Bottom-up (Tabulation)

There are two ways to implement a DP algorithm:

Let's take a quick look at each method.

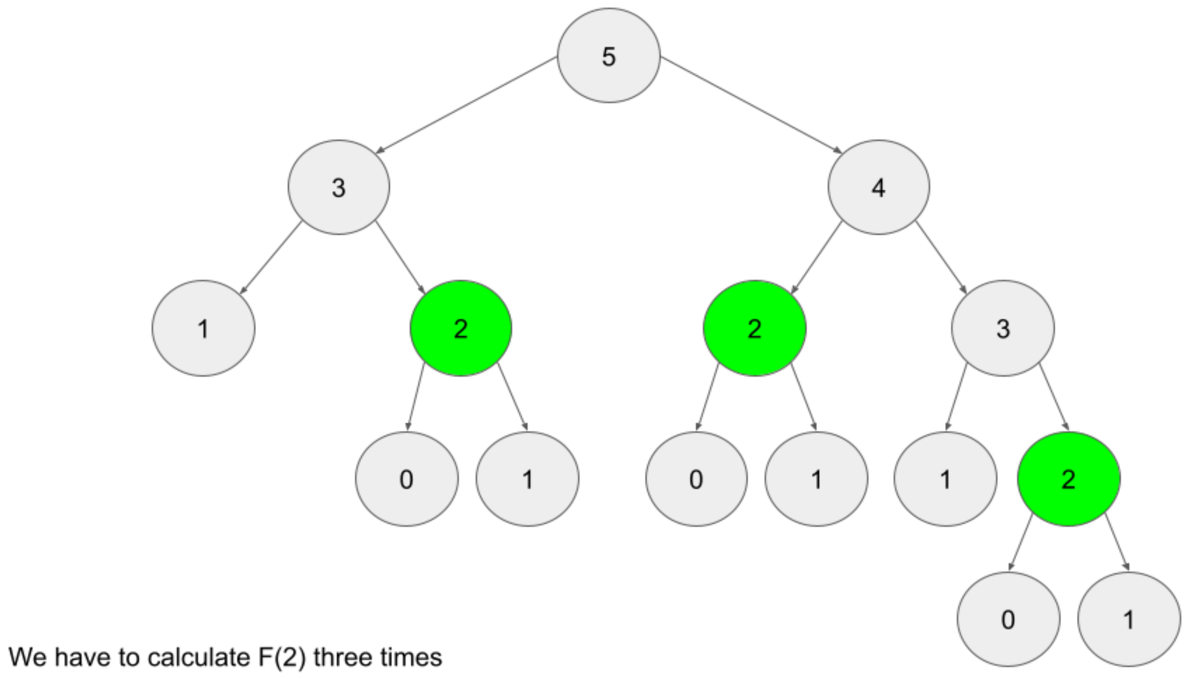
Bottom-up is implemented with iteration and starts at the base cases. Let's use the Fibonacci sequence as an example again. The base cases for the Fibonacci sequence are $F(0)=0$ and $F(1)=1$. With bottom-up, we would use these base cases to calculate $F(2)$, and then use that result to calculate $F(3)$, and so on all the way up to $F(n)$.

```
// Pseudocode example for bottom-up

F = array of length (n + 1)
F[0] = 0
F[1] = 1
for i from 2 to n:
    F[i] = F[i - 1] + F[i - 2]
```

Top-down (Memoization)

Top-down is implemented with recursion and made efficient with memoization. If we wanted to find the *nth* Fibonacci number $F(n)$, we try to compute this by finding $F(n−1)$ and $F(n−2)$. This defines a recursive pattern that will continue on until we reach the base cases $F(0)=F(1)=1$. The problem with just implementing it recursively is that there is a ton of unnecessary repeated computation. Take a look at the recursion tree if we were to find $F(5)$:



Notice that we need to calculate *F*(2) three times. This might not seem like a big deal, but if we were to calculate *F*(6), this **entire image** would be only one child of the root. Imagine if we wanted to find *F*(100) - the amount of computation is exponential and will quickly explode. The solution to this is to **memoize** results.

> **memoizing** a result means to store the result of a function call, usually in a hashmap or an array, so that when the same function call is made again, we can simply return the **memoized** result instead of recalculating the result.

After we calculate *F*(2), let's store it somewhere (typically in a hashmap), so in the future, whenever we need to find *F*(2), we can just refer to the value we already calculated instead of having to go through the entire tree again. Below is an example of what the recursion tree for finding *F*(6) looks like with and without memoization:

Current

```
// Pseudocode example for top-down
```

```
memo = hashmap
Function F(integer i):
    if i is 0 or 1:
        return i
    if i doesn't exist in memo:
        memo[i] = F(i - 1) + F(i - 2)
    return memo[i]
```

Which is better?

Any DP algorithm can be implemented with either method, and there are reasons for choosing either over the other. However, each method has one main advantage that stands out:

- A bottom-up implementation's runtime is usually faster, as iteration does not have the overhead that recursion does.
- A top-down implementation is usually much easier to write. This is because with recursion, the ordering of subproblems does not matter, whereas with tabulation, we need to go through a logical ordering of solving subproblems.

> We'll be talking more about these two options throughout the card. For now, all you need to know is that top-down uses recursion, and bottom-up uses iteration.