Theory: Introducing the first algorithm

14 minutes 3 / 4 problems solved

Start practicing

4760 users solved this topic. Latest completion was about 1 hour ago.

Computer algorithms are everywhere around us: they help us find the shortest paths on a map between two points of a city, quickly search the most relevant information on the Web, automatically perform spelling correction, and many other useful things. To understand how all these technologies work under the hood and to be able to efficiently solve similar problems, it is important for software developers to learn algorithms, get familiar with their pros and cons, and be able to choose when to apply them.

In this topic, we will continue learning the subject and will implement our first algorithm. Although the example will be simple, we will see and discuss some pitfalls developers may encounter while working on algorithms. We will also give a brief introduction about the efficiency of algorithms, explain why it is important, and how to estimate it.

§1. Introducing the Fibonacci sequence

Throughout this topic, we will use a mathematical structure called the Fibonacci sequence as a basis of our discussion. This sequence can be defined as follows: the first two elements are 0 and 1, and each of the next is the sum of the previous two. Here are the first nine elements of the sequence:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

More formally, the n-th element F_n of the sequence can be defined as follows:

$$F_n = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ F_{n-1} + F_{n-2} & \text{if } n > 1. \end{cases}$$

Our task will be to come up with an algorithm that for a given number n finds the n-th element of the sequence.

§2. A naive algorithm

Below is an algorithm for finding the n-th Fibonacci number written using pseudocode:

```
function fib(n):
    if n == 0: return 0
    if n == 1: return 1
    return fib(n-1) + fib(n-2)
```

Pseudocode is a simple language similar to real programming languages. Comparing with the latter, it has a much simpler syntax. Thus, it is often used to describe algorithms since its simplicity allows avoiding some implementation details and concentrating on the main idea of an algorithm. In the following topics, we will also often use pseudocode to describe algorithms.

The above function works as follows:

1. First, it checks whether the input number n is equal to zero. If it is the case, it returns the corresponding Fibonacci number. Otherwise, it performs a similar check comparing n with 1.
2. If none of the first two conditions holds, the function performs a recursive call to find the $n-1$ and $n-2$ Fibonacci numbers and then returns their sum as a final result.

Current topic:

✓ Introducing the first algorithm

Topic depends on:

✓ Computer algorithms

Topic is required for:

✓ The big O notation

Table of contents:

After we completed implementing an algorithm, the first thing we must ask ourselves is whether the algorithm is correct. For the above algorithm, we would like to know whether it indeed finds the n-th element of the Fibonacci sequence. Here, the answer is obviously yes, because the algorithm literally follows the definition of the n-th Fibonacci number.

The next question that we should ask is "how long does it take an algorithm to solve a problem?". For the above algorithm, the question is "how long does it take the algorithm to calculate the n-th Fibonacci number?". Comparing with the first one, this question is a bit trickier, so let's discuss it in more detail.

§3. Analyzing the naive algorithm

Before we begin analyzing the implemented algorithm, let's agree on some assumptions.

First, we don't actually want to know the exact amount of time the algorithm will be running. This is because it heavily depends on the computer used. Instead, we will estimate how many operations the algorithm requires to be completed. Thus, we will get an estimation that does not depend on a particular computer.

Secondly, we need to agree on what we mean by one operation. We will assume that checking the condition of the `if` statement requires one operation, and the addition of two integer numbers requires one operation as well.

The above assumption is a significant simplification. To sum two numbers, computers actually perform more than two operations. This is because CPU should at least read numbers from RAM, only then perform the addition, and then write the numbers back. Also, during the computations, some data might be cached. In other words, the real things are much more complex. However, we will apply the above assumptions in order to simplify our further analysis and get a metric that does not depend on a particular machine or computer architecture.

Now, let's estimate the number of operations the algorithm requires to calculate the n-th Fibonacci number taking the above assumptions into account. Let $T(n)$ be the number of such operations for a given n. Then, the number of operations can be expressed using the following recurrence formula:

$$T(n) = \begin{cases} 1 & \text{if } n = 0, \\ 2 & \text{if } n = 1, \\ T(n-1) + T(n-2) + 3 & \text{if } n \geq 2. \end{cases}$$

If n is equal to zero, the algorithm performs only one comparison. If n is 1, it performs two such checks. Otherwise, it also performs two checks first, and then makes $T(n-1)$ operations to find the $(n-1)$-th Fibonacci number, $T(n-2)$ to find the $(n-2)$-th element of the sequence, and one more addition thereafter.

