Framework for DP Problems

Now that we understand the basics of DP and how to spot when DP is applicable to a problem, we've reached the most important part: actually solving the problem. In this section, we're going to talk about a framework for solving DP problems. This framework is applicable to nearly every DP problem and provides a clear step-by-step approach to developing DP algorithms.

For this article's explanation, we're going to use the problem [Climbing Stairs](https://leetcode.com/problems/climbing-stairs/) as an example, with a top-down (recursive) implementation. Take a moment to read the problem description and understand what the problem is asking.

**Climbing Stairs**

You are climbing a staircase. It takes n steps to reach the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

|  |
| --- |
| **Example 1:**  **Input:** n = 2  **Output:** 2  **Explanation:** There are two ways to climb to the top.  1. 1 step + 1 step  2. 2 steps  **Example 2:**  **Input:** n = 3  **Output:** 3  **Explanation:** There are three ways to climb to the top.  1. 1 step + 1 step + 1 step  2. 1 step + 2 steps  3. 2 steps + 1 step    **Constraints:**   * 1 <= n <= 45 |

Before we start, we need to first define a term: **state**. In a DP problem, a **state** is a set of variables that can sufficiently describe a scenario. These variables are called **state variables**, and we only care about relevant ones. For example, to describe every scenario in Climbing Stairs, there is only 1 relevant state variable, the current step we are on. We can denote this with an integer i. If i = 6, that means that we are describing the state of being on the 6th step. Every unique value of i represents a unique **state**.

You might be wondering what "relevant" means here. Picture this problem in real life: you are on a set of stairs, and you want to know how many ways there are to climb to say, the 10th step. We're definitely interested in what step you're currently standing on. However, we aren't interested in what color your socks are. You could certainly include sock color as a state variable. Standing on the 8th step wearing green socks is a different state than standing on the 8th step wearing red socks. However, changing the color of your socks will not change the number of ways to reach the 10th step from your current position. Thus the color of your socks is an **irrelevant** variable. In terms of figuring out how many ways there are to climb the set of stairs, the only **relevant** variable is what stair you are currently on.

# The Framework

To solve a DP problem, we need to combine 3 things:

1. **A function or data structure that will compute/contain the answer to the problem for every given state**.

For Climbing Stairs, let's say we have an function dp where dp(i) returns the number of ways to climb to the *ith* step. Solving the original problem would be as easy as return dp(n).

How did we decide on the design of the function? The problem is asking "How many distinct ways can you climb to the top?", so we decide that the function will represent how many distinct ways you can climb to a certain step - literally the original problem but generalized for a given state.

Typically, top-down is implemented with a recursive function and hash map, whereas bottom-up is implemented with nested for loops and an array. When designing this function or array, we also need to decide on state variables to pass as arguments. This problem is very simple, so all we need to describe a state is to know what step we are currently on i. We'll see later that other problems have more complex states.

1. **A recurrence relation to transition between states.**

A recurrence relation is an equation that relates different states with each other. Let's say that we needed to find how many ways we can climb to the 30th stair. Well, the problem states that we are allowed to take either 1 or 2 steps at a time. Logically, that means to climb to the 30th stair, we arrived from either the 28th or 29th stair. Therefore, the number of ways we can climb to the 30th stair is equal to the number of ways we can climb to the 28th stair plus the number of ways we can climb to the 29th stair.

The problem is, we don't know how many ways there are to climb to the 28th or 29th stair. However, we can use the logic from above to define a recurrence relation. In this case, dp(i) = dp(i - 1) + dp(i - 2). As you can see, information about some states gives us information about other states.

Upon careful inspection, we can see that this problem is actually the Fibonacci sequence in disguise! This is a very simple recurrence relation - typically, finding the recurrence relation is the most difficult part of solving a DP problem. We'll see later how some recurrence relations are much more complicated, and talk through how to derive them.

1. **Base cases, so that our recurrence relation doesn't go on infinitely.**

The equation dp(i) = dp(i - 1) + dp(i - 2) on its known will continue forever to negative infinity. We need base cases so that the function will eventually return an actual number.

Finding the base cases is often the easiest part of solving a DP problem, and just involves a little bit of logical thinking. When coming up with the base case(s) ask yourself: What state(s) can I find the answer to without using dynamic programming? In this example, we can reason that there is only 1 way to climb to the first stair (1 step once), and there are 2 ways to climb to the second stair (1 step twice and 2 steps once). Therefore, our base cases are dp(1) = 1 and dp(2) = 2.

We said above that we don't know how many ways there are to climb to the 28th and 29th stairs. However, using these base cases and the recurrence relation from step 2, we can figure out how many ways there are to climb to the 3rd stair. With that information, we can find out how many ways there are to climb to the 4th stair, and so on. Eventually, we will know how many ways there are to climb to the 28th and 29th stairs.

