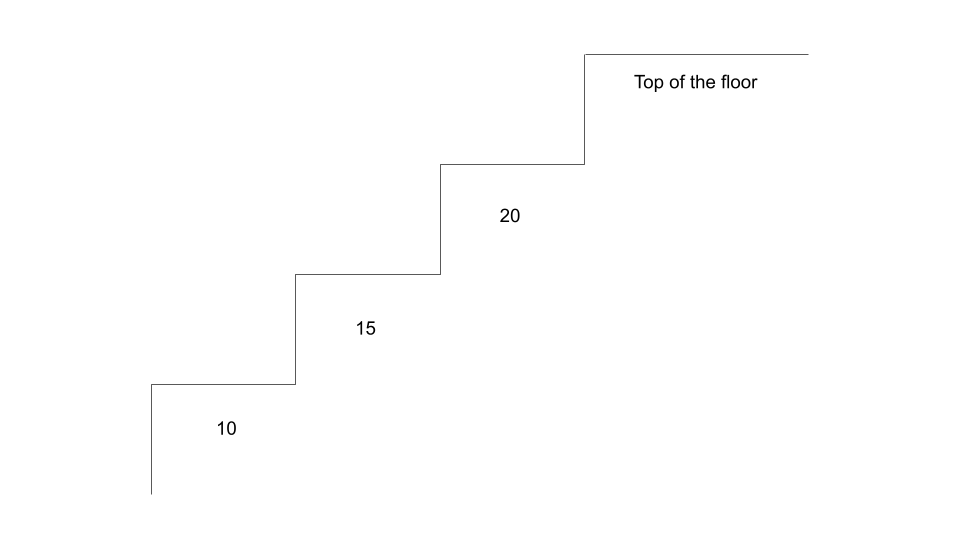
**Overview**

We can make two important observations about this problem. First, we need to find the maximum or minimum of something. Second, we have to make decisions that might look different depending on decisions we made previously. These characteristics are typical of a dynamic programming problem. In this case, we need to make decisions about either taking 1 step or 2 steps at a time, and our goal is to minimize the overall cost.

If you're new to dynamic programming, this question may seem more like a medium. Don't worry though, this is a great problem for getting started with dynamic programming. Generally, there are two main ways to implement a dynamic programming algorithm - top-down and bottom-up. In this article, we will take a look at both.

Before we begin, let's clear up some of the confusion surrounding the problem statement.



The "top of the floor" does not refer to the final index of costs. We actually need to "arrive" beyond the array's bounds.

**Approach 1: Bottom-Up Dynamic Programming (Tabulation)**

**Intuition**

Bottom-up dynamic programming is also known as **tabulation** and is done iteratively. Dynamic programming is based on the concept of **overlapping subproblems** and **optimal substructure**. This is when the solution to a problem can be constructed from solutions to similar and smaller subproblems. Solving a smaller version of the problem can be easier and faster, thus if we break up the problem into smaller subproblems, solving them can lead us to the final solution easier and faster.

Let's look at an example costs = [0,1,2,3,4,5]. Since we can take 1 or 2 steps at a time, we need to reach either step 4 or step 5 (0-indexed), and then pay the respective cost to reach the top. For this example, to reach step 4 optimally would cost **2** by taking path 0 --> 2 --> 4 (we're not counting the cost of step 4 yet since we are only talking about *reaching* the step right now). To reach step 5 optimally would cost **4** by taking path 1 --> 3 --> 5.

Now, imagine that before we started the problem, somebody came up to us and said "to optimally reach step 4 costs **2** and to optimally reach step 5 costs **4**." Well, then the problem is trivial - the answer is the minimum of 2 + cost[4] = 6 and 4 + cost[5] = 9. The only reason this was so easy was because we already knew the cost to reach steps 4 and 5.

So how do we find the minimum cost to reach step 4 or step 5? Well, you might notice that it's the exact same problem, just with a smaller input. For example, finding the minimum cost to reach step 4 is like solving the original problem with input [0,1,2,3] (step 4 is the "top of the floor" now). To solve this subproblem, we need to find the minimum cost to reach steps 2 and 3, which requires us to answer the original problem for inputs [0,1] and [0,1,2].

This pattern is known as a **recurrence relation**, and in this case, the minimum cost to reach the ithi^{th}*ith* step is equal to minimumCost[i] = min(minimumCost[i - 1] + cost[i - 1], minimumCost[i - 2] + cost[i - 2]). As you can see, we get the solution for the ithi^{th}*ith* step by using solutions from earlier steps. So, when does the sequence terminate? For this question, the base cases are given in the problem description - we are allowed to start at either step 0 or step 1, so minimumCost[0] and minimumCost[1] are both 0.

**Algorithm**

With our base cases and recurrence relation, we can now easily solve this problem.

1. Define an array minimumCost, where minimumCost[i] represents the minimum cost of reaching the ithi^{th}*ith* step. The array should be one element longer than costs and start with all elements set to 0.
   * The reason the array should contain one additional element is because we will treat the *top floor* as the *step* to reach.
2. Iterate over the array starting at the 2nd index. The problem statement says we are allowed to start at the 0th0^{th}0*th* or 1st1^{st}1*st* step, so we know the minimum cost to reach those steps is 0.
3. For each step, apply the recurrence relation - minimumCost[i] = min(minimumCost[i - 1] + cost[i - 1], minimumCost[i - 2] + cost[i - 2]). As you can see, as we populate minimumCost, it becomes possible to solve future subproblems. For example, before solving the 5th and 6th steps we are required to solve the 4th step.
4. At the end, return the final element of minimumCost. Remember, we are treating this "step" as the top floor that we need to reach.