It is hard to give any estimations just using namely this recurrence formula, and it would be more convenient to transform it into a direct equation for n. For the sake of simplicity, we will avoid a formal mathematical proof here and just give you the final result, which is as follows:

$$T(n) \geq 1.6^n$$

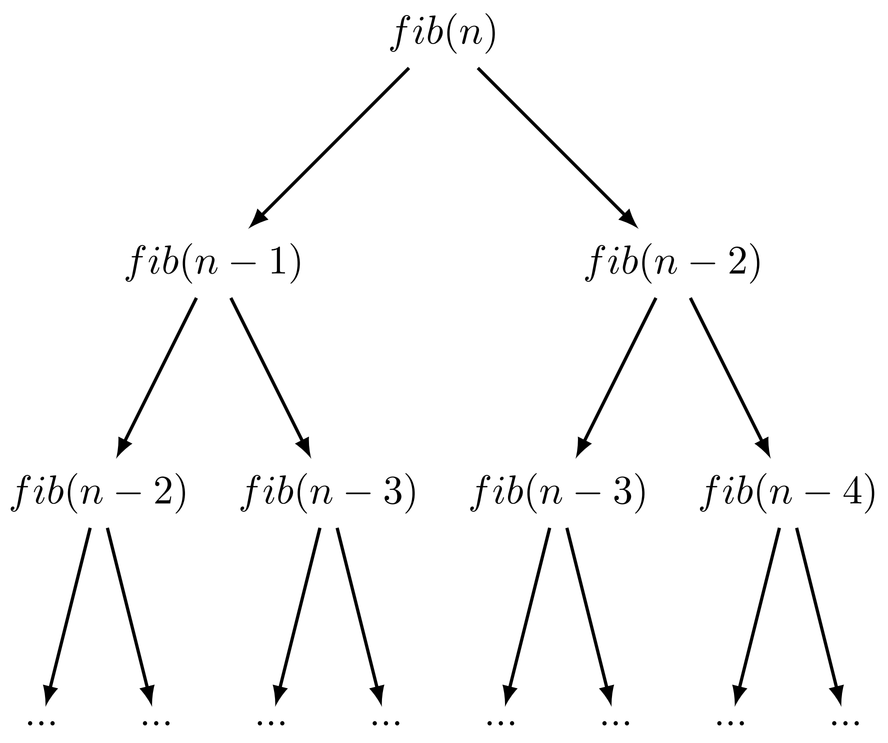
If you want to ensure that the equation indeed holds, you may try to prove it yourself by induction.

To understand why this is very slow, let's see how much time it takes to calculate the 90-th Fibonacci number. If we assume that our computer is able to perform 10^9 operations per second, the total amount of time needed to complete the calculations is:

$$\frac{1.6^{90}}{10^9} \approx 2.3 \cdot 10^9 \text{ seconds} \approx 74 \text{ years.}$$

You may wonder why we assume that the computer is able to perform namely 10^9 operations per second. The short answer is that this performance roughly corresponds to the performance of personal computers at the moment of writing this article. We should emphasize that this is one more simplification and actually the performance and thus running time may vary among different computers. However, the main point here is that the algorithm anyway takes decades to be completed thus being very inefficient.

Thus, our algorithm works correctly, but it is hopelessly slow. To understand why it is so, let's consider a tree of recursive calls the algorithm builds during the execution:



As you can see, there are a lot of calculations that the algorithm repeats several times. For example, $fib(n-3)$ is executed twice at the third level of recursion, and one more time at the fourth. Calculation for smaller numbers are performed even more times. Thus, many repeated calls cause such a huge number of operations.

A natural question that arises here is "can we improve our algorithm?". And the answer is yes. An obvious idea is to calculate each element of the sequence only once, save the result and reuse it then calculating the next elements. Let's see how we can improve our algorithm using this idea.

