Project report 1

—— the Maximum Submatrix Sum problem

Date:2021-10-02

Chapter 1: Introduction

Given an N×N integer matrix $(a_{ij})_{N\times N}$, find the maximum value of $\sum_{k=i}^{m}\sum_{l=j}^{n} a_{kl}$ 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 tasks are:
(1) Implement the $O(N^6)$ and the $O(N^4)$ versions of algorithms (similar to Algorithm 1 and Algorithm 2 given in Section 2.4.3) for finding the maximum submatrix sum;
(2) Analyze the time and space complexities of the above two versions of algorithms;
(3) Measure and compare the performances of the above two functions for N = 5, 10, 30, 50, 80, 100.
(4) **Bonus**: Give a better algorithm. Analyze and prove that your algorithm is indeed better than the above two simple algorithms.

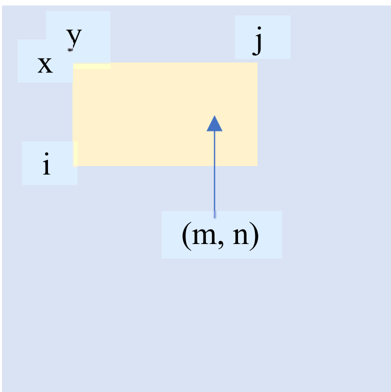
Chapter 2: Algorithm Specification

(1) Algorithm 1: $O(N^6)$

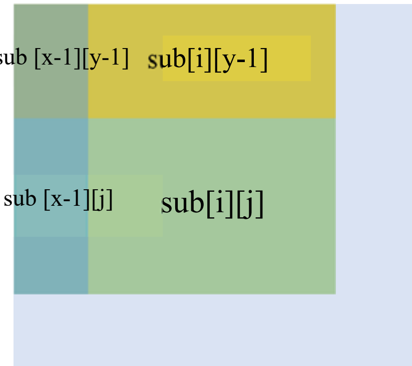
The simplest algorithm is to compute every possible submatrix sums and compare them. So I use four variants x, y, i and j to locate the submatrix, and use two variants m and n to sum the submatrix.

Pseudo code:

```
for x = 0 to N
    for y = 0 to N
        for i = x to N
            for j = y to N
                ThisSum = 0
                for m = x to i
                    for n = y to j
                        ThisSum += A[m][n]
                if ThisSum > MaxSum
                    MaxSum = ThisSum
```



(2) Algorithm 2: $O(N^4)$



To avoid repeated computing, we can save some of the sums. In this algorithm, I save the submatrix sums from (0, 0) to (i, j) in the matrix sub[N][N]. Then, using the including excluding principle, we can find the max submatrix sum within $O(N^4)$.

Pseudo code:

```
for i = 0 to N
    for j = 0 to N
        sub[i][j] = sub[i-1][j] + sub[i][j-1] – sub[i-1][j-1] + A[i][j]
```

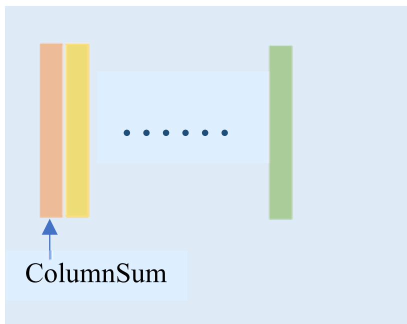
// When i or j = 0, this line (AND SOME LINES IN THE FOLLOWING REPORT) should be changed a little (with an if statement), because -1 is not available as an array subscript. (Complete code is in the appendix.)

```
for x = 0 to N
    for y = 0 to N
    ThisSum = 0
    for i = x to N
        for j = y to N
            ThisSum = sub[i][j] - sub[x-1][j] - sub[i][y-1] + sub[x-1][y-1]
            If ThisSum > MaxSum
                MaxSum = ThisSum
```

(3) Bonus Algorithm: $O(N^3)$

With respect to Algorithm 4 in Section 2.4.4 in the book, which calculates the max subsequence sum in O(N), I think of a better algorithm which can run in $O(N^3)$. It computes the submatrix sum separately in rows and columns. First, save the sums of the columns (from A[0][j] to A[i][j]) in matrix subx[N][N]; then add the columns in each row, using Algorithm 4 in the book.

