Maximum Contiguous Subarray Sum Problems

Contents

1 Abstract **2**

2 Part 1: Maximum Subsequence Sum Problem **2**

2.1 Problem Formulation 2

2.2 Algorithms 2

2.2.1 Brute-Force Search Algorithm 1 3

2.2.2 Brute-Force Search Algorithm 2 3

2.2.3 Divide&Conquer (DC) 3

2.2.4 Dynamic Programming (DP) 5

2.3 Algorithm Comparison 6

2.3.1 Time Complexity Comparison 6

2.3.2 Divide&Conquer V.S. Dynamic Programming 6

3 Part 2: SubMatrix, SubCube and M Subsequence Problems **7**

3.1 SubMatrix 7

3.2 SubCube 8

3.3 M Subsequence 9

4 Conclusion **10**

5 Appendix **11**

1 Abstract

In this project, I intend to investigate a class of problems which are aimed at finding a contiguous subarray whose sum of elements is maximized. The prototype of this class of problems is the maximum subsequence sum problem. Based on the investigation on this prototype problem, then I have explored how to extend the Dynamic Programming (DP) technique to solve other more complicated problems. Through this process, I expect to achieve a deeper understanding of DP.

To be more specific, I have done two parts of work:

1. Investigate the maximum subsequence sum problem. I have implemented four algorithms: two brute-force search algorithms, Divide&Conquer (DC) and DP. I have analyzed the time complexity of these four algorithms. In addition, a brief comparison between DC and DP is also included.

2. As an extension of the work in the first part, I have applied the DP technique to design algorithms to extract maximum submatrix, subcube and M subsequences.

2 Part 1: Maximum Subsequence Sum Problem

2.1 Problem Formulation

Given a sequence of n integers $A(1)...A(n)$, which could be positive or negative. Determine a contiguous subsequence $A(i)...A(j)$ for which the sum of elements is maximized among all possible contiguous subsequences. If all elements in a subsequence are negative integers, the maximum subsequence sum is set to be 0.

I choose this problem for a couple of reasons. First, we are easy to produce artificial data with random generators. Secondly, it is also convenient to control the size of the problem, the length of the sequence. Hence, this problem provides a good model for time complexity comparison among different algorithms. Thirdly, the brute-force search algorithm is feasible and easy to implement, so that the correct answer can be known. In this way, we will be able to know whether the algorithms are implemented correctly.

2.2 Algorithms

All algorithms are implemented in MATLAB. Since MATLAB has provided many useful functions, codes for some algorithms are pretty short. Therefore, I have not written pseudo-code for the report, and I will just paste some short codes in the report for explanation purposes.

2.2.1 Brute-Force Search Algorithm 1

The first straightforward idea is to enumerate all possible subsequences, which is implemented by two loops to enumerate all pairs of start index and end index. Then a nested loop, the most inner one, is used to calculate the sum of the subsequence. The code for this algorithm is shown in Figure 1.

```
1 -   N=10000; %N is the number of integers in the original sequence
2 -   A=randi([-100,100],1,N); %generate N random integers drawn from [-100,100]
3 -   maxSum=0; %max subsequence sum
4 -   beginIndex=1; %extracted subsequence begin index
5 -   endIndex=1; %extracted subseqeunce end index
6 -   for i=1:N  %loops on i and j to enumerate all possible subsequences
7 -       for j=i:N
8 -           tempSum=0;
9 -           for k=i:j %a nested loop to calculate the sum of the subsequence
10 -              tempSum=tempSum+A(k);
11 -          if (tempSum>maxSum)
12 -              maxSum=tempSum;
13 -              beginIndex=i;
14 -              endIndex=j;
15 -          end
16 -          end
17 -      end
18 -  end
19 -  dlmwrite('A.txt',A); %for comparison with other algorithms, we store sequence A
```

Figure 1: Code for Brute-Force Search Algorithm 1

Complexity Analysis There are three loops, indexed by i,j and k, respectively. Each of them executes $O(n)$ times. So, the total time complexity is $O(n^3)$.

