock();
        /* iterate K times to obtain a more accurate duration*/
        for (int i = 0; i < K; i++)
        {
                /* change the function if needed*/
                max1 = MaxSubSum1(A, N);
        }
        /* records the ticks at the end of the function call */
        stop = clock();
        /* use CLK_TCK to get the total time */
        times = ((double)(stop-start))/CLK_TCK;
        /* times / K is the value of duration */
        duration = times / K;
        /* output total time */
        printf("time=%f\n",times);
        /* output iterations(K) */
        printf("iterations=%d\n",K);
        /* output ticks,namely stop-start */
        printf("ticks=%d\n",stop-start);
        /* output duration */
        printf("duration=%f\n",duration);
        /* output maxsum */
        printf("max=%d\n",max1);
        return 0;
}
```

Algorithm 1

```
/* algorithm 1 */
/*simply traverse the matrix */
int MaxSubSum1( int (*A)[N], int N )
{
        /* initiate the varibles */
        int ThisSum = 0, MaxSum = 0, i, j, k, l, m, n;
        /* [i] control the row of start position    */
        for( i = 0; i < N; i++ )
        {
                /* [j] control the column of start position */
                for( j = 0; j < N; j++ )
```

```
        {
            /* [k] control the row of end position */
            for( k = i; k < N; k++ )
            {
                /* [l] control the column of end position */
                for( l = j; l < N; l++ )
                {
                    /* ThisSum is the sum of chosen submatrix */
                    ThisSum = 0;
                    /* add the member of submatrix to ThisSum */
                    /* Traverse submatrix */
                    for( m = i; m <= k; m++ )
                    {
                        for( n = j; n <= l; n++ )
                        {
                            ThisSum += A[m][n];
                        }
                    }
                    /* compare with the max */
                    if( ThisSum > MaxSum )
                        MaxSum = ThisSum;
                }
            }
        }
    }
    return MaxSum;
}
```

Algorithm 2

```
/* algorithm 2 */
/* traverse the matrix but with some skill to minimize the procedure of add */
int MaxSubSum2( int (*A)[N], int N )
{
    /* initiate the varibles */
    int MaxSum = 0, i, j, k, l;
    /* set a 2 dimensional array to store the sum calculated by former loop */
    int sum[N][N];
    /* [i] control the row of start position    */
    for( i = 0; i < N; i++ )
    {
        /* [j] control the column of start position */
        for( j = 0; j < N; j++ )
        {
            /* [k] control the row of end position */
            for( k = i; k < N; k++ )
            {
                /* [l] control the column of end position */
                for( l = j; l < N; l++ )
                {
                    /* sum[k][l] store the sum from [i][j] to [k][l] */
                    /* when no former loop start */
```

```
                    if( k == i && l == j)
                    {
                        /* sum[k][l] equals A[k][l] */
                        sum[k][l] = A[k][l];
                    }
                    /* when no sum[k-1][l],namely the first row */
                    else if ( k == i)
                    {
                        /* sum[k][l] equals the left sum plus A[k][l] */
                        sum[k][l] = sum[k][l-1] + A[k][l];
                    }
                    /* when no sum[k][l-1],namely the first column */
                    else if ( l == j)
                    {
                        /* sum[k][l] equals the upper sum plus A[k][l] */
                        sum[k][l] = sum[k-1][l] + A[k][l];
                    }
                    else
                    {
                        /*upper sum plus left sum minus repeted part and add new part */
                        sum[k][l] = sum[k-1][l] + sum[k][l-1] - sum[k-1][l-1] + A[k][l];
                    }
                    /* compare */
                    if( sum[k][l] > MaxSum )
                        MaxSum = sum[k][l];
                }
            }
        }
    }
    return MaxSum;
}
```

Algorithm 3

```
/* algorithm 3 */
/* compress the 2-dimensional array into 1-dimensional array,use algorithm 4 in book to solve */
int MaxSubSum3( int (*A)[N], int N )
{
    /* initiate the varibles */
    int MaxSum = 0, ThisSum = 0, i, j, k, l;
    /* set a 1-dimensional array to store the sum of value in one column */
    int B[N];
    /* [i] control the start row of submatrix */
    for( i = 0; i < N; i++)
    {
        /* initiate the storeage */
        for( l = 0; l < N; l++)
        {
            B[l]=0;
```

```
        }
        /* [j] control the end row of submatrix */
        for( j = i; j < N; j++)
        {
            /* initiate the sum for 1-dimensional array */
            ThisSum = 0;
            /* from the first column to N column */
            /* Traverse the new row */
            for( k = 0; k < N; k++)
            {
                /* since B[k] has already stored the sum of value in one column
before row j */
                /* if we wanna get the value of B[k],just add the value of A[j][k] */
                B[k] += A[j][k];
                ThisSum += B[k];
                /* compare */
                if( ThisSum > MaxSum)
                    MaxSum = ThisSum;
                /* if less than 0,the former part is useless */
                else if(ThisSum < 0 )
                    ThisSum = 0;
            }
        }
    }
    return MaxSum;
}
```

Declaration

I hereby declare that all the work done in this project titled "MSS" is of my independent effort.