Introduction to Dynamic Programming

Let’s go over the Dynamic Programming pattern, its real-world applications, and some problems we can solve with it.

We'll cover the following...

About the pattern

Many computational problems are solved by recursively applying a divide-and-conquer approach. In some of these problems, we see an **optimal substructure**, i.e., the solution to a smaller problem helps us solve the bigger one.

Let’s consider the following problem as an example: Is the string “rotator” a palindrome? We can start by observing that the first and the last characters match, so the string *might* be a palindrome. We shave off these two characters and try to answer the same question for the smaller string “otato”. The subproblems we encounter are: “rotator”, “otato”, “tat”, and “a”. For any subproblem, if the answer is *no*, we know that the overall string is not a palindrome. If the answer is *yes* for all of the subproblems, then we know that the overall string is indeed a palindrom

While each subproblem in this recursive solution is distinct, there are many problems whose recursive solution involves solving some subproblems over and over again (overlapping subproblems). An examp

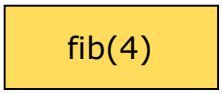
is the recursive computation of the n^{th} Fibonacci number. Let's review the definition of the Fibonacci series:

$$fib(0) = 0$$

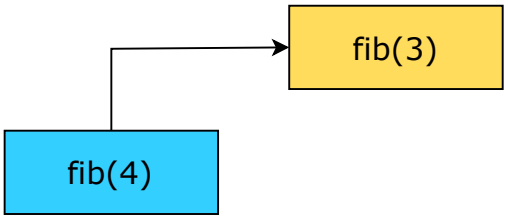
$$fib(1) = 1$$

$$fib(n) = fib(n-1) + fib(n-2), n > 1$$

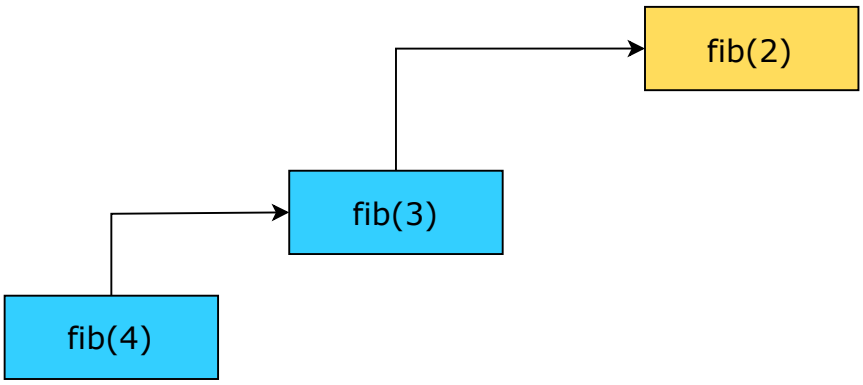
Here's the call tree for the naive recursive solution to this problem, with $n = 4$.



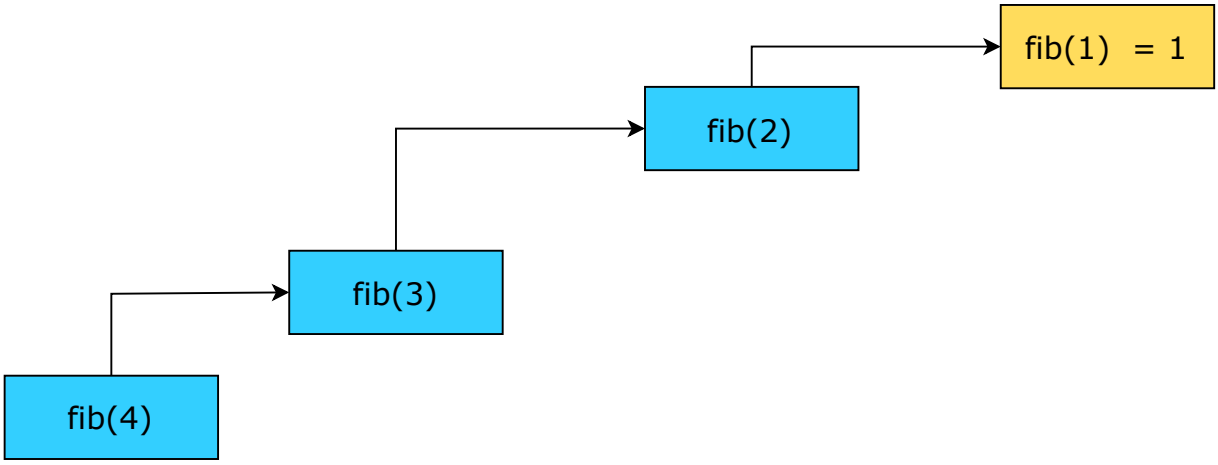
We start by calling our recursive function with **n = 4**.



