Big O

Edited from: sarahlim / big-o.md

Measuring performance

To measure an algorithm's efficiency, one intuitive idea is to consider the amount of time it takes to solve a problem (transform the input into the desired output).

Unfortunately, it's difficult to measure the *exact* runtime of an algorithm. In practice, runtime are influenced by a lot of factors unrelated to the algorithm itself. For instance, the same function might run quickly on a state-of-the-art gaming PC, and much more slowly on an ancient Windows 98 machine.

So instead of obsessing over nanosecond differences in runtime, many computer scientists like to use **asymptotic analysis** to classify algorithms according to their approximate efficiency.

Asymptotic analysis investigates *how an algorithm's performance degrades as the input size grows*. Generally speaking, we are interested in two performance factors:

- **Time complexity:** as the input grows infinitely large, how much *longer* does it take to solve the problem?
- **Space complexity:** as the input grows infinitely large, how much *more memory* does it take to solve the problem?

Let's consider the list searching example, where we have a list of numbers and we want to return a boolean indicating whether or not the number is in the list. We could write the following pseudocode algorithm:

```
function search(target, list):
  for each item in list:
    if item == target then return true
  return false
```

This algorithm starts at the beginning of the list and visits each item until we've either found our target, or gone through the entire list. In the *best case* scenario, our target is the first item of the list -- then the loop only runs once, and we're done!

In general, though, the best case scenario is not very interesting. When we compare algorithms, we're interested in how they perform when the rubber meets the road -- that is, complexity in the *worst case*

scenario.

For our example, the worst case scenario is when our list *doesn't contain our target element at all*. When this happens, our algorithm can't just find a match and terminate early. It has to check the entire list before calling it quits.

> **Note:** When we say "best case" and "worst case," we mean "the best case *in terms of our algorithm's performance*," not the best case in terms of external meaning we've ascribed to the results.
>
> In the previous example, our worst case isn't the worst case because we didn't find what we were looking for -- it's the worst case because *the algorithm has to go through the entire list instead of being able to terminate early*. Likewise, our best case isn't the best because we found our target; it's the best because we only had to run the loop once.
>
> An algorithm's *performance* (e.g. how long it takes to finish, or how much memory it uses) is totally separate from any subjective valuation we might attach to its *output* (e.g. "found = good and not found = bad!!!!!1"). For example, suppose we are searching a DNA sequence to determine whether a mutation exists. In the Real World, the ideal output is a result of "not found," but in Algorithm World, this is the worst case performance scenario. Conversely, finding the mutation at the very beginning of the patient's DNA sequence is the "best case" from a performance standpoint, even though it would be a bad outcome in real life.
>
> In short, "best" and "worst case" describe an algorithm's *inputs*, not its *outputs*.

Now we've established that in the worst case, our algorithm has to go through the entire input list. Intuitively, it takes longer to go through a long list than a short one. But roughly how much longer? This is the **time complexity** of our algorithm, and we use something called **asymptotic notation** to describe this value.

Big O analysis

Recall the worst case input requires our algorithm to visit each list item exactly once:

```
function search(list, target):
  for each item in list:
    if item == target then return true
  return false
```

Suppose our input list contains n items. Since our algorithm spends the same amount of time on each item, the total worst-case runtime is *directly proportional to the length of the input list*. In other words, if

we express runtime T as a function of input size n, then $T(n)$ is *linear*. **Equivalently, we say that the time complexity of our algorithm is $O(n)$**, pronounced "oh-N" or "oh of N."

This is an example of **big O notation**, which is the most common way computer scientists talk about algorithmic efficiency. Big O notation expresses an *upper bound* on a function. In particular, it expresses an upper bound in terms of *another function* that will always be *greater than or equal to* the first function, for any given value in both functions' domains.

When we analyze algorithms, we are interested in $T(n)$, or runtime as a function of input size. Specifically, we care about how quickly $T(n)$ grows. Saying T is $O(n)$ means that T grows *no faster than* the linear function n. This is the same as saying the linear function n is an *upper bound* on T.

> **Note:** Big O expresses relationships between different functions of some variable, as that variable grows infinitely large. In other words, it compares the *asymptotic behavior* of functions.
>
> There are a lot of ways to talk about big O mathematically. For two functions $f(n)$ and $g(n)$, the following expressions are more or less equivalent:
>
> - $f(n)$ is $O\big(g(n)\big)$, or $f(n) = O\big(g(n)\big)$, or f is $O(g)$
> - f grows (slower than/no faster than) g
> - f is upper bounded by g, or g is an upper bound on f
> - $f(n) \leq kg(n)$, for some constant factor k
> - as n approaches infinity, the ratio of f to g is finite
> - $\lim_{x\to\infty} \left|\frac{f(x)}{g(x)}\right| < \infty$

It turns out that $O(n)$ complexity is really good, relatively speaking (remember that big O is all about classifying functions relative to other functions). If you think about it, it's pretty reasonable to expect a function's runtime to increase proportionally to the input size. In fact, to do any better, our runtime would need to *not increase at all* -- that is, we would need $T(n)$ to remain *constant* as n grows infinitely large.

