DYNAMIC PROGRAMMING - SET 1 SOLUTIONS

Note: All solutions are written in C++.

Question 1:

- **Problem link**: https://leetcode.com/problems/maximum-subarray/
- **Difficulty level**: Easy

Explanation:

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.
3. Return `maxSubarray`.

Solution:

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {

        int local_sum=0;
```

```
        int global_sum=INT_MIN;
        for(int i=0;i<nums.size();i++)
        {
            local_sum=max(local_sum+nums[i],nums[i]);
            global_sum=max(global_sum,local_sum);
        }
        return global_sum;
    }
};
```

Complexity:

- Time: O(n)
- Space: O(n)

Question 2:

- **Problem link**: https://leetcode.com/problems/climbing-stairs/
- **Difficulty level**: Easy

Explanation:

As we can see this problem can be broken into subproblems, and it contains the optimal substructure property i.e. its optimal solution can be constructed efficiently from optimal solutions of its subproblems, we can use dynamic programming to solve this problem.

One can reach i^{th}step in one of the two ways:

1. Taking a single step from $(i-1)^{th}$ step.
2. Taking a step of 2 from $(i-2)^{th}$ step.

So, the total number of ways to reach i^{th} step is equal to the sum of ways of reaching $(i-1)^{th}$ step and ways of reaching $(i-2)^{th}$ step : $dp[i]=dp[i-1]+dp[i-2]$

Solution:

```
int climbStairs(int n) {
        int t[n+1];
        for(int i=0;i<=n;i++) t[i]=0;
        t[1]=1;
        if(n>=2) t[2]=2;
        for(int i=3;i<=n;i++){
             t[i] = t[i-1]+t[i-2];
        }
        return t[n];
    }
```

Complexity:

- Time: O(n)
- Space: O(n)

Question 3:

- **Problem link**: https://leetcode.com/problems/is-subsequence/
- **Difficulty level**: Easy

Explanation:

Traverse left to right string t and match character of string s from left to right. If character is matched then increase the j to next index to be compared from string s. If j is equal to the size of string s it implies that s is subsequence of t otherwise not.

Solution:

```
class Solution {
public:
    bool isSubsequence(string s, string t) {
        int n1 = s.length();
```

```
        int n2 = t.length();

        int j=0;

        for(int i=0; i<n2 and j<n1; i++){
            if(s[j]==t[i])
                j++;
        }

        if(j==n1)
            return true;
        else
            return false;
    }
};
```

Complexity:

- Time:O(n)
- Space:O(1)

Question 4:

- **Problem link**: https://leetcode.com/problems/decode-ways/
- **Difficulty level**: Medium

Solution:

```
/*    2 possibilities:
        1)consider it as individual(1-9) //not possible for 0
        2)consider as pair(<=26)
        1 2 3 4 5           1 2 2 3

        1 2 3 4 5           1 2 2 3      1 2 23
```

```
        12 3 4 5              12 2  3      12 23
        1 23 4 5              1 22  3
        */

class Solution {
public:

    int numDecodings(string s) {

        vector<int>dp(s.length()+1);
        dp[0]=1;
        for(int i=0;i<s.length();i++)
        {
            if(s[i]!='0')
            {
                dp[i+1]+=dp[i];
            }
            if(i>=1)
            {
                int x=stoi(s.substr(i-1,2));
                if(x>=10 and x<=26)
                {
                    dp[i+1]+=dp[i-1];
                }
            }
        }
        return dp[s.length()];
    }
};
```

Complexity:

- Time: O(n)
- Space: O(n)

Question 5:

- **Problem link**: https://leetcode.com/problems/coin-change/
- **Difficulty level**: Medium

Explanation:

We note that this problem has an optimal substructure property, which is the key piece in solving any Dynamic Programming problems. In other words, the optimal solution can be constructed from optimal solutions of its subproblems.

How to split the problem into subproblems?

Let's assume that we know F(S) where some change val_1, val_2,... for *S* which is optimal and the last coin's denomination is *C*. Then the following equation should be true because of optimal substructure of the problem: $F(S)=F(S-C)+1$

But we don't know which is the denomination of the last coin C. We compute $F(S - c_i)$ for each possible denomination $c_0, c_1, c_2 \dots c_{n-1}$ and choose the minimum among them. The following recurrence relation holds:

$$F(S) = \min_{i=0\dots n-1} F(S - c_i) + 1$$

subject to $S-c_i >= 0$

$$F(S) = 0, \text{when } S = 0$$

$$F(S) = -1, \text{when } n = 0$$

Now to get an iterative solution we think in a bottom-up manner. Before calculating $F(i)$ we have to compute all minimum counts for amounts up to i. On each iteration i of the algorithm $F(i)$ is computed as $\min_{j = 0\dots n-1} F(i - c_j) + 1$

Solution:

```
class Solution {
```

```
public:
    int coinChange(vector<int>& coins, int amount) {
        vector<double> dp(amount+1, INT_MAX);
        dp[0] = 0;

