Big O Classification

Robin Dawes

February 13, 2021

Offers a brief demonstration of different growth rates of the timing functions for algorithms. We are introduced to "Big O" classification of algorithms.

.

Groundwork

It's worth reviewing some basic details that relate to constructing a timing function for an algorithm. They are very straightforward and should be already familiar.

To determine the timing function for an algorithm we count the fundamental operations as a function of the size of the input. But when we do this, we usually just count the operations that involve the actual data. In other words we ignore things like index variables and execution control operations. As we will see, we don't even need to be completely precise in our counting. Consider this algorithm, which is written in the kind of informal pseudo-code that I will use throughout these notes. Notice that I'm leaving out all declarations.

```
CODE                    OPERATIONS
A1:
n = read()                 2        (1 I/O and 1 assignment)
for i = 1 to n:
    A[i] = read()          2*n      (1 I/O and 1 assignment, repeated n times)
```

We don't count any of the operations relating to the loop management because they don't involve the data. So we would write the timing function for A1 as

$$T_{A1}(n) = 2 + 2n$$

The size of the input here is actually $n + 1$ since that is the total number of read actions we execute. For our purposes here, calling it n is fine.

Now let's look at two more very simple algorithms:

```
CODE                      OPERATIONS
A2:
n = read()                   2       (1 I/O and 1 assignment)
for i = 1 to n:
    A[i] = read()*i          3*n     (1 I/O, 1 multiplication and 1 assignment, repeated n times)
for i = 1 to n:
    for j = 1 to n:
        print A[i] + A[j]    2*n^2   (2 ops, n^2 times)
```

So we would write the timing function for A2 as

$$T_{A2}(n) = 2 + 3n + 2n^2$$

```
CODE                      OPERATIONS
A3: n = read()               2   (1 I/O and 1 assignment)
for i = 1 to n:
    A[i] = read()            2*n (1 I/O and 1 assignment, repeated n times)
    B[i] = 2*A[i]            2*n (1 I/O and 1 assignment, repeated n times)
```

So we would write the timing function for A3 as

$$T_{A3}(n) = 2 + 4n$$

Our goal is to use the timing functions as a way of comparing the efficiency of algorithms. But as we have already seen, they are somewhat approximate because they don't count every single operation. So instead of comparing the explicit timing functions for different algorithms, we use the timing functions to organize algorithms into groups. Then to compare two algorithms, we compare the groups they are assigned to.

We group algorithms together based on the **growth-rate** of their timing functions. To illustrate this we can look at the three algorithms above and see what happens when we repeatedly double the value of n (ie. double the size of the input).

n	$T_{A1}(n)$	$T_{A2}(n)$	$T_{A3}(n)$
1	4	7	6
2	6	16	10
4	10	46	18
8	18	154	34
16	34	562	66
32	66	2146	130
64	130	8386	258
128	258	33154	514
256	514	131842	1026
512	1026	525826	2050
1024	2050	2100226	4098

Now how fast are these timing functions growing? Let's look at the ratios for successive values in the columns. For A1, the ratios are $\frac{3}{2}, \frac{5}{3}, \frac{9}{5}, \frac{17}{9} \ldots$ etc. We can see that these ratios are getting closer and closer to to 2. For A3, the sequence of ratios is almost identical (it's just missing the $\frac{3}{2}$ term) so it has the same behaviour.

Can you see why they will never quite reach 2?

For A2, the sequence of ratios is $\frac{16}{7}, \frac{46}{16}, \frac{154}{46}$ ... it's a bit harder to see the pattern. The ratios work out to (approximately) $2.3, 2.9, 3.3, 3.6, 3.8$ ... and if we went further, we would see that the ratios approach 4 but never quite reach it.

So when n increases by a factor of 2, $T_{A1}(n)$ and $T_{A3}(n)$ also increase by a factor of (slightly less than) 2, but $T_{A2}(n)$ increases by a factor of (slightly less than) 4.

Experiment: What if we try increasing n by a factor of 3? That is, start with $n = 1$, then $n = 3$, then $n = 9, 27, 81$, etc. You can work it out, but I'll jump to the results: $T_{A1}(n)$ and $T_{A3}(n)$ also increase by a factor of (slightly less than) 3, and $T_{A2}(n)$ increases by a factor of (slightly less than) 9.