Such constant $T(n)$s do exist, but are typically very simple. Consider the following function, which just adds 3 to a number:

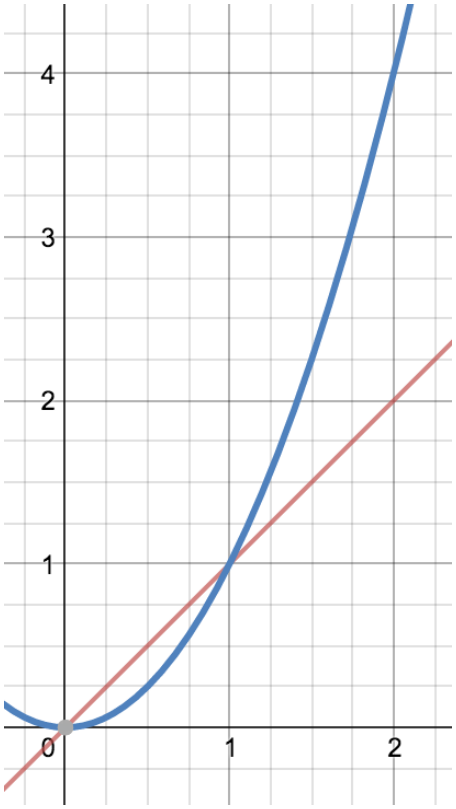
```
function add3(n):
  return n + 3
```

No matter how large n gets, `add3` takes the same amount of time to complete -- in other words, its runtime $T(n)$ is constant. **We say that `add3` is constant time, or $O(1)$**, pronounced "oh-one" or "oh of one."

Now we've seen two common **complexity classes**: $O(1)$ and $O(n)$. Let's meet some others:

Notation	Complexity class	Judgment
$O(1)$	Constant	Awesome
$O(\log n)$	Logarithmic	Good
$O(n)$	Linear	Decent
$O(n^2)$	Quadratic	Bad
$O(2^n)$	Exponential	Awful

Big O is all about classifying functions into groups like these. To understand big O in a nutshell, let's momentarily throw back to plotting graphs of linear and quadratic equations in middle school. Here is a graph of the functions $f(x) = x$ and $g(x) = x^2$.

