Fundamentals of Data Structures

Laboratory Projects

MAXIMUM SUBMATRIX SUM PROBLEM

林家丰，余瑞璟，郭泽均

Date: 2018-10-11

Chapter 1: Introduction

1.1-Problem:

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

1.2-Task:

We are supposed to implement the *O*(*N*^6) and the *O*(*N*^4) versions of algorithms for finding the maximum submatrix sum and analyze the time and space complexities of the above two versions of algorithms.

In this report,we will add the O(N^3) version and analyze the time and space complexities of three versions of algorithms by offering some graphics and instruction.

Chapter 2: Algorithm Specification

2.1-Algorithm1:

Pseudocode:

```
int algorithm1(int **A){
    Assign 0 to MaxSum,
    for(i = 0; i < N; i++)
        for(j = 0; j < N; j++)
            for(x = i; x < N; x++)
                for(y = j; y < N; y++){
                    Assign 0 to TempSum;
                    for(a = i; a <= x; a++){
                        for(b = j; b <= y; b++)
                            TempSum = the sum of the matrix
from[i,j]to[x,y]
                        if(TempSum > MaxSum)
                        MaxSum = TempSum;
                    }
                }
    return MaxSum;}
```

Algorithm1 is easy to think out,we just calculate every possible result to find the biggest sum.Finally we use 6 times circles to solve this problem.

Time Complexity:O(N^6)

Space Complexity:O(1)

Specifications of main data structures: Array.

2.2-Algorithm2:

Pseudocode:

```
int algorithm2(int **A){
    Assign 0 to MaxSum ,TempSum[N],CurrentSum[N];
    for(i = 0; i < N; i++)
        for(j = 0; j < N; j++){
            Assign 0 to TempSum;
            for(x = i; x < N; x++){
               Assign 0 to CurrentSum;
                for(y = j; y < N; y++){
                    CurrentSum[y] =the sum from A[i][j] to A[x][y];
                    if(y > j){
                        CurrentSum[y] += CurrentSum[y-1];
                    }
                    if(x == i){
                        TempSum[y] = CurrentSum[y];
                    }
                    if(x > i){
                        TempSum[y] += CurrentSum[y];
                    }
                    if(TempSum[y] > MaxSum){
                        MaxSum = TempSum[y];
                    }
                }
            }
        }
    return MaxSum;}
```

In algorithm2, we avoid the part of repetitive calculate work to realize a faster algorithm with the time complexity of N^4,as it has four layers of loops.

Time Complexity:O(N^4)

Space Complexity:O(N)

Specifications of main data structures: Array.

2.3-Algorithm3:

Pseudocode:

```
     int algorithm3(int **A)
1    {
 2       Assign 0 to TempSum[N]
 3       Assign 0 to CurrentSum[N];
 4       for(i = 0; i < N; i++){
 5           Assign 0 to TempSum[N]
 6           for(k = 0; k < N; k++){
 7               Assign 0 to CurrentSum[N];
 8               for(j = i; j < N; j++){
 910                 CurrentSum[j] += A[j][k];
11                  if(j > i){
12                      CurrentSum[j] += CurrentSum[j - 1];
13                  }
14                  if(k == 0){
15                      Assign CurrentSum[j] to TempSum[j];
16                  }
17                  if(k > 0){
18                      TempSum[j] += CurrentSum[j];
1920                }
21                  if(TempSum[j] > MaxSum){
22                      Assign TempSum[j] to MaxSum[j];
23                  }else if(TempSum[j] < 0){
24                      Assign 0 to TempSum[j];
25                  }
26              }
27          }
28      }
        return MaxSum;
```

In Algorithm3, we find that we can store the middle calculating result of the process to save the time of repeated calculating.

Time Complexity:O(N^3)

Space Complexity:O(N)

