FUNDAMENTALS OF DATA STRUCTURES

Laboratory Project 1
Performance Measurement

Programmer: xxx

Tester:xxx

Report writer: xxx

2018.10.28

Chapter 1: Introduction

In our textbook, several methods for solving the Maximum Subsequence Sum problem are discussed. So now we are going to extend this problem to 2-dimensional.

This project is given an $N \times N$ integer matrix $(a_{ij})_{N\times N}$, tofind the maximum value of $\sum_{k=i}^{m}\sum_{l=j}^{n} a_{ij}$ for all $1 \le i \le m \le N$ and $1 \le j \le n \le N$. For convenience, the maximum submatrix sum is 0 if all the integers are negative.

The simplest method is to compute every possible submatrix sum and find the maximum number. According to our textbook, an algorithm similar to Algorithm 1 given in section 2.4.3 will run in $O(N^6)$, and the one similar to Algorithm 2 in $O(N^4)$. Moreover we have got another algorithm which is indeed better than the above two simple algorithms.

After implementing the three version of algorithms, we can gain the durations to compare the worst case performances of the three functions for N = 5, 10, 30, 80, 100.So that, we can analyze the time and space complexities of them.

Chapter 2: Algorithm Specification

2.1 Specification of algorithm 1: "O（N6）version of algorithm"

C code:

```
int N6_Version(int **a, int N) {
    int row1, column1, row2, column2, row3, column3, currentSum, MaxSum=0;
    for (row1 = 0; row1 < N; row1++) {
        for (column1 = 0; column1 < N; column1++) {
            for (row2 = row1; row2 < N; row2++) {
                for (column2 = column1; column2 < N; column2++) {

                    currentSum = 0;
                    for (row3 = row1; row3 <= row2; row3++)
                        for (column3 = column1; column3 <= column2; column3++) {
                            currentSum += a[row3][column3];
                        }
                    if (currentSum > MaxSum)MaxSum = currentSum;
                }
            }
        }
    }
    return MaxSum;
}
```

Description: This is the most simplest and traditional method to solve this problem. Through compare the value of sub-matrix one by one, we can easily find the maximum. However, this ineffective loop makes the computer compare those for N times at the worst case.

Data Structure: Use integer data type and array data type.

2.2 Specification of algorithm 2: "O（N4）version of algorithm"

C code:

```
int N4_Version(int **a,int N) {
    int row1, column1, row2, column2, currentSum = 0, MaxSum = 0;
        int *Temporary_array = (int *)malloc(sizeof(int)*N);

    for (row1 = 0; row1 < N; row1++) {
        for (column1 = 0; column1 < N; column1++) {

            for (column2 = 0; column2 < N; column2++)Temporary_array[column2] = 0;

            for (row2 = row1; row2 < N; row2++) {
                currentSum = 0;
                for (column2 = column1; column2 < N; column2++) {
                    Temporary_array[column2] += a[row2][column2];
                    currentSum += Temporary_array[column2];
                    if (currentSum > MaxSum)MaxSum = currentSum;
                }
            }
        }
    }
    return MaxSum;
}
```

Description: This is the recursive version of algorithm one which also compare the value of sub-matrix one by one. The worst case is calling the function for N times.

Data Structure: Use integer data type and array data type.

2.3 Specification of algorithm 3: "Bonus version of algorithm"

C code:

```
int Bonus_Version(int **a, int N) {
    int row, column, currentSum = 0, MaxSum = 0, Temporary_row, Temporary_column;
    int **Temporary_array = (int *)malloc(sizeof(int*)*N);
    for (row = 0; row < N; row++)Temporary_array[row] = (int *)malloc(sizeof(int)*N);
        for (row = 0; row < N; row++) {
        for (column = 0; column < N; column++) {
            for (Temporary_row = N-1; Temporary_row >= row; Temporary_row--)Temporary_array[Temporary_row][column] += a[row][column];
        }
    }

    for (row = 0; row < N; row++) {
        for (Temporary_row = row; Temporary_row < N; Temporary_row++) {

                    currentSum = 0;
                    for (column = 0; column < N; column++) {
                    currentSum = Temporary_array[Temporary_row][column] - Temporary_array[row][column] + a[row][column] + currentSum;
                    if (currentSum > MaxSum)MaxSum = currentSum;
                    else if (currentSum < 0)currentSum = 0;
            }


        }
    }
    return MaxSum;
}
```

Description: This algorithm applies the ideology of "online refresh". Firstly define an array to store the sum of a[row1][column] to a[row2][column], then store the sum of a[row1][column] to a[row2][column] into it. Finally, take the "online refresh" way to calculate the Max sum.

