Theory: Dynamic programming basics

16 minutes 0 / 5 problems solved

For now, we already know one efficient approach to solving algorithmic problems: **divide and conquer**. Just to remind you, the main idea of this approach is to divide an initial problem into smaller subproblems, solve them, and then combine their solutions to get an answer for the original problem.

In some cases, however, this approach might not be ideal. If some subproblem turns up multiple times during the division, it will be solved repeatedly, thus leading to increased time complexity. An intuitive way to address this limitation is to solve each subproblem only once, save its solution and reuse when necessary. Such an approach that not only breaks down a problem into subproblems but saves and reuses their solutions is called **dynamic programming**.

This approach will be the main subject of discussion in this topic. Let's start with formulating some simple problem and then apply the dynamic programming approach to solve it.

§1. Fibonacci numbers

The **Fibonacci numbers** represent a sequence, the first element of which is 0, the second is 1, and each next element is the sum of the previous two. Below are the first 8 elements of the sequence:

$$0, 1, 1, 2, 3, 5, 8, 13, ...$$

For convenience, we will denote the n-th element of the sequence as F_n. For example, $F_0 = 0, F_1 = 1, F_6 = 8$.

Given a number n, our task will be to calculate F_n. Let's see how the dynamic programming approach can be applied to solve this problem.

§2. Dividing the problem into subproblems

First, we need to find a way to divide the problem into smaller subproblems. In such a simple case, the division is clear: to calculate F_n, we should calculate F_{n-1} and F_{n-2} first, and then find their sum to get F_n. Based on this idea and taking into account that $F_0 = 0$ and $F_1 = 1$, we may implement the following algorithm to find the n-th Fibonacci number:

```
function fib(n):
    if n < 2:
        return n

    return fib(n-1) + fib(n-2)
```

On the one hand, the function works correctly and solves our problem. On the other hand, it is inefficient since it performs many repeated calculations. To understand why it is so, let's consider a tree of recursive calls for `fib(4)`:

Current topic:

Dynamic programming basics

Topic depends on:

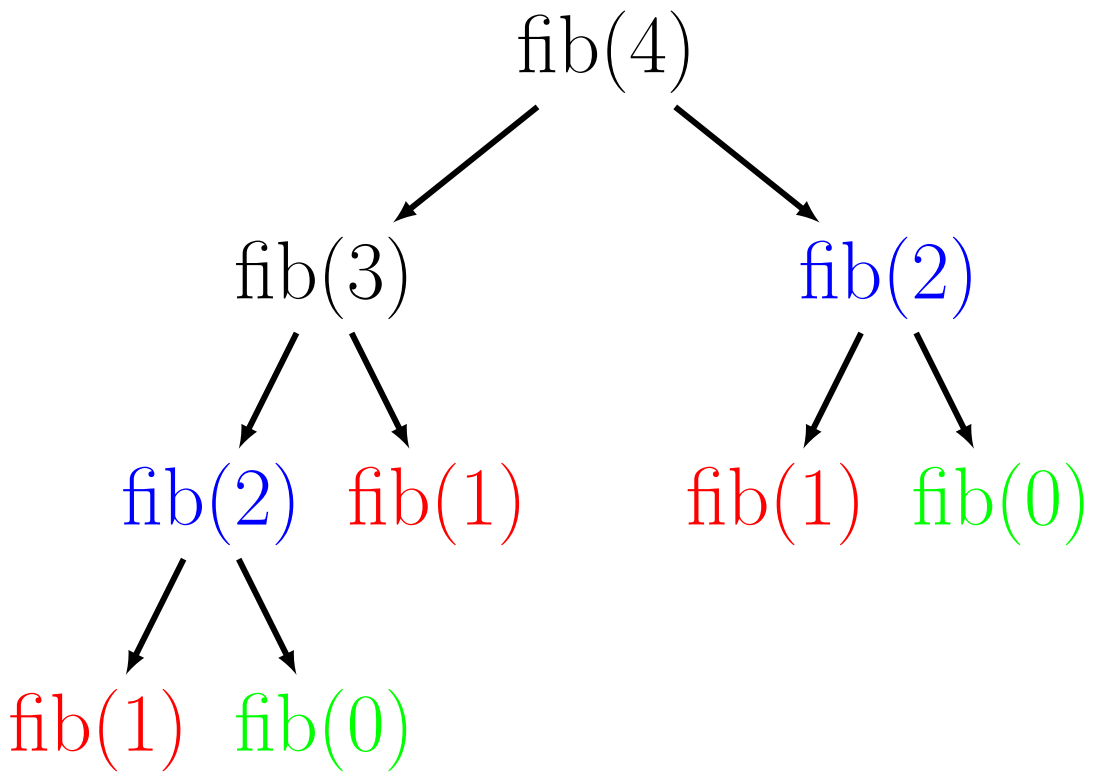
- ✓ Divide and conquer
- ✓ The big O notation