2.2.2 Brute-Force Search Algorithm 2

The second algorithm is also to enumerate all possible subsequences. But it is a little smarter than the first one, by calculating the sum of $A(i)...A(j+1)$ by adding $A(j+1)$ to the sum of $A(i)...A(j)$. In this way, the most inner loop is removed. The code for this algorithm is shown in Figure 2.

Complexity Analysis There are two loops, each of which executes $O(n)$ times. Thus, the total time complexity is $O(n^2)$.

2.2.3 Divide&Conquer (DC)

The idea behind DC is to recursively divide the original problem into two almost equal halves, until the subproblems can be solved instantly, then combine the solutions of subproblems to obtain the solution for the original problem. In our problem, we recursively divide a longer sequence into two halves until there is only one integer in each subsequence. To merge two subsequences, say subsequence A and B, we compare the maximum subsequence sum of A and B with the maximum sum of subsequence that bridges

```
1 -    A=dlmread('A.txt'); %read in the sequence A in algorithm 1
2 -    N=10000;
3 -    maxSum=0;
4 -    beginIndex=1;
5 -    endIndex=1;
6 -    for i=1:N
7 -        tempSum=0;
8 -        for j=i:N
9 -            tempSum=tempSum+A(j); %the smart part
10 -           if (tempSum>maxSum)
11 -               maxSum=tempSum;
12 -               beginIndex=i;
13 -               endIndex=j;
14 -           end
15 -       end
16 -   end
```

Figure 2: Code for Brute-Force Search Algorithm 2

A and B. Figure 3 shows how we determine the bridging subsequence. Since the subsequence needs to be contiguous, **the bridging subsequence surely includes the right most element of subsequence A and left most element of subsequence B**. Therefore, we can extend from these two points toward the ends to find the maximum sum bridging subsequence that includes these two points.

For the implementation of the DC algorithm, I have written a recursive function. I append the algorithm to the end of this report, since it is a little bit long (Figure 12).

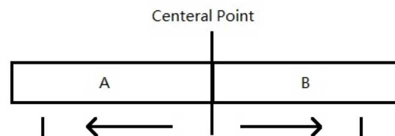

Figure 3: Determine the Bridging Subsequence

Complexity Analysis For each merging step, we need to calculate the bridging maximum subsequence by traversing the merged sequence once in $O(n)$ time. Thus, we have equations as follows:

$$T(n) = T(n/2) + T(n/2) + O(n) \tag{1}$$

$$T(n) = 2 \times T(n/2) + O(n) \tag{2}$$

$$T(n) = 2 \times \{2 \times T(n/4) + O(n/2)\} + O(n) \tag{3}$$

$$T(n) = 4 \times T(n/4) + 2 \times O(n) \quad (4)$$

$$... = ... \quad (5)$$

$$T(n) = n \times T(n/n) + logn \times O(n) \quad (6)$$

$$T(n) = O(nlogn) \quad (7)$$

The total time complexity is: $O(nlogn)$.

2.2.4 Dynamic Programming (DP)

Similar to DC technique, DP is also to synthesize the solution for big problems with the solutions of smaller problems. The core idea underlying DP is to develop a recursion function to transfer from one state to another. Suppose we have known the maximum subsequence sum for the first i elements ($A(1)...A(i)$). For sequence $A(1)...A(i+1)$, we need to determine whether the maximum subsequence includes element $A(i+1)$ or not. If it is, the maximum subsequence for the first $i+1$ elements is a subsequence ended with element $A(i)$. Otherwise, the maximum subsequence for the first $i+1$ elements is the same as that of the first i elements. The recursion function is defined as follows:

$$M(i) = \max\{M(i-1), tempSum(i)\} \quad (8)$$

where $tempSum(i)$ is the maximum subsequence sum ended with element $A(i)$.

