**DP Lecture 2**

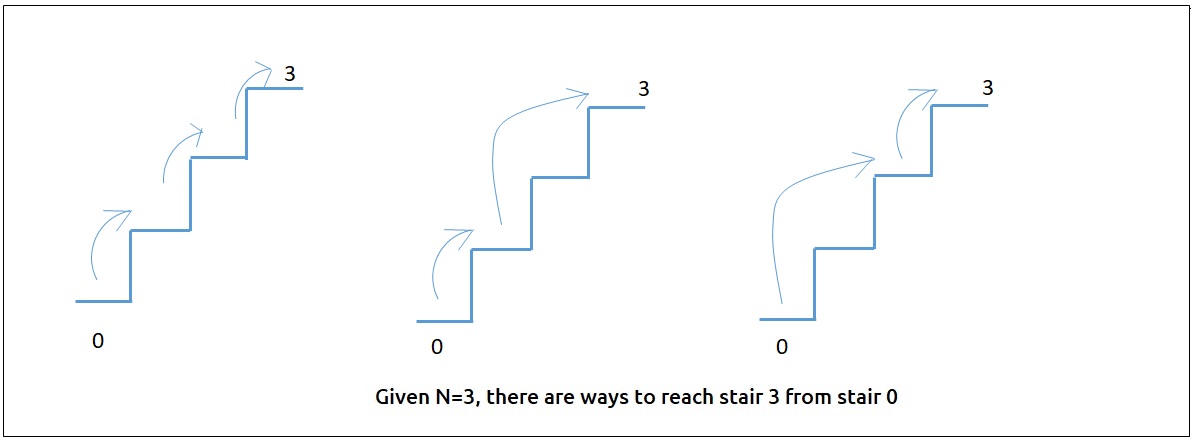
**1D – DP**

**Climbing Stairs**

**How to write 1-D Recurrence relation / Climbing Stairs**

In this article, we will learn to write 1-D Recurrence relations using the problem “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.



**Pre-req: Recursion,**[**Dynamic Programming Introduction**](https://takeuforward.org/data-structure/dynamic-programming-introduction/)

**Solution:**

**How to Identify a DP problem?**

When we see a problem, it is very important to identify it as a dynamic programming problem. Generally (but not limited to) if the problem statement asks for the following:

* Count the total number of ways
* Given multiple ways of doing a task, which way will give the minimum or the maximum output.

We can try to apply recursion. Once we get the recursive solution, we can go ahead to convert it to a dynamic programming one.

**Steps To Solve the Problem After Identification**

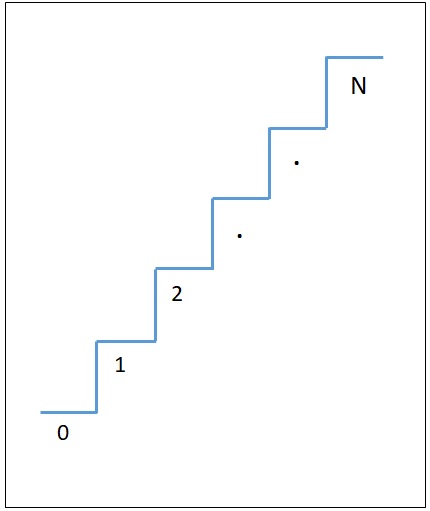
Once the problem has been identified, the following three steps comes handy in solving the problem:

* Try to represent the problem in terms of indexes.
* Try all possible choices/ways at every index according to the problem statement.
* If the question states
  + Count all the ways – return sum of all choices/ways.
  + Find maximum/minimum- return the choice/way with maximum/minimum output.

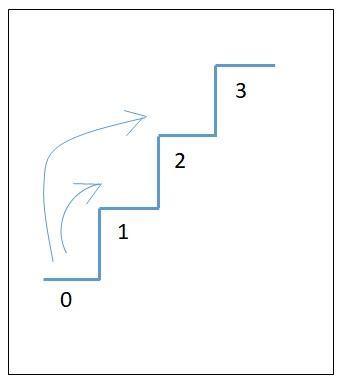
**Part 1: Recursion**

**Using these steps to solve the problem “Climbing Stairs”**

**Step 1:**We will assume n stairs as indexes from 0 to N.



**Step 2:**At a single time, we have 2 choices: Jump one step or jump two steps. We will try both of these options at every index.



**Step 3:**As the problem statement asks to count the total number of distinct ways, we will return the sum of all the choices in our recursive function.

The base case will be when we want to go to the 1st and 2nd stair, then we have 1 and 2 possible ways respectively.

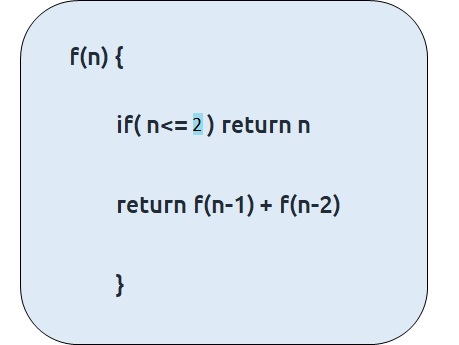
For 1st stair: There is only one way that is we directly take 1 step and jump to 1st stair

For 2nd stair:

1st way: 2 step and directly jump to 2nd stair

2nd way: 1 step and jump to 1st stair then again 1 step and jump to 2nd stair

The basic pseudo-code for the problem will be given as:



**Code:**

#include <bits/stdc++.h>

using namespace std;

int f(int n){

if(n<=2) return n;

return f(n-1) + f(n-2);

}

int main() {

int n=5;

cout<<f(n);

return 0;

}

**Output:** 5

**Time Complexity: O(2^N)**

Reason: As we are calling 2 recursive calls everytime for calculating the value of f(n).