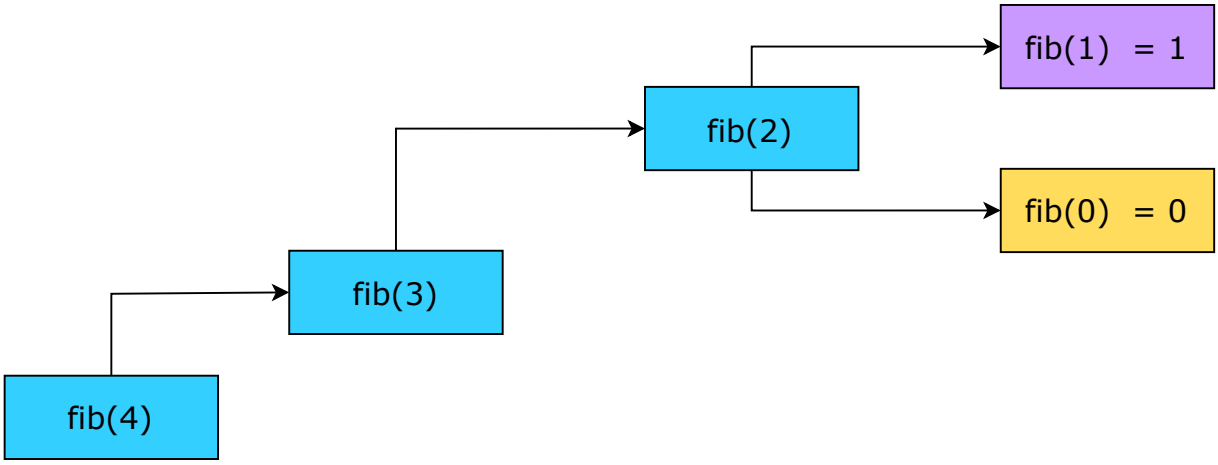
Which will further call **fib(3)**.



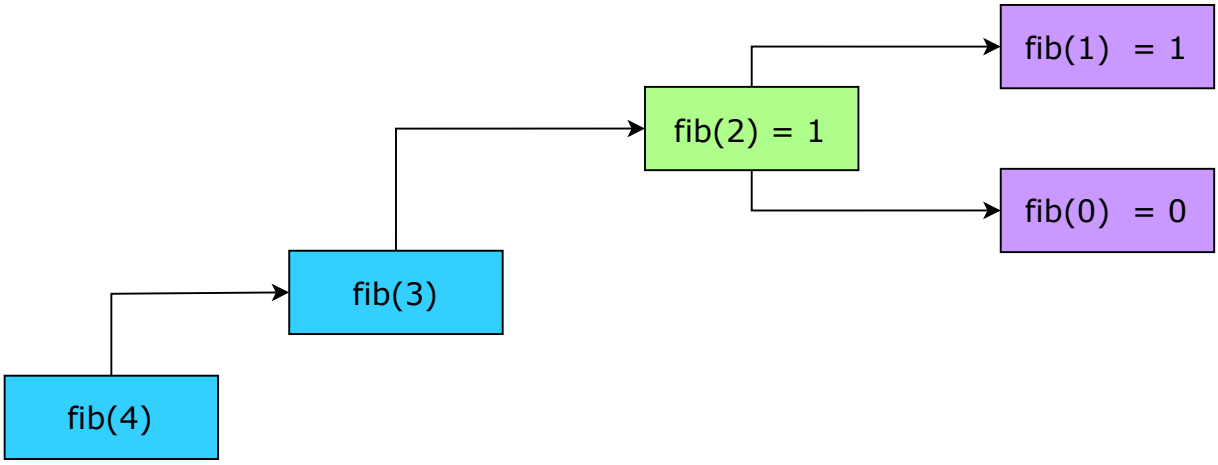
fib(3) will call **fib(2)**.



fib(2) will call **fib(1)** which is a base case and will return **1**.