The code for DP algorithm is shown in Figure 4.

```
1 -  A=dlmread('A.txt'); %read in the sequence A
2 -  N=100000; %N is the number of numbers in the array
3 -  maxSum=0; %store the maximum sum of the (i-1) numbers
4 -  tempSum=0; %store the maximum sum of the subsequence that includes i-th number
5 -  beginIndex=0;
6 -  endIndex=0;
7 -  for i=1:N
8 -      if (tempSum>0)
9 -          tempSum=tempSum+A(i); %the subsequence can be extend on the left side
10 -     else
11 -         tempSum=A(i); %start a new subsequence
12 -         beginIndex=i;
13 -     end
14 -     if (tempSum>maxSum)
15 -         maxSum=tempSum;
16 -         endIndex=i;
17 -     end
18 - end
```

Figure 4: Code for Dynamic Programming Algorithm

Complexity Analysis We only need to traverse the original sequence once. So, the time complexity is $O(n)$.

2.3 Algorithm Comparison

2.3.1 Time Complexity Comparison

$Big-O$ Complexity Analysis The $Big-O$ complexities of all algorithms are summarized in Table 1. This theoretical analysis shows that DP is the fastest algorithm. It has reduced the time complexity from $O(n^3)$ to a linear time $O(n)$. The complexity of DC is in the middle, better than naive algorithms and worse than DP.

Algorithm	Time Complexity
Brute-Force Search Algorithm 1	$O(n^3)$
Brute-Force Search Algorithm 2	$O(n^2)$
Divide&Conquer (DC)	$O(nlogn)$
Dynamic Programming (DP)	$O(n)$

Table 1: Theoretical Time Complexity Comparison

Real Running Time Analysis We have tested the four algorithms under different problem size n. Results are shown in Table 2. We can see that when n is small ($n = 10^2$), the brute-force search algorithm 2 is the fastest one. The DC is the slowest for this situation. This may be because the running time of DC algorithm contains more overhead in the form of recursive function calls. As the n increases, say $n = 10^4$, the DC and DP algorithms start to outperform the brute-force algorithms. This observation tells us that the $Big-O$ analysis only indicates that an algorithm with lower growth rate is faster than an algorithm with higher growth rate when the input size is very large. When the size is not large enough, this may be not the truth.

Algorithm	n=10^2	n=10^3	n=10^4
Brute-Force Search Algorithm 1	0.008	2.031	1895.581
Brute-Force Search Algorithm 2	0.003	0.009	0.583
Divide&Conquer	0.009	0.027	0.204
Dynamic Programming	0.004	0.004	0.004

Table 2: Real Execution Time (s)

2.3.2 Divide&Conquer V.S. Dynamic Programming

As a summary, a comparison between Divide&Conquer and Dynamic Programming has been made:

1. First of all, both techniques try to split their input into parts, find sub-solutions to the parts. Sub-solutions are then synthesized to give the solution to the original problem.

2. Divide&Conquer splits its input at prespecified deterministic points. It always split a big problem into two almost equal halves. It works best when the subproblems are independent/non-overlapping of each other. Since each subproblem is a small-sized original problem, we usually implement the Divide&Conquer algorithm in the form of recursive function.

3. When the sub-problems are dependent on each other, we do not know which splitting points are optimal. Dynamic Programming may be a better way to go. It is aimed at finding the optimal partitions by trying all possibilities.

3 Part 2: SubMatrix, SubCube and M Subsequence Problems

As an extension of the first part of my work, I have tried to use Dynamic Programming technique to solve several variants of the original maximum subsequence sum problem.