Pseudo code:




```
for i = 0 to N
    for j = 0 to N
        subx[i][j] = A[i][j] + subx[i-1][j]

for i = 0 to N
    for j = 0 to N
        ThisSum = 0
        for k = 0 to N
            ColumnSum = subx[j][k] – subx[i-1][k]
```

```
ThisSum += ColumnSum
if ThisSum > MaxSum          // Similar to Algorithm 4 in the book
      MaxSum = ThisSum
else if ThisSum < 0
      ThisSum = 0
```

Chapter 3: Testing Results

Algorithm 1, 2 and 3 correspond respectively to function MSS_N6, MSS_N4 and MSS_N3 in my code. I use clock() to measure the performance of the functions. Because the functions run too quickly to be measured on some occasions, I repeat the function calls for K times.

The results are as follows:

	N	5	10	30	50	80	100
$O(N^6)$ version: MSS_N6	Iterations (K)	10000	1000	100	1	1	1
	Ticks	29	99	5000	1015	16587	62536
	Total Time (sec)	0.029	0.099	5.000	1.015	16.587	62.536
	Duration (sec)	0.000003	0.000099	0.05	1.015	16.587	62.536
$O(N^4)$ version: MSS_N4	Iterations (K)	20000	2000	200	1	1	1
	Ticks	21	33	221	15	90	224
	Total Time (sec)	0.021	0.033	0.221	0.015	0.09	0.224
	Duration (sec)	0.000001	0.000017	0.001105	0.015	0.09	0.224
$O(N^3)$ version: MSS_N3	Iterations (K)	80000	8000	800	100	50	10
	Ticks	28	18	47	29	72	24
	Total Time (sec)	0.028	0.018	0.047	0.029	0.072	0.024
	Duration (sec)	0.000000	0.000002	0.000059	0.00029	0.00144	0.0024

Screen shots:

```
N = 5
maxsumN6 = 17   ticks = 29      repeat = 10000  total = 0.029000 secs    time = 0.000003 secs
maxsumN4 = 17   ticks = 21      repeat = 20000  total = 0.021000 secs    time = 0.000001 secs
maxsumN3 = 17   ticks = 28      repeat = 80000  total = 0.028000 secs    time = 0.000000 secs
```

```
N = 10
maxsumN6 = 64   ticks = 99      repeat = 1000   total = 0.099000 secs    time = 0.000099 secs
maxsumN4 = 64   ticks = 33      repeat = 2000   total = 0.033000 secs    time = 0.000017 secs
maxsumN3 = 64   ticks = 18      repeat = 8000   total = 0.018000 secs    time = 0.000002 secs
```

```
N = 30
maxsumN6 = 407  ticks = 5000    repeat = 100    total = 5.000000 secs    time = 0.050000 secs
maxsumN4 = 407  ticks = 221     repeat = 200    total = 0.221000 secs    time = 0.001105 secs
maxsumN3 = 407  ticks = 47      repeat = 800    total = 0.047000 secs    time = 0.000059 secs
```

```
N = 50
maxsumN6 = 600  ticks = 1015    repeat = 1      total = 1.015000 secs    time = 1.015000 secs
maxsumN4 = 600  ticks = 15      repeat = 1      total = 0.015000 secs    time = 0.015000 secs
maxsumN3 = 600  ticks = 29      repeat = 100    total = 0.029000 secs    time = 0.000290 secs
```

```
N = 80
maxsumN6 = 407  ticks = 16587   repeat = 1      total = 16.587000 secs  time = 16.587000 secs
maxsumN4 = 407  ticks = 90      repeat = 1      total = 0.090000 secs   time = 0.090000 secs
maxsumN3 = 407  ticks = 72      repeat = 50     total = 0.072000 secs   time = 0.001440 secs
```

```
N = 100
maxsumN6 = 582  ticks = 62536   repeat = 1      total = 62.536000 secs  time = 62.536000 secs
maxsumN4 = 582  ticks = 224     repeat = 1      total = 0.224000 secs   time = 0.224000 secs
maxsumN3 = 582  ticks = 24      repeat = 10     total = 0.024000 secs   time = 0.002400 secs
```