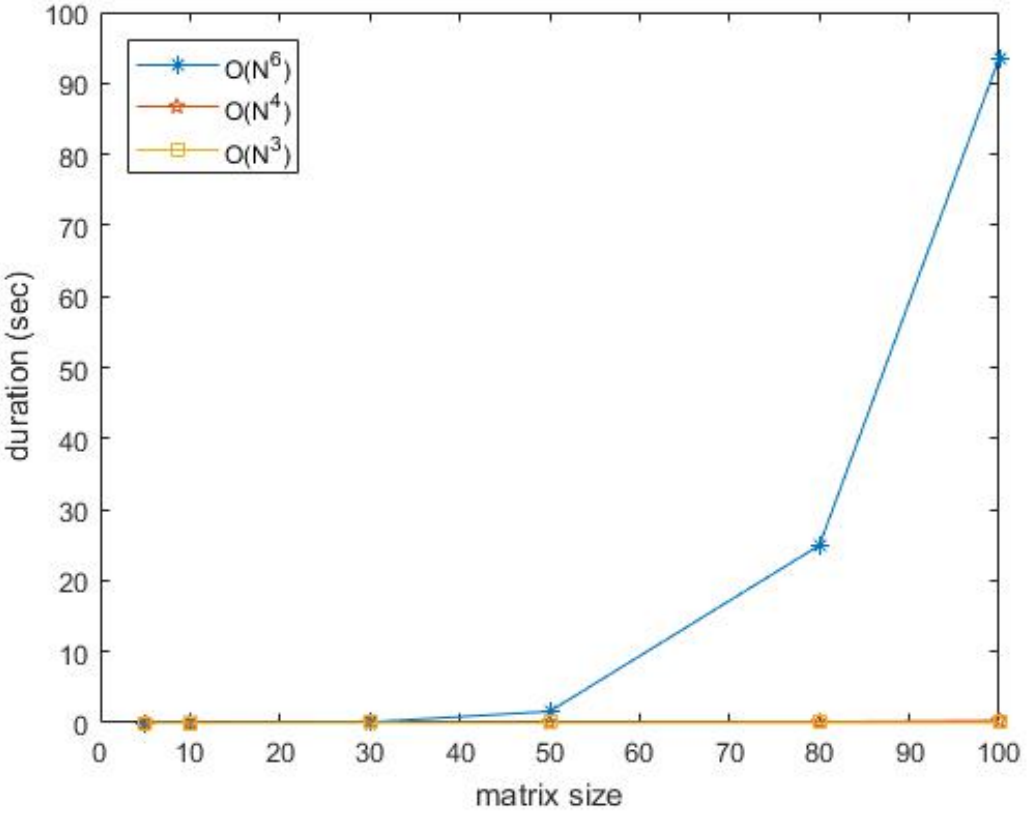
Specifications of main data structures: Array.

