Dynamic Programming

→ A bigger problem can be solved using the optimal soln of small subproblem.

→ Overlapping subproblems

⊛ Those who forget the past, are condemned to repeat it.

→ Top down (Recursion + Memoization)
→ Bottom up (Tabulation)

↓
Store the value of subproblem.

} Space Optimization

Fibonacci Sequence :-

```
// User function Template for C++
class Solution {
  public:
    int getFib(int n, vector<int> &dp) {
        if(n ≤ 1) return n;
        if(dp[n] ≠ -1) return dp[n];
        return dp[n] = getFib(n - 1, dp) + getFib(n - 2, dp);
    }
    int nthFibonacci(int n){
        vector<int> dp(n + 1, -1);
        return getFib(n, dp);
    }
};
```

→ Top down approach

Time Complexity : O(N)

Space Complexity : O(2*N)

```
class Solution {
  public:
    int nthFibonacci(int n){
        vector<int> dp(n + 1);
        dp[0] = 0;
        dp[1] = 1;

        for(int i = 2; i ≤ n; i++) {
            dp[i] = (dp[i - 1] + dp[i - 2]);
        }
        return dp[n];
    }
};
```

→ Tabulation

Time Complexity : O(N)

Space Complexity : O(N)

→ Space Optimization

```
class Solution {
  public:
    int mod = 1e9 + 7;
    int nthFibonacci(int n){
        long long prev2 = 0;
        long long prev1 = 1;

        for(int i = 2; i ≤ n; i++) {
            int curr = (prev1 + prev2) % mod;
            prev2 = prev1;
            prev1 = curr;
        }
        return prev1;
    }
};
```

Time Complexity : O(N)

Space Complexity : O(1)

Count ways to reach the nth stairs :-

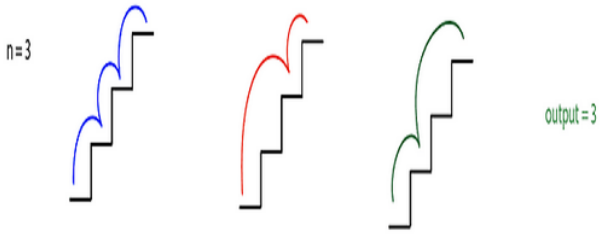
You have been given a number of stairs. Initially, you are at the 0th stair, and you need to reach the Nth stair.

Each time, you can climb either one step or two steps.

You are supposed to return the number of distinct ways you can climb from the 0th step to the Nth step.

Example :

N=3



We can climb one step at a time i.e. {(0, 1) ,(1, 2),(2,3)} or we can climb the first two-step and then one step i.e. {(0,2),(1, 3)} or we can climb first one step and then two step i.e. {(0,1), (1,3)}.

→ Recursion :-

```
long long countDistinctWays(long long nStairs) {
    if(nStairs < 0)
        return 0;
    else if(nStairs == 0)
        return 1;
    return countDistinctWays(nStairs - 2) + countDistinctWays(nStairs - 1);
}
```

→ Memoization :-

```
#define mod 1000000007;

int getWays(int nStairs, vector<long long> &dp) {
    if(nStairs <= 1) return 1;
    if(nStairs == 2) return 2;

    if(dp[nStairs] != -1) return dp[nStairs];

    return dp[nStairs] = (getWays(nStairs - 1, dp) + getWays(nStairs - 2, dp)) % mod;
}

int countDistinctWays(int nStairs) {
    vector<long long> dp(nStairs + 1, -1);
    return getWays(nStairs, dp);
}
```

→ Tabulation :-

```
#include <bits/stdc++.h>
#define mod 1000000007;

int getWays(int n) {
    vector<long long> dp(n + 1, -1);
    dp[0] = 1;
    dp[1] = 1;
    dp[2] = 2;

    for(int i = 3; i <= n; i++) {
        dp[i] = (dp[i - 1] + dp[i - 2]) % mod;
    }

    return dp[n];
}

int countDistinctWays(int nStairs) {
    return getWays(nStairs);
}
```