The results above are from random matrices whose elements are from -9 to 9. The same results of three functions show that they are right. To test some special cases, I put an all 0 matrix and a negative matrix (elements from -9 to -1) in the annotation below the normal matrix. I have tested these special cases:

Cases	Expected result	Result	Current status
Normal (-9 to 9)	Depends on the matrix	Depends on the matrix	Pass
All 0	0	0	Pass
Negative (-9 to -1)	0	0	Pass

Chapter 4: Analysis and Comments

Analyze the time and space complexities of the three algorithms:

(1) MSS_N6: $T(N) = N \times N \times N \times N \times N \times N = O(N^6)$

$S(N) = O(N^2)$

(2) MSS_N4: $T(N) = O(N^2) + O(N^4) = O(N^4)$

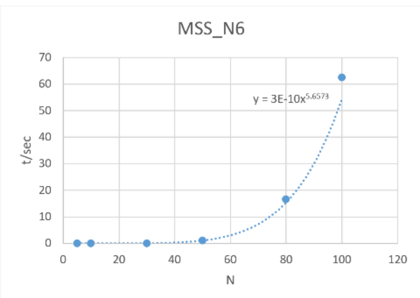
$S(N) = O(N^2) + O(N^2) = O(N^2)$

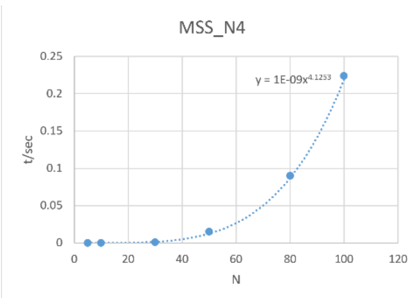
(3) MSS_N3: $T(N) = O(N^2) + O(N^3) = O(N^3)$

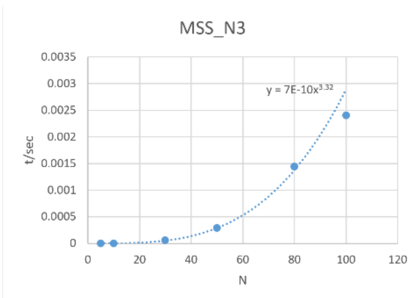
$S(N) = O(N^2) + O(N^2) = O(N^2)$

In order to check the time complexities, I input the data to Excel and generated their line charts.

Here are the line charts of the testing results above:







From the fitting curves, we can find:

$$MSS\_N6: t \propto N^{5.6573}$$
$$MSS\_N4: t \propto N^{4.1253}$$
$$MSS\_N3: t \propto N^{3.3200}$$

It is very close to the calculated time complexities.

In conclusion, with N increasing, the running time of $O(N^6)$ algorithm grows too rapidly for us to wait, while $O(N^4)$ and $O(N^3)$ algorithms are much better. When we sum the submatrices, we can save the sums to avoid repeated computing, reducing the time complexities of the functions.