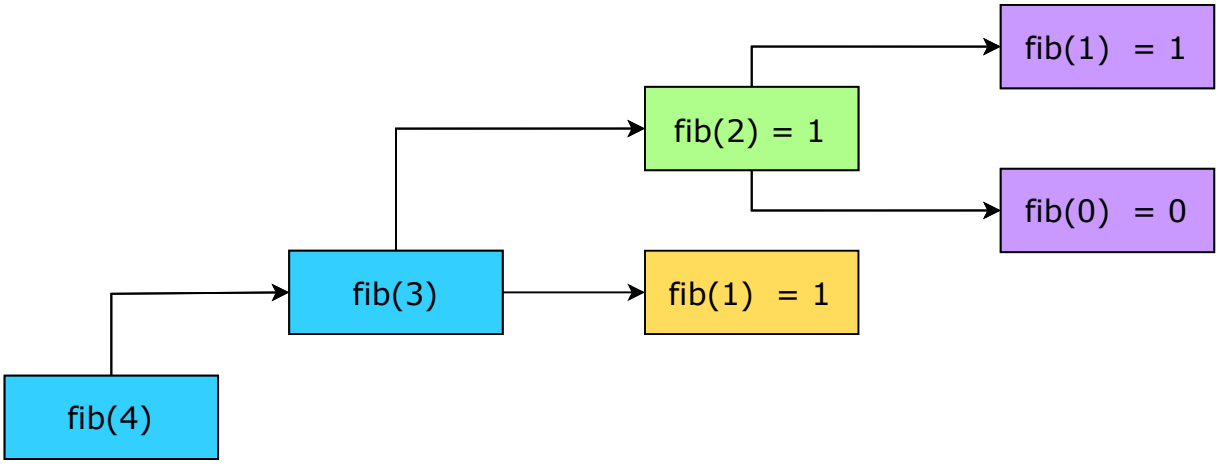


fib(2) will call **fib(0)**, which is a base case as well and will return **0**.



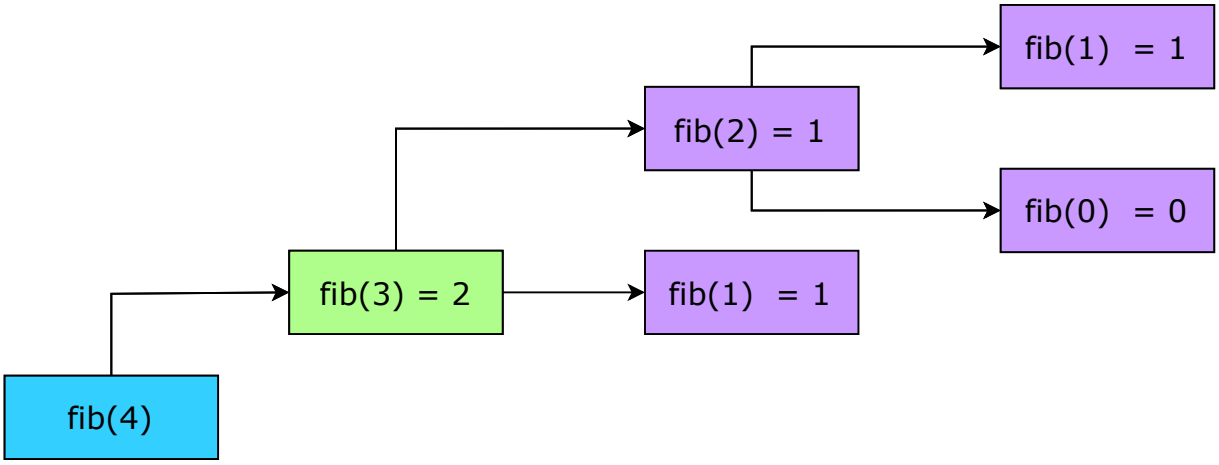
fib(1) + **fib(0)** returns **1** for the result of **fib(2)**.

