Algorithm analysis and Big O

Benjamin Cosman, Patrick Lin and Mahesh Viswanathan

Fall 2020

TAKE-AWAYS

- $f(n)$ is $O(g(n))$ if there are positive c, k such that for every $n \geq k$, $f(n) \leq c \cdot g(n)$.
- Informally, $f(n)$ is $O(g(n))$ if, as long as you ignore small inputs and constant factors, $f(n) \leq g(n)$.
- The following simple functions form a strictly increasing hierarchy for big-O:
  $1, \log(n), n, n\log(n), n^2, n^3, ..., 2^n, 3^n, ..., n!$.
- We say $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $g(n)$ is $O(f(n))$
- Dominant Term Method: If $f(n) = g_1(n) + g_2(n) + ... + g_k(n)$ and $g_1(n)$ is the dominant term (grows the fastest for large n; all other $g_i(n)$ are $O(g_1(n))$), then $f(n)$ is $\Theta(g_1(n))$
- The run time of an algorithm is expressed as a function $T(n)$ whose value is the worst-case run time for inputs of size n.
- Usually, an algorithm with (nested) loops will have a run time that is most easily expressed as a summation and a recursive algorithm will have a run time most easily expressed as a recurrence. Both can be simplified to closed forms (e.g. with unrolling or recursion trees), which are then frequently simplified further using big-O notation.

Big O

If on inputs of size n, one program takes $4n^2 + n + 5$ minutes to run and the other takes $3n^2 + 2$ minutes, then you can use the second one and save some time. But in this class, we'd like to look at a bigger picture, and note (for example) that those two functions are almost the same when compared with a program that takes n^3 minutes: n^3 grows faster than any quadratic, and so for large n it will be *much* slower than either of the quadratic options. We thus define the following notation which categorizes functions by their behavior on

large values:

Definition 1 (Big-O)**.** For functions $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$ [1], we say that $f(n)$ is $O(g(n))$ (pronounced "$f(n)$ is big-oh of $g(n)$") if the following is true:

$$\exists c \in \mathbb{R}^+ (\exists k \in \mathbb{R}^+ (\forall n \geq k (f(n) \leq c \cdot g(n))))$$

Informally, $f(n)$ is $O(g(n))$ if $g(n)$ is always at least as big as $f(n)$, with two important caveats:

- We can multiply $g(n)$ by any constant (c) first.
- We can ignore any constant number (k) of small values of n.

Example 2. n^2 is trivially $O(10n^2)$ since n^2 is never bigger than $10n^2$ (for any $n \in \mathbb{N}$). Using the definition, we can choose $c = k = 1$, and confirm that $\forall n \geq 1 (n^2 \leq 1 \cdot 10n^2)$.

Example 3. $10n^2$ is $O(n^2)$. Notice that $10n^2$ is bigger than n^2 for any positive number, but it's bigger only by a constant factor. In particular, using the definition, we can choose $c = 50$ and $k = 1$, and confirm that $\forall n \geq 1 (10n^2 \leq 50 \cdot n^2)$ is true. (We could also have picked any larger k, and any $c \geq 10$.)

It may seem strange that we ignore constant factors like this. Intuitively, $f(n)$ is $O(g(n))$ does not mean that f is actually smaller than g, but rather that f *scales up* in a way that is no worse than g. So in Example 3, note that n^2 and $10n^2$ scale up in the same way: if you make the input twice as big, the output gets four times bigger.

Example 4. $2n$ is $O(g(n))$, where $g(n)$ is defined piecewise by

$$g(n) = \begin{cases} 0 & \text{if } n = 100 \\ \frac{n}{3} & \text{otherwise} \end{cases}$$

Notice that for $n = 100$, $g(n)$ is not only less than $2n$, but since it's equal to 0, there is no multiple of $g(n)$ that is at least $2n = 200$. However, the definition allows us to skip any fixed number of values for n though our choice of k: in this case, if we choose $k = 101$ and $c = 6$, then we can confirm that $\forall n \geq 101 (2n \leq 6 \cdot \frac{n}{3})$. (As always, larger values of k and c would also have worked.)

Example 5. n^3 is NOT $O(10n^2)$. We can prove this by arguing that $\exists c \exists k (\forall n \geq k (n^3 \leq c \cdot 10n^2)$ is false - that is, that its negation $\forall c \forall k (\exists n \geq k (n^3 > c \cdot 10n^2)$ is true. Fix k and c. [2] Then let $n = max(k, 11c)$. This choice of n definitely satisfies $n \geq k$. Additionally, we have $n \geq 11c > 10c > 0$, so $n^2(n - 10c) > 0$, which implies $n^3 > c \cdot 10n^2$.

[1] $\mathbb{R}^{\geq 0}$ is the non-negative reals; $\mathbb{R}^+$ is the positive reals

[2] Notice that in the previous examples we chose a specific k and c because we were proving an existential statement, but the negation we're showing now is a universal statement so we need to leave them arbitrary. Meanwhile we now have an $\exists n$, so we do get to choose n.

In the same way that Big-O acts a little bit like a $\leq$ between functions, we can also define notation corresponding in the same way to other operators, the most important of which is equality:

Definition 6 (Big-Theta)**.** We say that $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $g(n)$ is $O(f(n))$. [3]

[3] You will not need the following for this class, but there is also standard notation corresponding similarly to $<$, $\geq$, and $>$, namely o, Ω, and ω.

Example 7. Using Example 2 and Example 3, we see that $10n^2$ is $\Theta(n^2)$ (and also n^2 is $\Theta(10n^2)$).