In general, we find that if the input n increases by a factor of k, $T_{A1}(n)$ and $T_{A3}(n)$ also increase by (slightly less than) a factor of k . We can write this as

$$\frac{T_{A1}(k * n)}{T_{A1}(n)} \leq k \quad \text{and} \quad \frac{T_{A3}(k * n)}{T_{A3}(n)} \leq k$$

Similarly, we find that when n increases by a factor of k, T_{A2} in-

creases by (slightly less than) a factor of k^2. We can write this as

$$\frac{T_{A2}(k*n)}{T_{A1}(n)} \leq k^2$$

We got to those conclusions by observation, but we can reach the same conclusion algebraically. For example, we can write

$$\begin{aligned} T_{A2}(n) &= 2n^2 + 3n + 2 \\ T_{A2}(k*n) &= 2(k*n)^2 + 3(k*n) + 2 \\ &= k^2 * (2n^2) + k * (3n) + 2 \\ &\leq k^2 * (2n^2 + 3n + 2) \end{aligned}$$

and we see that $\frac{T_{A2}(k*n)}{T_{A2}(n)}$ is always $\leq k^2$

Let's focus on $T_{A1}(n)$. We have seen that it grows LINEARLY (ie. at the same rate) as n grows. Can we use that information to give any information about the actual value of $T_{A1}(n)$?

Suppose there is some particular value n_0 for which we can determine that $T_{A1}(n_0) \leq c * n_0$ for some positive constant c.

Now consider $T_{A1}(k * n_0)$ where $k \geq 1$. From our previous discussion we know $\frac{T_{A1}(k*n_0)}{T_{A1}(n_0)} \leq k$ and from there it is a simple step to

$$T_{A1}(k * n_0) \leq c * (k * n_0)$$

Now if we replace "$k * n_0$" by a generic "n", we get

$$T_{A1}(n) \leq c * n \quad \forall n \geq n_0$$

Are there such an n_0 and constant c? Yes! We can see that if we let $n_0 = 1$ and $c = 4$, the requirements are satisfied.

Now what about $T_{A3}(n)$? You can work out that the same property holds (though you cannot use the same value for c).

But what about T_{A2}? Suppose we start by establishing that for some value n_0, $T_{A2}(n_0) \leq c * n_0$ for some constant c. Now we can consider $T_{A2}(k * n_0)$. From our previous analysis we know $\frac{T_{A2}(k*n_0)}{T_{A2}(n_0)} \leq k^2$ which gives

$$T_{A2}(k * n_0) \leq c * k * (k * n_0)$$

Now if we replace "$k * n_0$" by "n" we get

$$T_{A2}(n) \leq c * k * n \quad \forall \ n \geq n_0$$

.... which does not fit the same pattern as we saw for $T_{A1}(n)$ and $T_{A3}(n)$. In fact it is kind of confusing because it still has a k in it ... but remember that we used n to replace $k * n_0$, so $k = \frac{n}{n_0}$ ... and we can use this to replace the k in the right hand side! This gives

$$T_{A2}(n) \leq c * \frac{n}{n_0} * n \quad \forall\, n \geq n_0$$

ie.

$$T_{A2}(n) \leq \frac{c}{n_0} * n^2 \quad \forall\, n \geq n_0$$

Since c and n_0 are both constants, $d = \frac{c}{n_0}$ is a constant. Thus

$$T_{A2}(n) \leq d * n^2 \quad \forall n \geq n_0$$

Was there anything unique about the timing functions that we used? Not at all. To generalize what we have seen, consider this small theorem:

THEOREM: Suppose an algorithm A has timing function

$$T_A(n) = a_t * n^t + a_{t-1} * n^{t-1} + \cdots + a_1 * n + a_0$$

where the a_i coefficients are constants and $a_t > 0$.

Then $\exists$ a constant c such that

$$\forall\, n \geq 1, \quad T_A(n) \leq c * n^t$$

PROOF:

Let $a^* = max\{a_0, \ldots, a_t\}$

Let $c = (t+1) * a^*$

Let n_0 be any value ≥ 1.

$a^* * (n_0)^t \geq a_i * (n_0)^i \quad \forall\, i \in \{0, \ldots, t\}$

$\Rightarrow c * (n_0)^t \geq a_t * (n_0)^t + a_{t-1} * (n_0)^{t-1} + \cdots + a_1 * n_0 + a_0 = T_A(n_0)$

DJ Moose is used in these notes to mark the end of a proof.

