Problem 1: Maximum area contiguous subsequence

Solution:

We will use Divide-and-conquer approach to find the maximum area. We need to divide the sequence into two parts and find the maximum area of each of the two parts (A1 and A2) and the maximum area that begins from the first part and end in the second part (A3). Then decide which is the maximum area. First, we should determine the base condition. We divide the problem instance into sub-problems until we reach the base condition.

Base condition. There is only one value in the subsequence which is exactly the maximum area.

Inductive hypothesis. We assume that our function returns the maximum area in a given sequence.

Find the contiguous subsequence which starts from the first part and ends in the second part of sequence. In this scenario, the center element of the sequence must be included. Through while-loops, continually update the minimum value from the center element to the leftmost value and the rightmost value (In this process, if the value of the left-handed index is larger the value of the right-handed index, we firstly decrease the index of the left part to guarantee the min(seq) is no less than the min(seq) if we increase the right-handed index first. The crux is to keep the min(seq) as large as possible) and if the newly-calculated area is larger, update the maximum area A3.

Compare and return. Compare the three maximum areas (A1, A2 and A3) and return the maximum one as the result.

Time and space complexity:

(1) Every time we divide the problem into two sub-problems and till we reach the base condition, it is O(logn). In every sub-problem, compare and find the minimum value and calculate the area which takes O(n) times. We get T[n]=2T[n/2]+O(n). So the overall time complexity is O(nlogn).

(2) We define some variables to record the area, value, etc. To note that, we have only constant variables, so the space complexity is O(1).

pseudocode:

```
maxArea(int a[], int left, int right)
    int maxLeft=0, maxRight=0, center=(left+right)/2;
    if left==right
        return a[left];
    record the maximum area of the left part A1: maxLeft
    record the maximum area of the right part A2:maxRight;
    define the temporary maximum area A3 which has values in both two sides:
maxTemp=a[center];
    define minimum Value: a[center];
    int area1=0,le=center,ri=center;

    while(le>=left || ri<=right)
        if a[le]<a[ri] ri++;
        else le--;
        update the minimum value and recalculate the area:area1 ;
        if area1>maxTemp
            maxTemp=area1;
        end if
    end while

    while(le>left)
        le--;
        update the minimum value and recalculate the area: area1
        if(area1>maxTemp)
            maxTemp=area1;
        end if
    end while

    while(ri<right)
        ri++;
        update the minimum value and recalculate the area: area1
        if(area1>maxTemp)
```

```
            maxTemp=area1;
        end if
    end while

return max(maxLeft, maxRight, maxTemp);
```

Linear-time algorithm:

Check a given sequence of values one by one. Use a stack to preserve the ascending value and once the current value is smaller than the top element of the stack, we pop out the top element and find the index of next element in the stack. The values between the top element and next element must be larger than the top element because our stack is a non-decreasing stack. Thus we could work out a maximum area with minimum value (the popped one). The key idea is to find the maximum sequence of each value in which every value is larger. In case the sequence is non-decreasing, we need to add an element 0 to the stack to avoid no result.

Problem 2: Balanced Parentheses

Solution:

The property of balanced parentheses is that the number of left brackets is always no less than the number of right brackets and the total number of left brackets is the same as the number of right brackets and is the half of the string length.

To assure the two property, we define the string length as n and two variables to contain the current number of left bracket (x) and right bracket (y). In this case, the property could be indicated as x>=y and x<=n/2 and y<=n/2. We will use a for-loop to check every character of the string, and if the property is not satisfied, a change needs to be done to keep the parentheses balanced.

To be specific, when we have x>y and x<n/2, then no matter next element is left or right bracket, we do not need to flip it because it' s still satisfy the property. But if we have x=y, then next element must be left to guarantee x>=y, so we need to check next element if it needs to be flipped. If we have x>y and x==n/2, the next element must be right bracket to ensure the number of left bracket is the same as the right bracket at the end.