**Space Complexity: O(N)**

Reason: Auxilliary space taken by recursion tree (i.e height of recursion tree)

**Part 2: Memoization**

**Steps to memoize a recursive solution:**

Any recursive solution to a problem can be memoized using these three steps:

1. Create a dp[n+1] array initialized to -1.
2. Whenever we want to find the answer of a particular value (say n), we first check whether the answer is already calculated using the dp array (i.e dp[n]!= -1). If yes, simply return the value from the dp array.
3. If not, then we are finding the answer for the given value for the first time, we will use the recursive relation as usual but before returning from the function, we will set dp[n] to the solution we get.

**Code:**

#include <bits/stdc++.h>

using namespace std;

int f(int n, vector<int>& dp){

if(n<=2) return n;

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

return dp[n]= f(n-1, dp) + f(n-2, dp);

}

int main() {

int n=5;

vector<int> dp(n+1,-1);

cout<<f(n,dp);

return 0;

}

**Output:** 5

**Time Complexity: O(N)**

Reason: The overlapping subproblems will return the answer in constant time O(1). Therefore, the total number of new subproblems we solve is ‘n’. Hence total time complexity is O(N).

**Space Complexity: O(N) + O(N)**

Reason: We are using a recursion stack space(O(N)) and an array (again O(N)). Therefore, total space complexity will be O(N) + O(N) ≈ O(N)

**Part 3: Tabulation**

**Steps to convert memoization approach to Tabulation approach.**

* Declare a dp[] array of size n+1.
* First initialize the base condition values, i.e i=0 and i=1 of the dp array as 1.
* Set an iterative loop which traverses the array( from index 2 to n) and for every index set its value as dp[i-1] + dp[i-2].

**Code:**

#include <bits/stdc++.h>

using namespace std;

int main()

int n=3;

vector<int> dp(n+1,-1);

dp[0]= 0, dp[1]= 1, dp[2]= 2;

for(int i=3; i<=n; i++){

dp[i] = dp[i-1]+ dp[i-2];

}

cout<<dp[n];

return 0;

}

**Time Complexity: O(N)**

Reason: We are running a simple iterative loop

**Space Complexity: O(N)**

Reason: We are using an external array of size ‘n+1’.

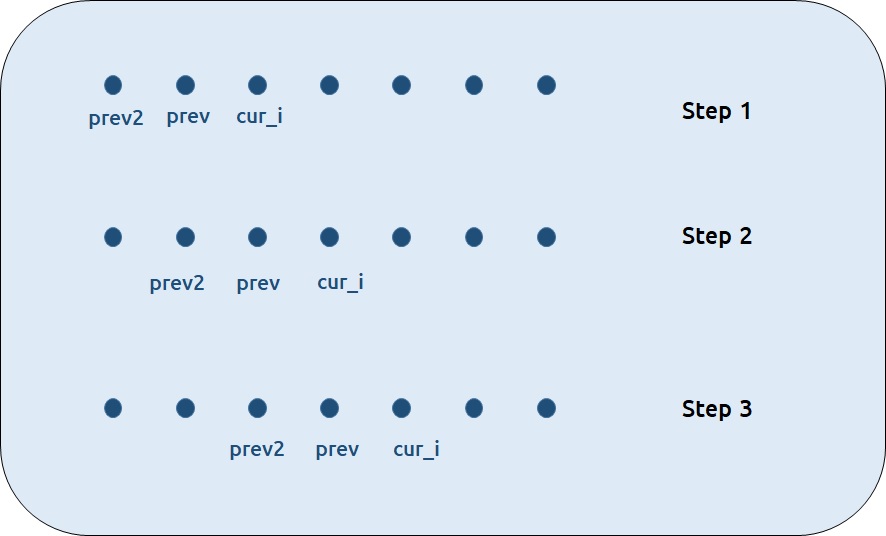
**Part 4: Space Optimization**

If we closely look the relation,

**dp[i] =  dp[i-1] + dp[i-2]**

we see that for any i, we do need only the last two values in the array. So is there a need to maintain a whole array for it?

The answer is ‘No’. Let us call dp[i-1] as prev and dp[i-2] as prev2. Now understand the following illustration.



* Each iteration’s cur\_i and prev becomes the next iteration’s prev and prev2 respectively.
* Therefore after calculating cur\_i, if we update prev and prev2 according to the next step, we will always get the answer.
* After the iterative loop has ended we can simply return prev as our answer.

**Code:**

#include <bits/stdc++.h>

using namespace std;

int main() {

int n=3;

int prev2 = 1, prev = 2;

if(n<=2) prev = n;

for(int i=3; i<=n; i++){

int cur\_i = prev2+ prev;

prev2 = prev;

prev= cur\_i;

}

cout<<prev;

return 0;

}

**Time Complexity: O(N)**

Reason: We are running a simple iterative loop

**Space Complexity: O(1)**

Reason: We are not using any extra space.