DEFINITION: Let $f(n)$ and $g(n)$ be non-negative valued functions on the set of non-negative numbers. If there are constants n_0 and c such that

$$f(n) \leq c * g(n) \quad \forall\ n \geq n_0$$

then we write

$$f(n) \in O(g(n))$$

and we say $f(n)$ IS IN BIG O OF $g(n)$ or (more formally) $f(n)$ IS IN ORDER $g(n)$.

The significance of this is that as n gets large, the growth-rate of $f(n)$ is no greater than the growth-rate of $g(n)$. In other words, the growth of $g(n)$ is an upper bound on the growth of $f(n)$.

So $O(g(n))$ actually defines a class of functions: all the functions that have that particular relationship to $g(n)$. What we're going to do is choose a bunch of canonical (and simple) functions as $g(n)$, and use

those functions to define useful classes of other functions that we can use to compare the efficiency of different algorithms.

The canonical $g(n)$ functions are mostly listed here (omitting a few that don't really concern us at this point). I'm also showing the names people typically use when these functions are used to classify the running time of algorithms.

$g(n)$	Common Name for $O(g(n))$
c	Constant Time or $O(1)$ Time
$\log n$	Logarithmic Time
n	Linear Time
$n \log n$	n log n Time
n^2	Quadratic Time
n^3	Cubic Time
n^k	Polynomial Time
$n!$	Factorial Time
$c^n \ \ (c > 1)$	Exponential Time

Putting all of this together, we find that $T_{A1}(n) \ \in \ O(n)$ and $T_{A3}(n) \in O(n)$ and $T_{A2}(n) \in O(n^2)$

That looks like a pretty clear distinction between $T_{A2}(n)$ and the other two ... but is it? Can we be sure that $T_{A2}(n)$ is not also in $O(n)$? Let's check that out.

Suppose $T_{A2}(n) \in O(n)$. Then there exist constants n_0 and c such that

$$T_{A2}(n) \leq c * n \quad \forall\, n \geq n_0$$

ie.

$$\begin{aligned} 2n^2 + 3 * n + 2 &\leq c * n \quad \forall n \geq n_0 \\ 2n^2 + 3 * n - c * n &\leq -2 \quad \forall n \geq n_0 \\ (2n + 3 - c)n &\leq -2 \quad \forall n \geq n_0 \end{aligned}$$

But $\forall\ n\ > \ \dfrac{c-3}{2}$, the left hand side is positive ...so the inequality *does not* hold for all $n \geq n_0$.

It makes no difference which of c and n_0 is larger.

Therefore $T_{A2}(n) \notin O(n)$

And now, finally, we are sure that $T_{A2}(n)$ does not belong to the same class of function as $T_{A1}(n)$ and $T_{A3}(n)$.

Nesting of Order Classes

THE RELATIONSHIP between different order classes is a bit more complex than it might seem. In the previous section we established that $T_{A1}(n) \in O(n)$. But observe that if $T_{A1}(n) \leq c * n$, then $T_{A1}(n) \leq c * n^2$ as well, so $T_{A1}(n) \in O(n^2)$ as well ...and the same can be said of any function in $O(n)$!

Assuming $n \geq 1$, which it always is when used as a measure of the size of the input to an algorithm that takes input.

Thus we can see that the class of functions described by $O(n)$ is a subset of the class described by $O(n^2)$, which is a subset of the $O(n^3)$ class, etc. In fact we can establish this:

$$O(c) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \ldots \text{ etc.}$$

In practice we **always** try to identify the **lowest** order class that a function belongs to. So when we say something like $T_A(n) \in O(n^2)$, we are also saying that $T_A(n)$ is **not** in any of the order classes that are subsets of $O(n^2)$.

A final observation: for simple algorithms like A1, A2 and A3, the process of classification is extremely easy: we just look for the step that is executed the most often and determine the power of n that relates to that step. Thus for A1 and A3, the most frequent step is executed $c * n$ times for some c (we don't care what c is) so these timing functions are in $O(n)$. For A2, the most frequent step is executed $c * n^2$ times for some c ...so $T_{A2} \in O(n^2)$.