6 of 13

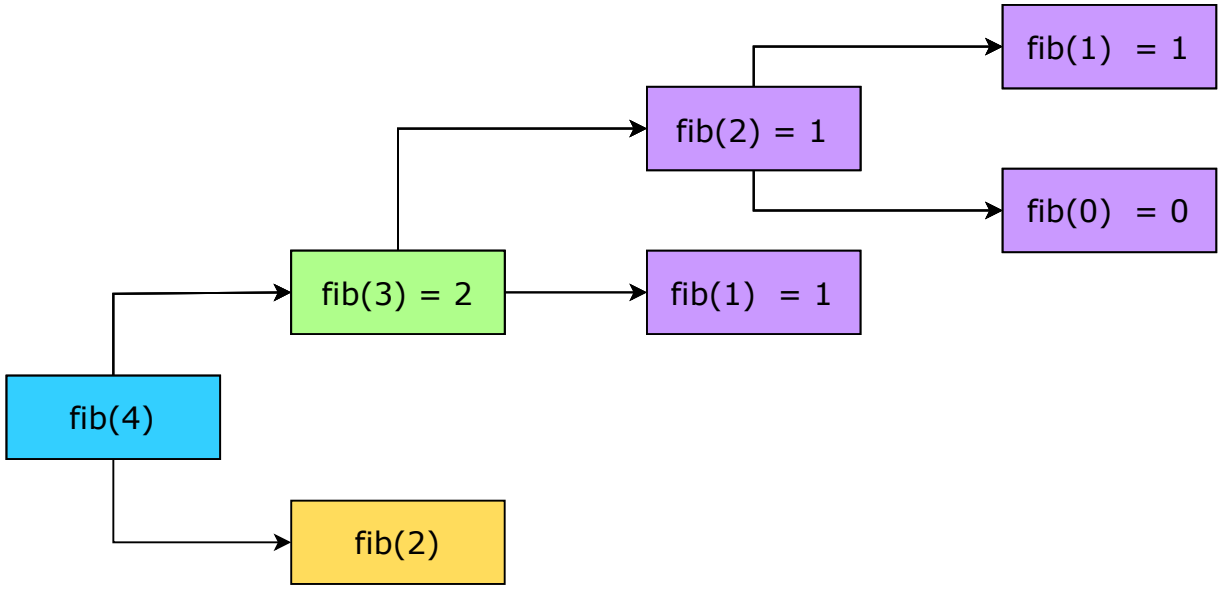


As **fib(2)** returned its value, now **fib(3)** will call **fib(1)**.