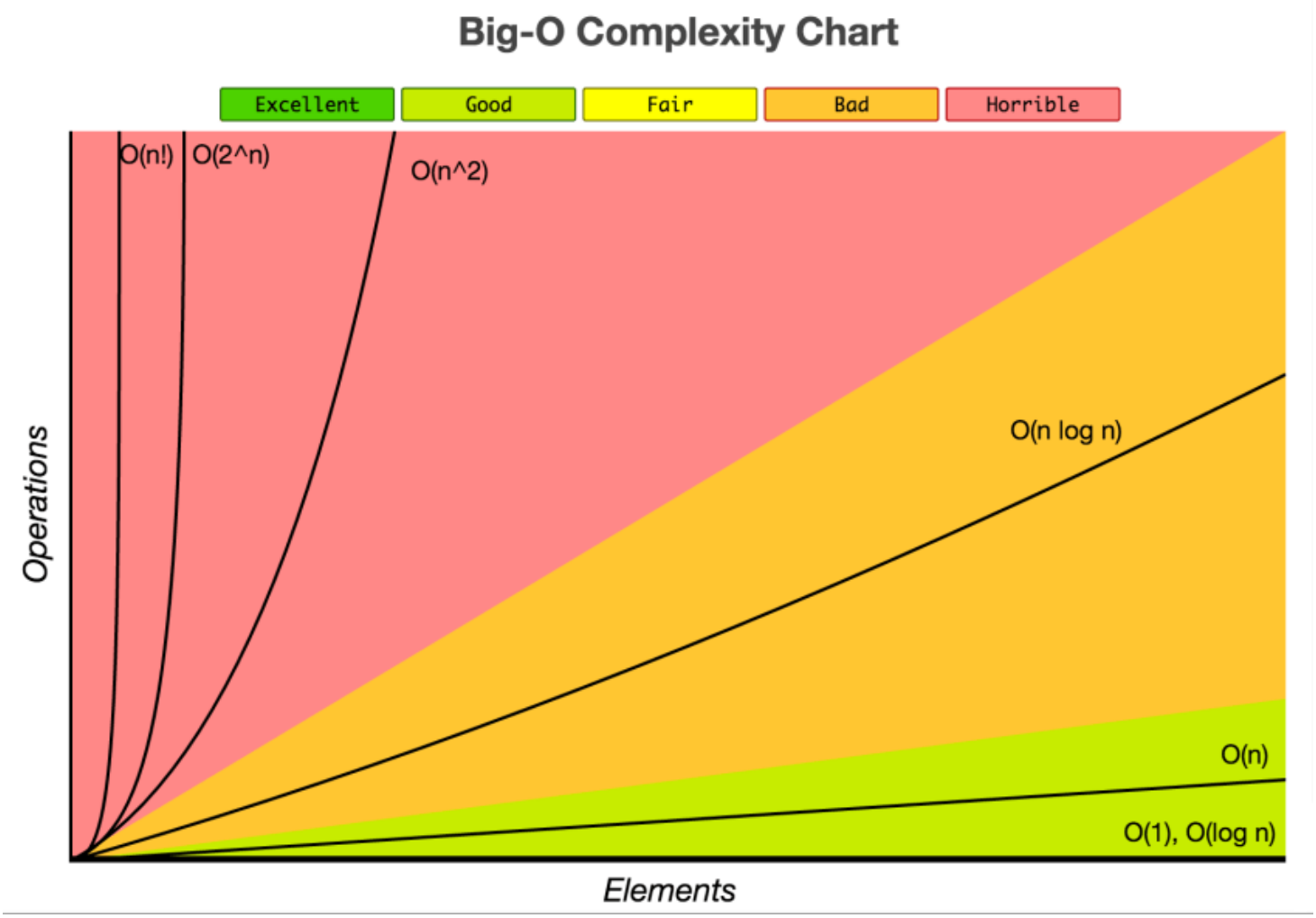


This graph makes it easy to see that for all values of $x > 1$, the quadratic function x^2 dominates the linear function x. As x grows infinitely large, not only will x^2 keep growing, but the *gap* between x^2 and x will keep growing as well.

Compare this behavior to the difference between linear functions x and $3x$. Although $3x$ always dominates x, the gap between the functions remains the same: We can intuitively say that linear and quadratic functions are in different "growth classes," whereas, say, the functions x and $3x$ are in the same linear growth class, even though $3x$ will always dominate x.

Here's the idea in a nutshell: "Let's divide all the functions, ever, into classes that describe approximately how fast a function grows, in terms of the closest general upper bound.

We can get a pretty good idea of how big O complexity works by comparing these growth rates for very large values of n:



Namely, notice how each function grows successively faster than its predecessor. Remember that we are using big O to measure the growth of our algorithm's running time $T(n)$. So,

- the faster $T(n)$ grows,
- the longer our algorithm takes, and
- the worse it performs.

Complexity classes

Look at the above graph again. We've already observed that each function grows successively faster than its predecessor -- the linear function n grows strictly faster than the constant function 1, the quadratic function n^2 grows strictly faster than the linear function n, and so on.

We can actually make an even stronger statement. Each successive function grows faster than not only its predecessor, but *all preceding functions*. That is,

$$1 \leq \log n \leq n \leq n^2 \leq 2^n$$

as n grows infinitely large.

Recall that big O notation expresses some function as the *upper bound* of another. (For instance, saying $T(n) = O(n^2)$ means that $T(n)$ is upper-bounded by the function n^2.)

If we say that a function f is $O(g)$, we are basically saying that f has an upper bound g, or

$$f \leq g.$$

Now, suppose this bound g is itself $O(h)$, meaning g has upper bound h:

$$g \leq h.$$

Chaining the two statements together, we can write h as the upper bound of f:

$$f \leq g \leq h$$

Now drop the g to relate f and h:

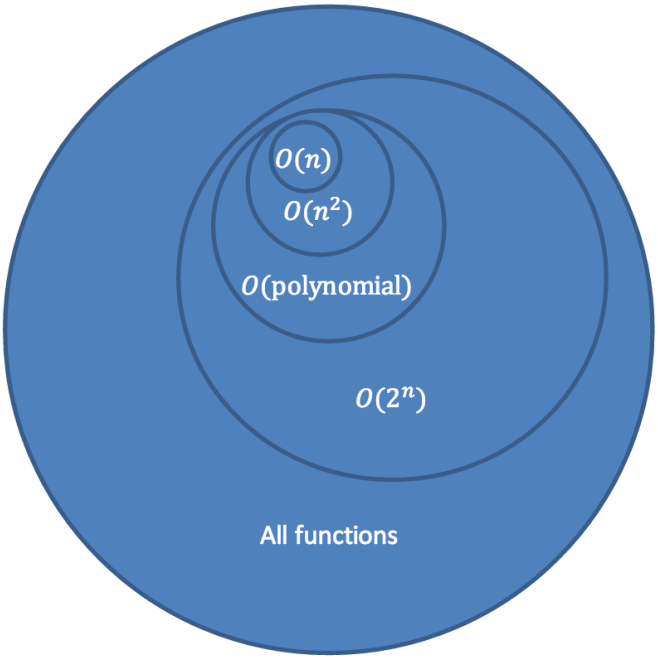
$$f \leq h$$

Translating this into big O notation, f is $O(h)$, in addition to being $O(g)$.

This seemingly trivial example illustrates an important property: a function f is $O(\text{all functions larger than f})$.

For instance, if $T(n)$ is $O(n)$, it's technically *also* $O(n^2)$, $O(n^3)$, $O(2^n)$, etc. But since we're trying to describe $T(n)$ with an upper bound, we want to provide the lowest bound we can, in order to best approximate $T(n)$. So in practice, we just say $T(n)$ is $O(n)$.

The following image neatly summarizes the relationship between complexity classes of functions.



Other asymptotic relationships

Although **big O** is the most common notation used for comparing functions, there are two others worth knowing.

The "opposite" of big O is **big** Ω (pronounced "big omega"), which defines a function as a lower bound on another function. The following statements are logically equivalent:

- f is $\Omega(g)$, pronounced "f is omega of g"
- f grows at least as quickly as g
- g is a lower bound on f
- $f(n) \geq kg(n)$, for some constant k

For example, we know that $\log n$ is $O(n)$ -- that is, n upper bounds the function $\log n$. Conversely, n is $\Omega(\log n)$ -- that is, the function $\log n$ lower bounds n.

Finally, **big** Θ (pronounced "big theta") defines a function as "tight bound" to another function. Two functions are "tight bound" (and thus Θ of one another) if they grow the same way/at the same rate. By definition, f is $\Theta(g)$ when we can "sandwich" f between bounds $k_1 g$ and $k_2 g$, for some constants k_1 and k_2.

For example, saying "$T(n)$ is $\Theta(n^2)$" means there exist constants k_1 and k_2 such that

$$k_1 n^2 \leq T(n) \leq k_2 n^2$$

which means that as n grows infinitely large, the function $T(n)$ will always stay between n^2 scaled by some constant k_1, and n^2 scaled by some constant k_2.

Usually these constants are different, and the two versions of n^2 "sandwich" the graph of $T(n)$. If the constants are the same, then $T(n)$ is exactly kn^2. Finally, if they are the same value *and* that value happens to be 1, then $T(n)$ is exactly n^2.

> **Note:** When we talk about algorithm performance, it's easy to confuse "bound terminology" (upper bound O, lower bound Ω, and tight bound Θ) with "case terminology" (best case, worst case, and average case).
>
> Recall that we use "(best/worst/average) case" to describe the features of an algorithm's input(s), and the corresponding impact upon the algorithm's performance. Each "case" produces a different $T(n)$ for some algorithm, depending on how favorable the inputs are.
>
> By contrast, we use asymptotic notation and "(upper/lower/tight) bounds" to describe functions in terms of other functions. Here, we already know the algorithm's performance $T(n)$, but we don't care how we got this function; we just want to classify its growth rate in relation to other functions.
>
> How do these ideas relate? If we let $T(n)$ denote the time complexity/performance of an algorithm, then the "best case" input *minimizes* $T(n)$ -- which minimizes the upper, lower, *and* tight bounds on $T(n)$. Conversely, the "worst case" input *maximizes* $T(n)$, which maximizes all three bounds.
>
> As a slightly more concrete example, consider our original searching algorithm:
>
> - **Best case:** our target is the *first item in the list*. Then no matter how large the list, the algorithm has the same runtime, because we only need to check the first item. So $T(n) = O(1)$.
>
> > But we could *also* write $T(n) = \Theta(1)$ or $T(n) = \Omega(1)$, because asymptotic notation just describes the function $T(n)$ in terms of other functions.
>
> - **Worst case:** our target *isn't in the list*. Then the algorithm's runtime grows proportionally to the size of the list. So $T(n) = O(n)$.
>
> > But we could *also* write $T(n) = \Omega(1)$, because T grows faster than any constant-valued function -- in other words, the constant function is a *lower bound* on $T(n)$.