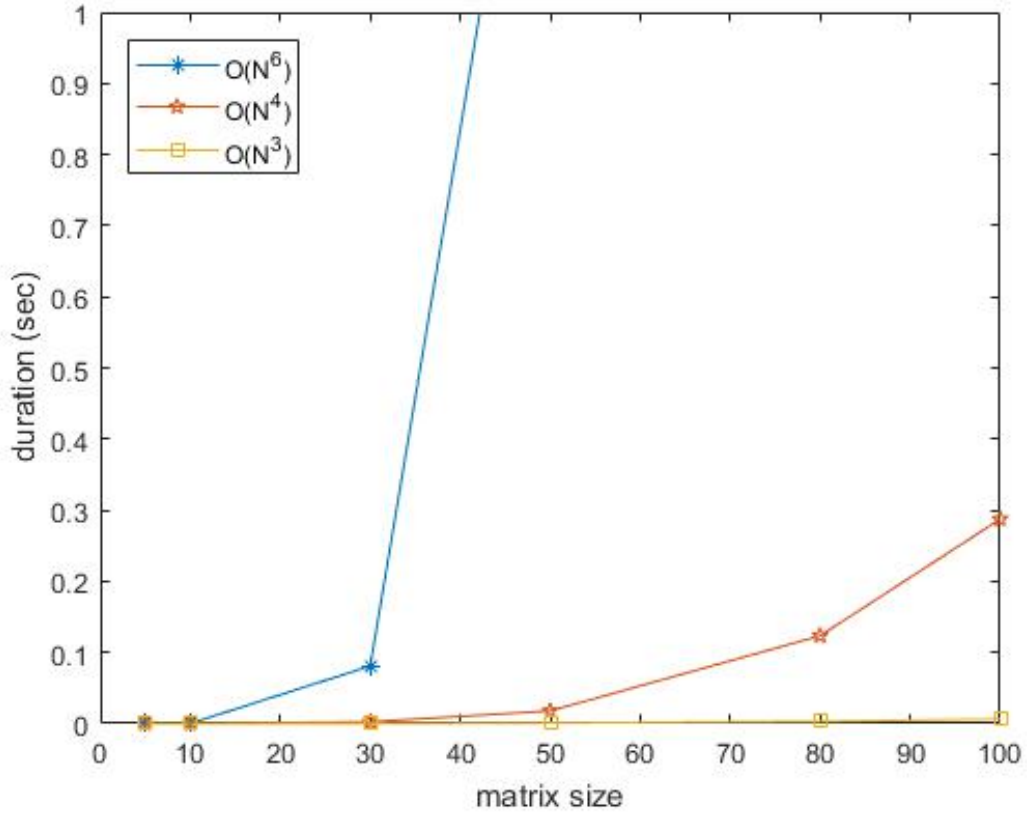
Chapter 3: Testing Results

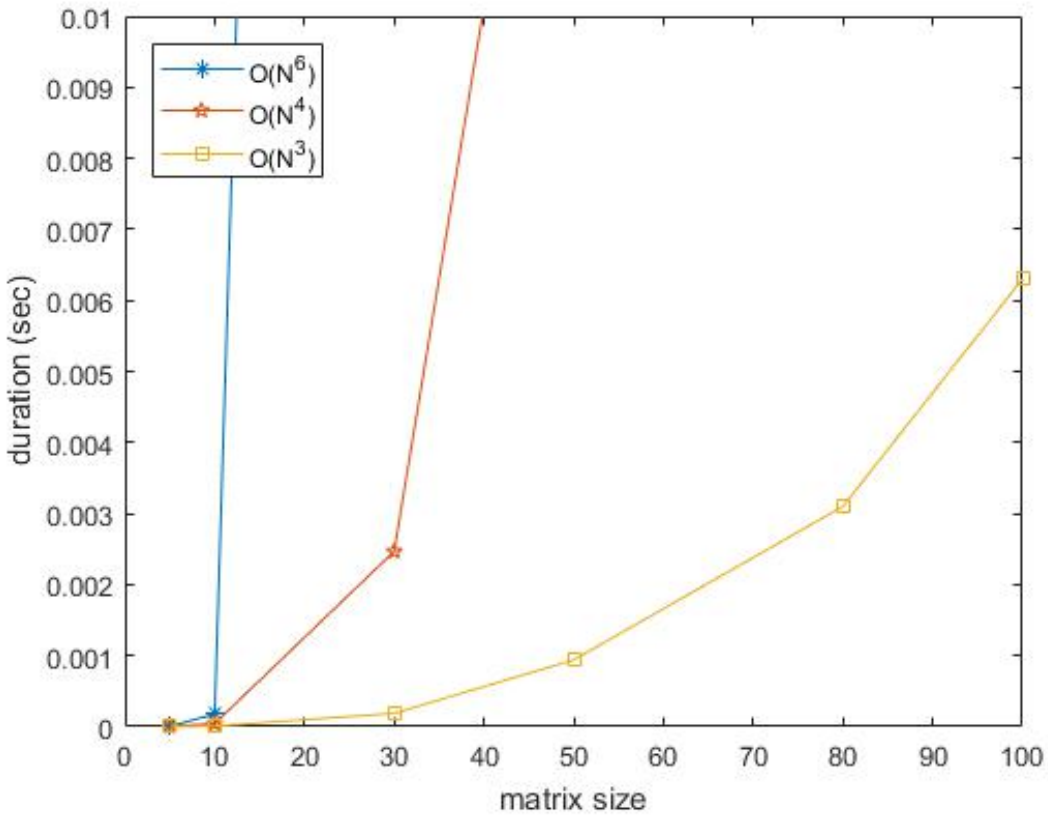
The data in test are generated by a program which randomly generates a specified-size matrix with the interge elements ranging from -16383 to 16384.The main function will get the input data files and iteration times and respectively run three algorithms functions, record the time and finally output the result to a file. Here is the result table.
(1) The result table of three algorithms:

	N	5	10	30	50	80	100
O(N^6) version	Iterations(K)	100000	10000	500	50	20	10
	Ticks	515	1781	40443	79450	499304	935309
	Total Time(sec)	0.515	1.781	40.443	79.450	499.300	935.309
	Duration(sec)	5.15×10-6	1.78×10-4	8.09×10-2	1.59	24.97	93.53
O(N^4) version	Iterations(K)	100000	10000	500	50	20	10
	Ticks	438	437	1234	890	2468	2874
	Total Time(sec)	0.438	0.437	1.234	0.890	2.468	2.874
	Duration(sec)	4.38×10-6	4.37×10-5	2.47×10-3	1.78×10-2	0.12	0.29
O(N^3) version	Iterations(K)	100000	10000	500	50	20	10
	Ticks	140	79	94	47	62	64
	Total Time(sec)	0.140	0.079	0.094	0.047	0.062	0.063
	Duration(sec)	1.40×10-6	7.90×10-6	1.88×10-4	9.40×10-4	3.10×10-3	6.30×10-3