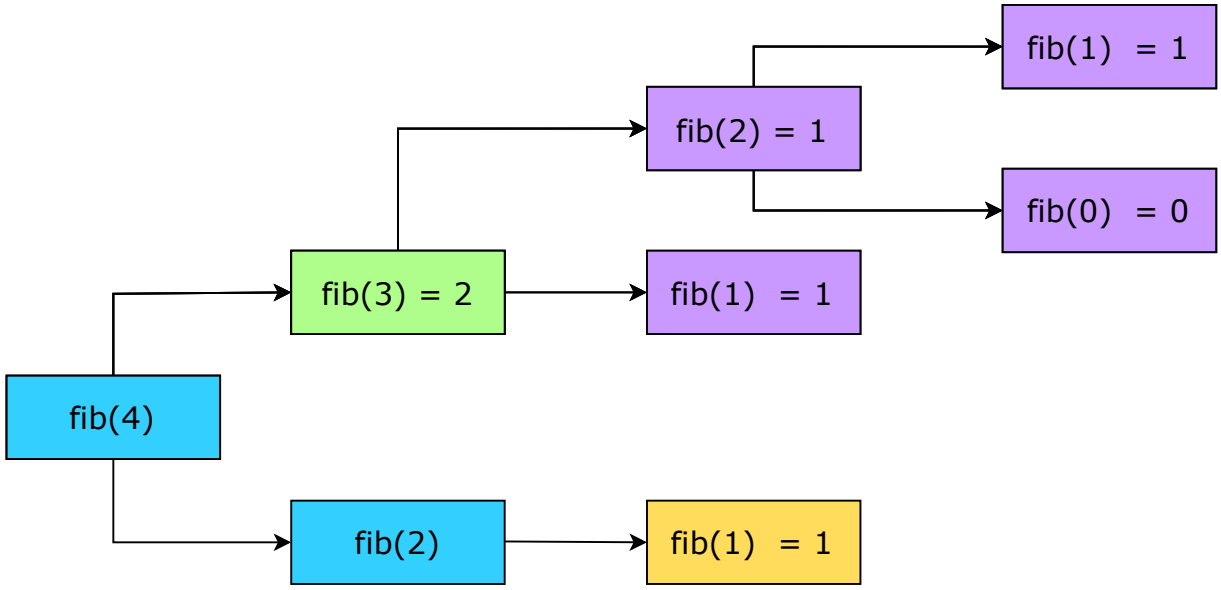
7 of 13



fib(2) + **fib(1)** returns **2** as the result of **fib(3)**.

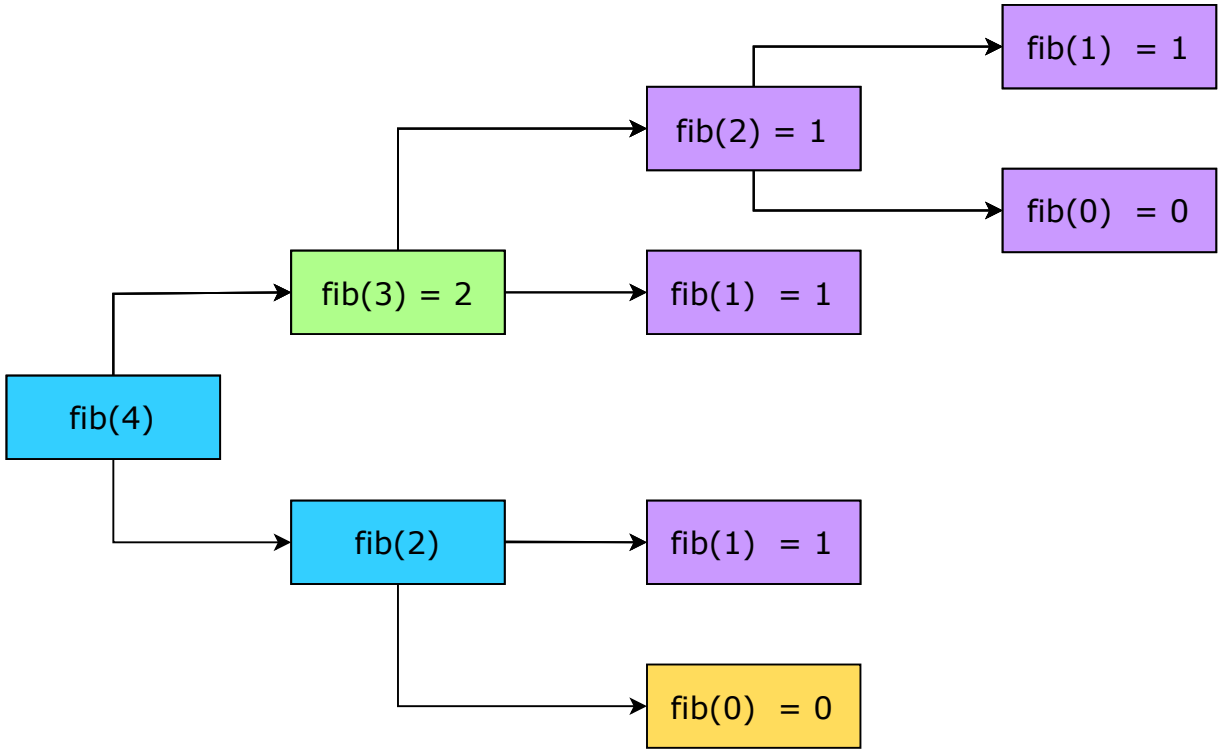


After **fib(3)** returns the result, **fib(4)** will call **fib(2)**.