3.1 SubMatrix

Similar to the subsequence problem, we want to obtain the maximum submatrix sum. As shown in Figure 5, the sum of the elements in the shaded area is maximized over all possible selection of contiguous submatrix. But the above method can only handle one dimension. Then, how can we solve this 2-D problem based on the method for 1-D problem? If we can fix the value for one dimension, say the start row and end row, then this 2-D problem can be solved in the same way as the maximum subsequence sum problem. How to fix the value for one dimension? We can simply enumerate all possibilities for this dimension, and then use the 1-D algorithm to calculate the start index and end index for the other dimension. In this way, the maximum submatrix problem can be transformed into a maximum subsequence sum problem. Steps are summarized as follows:

1. Enumerate all possible pairs of $(startRow, endRow)$.

2. Sum the rows from *startRow* to *endRow* together with respect to each dimension, from which we obtain a sequence (Figure 6).

3. Use the DP algorithm to find the maximum subsequence sum of the sequence generated in step 2.

The code of this algorithm is shown in Figure 7.

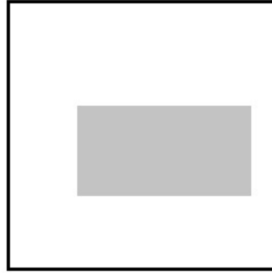

Figure 5: A Snapshot for the Submatrix Problem

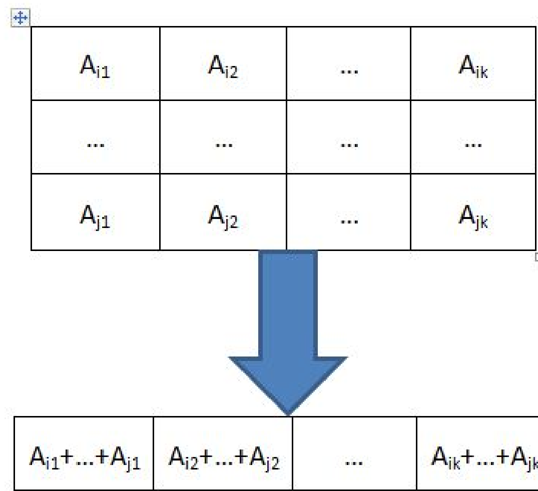

Figure 6: Transformation from 2-D to 1-D

Complexity Analysis For brute-force search, we need to enumerate all possible combinations of $(startRow, endRow, startColumn, endColumn)$. The time complexity would be $O(n^4)$. With the DP implementation, we reduced the time complexity to $O(n^3)$.

3.2 SubCube

With the idea in matrix, it is easy to extend the idea to 3-D arrays. Suppose a cube has x, y, z three dimensions. In analogy to the matrix, we only need to enumerate all possible combinations of $(xStart, xEnd, yStart, yEnd)$, and sum the elements together over x and y. Then, we use the DP algorithm to determine the start index and end index of z. The code is shown in Figure 8.

```
N=10;
A=randi([-100,100],N,N); %generate a NxN matrix
rowBeginIndex=0;
rowEndIndex=0;
columnBeginIndex=0;
columnEndIndex=0;
maxSum=0;
for i=1:N
    for j=1:N
        tempSum=0;
        %sum rows from i to j together
        V=sum(A(i:j,:));
        %maxSubSumDP is a function to calculate the maximum subsequence sum
        [tempSum,tempColumnBeginIndex,tempColumnEndIndex]=maxSubSumDP(V);
        if (tempSum>maxSum)
            rowBeginIndex=i;
            rowEndIndex=j;
            columnBeginIndex=tempColumnBeginIndex;
            columnEndIndex=tempColumnEndIndex;
            maxSum=tempSum;
        end
    end
end
```

Figure 7: Code for Submatrix Problem

Complexity Analysis For the brute-force search, we need to enumerate all possible combinations of $(xStart, xEnd, yStart, yEnd, zStart, zEnd)$. Thus, the time complexity is $O(n^6)$. With DP technique, we have reduced the time complexity to $O(n^5)$.