Time and space complexity:

(1) We need to use a for-loop to check every bracket in the string, so the time complexity is O(n).

(2) We need to store the changed parentheses which need O(n) space.

Problem 3: Maximum difference in an array

Solution:

To assure that A(i)-A(j)(j>i) is the maximum value of all choices of indexes, the property is that A(j)<=A(x) for all x>=j and A(i)>=A(x) for all x<j.

We define a variable maxVal to record the maximum value before current index j and a variable maxDiff to record the maximum difference until current index j.

We use a for-loop to check the array and update the maximum value(maxVal) of the values before a certain index j in this way we could ensure the difference of the current value and maxVal is the largest for current value. If the new difference is larger than maxDiff, update the maximum difference(maxDiff) when going through the entire array A.

Time and space complexity:

(1)In the process, we only use a for-loop and two comparison in each cycle which cost only constant time. So the entire time complexity is O(n).

(2) We define two variables to record the maximum value and maximum difference, and thus the space complexity is O(1).

Pseudocode:

```
Define int maxVal=A[0], maxDiff=0;
For i∈ (1, A.length)
  maxDiff=max(maxDiff, maxVal-A[i]);
  maxVal=max(maxVal, A[i]);
end for
return maxDiff;
```

Problem 4: Maximum difference in a matrix

Solution:

The request is the find the maximum difference of M[c,d]-M[a,b] over all choices of indexes such that both c > a and d > b. The property is that M[a,b] is the smallest for all M[i,j] ($0 \leq i < c, 0 \leq j < d$). Thus c and d must larger than 0. Define a new n*n matrix **N** in which element N[i][j] means the minimum value of all elements N[ii][jj] (for all ii<=I, jj<=j). To do this, we can use dynamic programming approach (as shown in pseudocode1). When we want to search the minimum M[a][b] for M[c][d], we could just make use N[c-1][d-1] to get the maximum difference for a certain M[c][d]. So in the entire process, we firstly generate new matrix N and then for each element M[c][d] in M, we calculate and update the maximum value by comparing M[c][d] and N[c-1][d-1] (for all c>=1, d>=1).

Time and space complexity:

(1) Generating new matrix needs a 2-layer nested loop which spends $O(n^2)$ times. Comparing M[c][d] with N[c-1][d-1] for every 0<c<=n, 0<d<=n needs $O(n^2)$ times. Their relationship is an addition: $O(n^2)+O(n^2)$, and thus the overall time complexity is $O(n^2)$.

(2) We defined a new matrix N to preserve the minimum value which takes $O(n^2)$ space. Record the maximum difference just need constant space. We can find out the space complexity is $O(n^2)$.

Pseudocode1:

```
N[0][0]=M[0][0];
For 1<i<n
    N[i][0]=min(N[i-1][0],M[i][0]);
    N[0][i]=min(N[0][i-1],M[0][i]);
End for
For 1<i<n
    For 1<j<n
       N[i][j]=min(N[i-1][j],N[i][j-1],M[i][j]);
    End for
End for
```

Problem 5: Pond sizes

Solution:

In order to find all regions of a pond, we use DFS approach to search all connected water region. We utilize a two-layer nested loop to check every value in the matrix. If the current value is 0, then a new pond is found. We use a variable to record the pond' s size. If a value V that connected vertically, horizontally or diagonally to current value is 0, we use a recursion to check value V' s neighborhood. In this recursion process, we will change the value 0 to 1 to represent that the value has already been detected and increase the pond' s size by one simultaneously. After the process of DFS, the function will go back to the next value of current value until the end of the matrix.

Time and space complexity:

(1) Assume the size of the matrix is m*n. For each element in the matrix, there is at most one time calling for DFS which cost O(mn). And each element will at most be changed once. Therefore, the overall time complexity is O(mn).

(2) Because we only change value based on the original matrix, the space is only used for record the sizes of all ponds. Hence the space complexity is O(mn).