Complexity Analysis[1]

[1] More than just a Theory ...

Robin Dawes

November 1, 2021

INTRODUCES an essential tool that computer scientists use to compare and classify algorithms.

.

Measuring the Running Time of an Algorithm

IN THE EARLY DAYS OF COMPUTING people discussed the efficiency of an algorithm by reporting the measured execution time of the algorithm. This is unsatisfactory when it comes to comparing different algorithms, due to the many factors that can affect the outcome: the processor, the operating system, the language of implementation, the compiler used, the IDE, the amount of memory available, the active background processes (updates, system maintenance, etc.) and others.

A better approach is to use a measurement that ignores all of those external factors and focuses on the number of operations required to execute the algorithm. However, even that involves some ambiguity: different programmers may implement an algorithm in slightly different ways, resulting in different numbers of operations. How can we tell if we are measuring the very best version of an algorithm?

The preferred method in use today measures the **growth rate** of the execution time, rather than the execution time itself. By **growth rate**, we mean this:

Let A be an algorithm. Let n be the size of the input to A (for example, n might be the length of a list of numbers that the algorithm will act on). Let $T_A(n)$ be a function that counts the number of operations A will execute when the input has size n. The question we want to answer is this. As n increases, how quickly (or slowly) does $T_A(n)$ increase?

For example, if we double n, does $T_A(n)$ also double? Or does it grow by a factor of more than 2, or less than 2?

First, we make a pretty strong assumption: we assume that all basic

operations (arithmetic, boolean operations, comparisons, flow control, assignment, input a value, output a value, function calls, etc.) take equal and constant time. With this assumption, we can justify the "counting operations instead of measuring time" approach.

Some examples in a more-or-less Pythony pseudocode ...

```
Algorithm A1:
        read(n)          # reads an integer from somewhere
        x = 1
        for i in range(n):
                x = (x + 2) * i
        print(x)
```

The first, second and final statements each involve 1 operation and they are each executed once. The statement inside the loop includes 3 operations and is executed n times. The "for i in range(n):" instruction creates several operations (assigning an initial value to i, incrementing i, making sure i doesn't go beyond n-1, etc.) some of which are performed once and some of which are performed n times.

Without doing an exact count, what we can say is that executing this algorithm includes some operations that are done once, and some operations that are done n times. If we let $T_{A1}(n)$ represent the total number of operations when the input is n, then

$$T_{A1}(n) = c_1 * n + c_2$$

for some fixed values of c_1 and c_2

It turns out that this is all we need to answer the question about the growth rate. Consider this table:

n	$T_{A1}(n) = c_1 * n + c_2$
1	$c_1 + c_2$
2	$c_1 * 2 + c_2$
4	$c_1 * 4 + c_2$
8	$c_1 * 8 + c_2$
16	$c_1 * 16 + c_2$
etc.	etc.

What we see in this table is that when n doubles, the part of $T_{A1}(n)$ that actually involves n also doubles. Regardless of what c_2 is, we will eventually reach a value of n where $c_1 * n > c_2$, and from then

on the difference between $c_1 * n$ and c_2 will get greater and greater. Eventually we will reach a point where c_2 is insignificant. At that point we will be able to say that when n doubles, $T_{A1}(n)$ also (almost) doubles.

Note that everything in the previous paragraph is true, no matter what the values of c_1 and c_2 are. So consider this algorithm:

```
Algorithm A2:
        read(n)          # reads an integer from somewhere
        x = 1
        y = 2
        for i in range(n+10):
                x = (y * x + 2)*(x - y)
                y = y + 1
    for i in range(n+50):
        x = x - i
        y = x + y
        print(x)
        print(y)
```

You should be able to convince yourself that even though $A1$ and $A2$ differ in a lot of details, the time function for $A2$ can be written as

$$T_{A2}(n) = c_3 * n + c_4$$

which looks exactly like the definition of $T_{A1}(n)$ (except for c_3 and c_4 instead of c_1 and c_2). This means that a table of n and $T_{A2}(n)$ would show exactly the same behaviour as the table for $T_{A1}(n)$: when n doubles, $T_{A2}(n)$ also (almost) doubles.

