Complexity

Usually there are many different ways to compute the same result — different algorithms with identical input-output behavior. Often, though, the different algorithms can behave very differently in other ways: memory use, runtime, etc. Complexity is the study of ways to measure an algorithm's usage of resources like these.

As an example, we saw two different ways to compute the Fibonacci function, one of which was clearly much faster than the other. We'll see below how to make this comparison precise.

The first step in a complexity measurement is to decide what resource we're measuring and how to measure it. A few examples are

- Wall-clock time. This resource can be hard to measure, and can differ greatly if we run the same algorithm on different computers. So we don't often use wall-clock time directly.
- Instruction count. This can also differ substantially: we can implement essentially the same algorithm in slightly different ways and come up with different answers. And, if we compile the same high-level code for different architectures, we also will get different instruction counts.
- Operation count. Instead of counting low-level instructions, we can count higher-level groups of instructions. For one example, we can count floating-point operations, or FLOPs: the number of times we add, subtract, multiply, or divide two real numbers. If we count only FLOPs, we get to ignore the details of branching, subtroutine calling, etc. For another example, we can pick a standard set of functions like $\cos$ and $\exp$, and count how many times we call them. The operation count approach has the advantage that we can hide a lot of differences between different computers and different architectures. But it has the disadvantage that we can miss important information in the abstraction: e.g., if we count calls to trig functions like $\sin$ and $\cos$, but our algorithm spends most of its time copying memory or evaluating conditionals for branches, we'll get an inaccurate idea of what's going on.
- Memory accesses. We can count how many bytes of memory we load and store. This approach again has the advantage of being relatively consistent

across computers (or at least computers with similar overall architectures and word lengths). But, it can measure something very different from operation count: there are algorithms that use a relatively smaller amount of memory but perform many operations per word, as well as algorithms that use a relatively larger amount of memory with fewer operations per word.

- Parallel work or span. If we are executing our algorithm on multiple cores or multiple networked computers, we can still measure the total operation count — this is often called the parallel *work*. But we can also measure the longest string of operations that we have to do in sequence, with each operation depending on the results of the previous one — this is called the *span* or the *depth*, and it limits how much benefit we can get by throwing more processors at the problem.
- Parallel communication. Just like the sequential case, we can measure how many bytes we send to and from memory; but we can also measure how many bytes we send between different processors.
- And more: we can measure other resources like information sent across a network, information stored to persistent storage, or the number of times we need to consult a human for input. There's not really a limit to what we can talk about: if someone is concerned about a resource, we can probably figure out a way to measure it.

The second step in measuring complexity is to relate the resource usage to how hard the problem is that we're trying to solve. After all, it doesn't seem fair to compare two algorithms' resource usages unless we standardize problem difficulty. Just like above, there are a lot of different ways we can measure difficulty. Here are a few common and useful ones:

- Problem dimension. In a lot of problems we're working in a vector space of a given dimension, and the problem gets harder as we move to higher dimensions. For example, if we're multiplying two matrices, we expect it to take longer in $\mathbb{R}^{1000}$ than we do in $\mathbb{R}^{10}$.
- Number of inputs. Instead of looking at the dimension of the problem, we can look at the amount of data required to specify a problem instance. This captures the intuition that we can have more or less complex inputs even if their maximum dimension is fixed: e.g., in $\mathbb{R}^{100}$, a vector takes 100 numbers to specify, while a matrix takes 10000, and a 3-mode tensor contains a million numbers.

- Input bit length. It might not be fair to compare a matrix made up of small integers to a matrix where every entry is specified to 64 bits of precision. Even worse, a real number can theoretically store an infinite number of bits of information (though in practice the methods we use for working with reals will have a much smaller limit). To compensate, we can count the total number of bits required to specify the problem input. This is probably the most general and widely used difficulty measure, since we can apply it to almost any problem. But it's less intuitive to work with than some other measures.
- Condition number. We saw earlier that gradient-based optimizers take longer to work when the objective function has widely varying curvature in different directions. We introduced the condition number to measure this property. This is a problem-specific difficulty measure: the condition number isn't relevant unless we have a function whose condition we can measure. But as it turns out, there are a lot of problems where conditioning is relevant, and a lot of algorithms whose complexity depends on conditioning.
- Other. As with resource measures, there are effectively infinitely many difficulty measures, tailored to different kinds of problems we might want to solve. Some examples are the diameter of a graph, the mixing time of a Markov chain, and the degree of a polynomial.

Scaling

Often the thing we care about most is not the precise resource usage — as mentioned above, the cost of a resource can depend a lot on the details of the computers we use. (And we can hope that future computers continue to get more powerful, so that the cost of executing an instruction or accessing memory continues to decline.) Instead, we care more about how resource usage scales with difficulty: this is a much stronger determiner of how difficult a problem we can solve.

To see why, imagine two algorithms for solving linear systems. For concreteness, say we're working in n dimensions, and the linear system has m nonzero coefficients. And, let's say that $m = 5n$. With these assumptions, the first algorithm takes n^3 FLOPs. The second one takes $4m \log^3 n = 20n \log^3 n$ FLOPs.