        // dp[i] represents the fewest number of coins to make up the
amount i
        for (int i = 1; i <= amount; i++) {

            for (auto coin : coins) {

                if (coin <= i) {

                    dp[i] = min(dp[i], dp[i-coin]+1);
                }
            }
        }

        return dp.back() == INT_MAX ? -1 : dp.back();
    }
};
```

Complexity:

- Time: $O(S*n)$, where S is the amount and n is denomination count.
- Space: $O(S)$, where S is the amount to change

Question 6:

- **Problem link**: https://leetcode.com/problems/perfect-squares/
- **Difficulty level**: Medium

Explanation:

The concept is similar to Coin Change Problem, except the coins array, we can use all the squares until `n` (example for n=18, your coins will be `[1,4,9,16]`)

Solution:

```
class Solution {
public:
    int solve(int n, vector<int> &dp) {
        if (n == 0) return 0;
        if (dp[n] != -1) return dp[n];
        dp[n] = INT_MAX;
        for(int i=1; i*i<=n; i++) {
            dp[n] = min(dp[n], solve(n-(i*i), dp)+1);
        }
        return dp[n];
    }
    int numSquares(int n) {
        vector <int> dp(n+1, -1);
        int ans = solve(n, dp);
        return ans;
    }
};
```

Complexity:

- Time: O(n)
- Space: O(n*sqrt(n))

Question 7:

- **Problem link**: https://leetcode.com/problems/minimum-path-sum/
- **Difficulty level**: Medium

Explanation:

I can reach any block in the grid in only two ways either from left or from top as I can move right or down from any point. So i can achieve my shortest path to (i,j) block by the equation `dp[i][j] = min(dp[i - 1][j], dp[i][j - 1]) + grid[i][j]`. Except top row and left columns shortest paths.

Solution:

Solution-1: space complexity - O(M*N)

```
class Solution {
public:
    int minPathSum(vector<vector<int>>& grid) {
        int m = grid.size();
        int n = grid[0].size();
        vector<vector<int>> dp(m,vector<int>(n));
        dp[0][0]=grid[0][0];


        for(int i=1; i<m; i++)
            dp[i][0]=grid[i][0]+dp[i-1][0];

        for(int i=1; i<n; i++)
            dp[0][i]=grid[0][i]+dp[0][i-1];


        for(int i=1; i<m; i++)
            for(int j=1; j<n; j++)
                dp[i][j]=(min(dp[i-1][j], dp[i][j-1])+grid[i][j]);

        return dp[m-1][n-1];
    }
};
```

Solution -2: space complexity - O(N)

```
class Solution {
public:
    int minPathSum(vector<vector<int>>& grid) {
        int m = grid.size();
        int n = grid[0].size();

        vector<int> dp1(n);
        vector<int> dp2(n);


        dp1[0]=grid[0][0];
        for(int i=1; i<n; i++)
            dp1[i]=dp1[i-1]+grid[0][i];

        for(int i=1; i<m; i++){
            for(int j=0; j<n; j++){
                if(j==0)
                    dp2[j]=dp1[j]+grid[i][j];
                else
                    dp2[j]=min(dp2[j-1],dp1[j])+grid[i][j];
            }

            swap(dp1,dp2);
        }

        return dp1[n-1];
    }
};
```

Complexity:

- Time: O(m*n)
- Space: O(m*n) or O(n)

Question 8:

- **Problem link**: https://leetcode.com/problems/regular-expression-matching/
- **Difficulty level**: Hard

Explanation:

We check the beginning of the text and pattern recursively. If the 2nd char of the pattern is not a star, then we check both the pattern and text from the first char depending on whether the first char were equal. If the 2nd char of the pattern is a star,

1. If the first characters match, then we can check for the first character against the text from the next index.
2. If the first characters do not match, then we check the pattern after the star with the text from beginning.

Solution:

```
class Solution {
public:

    bool isMatch(string s, string p) {
        int n=s.length();
        int m=p.length();
        vector<vector<int>>dp(n+1,vector<int>(m+1));
        dp[n][m]=1;
        for(int i=n;i>=0;i--)
        {
            for(int j=m-1;j>=0;j--)
            {
                bool first_match=(i<n and (s[i]==p[j] or p[j]=='.'));
                if(j+1<m and p[j+1]=='*')
                {
                    dp[i][j]=((first_match and dp[i+1][j]) or
dp[i][j+2]);
                }
```

```
                else
                {
                    dp[i][j]=first_match and dp[i+1][j+1];
                }
            }
        }
        return dp[0][0];
    }
};
```

Complexity:

- Time: O(n^2)
- Space: O(n^2)

Question 9:

- **Problem link**: https://leetcode.com/problems/trapping-rain-water/
- **Difficulty level**: Hard

Explanation:

Intuition is that height of water on ith bar is equal to minimum height of maximum left and right bar of ith bar minus actual height of ith bar. Sum all these water bars. Use an array to store the maximum left and right of ith bar.

Solution:

```
class Solution {
public:
    int trap(vector<int>& height) {
        int n = height.size();
        int ans =0;
        if(n==0)
            return ans;
```

```
        int Lmax[n];
        int Rmax[n];

        Lmax[0]=height[0];
        Rmax[n-1]=height[n-1];
        for(int i=1; i<n; i++)
            Lmax[i]=max(height[i],Lmax[i-1]);

        for(int i=n-2; i>=0; i--)
            Rmax[i]=max(height[i],Rmax[i+1]);

        for(int i=0; i<n; i++)
            ans+=min(Lmax[i],Rmax[i])-height[i];

        return ans;
    }
};
```

Complexity:

- Time: O(n)
- Space: O(n)