Lesson: Algorithm complexity

Bookmark this page

In this lesson you'll learn about the **complexity of an algorithm**, which helps you to understand its behavior with respect to the size of the input data.

Video

Start of transcript. Skip to the end.

It's me again, welcome back!

I can see you're hungry for some more concepts on algorithms.

In this lesson, we're going to take a look at the so-called complexity of algorithms.

An introduction to complexity of algorithms

The complexity of an algorithm is a measure

of the maximum number of elementary operations that are needed for its

---

An introduction to complexity of algorithms

The complexity of an algorithm is a measure of the maximum number of elementary operations that are needed for its execution, as a function of the size of the input.

It is obvious that different inputs of the same size may cause an algorithm to behave differently. As a consequence, the function describing its performance is usually an upper bound on the actual performance, determined from the worst case inputs to the algorithm.

So you should remember here that complexity contains a notion of *worst case*.

However, to fully understand the concept of complexity, we need to define two things more precisely. Ready?

- An *elementary operation* is usually an operation that can be executed in a very short time by the Central Processing Unit, or CPU, of the computer on which the algorithm is implemented. For example, addition, multiplication, and memory access are considered as elementary operations. Sometimes, adding or multiplying numbers can be very costly in terms of time. This can be true when adding or multiplying numbers that are very large integers, such as the ones used in certain cryptographic algorithms. So, in other words, defining elementary operations depends on the task.
- The second thing we should understand is that size of the input depends on the context of the algorithm. For example, if we consider the addition of two integers, the size would be the base 2 logarithm of the largest number, as the number of bits used to represent it in memory. In the case of algorithms involving graphs, size may be the number of edges or vertices.

Let's move on. Complexity is generally expressed in the form of *asymptotical behavioral* using the "big-O" $\mathcal{O}$ notation because knowing this asymptotical behavioral is often sufficient to decide if an algorithm is fast enough, and it's often useless to have more accurate estimations.

The asymptotical behavior expressed by the big-O notation makes us drop constants and low-order terms. This is because when the problem size gets large enough, those terms don't matter anymore.

Note that two algorithms can have the same big-O time complexity, even though one is always faster than the other.

For example, suppose that algorithm 1 requires n^2 time, and algorithm 2 requires $10n^2 + 3n$ time. For both algorithms, the complexity is $\mathcal{O}(n^2)$, but algorithm 1 will always be faster than algorithm 2 with the same input. In this case, the constants and

low-order terms do matter.

However, constants do not matter for the scalability question, which asks how the algorithm's execution time changes as a function of the problem size. Although an algorithm that requires n^2 time will always be faster than an algorithm that requires $10n^2 + 3n$ time, for both algorithms, for n large enough and if the problem size doubles, the actual time will approximately quadruple.

Typically, algorithms considered as fast are the ones with a linear complexity, $\mathcal{O}(\mathbf{n})$, or the ones for which the complexity is $\mathcal{O}(\mathbf{n}log(\mathbf{n}))$. Algorithms for which complexity is $\mathcal{O}(\mathbf{n^k})$ for $k \geq 2$ are considered to have reasonable complexity, depending on the application field. Algorithms that have at least an exponential complexity are rarely used in practice.

Complexity of the algorithms that we have seen so far

Let's put this into context with the algorithms we've seen so far. Both the DFS and BFS examine each vertex of a graph once, and for each vertex, they consider all of its neighbors. Therefore, their complexity is in the order of the number of edges in the graph, $\mathcal{O}(|E|)$.

Regarding Dijkstra's algorithm, the complexity will largely depend on the implementation of the min-heap. In practice, it's possible to obtain a complexity of $\mathcal{O}(|E| + |V| \ln(|V|))$. However, we won't demonstrate this result here.

This ends today's lesson on the complexity of algorithms. As you might have guessed, complexity is an important tool to evaluate the performance of an algorithm before starting to implement it. For example, imagine that you want to implement an artificial intelligence in a game. As time complexity is directly linked to calculation time, and calculation time directly relates to the response time of your artificial intelligence, you can see how important this matter is if you wish to beat your opponents.

See you next week, where you'll learn about a very important problem in computer science : the traveling salesman problem, or TSP. Unfortunately, we'll also see that the algorithms to solve it, are very complex. But luckily, this does not mean that they are complicated to understand.

‹ Previous

Next ›

Media Kit
Donate

深圳市恒宇博科技有限公司 粤ICP备17044299号-2