Topic is required for:

Edit distance

Table of contents:

↑ Dynamic programming basics
§1. Fibonacci numbers
§2. Dividing the problem into subproblems
§3. Memoization
§4. Bottom-up method
§5. Summary
Feedback & Comments



In the picture above, nodes of the same color correspond to similar subproblems. As you can see, $fib(1)$ is calculated three times, $fib(0)$ and $fib(2)$ are calculated twice. The larger the n, the more repeated calculations the algorithm performs, which increases the running time.

§3. Memoization

To solve this unpleasant issue, we need to apply **memoization**. The idea here is to use an array to store the answers for subproblems and reuse these answers when necessary. The pseudocode below is the previous algorithm complemented by memoization:

```
function fib(n):
    numbers = array of size (n + 1)

    for (i = 0; i <= n; i = i + 1):
        numbers[i] = -1

    return fib_helper(n, numbers)

function fib_helper(n, numbers):

    if numbers[n] != -1:

        return numbers[n]



    if n < 2:

        numbers[n] = n

    else:

        numbers[n] = fib_helper(n-1, numbers) + fib_helper(n-2, numbers)



    return numbers[n]
```

In the `fib` function, we create an array `numbers` to store the answers for subproblems. Initially, each element of the array is assigned to -1 indicating that the element is not calculated yet. Next, we call the `fib_helper` function passing the number n to be calculated and the `numbers` array as parameters.

The first condition in the `fib_helper` function is the very place where memoization starts working: we check if the current number n is already calculated and if it is so, we immediately return it as an answer. Otherwise, we calculate this number either directly or using recursion, save it into the array and only then return it.

Since each number is calculated exactly once here, the algorithm works in $O(n)$, thus being more efficient than the previous one.

§4. Bottom-up method

The way the previous function is implemented is called the **top-down** method: we start with an initial problem (top), and then break it down into smaller subproblems (down). However, there is another way to solve the same problem called the **bottom-up** method. In this method, we start calculating answers directly from the smallest subproblems (bottom) and based on their solutions find answers to larger subproblems (up). Below is an algorithm for finding the n-th Fibonacci number implemented with the bottom-up method:

```
function fib_bottom_up(n):
    if n < 2:
        return n

    numbers = array of size (n + 1)
    numbers[0] = 0
    numbers[1] = 1

    for (i = 0; i <= n-2; i = i + 1):

        numbers[i + 2] = numbers[i + 1] + numbers[i]



    return numbers[n]
```

Similarly to the previous function, we create an array `numbers` to store the answers for subproblems. However, we don't use recursion here and start calculating numbers from smallest to largest: the first two are initialized directly, and the remaining are calculated with the `for` loop. Finally, we return the n-th Fibonacci number as an answer.

As you can see, this function is simpler than the previous. Also, it requires less additional memory since it doesn't use recursion. For these reasons, this method is used more often when applying the dynamic programming approach.

§5. Summary

In this topic, we learned what dynamic programming is and applied it to calculate the n-th Fibonacci number. In addition to this problem, there are many others that can be solved using the DP approach. In general, use dynamic programming when your problem can be broken down into smaller subproblems identical to the initial one. Remember about applying memoization to avoid repeated computations and thus make the algorithm even more efficient.