3.3 M Subsequence

This problem is a little different from the above two variant problems. For the M subsequence problem, we are given a sequence and a number M. We are asked to extract M non-overlapping subsequences whose total sum is maximized. In stead of only one subsequence in maximum subsequence sum problem, the M subsequence problem asks us to extract M subsequences. Thus, an additional constraint has been put onto this problem. Just as the knapsack problem, in which a weight constraint is added. What we need to do is to increase one dimension for the OPT value. Here, we only need to calculate a $D(i)$ for the maximum subsequence sum problem. But we may need to compute a $D(i, j)$ to transfer from one state to another state for the M subsequence problem. With this idea, it is not difficult to design a DP algorithm to solve the M subsequence problem.

Suppose the original sequence is of size N, and we want to maximize the total sum of M subsequences. We calculate the matrix D, with $D(i, j)$ as the maximum sum of first j elements with i subsequences. For element $j + 1$, we have two choices: one is element $j + 1$ constructs a subsequence exclusively; the other is this element is integrated into the last subsequence. So, the recursion function is defined as follows:

$$D(i, j) = \max\{[\max\{D(i-1, k)\} + A(j)], D(i, j-1) + A(j)\}, (i-1) \leq k \leq (j-1) \tag{9}$$

```
N=10;
A=randi([-100,100],N,N,N);
xBeginIndex=0;
xEndIndex=0;
yBeginIndex=0;
yEndIndex=0;
zBeginIndex=0;
zEndIndex=0;
maxSum=0;
for i1=1:N
    for i2=(i1+1):N
        for j1=1:N
            for j2=(j1+1):N
                V=sum(sum(A(i1:i2,j1:j2,:)));
                [tempSum,tempZBeginIndex,tempZEndIndex]=maxSubSumDP(V);
                if (tempSum>maxSum)
                    xBeginIndex=i1;
                    xEndIndex=i2;
                    yBeginIndex=j1;
                    yEndIndex=j2;
                    zBeginIndex=tempZBeginIndex;
                    zEndIndex=tempZEndIndex;
                    maxSum=tempSum;
                end
            end
        end
    end
end
```

Figure 8: Code for Subcube Problem

The code for this algorithm is shown in Figure 9.

```
N=20; %N is the number of numbers in the array
M=2; %number of subsequences
A=randi([-100,100],1,N); %generate N random numbers drawn from [-100,100]
D=zeros(M+1,N+1);
for i=2:(M+1)
    for j=i:(N+1)
        tempSum=max(D((i-1),((i-1):(j-1))));
        tempSum=tempSum+A(j-1);
        if (j~=i) %since we did not claculate the value D(i,(j-1)) (i=j)
            D(i,j)=max(tempSum,(D(i,(j-1))+A(j-1)));
        else
            D(i,j)=tempSum;
        end
    end
end
```

Figure 9: Code for M Subsequence Problem

Complexity Analysis There are two loops, the time complexity is $O(n^2)$.

4 Conclusion

In this project, I have implemented four existing algorithms for the maximum subsequence sum problem. Time complexity of the four algorithms have been compared. I have also made a brief comparison between

Divide&Conquer and Dynamic Programming.

The innovative part of this project is the extension of the DP algorithm to maximum submatrix, subcube and M subsequence problems. Although these problems may have already been solved with DP by others before, I have developed all these algorithms completely by myself. Through this process, two main principles for DP algorithm designing are learnt. First, complicated problems, like maximum submatrix and subcube problems, can be reduced to simple problems, say maximum subsequence sum problem. Through this transformation, the seemingly complicated problems can be solved in the same way as the simple problem. Secondly, when there is an additional constraint added to the problem, an additional constraint of M for the M subsequence problem when compared with the maximum subsequence sum problem, we can add one dimension to the OPT value correspondingly.

In summary, I do learn a lot from this project, although it may be not a complicated one. Through this project, I have achieved a much better understanding of DP technique than before. And my ability in algorithm designing has also been improved.

