Algorithm Efficiency

Moses Hansen

12 October 2022

1 Introduction

When designing and implementing algorithms, we are obviously very concerned with how long the algorithm will take to run, and how much data storage will be required. For example, if we have a list of numbers, and we want to sort them, that task is very easy for the list $\{7, 11, 2\}$, but much more difficult for a list with 1000 numbers in it.

There are two types of complexity with which we are concerned:

1. Temporal Complexity: How long an algorithm takes to run
2. Spacial Complexity: How many storage resources are required for the algorithm to operate

And two ways of expressing each:

1. Big-O $O(f)$
2. Little-O $o(f)$

... with f being a function, usually a polynomial. Intuitively, this means that the complexity of an algorithm increases in a way similar to f as the length n of the inputs increases.
For example, if the temporal complexity of an algorithm is $O(n)$, then doubling the size of the inputs will make the algorithm take twice as long. But, if its temporal complexity is $O(n^3)$, then doubling the size of the inputs will make the algorithm take eight times as long. As a side note, this is why efficient algorithm design and programming is so crucial: inefficient design can make an algorithm impractical.

2 Counting Operations

How do we know the temporal complexity of an algorithm? Think of the algorithm like a to-do list, and ask yourself "how many items are on my to-do list?" For example if your list were something like:

1. Put clothes in dryer
2. Run the dishwasher
3. Take out the trash

Then you have three tasks. We can express the complexity of your to-do list as $f(n) = 3$.

On the other hand, if your list were:

1. Put clothes in dryer
2. Wash the dishes by hand
3. Take out the trash

Then the time it will take you to complete your list depends on how many dishes there are. Whereas running the dishwasher is only one action no matter how many dishes are in it, washing them by hand requires every dish to be addressed individually. Hence, the complexity of this list can be expressed as $f(n) = n+2$. If we have no dishes, we have only two tasks to complete. If we have a thousand dishes, then we have 1002 tasks to complete.

It's important to note that we're not super concerned with the actual duration of the tasks, only how that duration is affected by the size of the input to the task. The reason why will become clear in the next section.

3 Comparing Complexities

Now that we have a way of quantifying the complexity of a given algorithm, we're interested in being able to compare algorithms. For instance, if algorithm A has complexity given by $f(n) = n$ and algorithm B has complexity given by $g(n) = \log(n) + 10$, which algorithm is better?
At small input sizes, if we know the inputs, we can compare them directly. For the given example, $f(n) < g(n)$ all the way up to eleven-ish, so if our input size is going to be smaller than eleven, we would prefer the first algorithm, while the opposite is true if the input size exceeds eleven.
Generally, we care more about the performance of an algorithm with large input sizes, and to make those comparisons effectively, we have Big-O and Little-O notation.

4 Big-O

Big-O is an equivalence relation between complexity functions. That is, we can create equivalence classes, and partition every possible complexity function into

exactly one class. For example, $O(n)$ is the class of complexity functions that increase linearly with an increase in input size, $O(\log(n))$ is the class of complexity functions that increase logarithmically with an increase in input size, and $O(1)$ is the class of complexity functions that don't depend on input size at all.

We determine the Big-O of an algorithm with the following theorem:

$f(n) \sim O(g(n))$ if there exists $M, N \in \mathbb{N}$ such that $|f(x)| \leq M|g(x)|$ whenever $n \geq N$

If we have $f(n) = 3n$, we can show that $f(n) \sim O(n)$ by picking $M = 3$, which satisfies the above condition. We can do similarly for $g(n) = 12n$, $h(n) = 5n+3$, and $j(n) = x + \log(x)$, hence the complexity all of these functions increases linearly with an increase in input size.

NOTE: It should be understood that the Big-O of a function is the LEAST function for which the condition holds. We can find M, N such that $|n| \leq M|n!|$ easily, however this is not the tightest fit.

5 Little-O

Little-O is similar to Big-O, but far less common. The Little-O condition is:

$f(n) \sim o(g(n))$ if $\forall \epsilon > 0$, $\exists N > 0$ such that $|f(n)| \leq \epsilon|g(n)|$ whenever $n \geq N$

Note that this is not the same thing as Big-O. For $f(n) = n$, $f(n) \sim O(n)$, but $f(n) \not\sim o(n)$, since for any $\epsilon < 1$, the Little-O condition does not hold.