# Example Implementations

Here is a basic top-down implementation using the 3 components from the framework:

|  |
| --- |
| *map*<int, int> mp;  int climbStairs(int n) {  if (n <= 2) return n;  return climbStairs(n - 1) + climbStairs(n - 2);  }  int main() {  *cout* << climbStairs(5) << *endl*;  return 0;  } |

Do you notice something missing from the code? We haven't memoized anything! The code above has a time complexity of *O*(2n) because every call to dp creates 2 more calls to dp. If we wanted to find how many ways there are to climb to the 250th step, the number of operations we would have to do is approximately equal to the number of atoms in the universe.

In fact, without the memoization, this isn't actually dynamic programming - it's just basic recursion. Only after we optimize our solution by adding memoization to avoid repeated computations can it be called DP. As explained in chapter 1, memoization means caching results from function calls and then referring to those results in the future instead of recalculating them. This is usually done with a hashmap or an array.

|  |
| --- |
| *map*<int, int> mp;  int climbStairs(int n) {  if (n <= 2) return n;  if (mp[n] == 0) {  mp[n] = climbStairs(n - 1) + climbStairs(n - 2);  }  return mp[n];  }  int main() {  *cout* << climbStairs(5) << *endl*;  return 0;  } |

With memoization, our time complexity drops to *O*(*n*) - astronomically better, literally.

You may notice that a hashmap is overkill for caching here, and an array can be used instead. This is true, but using a hashmap isn't necessarily bad practice as some DP problems will require one, and they're hassle-free to use as you don't need to worry about sizing an array correctly. Furthermore, when using top-down DP, some problems do not require us to solve every single subproblem, in which case an array may use more memory than a hashmap.

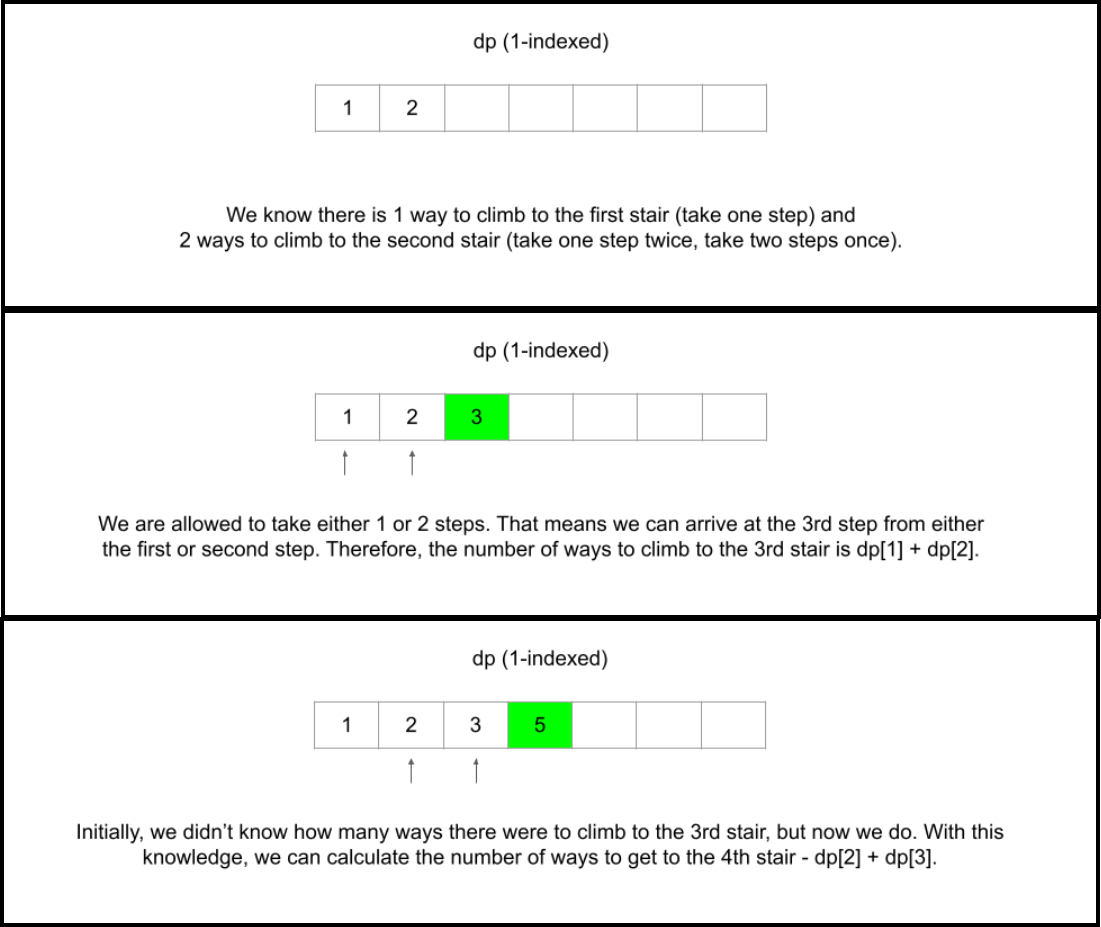
We just talked a whole lot about top-down, but what about bottom-up? Everything is pretty much the same, except we will start from our base cases and iterate up to our final answer. As stated before, bottom-up implementations usually use an array, so we will use an array dp where dp[i] represents the number of ways to climb to the *ith* step.