Appendix: Source Code (in C)

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

clock_t start,stop;                 // record the ticks at the beginning and the end
double duration,once;               // record the total time and time of the functions
int ticks;                          // records the number of ticks
const int N=10;                     // the size of the matrices


int MSS_N6(const int A[][N],int N)       // the O(N^6) Algorithm is the simplest one
{      // which computes every possible submatrix sum and find the maximum number.
   int ThisSum,MaxSum,x,y,i,j,m,n;

   MaxSum = 0;                                        // initialize the MaxSum
   for(x=0;x<N;x++){
      for(y=0;y<N;y++){  // x and y are used to locate the beginning of the submatrix
         for(i=x;i<N;i++){
            for(j=y;j<N;j++){       // i and j are used to locate the end of the submatrix
               ThisSum = 0;
               for(m=x;m<=i;m++){
                              // m and n are used to locate every element of the submatrix
                  for(n=y;n<=j;n++){
                     ThisSum += A[m][n];
                                          // compute the submatrix sum from (x,y) to (i,j)
                  }
               }
               if(ThisSum > MaxSum)
                  MaxSum = ThisSum;       //compare all the sums to find the max sum
            }
         }
      }
   }
```

```
    return MaxSum;
}


int MSS_N4(const int A[][N],int N)
                                      // use the prefix sum to obtain a O(N^4) Algorithm
{                                     // T(N) = O(N^2) + O(N^4) = O(N^4)
    int MaxSum = 0;
    int ThisSum;

    int sub[N][N];
          // First save the submatrix sum from (0,0) to (i,j), which is called prefix sum.
    for(int i=0;i<N;i++){
                          // If i or j == 0, i-1 or j-1 are not available as array subscripts,
        for(int j=0;j<N;j++){ // so we should change the way to calculate the prefix sum.
            if(!i && !j)
                sub[i][j] = A[i][j];
                // When i == 0 and j == 0, the prefix sum equals the first element A[0][0]
            else if(!i && j)
                sub[i][j] = sub[i][j-1] + A[i][j];
            else if(i && !j)
                sub[i][j] = sub[i-1][j] + A[i][j];
            else
                sub[i][j] = sub[i-1][j] + sub[i][j-1] - sub[i-1][j-1] + A[i][j];

    // Use the including excluding principle to calculate prefix sums on usual occasions.
        }
    }

    for(int x=0;x<N;x++){
        for(int y=0;y<N;y++){
                                // x and y are used to locate the beginning of the submatrix
            ThisSum = 0;
            for(int i=x;i<N;i++){       // i and j are used to locate the end of the submatrix
                for(int j=y;j<N;j++){
                    if(!x && !y)
                            // If x or y == 0, x-1 or y-1 are not available as array subscripts.
                        ThisSum = sub[i][j];
                          // When x == 0 and y == 0, the submatrix sum equals the prefix sum.
                    else if(!x && y)
                        ThisSum = sub[i][j] - sub[i][y-1];
                    else if(x && !y)
                        ThisSum = sub[i][j] - sub[x-1][j];
```

```
                ThisSum = sub[i][j] - sub[x-1][j] - sub[i][y-1] + sub[x-1][y-1];
// Use the including excluding principle to calculate submatrix sums with prefix sums.
            if(ThisSum > MaxSum)
                MaxSum = ThisSum;
            }
        }
    }
  }

  return MaxSum;
}


int MSS_N3(const int A[][N],int N)                       // Bonus: a better algorithm
{                                                    // T(N) = O(N^2) + O(N^3) = O(N^3)
  int MaxSum = 0;                 // compute the sums of columns and rows separately
  int ThisSum;

                                                   // First, calculate the sums of columns.
  int subx[N][N];          // The array subx[][] records the sum from A[0][j] to A[i][j].
  for(int i=0;i<N;i++){
    for(int j=0;j<N;j++){          // If i == 0, i-1 is not available as array subscripts.
      if(i)
        subx[i][j] = A[i][j] + subx[i-1][j];
      else
        subx[i][j] = A[i][j];
    }
  }

  for(int i=0;i<N;i++){
                                         // Compare all the ColumnSums in the ROW,
                                   // using an algorithm similar to Algorithm 4 in the book.
    for(int j=i;j<N;j++){    // i and j locate the beginning and the end of ColumnSum.
      ThisSum = 0;                                          // reset ThisSum every loop
      for(int k=0;k<N;k++){         // k locates the horizontal ordinate of the matrix.
        int ColumnSum;      // ColumnSum records the sums from A[i][k] to A[j][k].
        if(i)
          ColumnSum = subx[j][k] - subx[i-1][k];
        else
          ColumnSum = subx[j][k];                                  // in case i == 0