This leads us to an interesting distinction: Big-O is essentially the greatest lower bound, while Little-O is the least upper bound.

6 Theorems about Big and Little O

Here are some useful theorems for evaluating the Big and Little O of a given algorithm:

1. If $\lim_{x\to\infty} \frac{|f(x)|}{|g(x)|} \leq M$ for some $M > 0$, then $f(x) \sim O(g(x))$
2. $\lim_{x\to\infty} \frac{|f(x)|}{|g(x)|} = 0$ if and only if $f(x) \sim o(g(x))$
3. If $\lim_{x\to\infty} \frac{|f(x)|}{|g(x)|} = \infty$, then $f(x) \not\sim O(g(x))$

These can be much more intuitive and useful than the formal definitions.

7 Leading Order (Asymptotic Equivalence)

Say we have to functions with identical Big-O complexity. How do we find the faster of the two? Leading order complexity allows us to measure that difference.
Say we have an algorithm with complexity $f(n) = 3n + 99$ and another with complexity $g(n) = 5n + 2$. Both are $O(n)$, but we would expect $f(n)$ to be considerably less complex than $g(n)$. Mathematically, we can express this relationship in the following way:

$$f(n) \sim g(n) \text{ if } \lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$$

. Thus, $f(n) \sim 3n$ and $g(n) \sim 5n$. $3n < 5n$ for any positive n, so we conclude that $f(n)$ is the better algorithm.

8 The Master Theorem

Basic, iterative algorithms are easy to analyze since counting the necessary operations is relatively straightforward. However, sometimes it's unclear how many times a given part of the task needs to be performed.

For example, if your task is "remove half of the blue MM's from a bag until only 1 remains", and you go about the task by counting the number of blue MM's, removing half, and repeating, you may not know out of the gate how many times you'll have to repeat that process. This is an example of a recursive algorithm: rather than performing a task a certain number of times, you perform a task over and over until a pre-set condition (base case) is satisfied.

This makes counting the total number of operations performed by the algorithm in the way we've used up until now nearly impossible. Luckily, the Master Theorem provides a convenient alternative. Given a recursive function of the form:

$$T(n) = \begin{cases} aT(\lceil \frac{n}{b} \rceil) + f(n) & n > 1 \\ T_1 & n = 1 \end{cases}$$

We have:

$$T(n) \sim \begin{cases} O(n^d) & b^d > a \\ O(n^d \log(n)) & b^d = a \\ O(n^{\log_b(a)}) & b^d < a \end{cases}$$

Where d is the least real number such that $f(n) \sim O(n^d)$.

Lost yet? Me too.

Taking our example from earlier, let's construct $T(n)$.
Since we are removing half of the blue MM's each time, the next iteration will be $T(\lceil \frac{n}{2} \rceil)$. We scale that by how many such recursive paths are pursued, and add in any other operations necessary. Removing $\frac{n}{2}$ MM's takes $\frac{n}{2}$ operations,

so our $f(n) = \frac{n}{2}$. Therefore, $T(n) = \begin{cases} T(\lceil \frac{n}{2} \rceil) + \frac{n}{2} & n > 1 \\ 1 & n = 1 \end{cases}$

Now, we can find a, b, and d. $a = 1$, $b = 2$ are easily obtained from the equation itself, and $\frac{n}{2} \sim O(n^1)$ is as tight as possible, so $d = 1$.
It then follows that $b^d = 2^1 = 2 > a$, so by the Master Theorem, $T(n) \sim O(n^d) = O(n)$.

If we modify the process slightly, so that we count the blue MM's, separate half of them to a separate pile, and then continue the process until all piles have only one MM remaining, then our $T(n)$ changes. In this case, $T(n) = 2T(n) + \frac{n}{2}$, since we are now recursing on TWO branches. Consequently, $a = 2$, $b = 2$, and $d = 1$, so $b^d = 2 = a$, so $T(n) \sim O(n^d \log(n)) = O(n \log(n))$. Intuitively, you might think that the task is only twice as complex now, but its order of complexity has changed entirely. This new algorithm is much less efficient than the first.

9 Conclusion

Understanding how algorithms behave as input size increases is crucial to designing efficient algorithms. We should always strive to find the simplest process for a task, whether that be code that we're writing, processes for a manufacturing facility, etc.