5 Appendix

```
1 -     N=1000000; %N is the number of integers in the original sequence
2 -     A=randi([-100,100],1,N); %generate N random integers drawn from [-100,100]
3 -     maxSum=0; %max subsequence sum
4 -     beginIndex=1; %extracted subsequence begin index
5 -     endIndex=1; %extracted subseqeunce end index
6 -     for i=1:N  %loops on i and j to enumerate all possible subsequences
7 -         for j=i:N
8 -             tempSum=0;
9 -             for k=i:j %a nested loop to calculate the sum of the subsequence
10 -                tempSum=tempSum+A(k);
11 -            if (tempSum>maxSum)
12 -                maxSum=tempSum;
13 -                beginIndex=i;
14 -                endIndex=j;
15 -            end
16 -            end
17 -        end
18 -    end
19 -    dlmwrite('A.txt',A); %for comparison with other algorithms, we store sequence A
```

Figure 10: Brute-Force Search Algorithm 1 for Maximum Subsequence Sum Problem

```
A=dlmread('A.txt'); %read in the sequence A in algorithm 1
N=10000;
maxSum=0;
beginIndex=1;
endIndex=1;
for i=1:N
    tempSum=0;
    for j=i:N
        tempSum=tempSum+A(j); %the smart part
        if (tempSum>maxSum)
            maxSum=tempSum;
            beginIndex=i;
            endIndex=j;
        end
    end
end
```

Figure 11: Brute-Force Search Algorithm 2 for Maximum Subsequence Sum Problem

```
function [maxSum,beginIndex,endIndex]=maxSubSum(A,leftIndex,rightIndex)
%I have written a recursive function maxSubSum
maxSum=0;
if(leftIndex==rightIndex) %temination condition
    beginIndex=leftIndex;
    endIndex=rightIndex;
    if (A(leftIndex)>0)
        maxSum=A(leftIndex);
    else
        maxSum=0;
    end
else
    centerIndex=floor((leftIndex+rightIndex)/2); %calculate the partition point
    %calculate the maxSum for two halves
    [leftMaxSum,leftBeginIndex,leftEndIndex]=maxSubSum(A,leftIndex,centerIndex);
    [rightMaxSum,rightBeginIndex,rightEndIndex]=maxSubSum(A,centerIndex+1,rightIndex);
    %To calculate the max subsequence that bridges two halves, I extend
    %toward left and right from the central point
    leftExtendMaxSum=0;
    leftExtendTempSum=0;
    tempIndex1=0;
    %since the subsequence needs to be contiguous, we only need to
    %tranverse the sequence once
    for i=centerIndex:-1:leftIndex
        leftExtendTempSum=leftExtendTempSum+A(i);
        if (leftExtendTempSum>leftExtendMaxSum)
            tempIndex1=i;
            leftExtendMaxSum=leftExtendTempSum;
        end
    end
    rightExtendMaxSum=0;
    rightExtendTempSum=0;
    tempIndex2=0;
    for i=(centerIndex+1):rightIndex
        rightExtendTempSum=rightExtendTempSum+A(i);
        if (rightExtendTempSum>rightExtendMaxSum)
            tempIndex2=i;
            rightExtendMaxSum=rightExtendTempSum;
        end
    end

    maxSum=leftExtendMaxSum+rightExtendMaxSum;
    beginIndex=tempIndex1;
    endIndex=tempIndex2;
    %compare the three max subsequence sum
    if(maxSum<leftMaxSum)
        beginIndex=leftBeginIndex;
        endIndex=leftEndIndex;
        maxSum=leftMaxSum;
    end
    if(maxSum<rightMaxSum)
        beginIndex=rightBeginIndex;
        endIndex=rightEndIndex;
        maxSum=rightMaxSum;
    end
end
end
```