        ThisSum += ColumnSum;
      // When ThisSum < 0, reset ThisSum.(Similar to Algorithm 4 in Section 2.4.4)
```

```
            if(ThisSum > MaxSum)
               MaxSum = ThisSum;
            else if(ThisSum < 0)
               ThisSum = 0;
         }
      }
   }

   return MaxSum;
}

int main()
{
   srand(time(0));                     // make sure the numbers are true random
   int a[N][N];
   for(int i=0;i<N;i++){
      for(int j=0;j<N;j++){
         a[i][j] = rand()%19 - 9;       // the numbers are from -9 to 9 (a normal matrix)
         // a[i][j] = 0;                          // a 0 matrix (results should be 0)
         // a[i][j] = rand()%8 - 10;
                         // a negative matrix from -9 to -1 (results should be 0)
         printf("%3d",a[i][j]);
      }
      printf("\n");
   }
   printf("\n");
                                        // generate a random matrix of size N*N
   printf("N = %d\n",N);                          // show the size of the matrix

   // The function may run too fast to be measured,
   // so I repeat the function calls for K times.

   // test Algorithm 1,T(N)=O(N^6)
   start = clock();
   int maxsumN6 = MSS_N6(a,N);                          // the max submatrix sum
   int K_N6 = 1;                             // repeat the function calls for K times
   for(int t=0;t<K_N6 - 1;t++){
      MSS_N6(a,N);
   }
   stop = clock();
   ticks = stop - start;                                     // the number of ticks
   duration = ((double)(stop - start))/CLK_TCK;           // the total times of K calls
   once = duration/K_N6;                      // the more accurate time for a single run
   printf("maxsumN6 = %d\tticks = %d\trepeat = %d \ttotal = %f secs\ttime = %f sec
```

```
s\n",maxsumN6,ticks,K_N6,duration,once);
   // print the result and the time

   // test Algorithm 2,T(N)=O(N^4)
   start = clock();
                                              // details are the same as the former paragraph
   int maxsumN4 = MSS_N4(a,N);
   int K_N4 = 1;        // K should be larger than Algorithm 1, since this one runs faste
r
   for(int t=0;t<K_N4;t++){
      MSS_N4(a,N);
   }
   stop = clock();
   ticks = stop - start;
   duration = ((double)(stop - start))/CLK_TCK;
   once = duration/K_N4;
   printf("maxsumN4 = %d\tticks = %d\trepeat = %d \ttotal = %f secs\ttime = %f sec
s\n",maxsumN4,ticks,K_N4,duration,once);

   // test Algorithm 3,T(N)=O(N^3)
   start = clock();
                                              // details are the same as the former paragraph
   int maxsumN3 = MSS_N3(a,N);
   int K_N3 = 10;                  // K should be the largest, since this one runs faster
                                        // than both Algorithm 1 and Algorithm 2
   for(int t=0;t<K_N3;t++){
      MSS_N3(a,N);
   }
   stop = clock();
   ticks = stop - start;
   duration = ((double)(stop - start))/CLK_TCK;
   once = duration/K_N3;
   printf("maxsumN3 = %d\tticks = %d\trepeat = %d \ttotal = %f secs\ttime = %f sec
s\n",maxsumN3,ticks,K_N3,duration,once);

   // compare the results to check if the Algorithms are right
   // compare the time to decide which one is faster
}
```

Declaration:

I hereby declare that all the work done in this project titled " Project report 1 —— the Maximum Submatrix Sum problem" is of my independent effort.