Day 50 / 100 :

Topic - Backtracking, DP

1 Problem statement: Check for valid partition of array (medium)

You are given a 0-indexed integer array nums. You have to partition the array into one or more contiguous subarrays.

We call a partition of the array valid if each of the obtained subarrays satisfies one of the following conditions:

The subarray consists of exactly 2 equal elements. For example, the subarray [2,2] is good.

The subarray consists of exactly 3 equal elements. For example, the subarray [4,4,4] is good.

The subarray consists of exactly 3 consecutive increasing elements, that is, the difference between adjacent elements is 1. For example, the subarray [3,4,5] is good, but the subarray [1,3,5] is not.

Return true if the array has at least one valid partition. Otherwise, return false.

Example 1:

Input: nums = [4,4,4,5,6]

Output: true

Explanation: The array can be partitioned into the subarrays [4,4] and [4,5,6].

This partition is valid, so we return true.

Example 2:

Input: nums = [1,1,1,2]

Output: false

Explanation: There is no valid partition for this array.

Solutions :

Approach 1 - Bottom up - DP

Intuition:

You can solve this problem by Bottom-Up Dynamic approach.

Why to use this approach?

Problem is dependent on whether you have solved the problem for previous indices (i+2 and i+3).

Approach:

- Initialize a dp array, which tells if it is possible to find a solution starting at an index i in nums.
- Add the base case i.e. dp[nums.size()] = true.
- For every index i starting from nums.size()-1 to 0, check for the 2 conditions:
  i. nums[i] == nums[i+1]
  ii. nums[i] == nums[i+1] and nums[i] == nums[i+2]
  iii. nums[i+1] - nums[i] == 1 and nums[i+2] - nums[i+1] == 1

Return dp[0] as the answer.

Time complexity:

O(n)

Space complexity:

O(n)

Code

```
class Solution {
public:
    bool validPartition(vector<int>& nums) {
        int n = nums.size();
        bool dp[n+1];
        // You can use vector instead of array.
        // vector<bool> dp(n+1, false);
        memset(dp, false, n+1);
        dp[n] = true;
        for(int i{n-2}; i>=0; i--){
            if(dp[i+2] && nums[i] == nums[i+1]) dp[i] = true;
            if(i < n-2 && dp[i+3]){
                if(nums[i] == nums[i+1] && nums[i]==nums[i+2]) dp[i] =
true;
                if(nums[i+1] - nums[i] == 1 && nums[i+2] - nums[i+1] == 1)
```

```
                    dp[i] = true;
                }
            }
            return dp[0];
        }
};
```

2️⃣ Problem statement: [Nth Fibonacci Number](#) (Easy)

Given a positive integer n, find the nth Fibonacci number. Since the answer can be very large, return the answer modulo 1000000007.

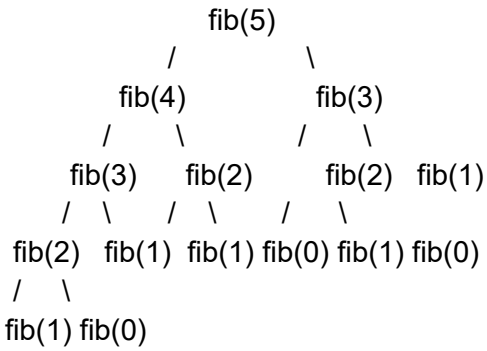
Example 1:
Input:
n = 2
Output:
1
Explanation:
1 is the 2nd number of Fibonacci series.

Example 2:
Input:
n = 5
Output:
5
Explanation:
5 is the 5th number of Fibonacci series.

Solutions :

Approach 1 - Bottom up - DP

- Consider the Recursion Tree for the 5th Fibonacci Number from the above approach:



- If you see, the same method call is being done multiple times for the same value. This can be optimized with the help of Dynamic Programming. We can avoid the repeated work done in the Recursion approach by storing the Fibonacci numbers calculated so far.

```
class GFG {
public:
    int fib(int n)
    {

        // Declare an array to store
        // Fibonacci numbers.
        // 1 extra to handle
        // case, n = 0
        int f[n + 2];
        int i;

        // 0th and 1st number of the
        // series are 0 and 1
        f[0] = 0;
        f[1] = 1;

        for (i = 2; i <= n; i++) {

            // Add the previous 2 numbers
            // in the series and store it
            f[i] = f[i - 1] + f[i - 2];
        }
        return f[n];
    }
};
```