Now consider this algorithm:

```
Algorithm A3:
        read(n)          # reads an integer from somewhere
        x = 1
        y = 2
        for i in range(n):
                x = (y * x + 2)*(x - y)
                for j in range(n):
                        y = y + 1
        print(x)
        print(y)
```

It should be clear that the time function for this algorithm will look like this:

$$T_{A3}(n) = c_5 * n^2 + c_6 * n + c_7$$

for some specific c_5, c_6 and c_7

To get a feeling for the growth rate of $T_{A3}(n)$, let's make a table like the one we made before:

n	$T_{A3}(n) = c_5 * n^2 + c_6 * n + c_7$
1	$c_5 + c_6 + c_7$
2	$c_5 * 4 + c_6 * 2 + c_7$
4	$c_5 * 16 + c_6 * 4 + c_7$
8	$c_5 * 64 + c_6 * 8 + c_7$
16	$c_5 * 256 + c_6 * 16 + c_7$
etc.	etc.

Once again we can see that even if c_6 and c_7 are very large numbers, eventually the rapidly growing multiplier for c_5 will make the first term larger than the sum of the other two terms, and its dominance only gets larger and larger as n continues to grow.

Consider what happens to $T_{A3}(n)$ when n goes from $n = k$ to $n = 2k$ (i.e. when n doubles). $T_{A3}(n)$ goes from (about) $c_5 * k^2$ to (about) $c_5 * (2 * k)^2$, which is equal to $c_5 * 4 * k^2$. We can see that the growth rate of $T_{A3}(n)$ is the **square** of the growth rate of n.

We can easily (and correctly) extrapolate from this: If an algorithm A has a time function that looks like this:

$$T_A(n) = c_k * n^k + c_{k-1} * n^{k-1} + \ldots c_1 * n + c_0$$

then for sufficiently large values of n, when n doubles, $T_A(n)$ will increase by a factor of (about) 2^k

We can also observe that there is nothing magical or essential about doubling n in the tables shown above. If we triple n (i.e. if we look at $n = 1, 3, 9, 27...$) then the values of $T_{A1}(n)$ and $T_{A2}(n)$ will also triple (same growth rate as n), and the values of $T_{A3}(n)$ will grow by a factor of 9 (the square of the growth rate of n).

This analysis illustrates why we **never** have to do an exact count of the number of operations in an algorithm. The growth rate of the time function is independent of the exact number of operations - it depends on the highest power of n in the function.

Big-O Notation

WHAT WE NEED now is some simple notation to summarize this. The notation we use is called "Order Notation" or "Big-O Notation". I actually prefer to call this "Big-O Classification" because that is what we will use it for: we are going to group algorithms together into sets based on the growth rate of their time functions.

DEFINITION: Let $f(n)$ and $g(n)$ be functions. Then we say
"$f(n)$ is in $Order(g(n))$" or " "$f(n) \in O(g(n))$"
if there exist values n_0 and c such that $\forall n \geq n_0,\ f(n) \leq c * g(n)$

EXAMPLE 1: Let $f(n) = 7n + 50$ and $g(n) = n$

Let $c = 8$. It is easy to show that $\forall n \geq 51, f(n) \leq 8n$. Thus there exist values of c and n_0 that satisfy the requirement, so $f(n) \in O(n)$

Note that the choice of $c = 8$ is not unique. We could have used $c = 9, c = 7.01, c = 2000 \ldots$ basically any value of $c > 7$ lets us find an appropriate n_0.

EXAMPLE 2: Let $f(n) = c_2n^2 + c_1n + c_0$. I claim that $f(n) \in O(n^2)$

PROOF: Let $c = c_2 + 1$. We need only show that there exists an appropriate value for n_0 such that $\forall n \geq n_0, f(n) \leq (c_2 + 1)n^2$

We can simplify this inequality:

$$\begin{aligned} f(n) &\leq (c_2 + 1)n^2 \\ c_2n^2 + c_1n + c_0 &\leq c_2n^2 + n^2 \\ c_1n + c_0 &\leq n^2 \\ c_0 &\leq n(n - c_1) \end{aligned}$$

We see that the right hand side is postive and increases without limit when $n > c_1$, whereas the left hand side is fixed. Thus there must be a value of n_0 such that $\forall n \geq n_0$, the just-stated inequality is satisfied. This completes the demonstration that $f(n) \in O(n^2)$

These two examples can be generalized into the following very useful theorem, which I present without proof.