→ Space Optimization :-

```
#include <bits/stdc++.h>
#define mod 1000000007;

int getWays(int n) {
    long long prev0 = 1, prev1 = 1, prev2 = 2;

    for(int i = 3; i <= n; i++) {
        int curr = (prev2 + prev1) % mod;
        prev1 = prev2;
        prev2 = curr;
    }
    return prev2;
}

int countDistinctWays(int nStairs) {
    if(nStairs == 0 || nStairs == 1) return 1;
    return getWays(nStairs);
}
```

Minimum cost climbing stairs

You are given an integer array `cost` where `cost[i]` is the cost of `i^th` step on a staircase. Once you pay the cost, you can either climb one or two steps.

You can either start from the step with index `0`, or the step with index `1`.

Return *the minimum cost to reach the top of the floor.*

Example 1:

```
Input: cost = [10,15,20]
Output: 15
Explanation: You will start at index 1.
- Pay 15 and climb two steps to reach the top.
The total cost is 15.
```

Example 2:

```
Input: cost = [1,100,1,1,1,100,1,1,100,1]
Output: 6
Explanation: You will start at index 0.
- Pay 1 and climb two steps to reach index 2.
- Pay 1 and climb two steps to reach index 4.
- Pay 1 and climb two steps to reach index 6.
- Pay 1 and climb one step to reach index 7.
- Pay 1 and climb two steps to reach index 9.
- Pay 1 and climb one step to reach the top.
The total cost is 6.
```

Constraints:

- `2 <= cost.length <= 1000`
- `0 <= cost[i] <= 999`

→ Recursion :-

```
class Solution {
public:
    int solve(vector<int> cost, int n) {
        if(n == 0) return cost[0];
        if(n == 1) return cost[1];
        int ans = cost[n] + min(solve(cost, n - 1), solve(cost, n - 2));

        return ans;
    }


    int minCostClimbingStairs(vector<int>& cost) {
        int n = cost.size();
        int ans = min(solve(cost, n - 1), solve(cost, n - 2));
        return ans;
    }
};
```

→ Memoization :-

```
class Solution {
public:
    int solve(vector<int> cost, int n, vector<int> &dp) {
        if(n == 0) return cost[0];
        if(n == 1) return cost[1];

        if(dp[n] != -1) return dp[n];

        dp[n] = cost[n] + min(solve(cost, n - 1, dp), solve(cost, n - 2, dp));

        return dp[n];
    }
    int minCostClimbingStairs(vector<int>& cost) {
        int n = cost.size();
        vector<int> dp(n + 1, -1);
        int ans = min(solve(cost, n - 1, dp), solve(cost, n - 2, dp));
        return ans;
    }
};
```

→ Tabulation :-

```
class Solution {
public:
    int solve(vector<int> &cost, int n) {
        vector<int> dp(n + 1);
        dp[0] = cost[0];
        dp[1] = cost[1];
        for(int i = 2; i < n; i++) {
            dp[i] = cost[i] + min(dp[i - 1], dp[i - 2]);
        }
        return min(dp[n - 1], dp[n - 2]);
    }
    int minCostClimbingStairs(vector<int>& cost) {
        int n = cost.size();
        return solve(cost, n);
    }
};
```

→ Space Optimization :-

```
class Solution {
public:
    int solve2(vector<int> &cost, int n) {
        int prev1 = cost[1];
        int prev2 = cost[0];
        for(int i = 2; i < n; i++) {
            int curr = cost[i] + min(prev1, prev2);
            prev2 = prev1;
            prev1 = curr;
        }
        return min(prev1, prev2);
    }
    int minCostClimbingStairs(vector<int>& cost) {
        int n = cost.size();
        return solve2(cost, n);
    }
};
```