The dominant term method

Using similar techniques to the examples above, we can show that in the following list of functions, each function $f(n)$ is $O(g(n))$ for every g to its right, and NOT $O(g(n))$ for every g to its left: [4]

$1, \log(n), n, n\log(n), n^2, 2^n, n!$

Additionally, if $0 < a < b$ then the same holds for the lists n^a, n^b and a^n, b^n.

[4] In other words, each $f(n)$ is $o(g(n))$ for each g to its right - see previous footnote

For large enough n, any function later in one of these lists will grow arbitrarily larger and faster than any function earlier in the list. We call the fastest growing term in a sum the *dominant* term, and when working with big-O it turns out that we can ignore all other terms.

Example 8. The function $100n\log(n) + 3n^8 + 42n^6$ behaves, for large enough n, like its dominant term $3n^8$. We can thus immediately decide that $100n\log(n) + 3n^8 + 42n^6$ is $\Theta(n^8)$.

Pitfalls with Big-O

If you look at other references, be aware that there are several minor variations in the way big-O is defined and used. Some texts write $f(n) = O(g(n))$ instead of our "$f(n)$ is $O(g(n))$". Some texts define big-O to work on real- or even negative-valued functions (by replacing e.g. $f(n)$ with $|f(n)|$ in the definition).

You can find a selection of common big-O pitfalls in our MCS textbook, section 14.7.4 (p634).

Algorithm Analysis

Consider the following algorithm, presented in pseudocode:

Insertion Sort

```
1. insertionSort(A): // A: (1-indexed) array of size n
2.   for each i from 2 to n:
3.     j = i
4.     while j > 1 and A[j] < A[j-1]:
5.       swap the values in A[j] and A[j-1]
6.       j = j-1
```

This algorithm sorts its input array A in ascending order.[5] This week, we're interested in the deceptively simple question: how fast is the algorithm?

[5] We won't be concerned in this class with proving that the algorithm successfully sorts the array.

We certainly can't give an answer like "5 seconds", for many reasons. First, we don't actually want to give an answer in units of "seconds", since that would depend on various details we don't have and don't really care about at this level of abstraction, like how fast the processor runs and what language the algorithm is coded in. Thus we will usually come up with some other unit - either something problem-specific (like "swaps" for insertion sort), or something generic like "steps" or "operations". Next, we can't give a constant answer like "5" since the answer depends in part on the size of the input, so instead we will give our answer as a function of the input size n - in our example, "$n(n-1)/2$ swaps". Finally, even for inputs of the same size, two different inputs might take different amounts of time - in our example, an array that is already sorted will take less time to sort because the while loop will end early whenever it finds $A[j] >= A[j-1]$. We will consider only the *worst-case* run time, so in our example, we assume that the $A[j] < A[j-1]$ always comes up true and we have to keep swapping until $j = 1$. [6] Finally, we will often give our answer entirely in big-O notation - in our example, we say insertion sort takes "$O(n^2)$ time".

[6] Outside the scope of this class, there are other analyses like *average*-case run time that may also be useful.

Computing the run time of an algorithm with loops usually involves creating a summation, computing the closed form of the summation, and then using big-O notation to simplify the answer.

Example 9. Let's count the (worst-case) number of swaps done by insertion sort on an input of length n. Since we assume in the worst case that $A[j] < A[j-1]$ will always be true, for each fixed value of i the while loop on lines 4-6 will have to perform one swap for each j from i down to 2, i.e. $i-1$ swaps. i ranges from 2 to n, so the total number of swaps is $T(n) = \sum_{i=2}^{n}(i-1)$. One way to compute a closed form for that summation is to rewrite it in terms of a new variable $k = i-1$, so we get $\sum_{k=1}^{n-1} k$, which we have seen comes out to $\frac{(n-1)n}{2}$. Finally, if we want an answer in big-O notation, we can write this as $\frac{n^2}{2} - \frac{n}{2}$, and we see that the dominant term is $\frac{n^2}{2}$, so we can say

the run time is $O(n^2)$.

Recursive algorithms can be analyzed in a similar way, except that instead of creating a summation, we will create a recurrence.

Example 10. Consider the following pseudocode: [7]

Merge Sort

```
1. mergeSort(A): // A: array of size n
2.   if A has size 1:
3.     return A
4.   otherwise:
5.     split A in half into L and R
6.     L = mergeSort(L)
7.     R = mergeSort(R)
8.     Res = []   // an empty array
9.     while L or R is non-empty:
10.      move the smaller of L[1] and R[1] to end of Res
11.    return Res
```

[7] Again we will only be concerned with analyzing how long this code takes to run; proving that it successfully sorts the input array is beyond the scope of this class but you can find animations/proofs online if interested.

For simplicity we will assume that n is a power of 2, and we will compute the (worst-case) run time $T(n)$ for mergeSort. When n is 1, we do some (small) constant c amount of work to just check that the size is 1 and then return the given array, so $T(1) = c$. For larger n, we make two recursive calls to mergeSort, each with an array of size $\frac{n}{2}$. We also do an additional amount of work that is proportional to n: naively splitting A on line 5 may require copying n values into new arrays, and the while loop on lines 9-10 does a constant amount of work for each of the n values contained between L and R. This gives us the following recurrence:

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(\frac{n}{2}) + dn & \text{otherwise} \end{cases}$$

This has the closed form $dn\log_2(n) + cn$, [8] so the dominant term is $dn\log_2(n)$ and thus the run time is $O(n\log_2(n))$. (And since $\log_2(n)$ differs from $\log_{10}(n)$ - or indeed any other log with fixed base - by a constant, it is also accurate and more common to write that the run time is just $O(n\log(n))$.)

[8] You can compute this with e.g. the recursion tree method from last week