Figure 12: Divide&Conquer Algorithm for Maximum Subsequence Sum Problem

```
A=dlmread('A.txt'); %read in the sequence A
N=100000; %N is the number of numbers in the array
maxSum=0; %store the maximum sum of the (i-1) numbers
tempSum=0; %store the maximum sum of the subsequence that includes i-th number
beginIndex=0;
endIndex=0;
for i=1:N
    if (tempSum>0)
        tempSum=tempSum+A(i); %the subsequence can be extend on the left side
    else
        tempSum=A(i); %start a new subsequence
        beginIndex=i;
    end
    if (tempSum>maxSum)
        maxSum=tempSum;
        endIndex=i;
    end
end
```

Figure 13: Dynamic Programming Algorithm for Maximum Subsequence Sum Problem

```
function [maxSum,beginIndex,endIndex]=maxSubSumDP(V)
maxSum=0; %store the maximum sum of the (i-1) numbers
tempSum=0; %store the maximum sum of the subsequence that includes i-th number
beginIndex=0;
endIndex=0;
N=size(V,2); %for matrix, it is N=size(V,2); for cubic, it is N=size(V,3)
for i=1:N
    if (tempSum>0)
        tempSum=tempSum+V(i); %the subsequence can be extend on the left side
    else
        tempSum=V(i); %start a new subsequence
        beginIndex=i;
    end
    if (tempSum>maxSum)
        maxSum=tempSum;
        endIndex=i;
    end
end
```

Figure 14: A Function Called in Submatrix and Subcube Algorithms

```
N=10;
A=randi([-100,100],N,N); %generate a NxN matrix
rowBeginIndex=0;
rowEndIndex=0;
columnBeginIndex=0;
columnEndIndex=0;
maxSum=0;
for i=1:N
    for j=1:N
        tempSum=0;
        %sum rows from i to j together
        V=sum(A(i:j,:));
        %maxSubSumDP is a function to calculate the maximum subsequence sum
        [tempSum,tempColumnBeginIndex,tempColumnEndIndex]=maxSubSumDP(V);
        if (tempSum>maxSum)
            rowBeginIndex=i;
            rowEndIndex=j;
            columnBeginIndex=tempColumnBeginIndex;
            columnEndIndex=tempColumnEndIndex;
            maxSum=tempSum;
        end
    end
end
```

Figure 15: Algorithm for Maximum Submatrix Sum Problem

```
N=10;
A=randi([-100,100],N,N,N);
xBeginIndex=0;
xEndIndex=0;
yBeginIndex=0;
yEndIndex=0;
zBeginIndex=0;
zEndIndex=0;
maxSum=0;
for i1=1:N
    for i2=(i1+1):N
        for j1=1:N
            for j2=(j1+1):N
                V=sum(sum(A(i1:i2,j1:j2,:)));
                [tempSum,tempZBeginIndex,tempZEndIndex]=maxSubSumDP(V);
                if (tempSum>maxSum)
                    xBeginIndex=i1;
                    xEndIndex=i2;
                    yBeginIndex=j1;
                    yEndIndex=j2;
                    zBeginIndex=tempZBeginIndex;
                    zEndIndex=tempZEndIndex;
                    maxSum=tempSum;
                end
            end
        end
    end
end
```

Figure 16: Algorithm for Maximum Subcube Sum Problem

```
N=20; %N is the number of numbers in the array
M=2; %number of subsequences
A=randi([-100,100],1,N); %generate N random numbers drawn from [-100,100]
D=zeros(M+1,N+1);
for i=2:(M+1)
    for j=i:(N+1)
        tempSum=max(D((i-1),((i-1):(j-1))));
        tempSum=tempSum+A(j-1);
        if (j~=i) %since we did not claculate the value D(i,(j-1)) (i=j)
            D(i,j)=max(tempSum,(D(i,(j-1))+A(j-1)));
        else
            D(i,j)=tempSum;
        end
    end
end
```

Figure 17: Algorithm for M Subsequence Problem