Comparing Algorithms

This chapter will cover different types of complexity measures like Big O and their uses.

We'll cover the following

Introduction

There are typically several different algorithms to solve a given computational problem. It is natural, then, to compare these alternatives. But how do we know if algorithm A is better than algorithm B?

Important Criteria: Time and Space

One important factor that determines the “goodness” of an algorithm is the amount of time it takes to solve a given problem. If algorithm A takes less time to solve the same problem than does algorithm B, then algorithm A is considered better.

Another important factor to compare two algorithms is the amount of memory required to solve a given problem. The algorithm that requires less memory is considered better.

Comparing Execution Time

For the remainder of this lesson, we will focus on the first factor, i.e., execution time. How do we compare the execution time of two algorithms?

Experimental Evaluation

Well, we could implement the two algorithms and run them on a computer while measuring the execution time. The algorithm with less execution time wins. One thing is for sure, this comparison must be done in a fair manner. Let’s try to punch holes into this idea:

- An algorithm might take longer to run on an input of greater size. Thus, algorithms being compared must be tested on the same input size. But that's not all. Due to the presence of conditional statements, for a given input size, even the same algorithm's running time may vary with the actual input given to it. This means that the algorithms being compared must be tested on the same input. Since one algorithm may be disadvantaged over another for a specific input, we must test the algorithms exhaustively on all possible input values. This is just not possible.
- The algorithms being compared must first be implemented. What if the programmer comes up with a better implementation of one algorithm than the other? What if the compiler optimizes one algorithm more than it does the other? There's so much that can compromise the fairness at this stage.
- The programs implementing the two algorithms must be tested on exactly the same hardware and software environment. Far fetched as it may be, we could assign a single machine for all scientists to test their algorithms on. Even if we did, the task scheduling in modern day operating systems involves a lot of randomness. What if the program corresponding to "the best" algorithm encounters an excessive number of hardware interrupts? It is impossible to guarantee the same hardware / software environment to ensure a fair comparison.

Analytical Evaluation

The above list highlights some of the factors that make fair experimental evaluation of algorithms impossible. Instead, we are forced to do an analytical / theoretical comparison. The two key points that we hold on to, from the previous discussion, is that we must compare algorithms for the same input size and consider all possible inputs of the given size. Here is how it is done.

We assume a hypothetical computer on which some primitive operations are executed in a constant amount of time. We consider a specific input size, say, n. We, then, count the number of primitive operations executed by an algorithm for a given input. The algorithm that results in fewer primitive operations is considered better.

What constitutes a primitive operation, though? You can think of these as simple operations that are typically implemented as processor instructions. These operations include assignment to a variable or array index, reading from a variable or array index, comparing two values, arithmetic operations, context switch that results from a function call.

> When the control is transferred from the calling function to the called function, that is what we call a *context switch*.

What is not considered a primitive operation? While the context switch is a primitive operation, the function invocation as a whole is not always a single primitive operation. A function call's cost is considered to be the sum of the primitive operations in the function body. Similarly, displaying an entire array is not a primitive operation.

The number of times conditional statements run depends on the actual input values. Sometimes a code block is executed, some times it isn't. So, how do we account for conditional statements? We can adopt one of three strategies: best case analysis, average-case analysis, and worst-case analysis.

Best Case Analysis

In the best case analysis, we consider the specific input that results in the execution of the fewest possible primitive operations. This gives us a lower bound on the execution time of that algorithm for a given input size.

Worst-Case Analysis

In the worst-case analysis, we consider the specific input that results in the execution of the maximum possible primitive operations. This gives us an upper bound on the execution time of that algorithm for a given input size.

Average Case Analysis

In the average case analysis, we try to determine the average number of primitive operations executed for all possible inputs of a given size. This is not as easy as it may sound to the uninitiated. In order to compute the average-case running time of an algorithm, we must know the relative frequencies of all possible inputs of a given size. We compute the weighted average of the number of primitive operations executed for each input. But how can we accurately predict the distribution of inputs? If the algorithm encounters a different distribution of inputs in the field, our analysis is useless.

The Verdict

The best-case analysis has limited value because what if you deploy that algorithm and the best case input rarely occurs. We feel that the worst-case analysis is more useful because whatever answer it gives you, you can be sure that no matter what, algorithm A wouldn't incur more time than that. Unless otherwise specified, our analysis in this course will be worst-case running time.

The running time of an algorithm computed in the aforementioned way is also known as its **time complexity**. Another term that you will often hear is an algorithm's **space complexity**. The space complexity of an algorithm is the amount of additional or auxiliary memory space that the algorithm requires. This is memory space other than the actual input itself. We will see examples of evaluating the space complexity of an algorithm later on in this course.

Analyzing a Simple Python Program

Suppose that instead of an algorithm, we were given Python code, instead. Here's how we can analyze the algorithm underlying the given program. Let's count the number of primitive operations in the program given below:

```
1  x = 0
2  x += 1
3  print(x)
```

There is a variable assignment on line number 1, so that's one primitive operation. A variable's value is accessed, an addition is performed and then an assignment takes place on line 2, so that's three primitive operations. A variable's value is accessed and then displayed on line 3, so that's two primitive operations. So, overall there are **six primitive operations** in the above program. This analysis is also presented in the following table:

Line No.	Primitive operations	No. of Primitive Operations
1	Variable assignment	1
2	Variable access, addition, variable assignment	3
3	Variable access, display value	2

For the above program, the time complexity can be specified as:

$$\text{Time Complexity} = 1 + 3 + 2 \ \Rightarrow 6$$

Note that there is no notion of input size in this simple example, since there is no input to this program. Algorithms for which the time complexity is independent of the input size are known as constant-time algorithms.

In the next lesson, we will analyze the running time of an algorithm involving a simple loop.

Next →

Example 1: Measuring Time Complexity

Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/comparing-algorithms__introduction-to-complexity-measures__data-structures-for-coding-interviews-in-python)