§4. Improving the naive algorithm

Here is an improved version of an algorithm for finding the n-th Fibonacci number:

```
function fib(n):
    if n == 0: return 0
    f = array of size (n + 1)
    f[0] = 0
    f[1] = 1
    for (i = 2; i <= n; i = i + 1):
        f[i] = f[i-1] + f[i-2]
    return f[n]
```

The above function works as follows:

1. First, it checks if the input number n is equal to zero. If it is the case, there is nothing to calculate and it returns 0 as an answer.
2. Otherwise, the function creates an array of size $n + 1$ to store the elements of the Fibonacci sequence.
3. Next, it starts calculating the elements in the loop one by one reusing the elements calculated at the previous steps.
4. Finally, it returns the n-th element of the array as an answer.

Now let's calculate the number of operations this algorithm performs and compare it with the naive one.

§5. Analyzing the improved algorithm

At the first step, the algorithm makes one operation since it simply needs to check the condition of the `if` statement.

Next, it creates an array of size $n+1$:

```
f = array of size (n + 1)
```

We will assume that this requires $n+1$ operations since the algorithm needs to initialize each element of the array with zero.

Then, the algorithm initializes the first two elements of the sequence:

```
f[0] = 0
f[1] = 1
```

We will assume that getting the i-th element of the array requires 1 operation, as well as the assignment.

Next, it performs $n-1$ iterations in the loop, making 7 operations at each iteration (it gets an element of an array 3 times, makes 1 addition, 2 subtractions, and 1 assignment):

```
for (i = 2; i <= n; i = i + 1):
    f[i] = f[i-1] + f[i-2]
```

We should also take into account that the algorithm performs some operations with the `i` counter. Namely, it first initializes it, which requires one operation. Next, it compares the counter with n at each iteration making n comparisons in total. After each of the comparisons except the last one, the algorithm needs to increment and reassign the counter. It requires $2 \cdot (n-1)$ more operations.

Finally, it returns the n-th element of the array as an answer:

```
return f[n]
```

Thus, the total number of operations is

$$T(n) = 1 + (n+1) + 2 + 2 + 7 \cdot (n-1) + 1 + n + 2 \cdot (n-1) + 1 = 11 \cdot n - 1.$$

The above approach for calculating the number of operations is usually not used to estimate the efficiency of algorithms. This is because it is too detailed and it would be tedious to apply it for more sophisticated algorithms. There is the so-called big O notation commonly used for such an analysis. Since the big O notation requires a separate topic to be discussed (and we will discuss it later), we used another way to calculate the number of operations that requires no specific prior knowledge.

To calculate the 90-th Fibonacci number, a computer with the same performance as above needs

$$\frac{11 \cdot 90 - 1}{10^9} \approx 9.9 \cdot 10^{-7} \text{ seconds.}$$

As you can see, a small algorithmic change has led to a significant efficiency improvement.

Although the two given algorithms solve the same problem, the naive algorithm is inefficient and impossible to use to find large Fibonacci numbers. The second one is much more efficient and can be applied to find large Fibonacci numbers in a reasonable time. Thus, it is always important to think about the efficiency of algorithms, since depending on how efficient your algorithm is it might be suitable (or not) to be applied for real tasks and on real data.

§6. Summary

In this topic, we have discussed an algorithm for finding the n-th element of the Fibonacci sequence. First, we have implemented a naive version, that turned out to be very inefficient. Then, we have improved this version and have got a much more efficient algorithm for solving the same problem. The main points to pay attention to are:

- When you come up with some algorithm, it is important to ask yourself whether the algorithm is correct. In our simple example, the correctness was obvious. However, there are many sophisticated algorithms whose correctness is not that obvious and it might require a lot of effort to prove that some algorithm indeed works. We will consider such algorithms in the following topics.
- Another important thing to think about is efficiency. As you can see from the above examples, there might be a significant difference in efficiency between two algorithms that solve the same problem.

We have also given a basic example of how to estimate the number of operations the algorithms require to be completed. It's important to have an estimation that does not depend on a particular machine or computer architecture. In the following topics, we will continue discussing this approach and will talk about the so-called big O notation.

Report a typo

Feedback sent!

Start practicing

Comments (17)   Hints (0)   Useful links (1)   Show discussion