(2) The line charts of input size and run time of each function (plot in differnt range):


O(N^6)
O(N^4)
O(N^3)
duration (sec)
matrix size
0
10
20
30
40
50
60
70
80
90
100


O(N^6)
O(N^4)
O(N^3)
duration (sec)
matrix size
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
10
20
30
40
50
60
70
80
90
100



Analysis in test result:

As we see, the time three algorithms consumed in same size implies that the rank of speed on three algorithms is algorithm1 < algorithm2 < algorithm 3 when the input size are the same. More importantly, there is a huge difference on the growth rate of the consume time of three algorithms. Compared with each other, the growth rate of algorithm 1 is very fast, algorithm takes second place, and algorithm 3 is very slow, which means that with the increase of the matrix size, the difference of consumed time among three algorithms will be bigger and bigger. (When the size is small, the durations are similar, but when the size comes to 100, algorithm1 uses nearly 100 seconds and algorithm 3 uses only nearly 0.006 second) The result meets our theoretical expectations.

Chapter 4: Analysis and Comments

Analysis:

Premise：

①Basic operation such as judge, assignment, definition, arithmetic operation and increment will all consume O(1) time.

② Variable of basic data types in C programming language occupy O(1) memory space.

Method：

①We add all the time consumed by all operations in the program to measure time complexity.

②We calculate the number of all the extra memory space used in program to measure the space complexity.（We only consider and calculate the asymptotic result）

(1) Algorithm1:

Algorithm 1 uses four fold nested loop to traverse all the submatrices of input matrix, which uses O(n^4) steps. Then, for each submatrix, we use double fold nested loop to calculate the sum of elements in the submatrix and compare with current maximum sum and update it if necessary, which uses O(n^2) steps. So the whole algorithm use O(n^6) steps to finish all operations, and the time complexity of it is O(n^6). In addition, we only use a few temporary variables to traverse and store the maximum sum and current sum, so the space complexity is only O(1).The following is the theoretical calculation.

```
1  int algorithm1(int **A){
2      /*Submatrix head*/
3      int i, j;
4      /*Submatrix end*/
5      int x, y;
6      /*Traversal tool*/
7      int a, b;
```

```
8       int MaxSum = 0, TempSum;
9       /*Move the position of head*/
10      for(i = 0; i < N; i++){
11          for(j = 0; j < N; j++){
12              /*Move the position of end*/
13              for(x = i; x < N; x++){
14                  for(y = j; y < N; y++){
15                      TempSum = 0;
16                      /*Traverse addition*/
17                      for(a = i; a <= x; a++){
18                          for(b = j; b <= y; b++){
19                              TempSum += A[a][b];
20                          }
21                          if(TempSum > MaxSum){
22                          MaxSum = TempSum;
23                          }
24                      }
25                  }
26              }
27          }
28      }
29      return MaxSum;}
30
```

Time complexity:

$$O(1)+\sum_{i=0}^{N-1}\sum_{j=0}^{N-1}\sum_{x=i}^{N-1}\sum_{y=j}^{N-1}\left(O(1)+\sum_{a=i}^{x}\sum_{b=j}^{y}O(1)\right)=O(N^6)$$

Space complexity:

$O(1)$ (Just a few temporary variables)

(2) Algorithm2:

Algorithm2 uses double fold nested loop to traverse all the elements, which use O(n^2) steps. Then, use double fold nested loop to traverse all the submatrices that take this element as the upper left corner element, which use O(n^2) steps. To calculate the sum of all elements in submatrix, we use two auxiliary arrays to store previous results. (TempSum[t] stores the sum of the submatrix with (i,j) as the upper left

corner and (x,t) as the lower right corner. CurrentSum[t] stores the sum of the linear vector with (x,y) as the start point and (x,t) as the end point) By using the them, we can only use O(1) step to calculate the sum of current submatrix and update the auxiliary arrays for the following calculations. So we use O(n^4) steps in all, the whole time complexity is O(n^4). Because we use two n-size arrays, the space complexity is O(n). The following is the theoretical calculation.

```
int algorithm2(int **A){
    int i, j;
    int x, y;
    int MaxSum = 0, TempSum[N], CurrentSum[N];
    /*Move the position of head*/
    for(i = 0; i < N; i++){
        for(j = 0; j < N; j++){
            memset(TempSum, 0, N * sizeof(int));
            /*Traverse the matrix*/
            for(x = i; x < N; x++){
                memset(CurrentSum, 0, N * sizeof(int));
                for(y = j; y < N; y++){
                    CurrentSum[y] += A[x][y];
                    /*After the fist line*/
                    if(y > j){
                        CurrentSum[y] += CurrentSum[y-1];
                    }
                    if(x == i){
                        TempSum[y] = CurrentSum[y];
                    }
                    if(x > i){
                        TempSum[y] += CurrentSum[y];
                    }
                    if(TempSum[y] > MaxSum){
                        MaxSum = TempSum[y];
                    }
                }
            }
        }
    }
    return MaxSum;}
```