The proof is just a more general version of the argument I used in Example 2.

THEOREM:

Let $f(n) = c_kn^k + c_{k-1}n^{k-1} + \cdots + c_1n + c_0$. Then $f(n) \in O(n^k)$

What Do We Do With It?

Remember, our goal is to find a simple way to compare algorithms that will be independent of extraneous details. We're close to achieving that.

Suppose A_1 and A_2 are two algorithms that solve exactly the same problem. Further, suppose that we have determined that the time function for A_1 is $T_{a_1}(n) = c_1 n + c_0$ and the time function for A_2 is $T_{A_2}(n) = c_4 n^2 + c_3 n + c_2$. What can we say about A_1 compared to A_2?

We know $T_{A_1}(n) \in O(n)$ and $T_{A_2}(n) \in O(n^2)$. This suggests that $T_{A_1}(n)$ grows slower than $T_{A_2}(n)$, which implies that once we get past some (possibly large) value of n, $T_{A_1}(n)$ will always be less than $T_{A_2}(n)$, i.e. A_1 will be faster than A_2

This is pretty much exactly the conclusion we want to make, so be sure you understand it. Unfortunately, there is *one more thing* we need to work out before we can say we are done.

Let's continue with A_1 and A_2. We know $\exists n_0$ and $c \ni \forall n \geq n_0, T_{A_1}(n) \leq cn$.

But consider this: $n \leq n^2$ so $cn \leq cn^2$ ... and thus it is completely accurate to say that $T_{A_1}(n) \in O(n^2)$

And we can also say that $T_{A_1}(n) \in O(n^3)$, and $O(n^4)$, $O(n^5)$, etc.

But wait ... didn't we just a minute ago say we prefer A_1 over A_2 because $T_{A_1} n \in O(n)$ and $T_{A_2}(n) \in O(n^2)$? Now we're saying that both T_{A_1} and $T_{A_2}(n)$ are in $O(n^2)$ ... doesn't that mean that we have no reason to prefer one of them over the other? Has it all been a waste of time?

Happily, the answer to those questions is "no": we **do** have good reason to prefer A_1 over A_2, and therefore it has not been a waste of time.

It is true that both of these algorithms are in $O(n^2)$, but ***only one of them is in*** $O(n)$!

More precisely, their *time functions* are in $O(n^2)$

Some people[2] claim that you cannot prove a negative. But you can of course, and we will now prove that $T_{A_2}(n) \notin O(n)$

[2] including Scott Adams, the creator of Dilbert ... who really should know better!

Suppose $T_{A_2}(n) \in O(n)$

That implies $\exists n_0$ and $c \ni \forall n \geq n_0, T_{A_2}(n) \leq cn$

$$\Rightarrow \forall n \geq n_0,\ c_4n^2 + c_3n + c_2 \leq cn$$
$$\Rightarrow \forall n \geq n_0,\ c_4n^2 + (c_3 - c)n + c_2 \leq 0$$
$$\Rightarrow \forall n \geq n_0,\ (c_4n + c_3 - c)n + c_2 \leq 0$$
$$\Rightarrow \forall n \geq n_0,\ (c_4n + c_3 - c)n \leq -c_2$$

But for all values of $n > \left\lfloor \frac{c_3 - c}{c_4} \right\rfloor$, the left side is positive and increasing, while the right side is constant. Thus no matter what c_2 is, eventually the left side will be $> -c_2$ and the statement will be false.

Thus there does not exist any such n_0, and thus $T_{A_2} \notin O(n)$

And now **finally** we can make our assertion with complete confidence: There exists some value x such that $\forall n \geq x, T_{A_1}(n) \leq TA_2(n)$ ... in other words, when n is sufficiently large, A_1 is faster than A_2

A Note on Standard Usage

We adopt a simple convention to avoid having to go through that kind of effort every time we want to compare two algorithms:

> CONVENTION: When we state $f(n) \in O(g(n))$, we are also asserting that $f(n)$ is **not in** any lower complexity class ... i.e. we are asserting that $g(n)$ is the smallest function that gives an upper bound on the growth rate of $f(n)$.

For simple time functions of the form $T_A(n) = c_kn^k + \ldots c_1n + c_0$, we know that we can state $T_A(n) \in O(n^k)$ and it is easy to show that this is the lowest complexity class to which $T_A(n)$ belongs.