Problem 1
[25 pts] Pascal's triangle:
The first entry in a row is 1 and the last entry is 1 (except for the first row which contains only 1), and every other entry in Pascal's triangle is equal to the sum of the following two entries: the entry that is in the previous row and the same column, and the entry that is in the previous row and previous column.

1. **[10 pts]** Give a recursive definition (relation) for the entry $C[i, j]$ at row i and column j of Pascal's triangle. Make sure that you distinguish the base case(s).
2. **[5 pts]** Give a recursive algorithm to compute $C[i, j]$, $i \geq j \geq 1$. Illustrate by drawing a diagram (tree) the steps that your algorithm performs to compute $C[6, 4]$. Does your algorithm perform overlapping computations?
3. **[10 pts]** Use dynamic programming to design an $O(n^2)$ time algorithm that computes the first n rows in Pascal's triangle.

Solution:

1. Recursive definition: $C(i, j) = C(i-1, j-1) + C(i-1, j)$ where $i > j > 0$; (base case) 1 if $j = 0$ or $j = i$.

2.

```
def pascalNumber(int i, int j):
    if i = 0 or j = i then
        return 1;
    end
    return pascalNumber(n - 1, k - 1) + pascalNumber(n - 1, k);
```

This is same approach as Fibonacci Sequence F(n) = F(n - 1) + F(n - 2) and therefore there are many overlapping subproblems.

3.

```
def pascalNumber(int k, int m):
    A ← new 2 dimension Array;
    for i = 0; i < k + 1; i++ do
        for j = 0; j < i; j++ do
            if j == i or j == 0 then
                A[i][j] == 1;
            end
            else
                A[i][j] = A[i - 1][j - 1] + A[i - 1][j];
            end
        end
    end
    return A[k][m];
```

Nested loop has been applied in this algorithm, the totally time complexity is $O(n^2)$.

Problem 2
[25 pts] In a previous life, you worked as a cashier in the lost Antarctican colony, spending the better part of your day giving change to your customers. Because paper is a very rare and valuable resource in Antarctica, cashiers were required by law to use the fewest bills possible whenever they gave change.

1. **[5 pts]** Suppose that the currency of the colony was available in the following denominations: 1, 4, and 6. Consider an algorithm that repeatedly takes the largest bill that does not exceed the target amount. For example, to make 11 using this algorithm, we first take a 6 bill, then a 4 bill, and finally a 1 bill. Give an example where this greedy algorithm uses more bills than the minimum possible.

2. **[10 pts]** Describe and analyze a recursive algorithm that computes, given an integer n and an arbitrary system of k denominations $< d_1 = 1, ..., d_k >$, the minimum number of bills needed to make the amount n.

3. **[10 pts]** Describe a dynamic programming algorithm that computes, given an integer n and an arbitrary system of k denominations $< d_1 = 1, ..., d_k >$, the minimum number of bills needed to make amount n.

Solution:

1. Example: 8, greedy algorithm will take a 6 bill then a 1 bill, and finally a 1 bill again, a total of 3 bills. As we all know the best solution is two of 4 bill.

2.

```
result ← MAXIMUM;  def findMinBill(List<Integer> bills, int value):
    if value == 0 then
        return 0;
    end
    for i = 0 to length of bills do
        if bills[i] ≤ value then
            currrentRes = findMinBill(bills, value - bills[i]);
            if currentRes + 1 ¡ result then
                result = currentRes + 1;
            end
        end
    end
```

3.

```
def findMinBill(List<Integer> bills, int value):
    A ← new 2 dimension Array;
    A[0] ← 0; for i = 1 to value do
        for j = 0 to length of bills do
            if (bills[j] ¡ i) and (A[i - bills[j]] + 1 ¡ A[i]) then
                A[i] = A[i - bills[j]] + 1;
            end
        end
    end
    return A[value];
```

Problem 3
[25 pts] Suppose that you are given an array $A[1..n]$ of numbers, which may be positive, negative, or zero, and which are not necessarily integers. Give an O(n)-time algorithm that finds the largest sum of elements in a contiguous subarray $A[i..j]$ of A. For example, given the array [-6, 12, -7, 0, 14, -7, 5] as input, your algorithm should return 19, which is the content of $A[2..5]$.

Solution:
Approach steps:

1. Initialize 2 integer variables. Set both of them equal to the first value in the array.
   currentSubarray will keep the running count of the current subarray we are focusing on.
   maxSubarray will be our final return value. Continuously update it whenever we find a bigger subarray.

2. Iterate through the array, starting with the 2nd element (as we used the first element to initialize our variables). For each number, add it to the currentSubarray we are building. If currentSubarray becomes negative, we know it isn't worth keeping, so throw it away. Remember to update maxSubarray every time we find a new maximum.

```
def maxSubArray(List<Integer> nums):
    currentSubarray ← nums[0];
    maxSubarray ← nums[0];
    for i = 0 to length of nums do
        num ← nums[i];
        currentSubarray ← max(num, currentSubarray + num);
        maxSubarray ← max(maxSubarray, currentSubarray);
    end
    return maxSubarray;
```

Problem 4
[15 pts] A subsequence of a sequence is anything obtained from a sequence by extracting a subset of elements, but keeping them in the same order; the elements of the subsequence need not be contiguous in the original sequence. For example, the strings C, DAMN, YAIOAI, and DYNAMICPROGRAMMING are all subsequences of the string DYNAMICPROGRAMMING.

1. **[15 pts]** Let A[1..m] and B[1..n] be two arbitrary arrays. A common subsequence of A and B is another sequence that is a subsequence of both A and B. A longest common subsequence of A and B is a subsequence of A and B of maximum length. For example, if A = $< CTGCGTGTC >$ and B = $< GTCGTGGC >$, then the length of the longest common subsequence of A and B is 6, and the sequence ⟨TCGTGC⟩ is such a longest common subsequence of A and B. Describe an O(nm)-time algorithm to compute the length of the longest common subsequence of two given sequences A and B. (Hint. You may modify the Edit Distance algorithm so that you consider only the two operations: Insert and Delete. What is the connection between this variant of Edit Distance and the problem of computing a longest common subsequence?)

2. **[10 pts]** A palindrome is any sequence that is exactly the same as its reversal, like I, or DEED, or RACECAR, or AMANAPLANACATACANALPANAMA. Use part (a) above to give an $O(n^2)$-time algorithm to find the length of the longest subsequence of a given sequence of length n that is also a palindrome.

Solution:

1.

```
def longestCommonSubsequence(String a, String b):
    A ← new 2 dimension Array;
    for i = length of b; i ≥ 0; i ← i - 1 do
        for j = length of a; j ≥ 0; j ← j - 1 do
            if a[j] == b[i] then
                A[j][i] = 1 + A[j + 1][i + 1];
            end
            else
                A[j][i] = max(A[j + 1][i], A[j][i + 1]);
            end
        end
    end
    return A[0][0];
```

Let M be the length of the string a, and N be the length of the string b. Total time complexity will be $O(M \cdot N)$.

2. Part (a) is to find the longest common subsequence between string A and string B, to determine whether a string C is palindrome, we can compare the longest common subsequence between string C and reverse of C, if the result is equivalent to the length of C, which means this is palindrome.

```
def palindrome(String c, String revserseC):
    res = longestCommonSubsequence(c, revserseC);
    if res = length of c then
        return true;
    end
    return false;
```

Let N be the length of the string c, thus reverseC has same length. Total time complexity will be $O(N^2)$.