Day 43 / 100 :

Topic - DP, Memorization

1 Problem statement: Number of Music Player (Hard)

Your music player contains n different songs. You want to listen to goal songs (not necessarily different) during your trip. To avoid boredom, you will create a playlist so that:

- Every song is played at least once.
- A song can only be played again only if k other songs have been played.

Given n, goal, and k, return the number of possible playlists that you can create. Since the answer can be very large, return it modulo 109 + 7.

Example 1:
Input: n = 3, goal = 3, k = 1
Output: 6
Explanation: There are 6 possible playlists: [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], and [3, 2, 1].

Example 2:
Input: n = 2, goal = 3, k = 0
Output: 6
Explanation: There are 6 possible playlists: [1, 1, 2], [1, 2, 1], [2, 1, 1], [2, 2, 1], [2, 1, 2], and [1, 2, 2].

Example 3:
Input: n = 2, goal = 3, k = 1
Output: 2
Explanation: There are 2 possible playlists: [1, 2, 1] and [2, 1, 2].

Solutions :

Approach 1 - DP + Memorization

Intuition:

The goal is to find the number of distinct playlists of length goal that can be created using n different songs, with the constraint that the same song cannot appear in consecutive positions in the playlist, and there should be at least k songs between any two occurrences of the same song.

The function numMusicPlaylists takes three parameters: n (number of songs), goal (length of the playlist), and k (minimum number of songs between repeated songs). It uses dynamic programming to find the answer efficiently.

The solve function is a recursive helper function that calculates the number of valid playlists of length goal using n songs and satisfying the constraint of k. It uses memoization (dynamic programming) to avoid redundant calculations and improve efficiency.

Approach:

1. The base cases of the recursive function solve are:
    - If there are no songs left (n == 0) and the playlist length is also zero (goal == 0), we have found a valid playlist, so return 1.
    - If there are no songs left (n == 0) but the playlist length is still greater than zero (goal > 0), or vice versa, it is not possible to create a valid playlist, so return 0.
2. If the value of dp[n][goal] is already calculated (not equal to -1), return it. This is the memoization step to avoid redundant calculations.
3. The main recursive steps:
    - pick: We choose a song for the current position in the playlist. There are n choices for the first position, n - 1 for the second, and so on. So, we multiply solve(n - 1, goal - 1, k, dp) by n to count all valid playlists when we pick a song for the current position.
    - notpick: We do not choose a song for the current position. In this case, we can select any song from the remaining max(n - k, 0) songs (to ensure there are at least k songs between repeated songs). So, we multiply solve(n, goal - 1, k, dp) by max(n - k, 0).
4. Finally, we return the result of the recursive calculation and store it in dp[n][goal] for future use.

Complexity:

- Time complexity: O(n * goal)
- Space complexity: O(n * goal)

```
class Solution {
public:
    #define ll long long
    const int MOD = 1e9 + 7;
    ll solve(int n, int goal, int k, vector<vector<int>>& dp) {
```

```
        if (n == 0 && goal == 0) return 1;
        if (n == 0 || goal == 0) return 0;
        if (dp[n][goal] != -1) return dp[n][goal];
        ll pick = solve(n - 1, goal - 1, k, dp) * n;
        ll notpick = solve(n, goal - 1, k, dp) * max(n - k, 0);
        return dp[n][goal] = (pick + notpick) % MOD;
    }

    int numMusicPlaylists(int n, int goal, int k) {
        vector<vector<int>> dp(n + 1, vector<int>(goal + 1, -1));
        return solve(n, goal, k, dp);
    }
};
```

2 Problem statement: Longest Substring without repeating character (Medium)

Given a string s, find the length of the longest **substring** without repeating characters.

Example 1:
Input: s = "abcabcbb"
Output: 3
Explanation: The answer is "abc", with the length of 3.

Example 2:
Input: s = "bbbbb"
Output: 1
Explanation: The answer is "b", with the length of 1.

Example 3:
Input: s = "pwwkew"
Output: 3
Explanation: The answer is "wke", with the length of 3.
Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

Solutions :

Approach 1 - Hash Set + sliding window

Intuition:

if you know sliding window...then it can be intuitive. But if you don't know ...no worry i will teach you...
Refer below approach points.....

Approach:

1. Use sliding window with hashset, use left and right pointers to move the window .
2. If the set doesn't contains character then first add into the set and calculate the maxLength hand-in-hand...
3. if character already present in the set that means you have to move your sliding window by 1 , before that you have to remove all the characters that are infront of the character that is present already in window before.
4. Now you have to remove that character also and move the left pointer and also add the new character into the set.
5. THAT'S ALL.....EASY APPROACH USING SIMPLE HASHSET+SLIDING WINDOW

Complexity:

- Time complexity: O(n)
- Space complexity: O(k), where k is the number of distinctive characters prsent in the hashset.

```java
class Solution {
    public int lengthOfLongestSubstring(String s) {
        Set<Character>set=new HashSet<>();
        int maxLength=0;
        int left=0;
        for(int right=0;right<s.length();right++){

            if(!set.contains(s.charAt(right))){
                set.add(s.charAt(right));
                maxLength=Math.max(maxLength,right-left+1);

            }else{
                while(s.charAt(left)!=s.charAt(right)){
                    set.remove(s.charAt(left));
                    left++;
                }
                set.remove(s.charAt(left));left++;
                set.add(s.charAt(right));
            }

        }
        return maxLength;
    }
}
```