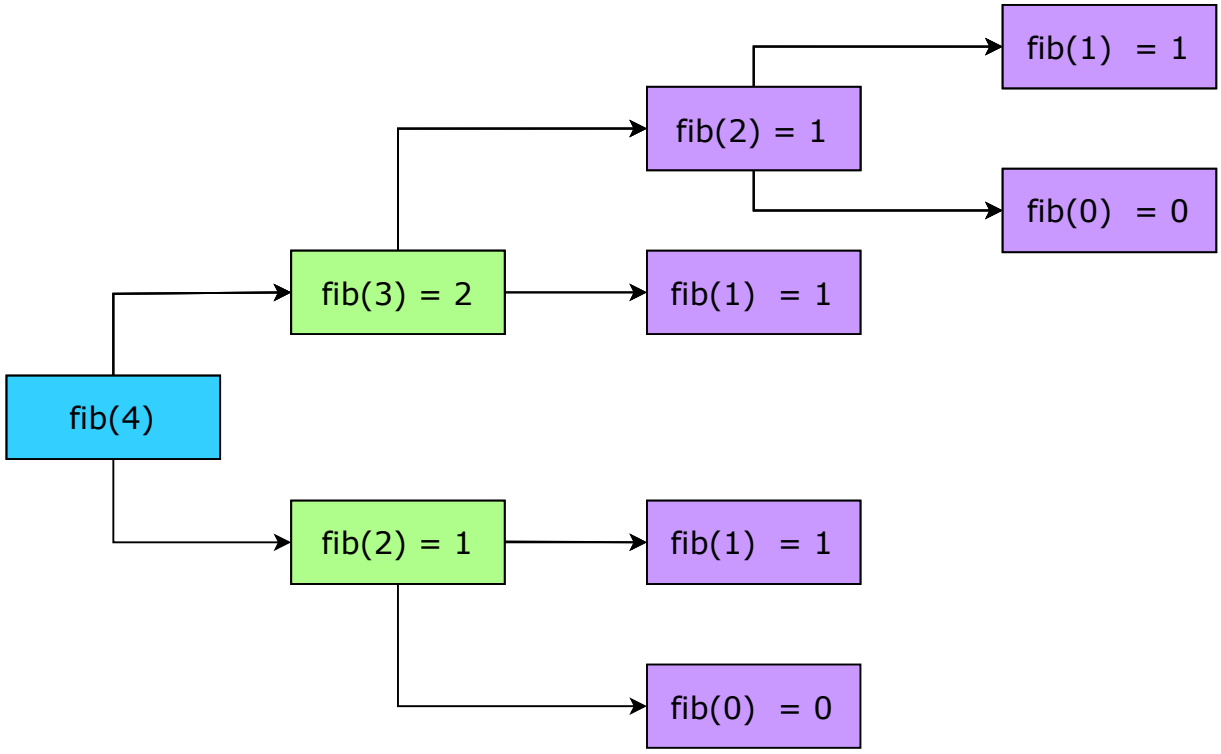


Similarly, **fib(2)** will first call **fib(1),** which returns **1**.