Time complexity:

$$O(1)+\sum_{i=0}^{N-1}\sum_{j=0}^{N-1}\left(O(N)+\sum_{x=i}^{N-1}\left(O(N)+\sum_{y=j}^{N-1}O(1)\right)\right)=O(N^4)$$

Space complexity:

$$O(1)+2\times O(N)=O(N)$$ (Two N-size extra arrays)

(3) Algorithm3 (Bonus):

Algorithm3 uses double fold nested loop to traverse all the staring row and ending row, which use O(n^2) steps. Then use a loop to traverse the column to use online processing algorithm (which is similar to maximum sum of subarray problem) to find the maximum sum of the submatrix with specified starting row and ending row, which use O(n) steps. Finally, comparing all the result and get the maximum sum. We use two auxiliary arrays to store the previous sum of the submatrices in order to compress the current matrix into vector and process it like one-dimension problem. (CurrentSum[t] stores the sum of the vector with (i,k) as start point and (t,k) as end point, TempSum[t] stores the maximum sum of the matrix with i and j as the boundary rows and t as the right-hand column.) By using them, we can use O(1) step to calculate the sum of current matrix and update auxiliary arrays. So the whole algorithms use O(n^3) steps, the time complexity is O(n^3) and because we use two n-size arrays ,the space complexity is O(n). The following is theoretical calculation.

```
1   int algorithm3(int **A){
2       int i, j, k;
3       int MaxSum = 0, TempSum[N], CurrentSum[N];
4       memset(TempSum, 0, N * sizeof(int));
5       memset(CurrentSum, 0, N * sizeof(int));
6       /*The start row*/
7       for(i = 0; i < N; i++){
8           memset(TempSum, 0, N * sizeof(int));
9           /*The end row*/
10          for(k = 0; k < N; k++){
11              memset(CurrentSum, 0, N * sizeof(int));
```

```
            /*Use a online algrithm by moving the fist column*/
            for(j = i; j < N; j++){
                CurrentSum[j] += A[j][k];
                if(j > i){
                    CurrentSum[j] += CurrentSum[j - 1];
                }
                if(k == 0){
                    TempSum[j] = CurrentSum[j];
                }
                if(k > 0){
                    TempSum[j] += CurrentSum[j];
                }
                if(TempSum[j] > MaxSum){
                    MaxSum = TempSum[j];
                }else if(TempSum[j] < 0){
                    TempSum[j] = 0;
                }
            }
        }
    }
    return MaxSum;}
```

Time complexity:

$$O(N)+\sum_{i=0}^{N-1}\left(O(N)+\sum_{k=0}^{N-1}\left(O(N)+\sum_{j=i}^{N-1}O(1)\right)\right)=O(N^3)$$

Space complexity:

$$O(1)+2\times O(N)=O(N)$$ (Two N-size extra arrays)

Comments:

As we see, there algorithms have different time complexity and space complexity. For time complexity, algorithm 3 is superior to algorithm 2 over algorithm 1. For space complexity, algorithm 1 is better than algorithm 2 and 3. Under comprehensive consideration, generally algorithm 3 is best because in practical use, time complexity is prior to space complexity, and the space complexity of algorithm 2 and 3 is just O(N), which is not large for memory. However, if the concrete situation requires as less the consumed memory as possible, maybe algorithm1 is the best choice if the matrix size isn't too large.

The algorithms' asymptotic time complextiy is fixed, but we can optimize the implementaion code to reduce the constant factor of the highest term in time function and shorten the running time of the program.

Concrete suggestions：

In Algorithm 2 and 3, we can remove all the steps of array-initialization ('memset' functions) and get the same right answers. It is because that the following assignment steps will cover the random data, which means that the array-initialization steps are all superfluous.In this respect, we can remove the code of calling 'memset' functions to optimize program code.

Declaration

We hereby declare that all the work done in this project titled " MAXIMUM SUBMATRIX SUM PROBLEM " is of our independent effort as a group.

Duty Assignments:

Programmer: 郭泽均
Tester: 余瑞璟
Report Writer: 林家丰