Data Structure: Use integer data type, array data type and pointer data type.

Chapter 3: Testing Results

The table below show the time used to process the function as N is 5, 10, 30, 80, 100. Because the funtion runs so quickly that it takes less than a tick, we may repeat the function calls for K times to obtain total run time, which we make from 100 to 10000.

	N	5	10	30	50	80	100
$O(N^6)$ version	Iterations(K)	10000	1000	100	10	1	1
	Ticks	42	120	5335	11091	18749	66946
	Total time(sec)	0.042	0.12	5.335	11.091	18.749	66.946
	Duration(sec)	$4.2x10^{-6}$	$1.2 x10^{-4}$	0.05335	1.109	18.749	66.946
$O(N^4)$ version	Iterations(K)	10000	1000	100	10	1	1
	Ticks	16	16	103	792	49	110
	Total time(sec)	0.016	0.016	0.103	0.792	0.049	0.11
	Duration(sec)	$1.6x10^{-6}$	$1.6x10^{-5}$	0.00103	0.00792	0.049	0.11
$O(N^3)$ version	Iterations(K)	10000	10000	1000	1000	100	100
	Ticks	13	63	111	582	239	405
	Total time(sec)	0.013	0.063	0.111	0.582	0.239	0. 405
	Duration(sec)	$1.3x10^{-6}$	$6.3x10^{-6}$	0.000111	0.000582	0.00239	0. 00405

Fig. 1 Time used to process the function (as N is 5, 10, 30, 80, 100).

Chapter 4: Analysis and Comments

Analysis:

The $O(N^6)$ version of function calculates the sum of every possible sub-matrix to find the maximum sum. It searches for every sub-matrix by locating its first and last elements, thus having a time complexity of $O(N^6)$ and a space complexity of $O(1)$. The $O(N^4)$ version of function saves the current sum into an array. For each element in the matrix, it takes N^2 to find the biggest sum. It has a time complexity of $O(N^4)$ and a space complexity of $O(N)$. The third version is an upgraded and advanced version of the $O(N^4)$ version with a time complexity of $O(N^3)$ and space complexity of $O(N^2)$.

Comments:

The three algorithms our team design fit the demand of time and space complexity. They all can correctly find the maximum matrix sum. They are well designed and perfectly functional. As the table illustrated, when N is small enough, three versions are almost as fast as each other. When N is bigger, the gap between the efficiency of the algorithms becomes more evident, which is because the time complexity of the last two algorithms grows way more dramatically than the first one.

Appendix: Rest Source Code(in C)

```c
int main(void) {

    //define the necessary variables
    clock_t start, stop;
    int row,column,N,N6_Result,N4_Result,Bonus_Result;
    double N6_Duration, N4_Duration,Bonus_Duration;
    printf("please input the size of the matrix:");

    //intput the size of the matrix and Dynamic allocation of arrays
    scanf("%d", &N);
    int **a = (int **)malloc(sizeof(int *)*N);
    for (row = 0; row < N; row++)a[row] = (int *)malloc(sizeof(int)*N);

    //intput the matrix
    for (row = 0; row < N; row++)
        for (column = 0; column < N; column++)
            scanf("%d", &a[row][column]);

    //test programe
    start = clock();
    N6_Result = N6_Version(a, N);
    stop = clock();
    N6_Duration = ((double)(stop - start)) / CLK_TCK;


    start = clock();
    N4_Result = N4_Version(a, N);
    stop = clock();
    N4_Duration = ((double)(stop - start)) / CLK_TCK;

    start = clock();
    Bonus_Result = Bonus_Version(a, N);
    stop = clock();
    Bonus_Duration = ((double)(stop - start)) / CLK_TCK;

    printf("The result of O(N^6) version, O(N^4) version and bonus version are :%d  %d  %d\n", N6_Result, N4_Result, Bonus_Result);
    printf("The run time of O(N^6) version, O(N^4) version and bonus version are :%lfs  %lfs  %lfs\n", N6_Duration, N4_Duration, Bonus_Duration);
    return 0;
```

Declaration

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

Duty Assignments:

Programmer: XXX

Tester: XXX

Report Writer: XXX