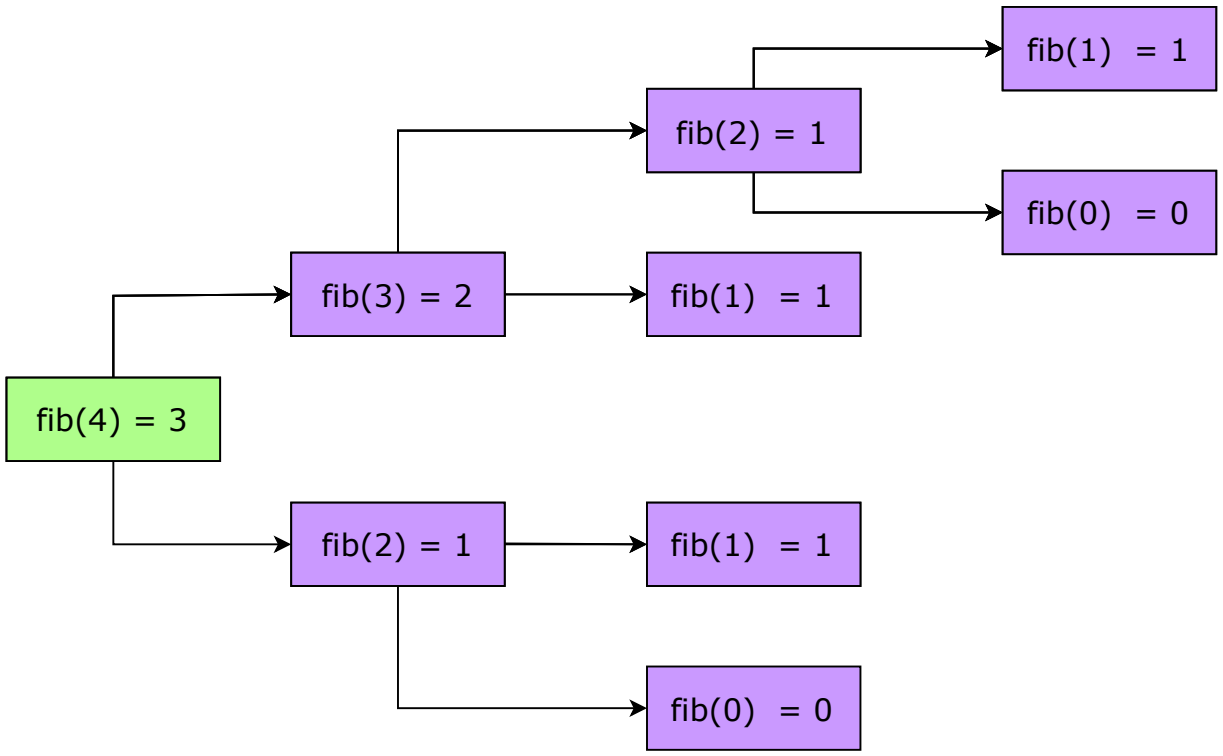
10 of 13



fib(2) will now call **fib(0)**, which returns **0**.



fib(1) + **fib(0)** returns **1** for the result of **fib(2)**.



fib(3) + **fib(2)** returns **3** for the result of **fib(4)**, which is our final answer.

13 of 13

We can see above that the subproblem $fib(2)$ is evaluated twice, $fib(1)$ is evaluated thrice, and $fib(0)$ is evaluated twice. These are repeated or overlapping subproblems. As every subproblem is reevaluated every time it appears, the naive recursive implementation of this solution will have exponential time complexity.

An optimization of such a recursive solution would be to store and reuse solutions to subproblems, reducing the time complexity to polynomial time. Such an approach is called dynamic programming (DP).

We've discussed that we need to save the computations, but how can we save and use them? We use the following two approaches that primarily

save our computations by reusing previous calculations of subproblems:

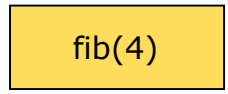
- **Top-down approach:** It is a recursive approach that stores the results of redudant function calls to avoid repeating calculations for the same subproblems.
- **Bottom-up approach:** It is an iterative strategy that systematically fills a table with results of subproblems to solve larger problems efficiently.

Top-down approach

The top-down approach is also known as memoization. It is usually implemented as an enhancement of the naive recursive solution. It uses recursion to break down larger subproblems into smaller ones. The smallest one is solved and the result is stored in a lookup table for use in computing larger subproblems.

To take advantage of the stored results of subproblems, in every call, the top-down recursive function *first* checks if a solution to a subproblem already exists. If it does, the result is fetched from the lookup table *instead* of making a recursive call to compute it. Otherwise, the recursive call is made. As we can see in the illustration below, using the top-down approach, we are able to avoid recomputing the subproblems $fib(2)$, $fib(1)$, and $fib(0)$.

Compared to the naive recursive solution, the top-down approach takes up additional space in memory because it stores intermediate results in a lookup table.



We start by initializing all the values of a lookup table with the **-1**.

lookup table	-1	-1	-1	-1	-1

1 of 12

Bottom-up approach

The bottom-up approach is also known as tabulation. In this approach, the smallest problem is solved first, the results saved, and then larger subproblems are computed based on the evaluated results. In contrast to the top-down approach, which uses recursion to first break down a larg problem into smaller subproblems, the bottom-up approach starts by