When the question is like asking to find all the **possible ways / find the best or worst** etc , we can go for recursion.

# Climbing Stairs

**Problem Statement:** Given a number of stairs. Starting from the 0th stair we need to climb to the “Nth” stair. At a time we can climb either one or two steps. We need to return the total number of distinct ways to reach from 0th to Nth stair.

**Brute force** : either we can take steps of 1 stair or 2 stairs , hence the recurrence relation will be very similar to the fibonnacci series.

Time Complexity : O(2^n)

**Best Approach :** Use DP

TC : O(n)

# Frog Jump

**Problem Statement:**

Given a number of stairs and a frog, the frog wants to climb from the 0th stair to the (N-1)th stair. At a time the frog can climb either one or two steps. A height[N] array is also given. Whenever the frog jumps from a stair i to stair j, the energy consumed in the jump is abs(height[i]- height[j]), where abs() means the absolute difference. We need to return the minimum energy that can be used by the frog to jump from stair 0 to stair N-1.

**Approach :**

We have two options: we can jump from the previous stone (or latter one) to the i-th stone, or we can jump from the stone two positions back (or two positions after ith ) to the i-th stone. We choose the option with the minimum cost, where cost is the absolute difference in height between the two stones.

TC : O (2^n)

**Best Approach :** Use DP

TC : O(n)

# Frog Jump with k Distances

This is a follow-up question to “Frog Jump” In the previous question, the frog was allowed to jump either one or two steps at a time. In this question, the frog is allowed to jump up to ‘K’ steps at a time. If K=4, the frog can jump 1,2,3, or 4 steps at every index.

**Brute Force :** This is basically nothing but the extension of the same logic to k , by a for loop.

The time complexity of the plain recursive solution without memoization for this problem is O(K^N), where N is the number of stones and K is the maximum jump distance.

**Best Approach :** Use DP

If we use dynamic programming instead of recursion, we can reduce the time complexity to O(N\*K) as well, but without the overhead of function calls and recursion.

# Maximum sum of non-adjacent elements

**Problem Statement:** Given an array of ‘N’ positive integers, we need to return the maximum sum of the subsequence such that no two elements of the subsequence are adjacent elements in the array.

**Brute force :** For the general case, we have two options: either include the last element and skip the second-to-last element (since we cannot include adjacent elements), or skip the last element and consider the maximum sum up to the second-to-last element. We take the maximum of these two options.

TC : This recursive solution has a time complexity of O(2^n)

**Best Approach :** Use DP

TC : O(n)

# House Robber

**Problem Statement :** A thief needs to rob money in a street. The houses in the street are arranged in a circular manner. Therefore the first and the last house are adjacent to each other. The security system in the street is such that if adjacent houses are robbed, the police will get notified.

Given an array of integers “arr” which represents money at each house, we need to return the maximum amount of money that the thief can rob without alerting the police.

**Approach :** Same as above.

For a circular street, the first and last house are adjacent, therefore one thing we know for sure is that the answer will not consider the first and last element simultaneously (as they are adjacent).

Make two reduced arrays – arr1(arr-last element) and arr2(arr-first element).