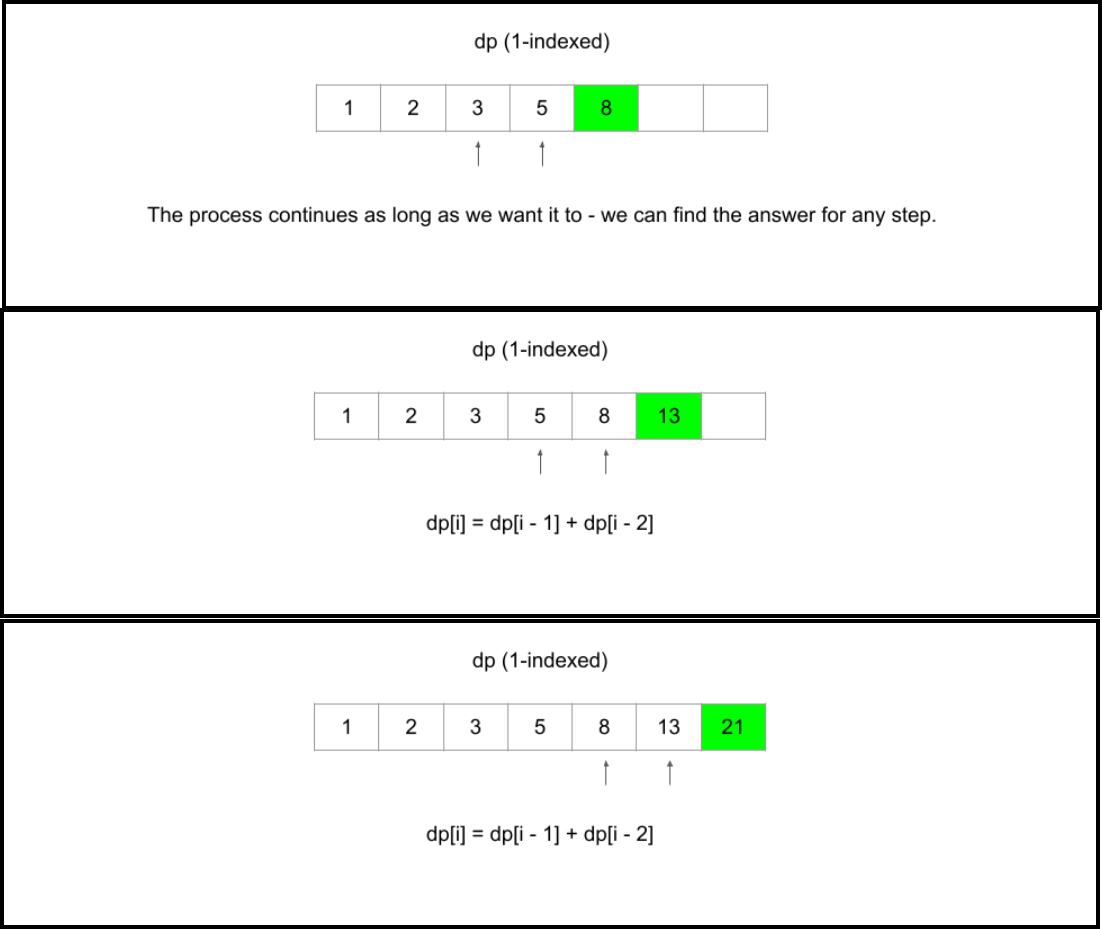
|  |
| --- |
| int climbStairs(int n) {  if (n == 1) return 1;  // An array that represents the answer to the problem for a given state  int\* dp = new int[n + 1];  dp[1] = 1; // Base cases  dp[2] = 2; // Base cases  for (int i = 3; i <= n; i++) {  dp[i] = dp[i - 1] + dp[i - 2]; // Recurrence relation  }  return dp[n];  }  int main() {  *cout* << climbStairs(5) << *endl*;  return 0;  } |

# To Summarize

With DP problems, we can use logical thinking to find the answer to the original problem for certain inputs, in this case we reason that there is 1 way to climb to the first stair and 2 ways to climb to the second stair. We can then use a recurrence relation to find the answer to the original problem for any state, in this case for any stair number. Finding the recurrence relation involves thinking about how moving from one state to another changes the answer to the problem.

This is the essence of dynamic programming. Here's a quick animation for Climbing Stairs:





### Next Up

For the rest of the explore card, we're going to use the framework to solve multiple examples, while explaining the thought process behind how to apply the framework at each step. It may be useful to refer back to the section of this article titled "The framework" as you move along the card. For now, take a deep breath - this was a lot to take in, but soon you will be equipped to start solving DP problems on your own.