In case you're interested, under some assumptions there do exist linear solvers that take approximately this many operations. The second one is based on a famous result of Spielman and Teng for a class of matrices derived from large graphs.

If we plot the FLOP count vs. problem size, we see some interesting things: if we look at small problem sizes, the first algorithm is much better. But if we zoom out to look at bigger problems, the situation changes: now the first algorithm is getting beaten badly.

To see just how badly, suppose we have a budget of one petaFLOP, i.e., 10^{15} FLOPs. (Right now, depending on the problem, it's order-of-magnitude reasonable to guess a teraFLOP per second on reasonable hardware; if we do this for 1000 seconds we get a petaFLOP.) The first algorithm can then handle n up to 100,000 — not too shabby. But the second algorithm can handle n up to about 1.7 *billion*. This difference is night and day — especially if the n for the problem we care about happens be around a million.

There's another, smaller reason that we care more about scaling than about precise resource usage at a given problem size: the resource usage isn't necessarily a smooth function. Lots of not-so-relevant details can cause jumps or dips. For example, we might get a jump when our problem exceeds the size of some level of our memory hierarchy (leading to a jump in the use of slower memory), or a dip when our problem is some fortunate size like an exact power of 2. While details like these do matter, especially when they happen to the problem instance we care about, they don't have a long-term effect on the most difficult problem we can solve.

O notation

To talk about scaling, we'll use a tool called O notation, or "big-O". The point of O notation is to compare the growth rates of different functions like n^3 or $20n\log^3 n$, while hiding their behavior at small values of n. We'll focus on *nonnegative* functions and on $n \geq 1$.

It's also OK if the function only becomes nonnegative for big enough n; we can modify the definitions below to account for this case.

When we compare two functions using O notation, we find out that one function eventually grows faster, slower, or about the same as the other. There are two big caveats here:

- "About the same" does not mean exactly the same, nor even the same in practice. Instead it means that we need a more precise tool to compare.
- "Eventually" is a dangerous word: we might have to build a universe-sized computer before we see the eventually-faster-growing function win.

With those caveats, for any function $g(n)$, we define the set of functions $O(g(n))$ as

> $O(g(n))$ is the set of functions $f(n)$ such that there exists a scaling factor $c > 0$ and a minimum argument $n_0 > 0$ for which
>
> $$f(n) \leq cg(n) \quad \forall n \geq n_0$$

If $f \in O(g)$, we say "f is O of g". The interpretation is that f grows no faster than g (eventually, and at the level of detail that we're paying attention to).

> *You will also see people write $f = O(g))$ to mean $f \in O(g)$. Just be careful to realize that this is not really an equality: $O(g)$ is not a single function, but a set.*

There are two important numbers in the definition of big-O: the scaling factor c and the minimum argument n_0.

- The scaling factor says what we mean by "grows faster than": f grows faster than g only if f eventually beats *any* constant scaling factor applied to g.
- The minimum argument encodes "eventually". We don't care about behavior before n_0; we just care that, after some point, the inequality holds.

For example, let $f(n) = 3n^2 + 5n + 99$. We can show

$$f(n) \in O(n^2)$$

which we say as "f is O of n^2" or "f grows quadratically".

To see why $f \in O(n^2)$, we can place a bound on each term of f using a multiple of n^2. For the first term, trivially, $3n^2 \geq 3n^2$. For the second term, we have $n^2 \geq 5n$ for any $n \geq 5$. For the third term, we have $n^2 \geq 99$ for any $n \geq 10$.

Adding these bounds together, we have $3n^2 + n^2 + n^2 \geq f(n)$ for any $n \geq 10$. This matches the definition of $O(n^2)$ using $c = 5$ and $n_0 = 10$.

Of course, we can also show

$$f(n) \in O(n^3)$$

and

$$f(n) \in O(527n^2 - 3.14159n + 10^{100})$$

> Exercise: do so.

Given that we know $f \in O(n^2)$, these latter two results are not very helpful. The first one loses information: saying $f \in O(n^3)$ is strictly weaker than $f \in O(n^2)$. The second one is unnecessarily complicated: there's no need for the constant 527 in front of n^2, nor for the lower-degree terms.

Partial order

Big-O is transitive: if $f \in O(g)$ and $g \in O(h)$ then $f \in O(h)$. So, we can think of big-O as a generalization of $\leq$. Just like $\leq$, it's nice to have notation for the opposite direction $\geq$. For big-O, the opposite is Ω:

$$f \in O(g) \quad \leftrightarrow \quad g \in \Omega(f)$$

Unsurprisingly, we also have a notation analogous to $=$: if $f \in O(g)$ and $g \in O(f)$, we write

$$f \in \theta(g) \qquad g \in \theta(f)$$

Unlike $\leq$, with big-O, it's possible for two functions to be incomparable. An example is

$$f(n) = \sin^2 \frac{n}{100} \qquad g(n) = \cos^2 \frac{n}{100}$$

The ratio between $f(n)$ and $g(n)$ can be arbitrarily large or arbitrarily small, so there's no constant factor we can find to bound f by a multiple of g or g by a multiple of f.

Properties of O

We can add big-O expressions: if $f(n) \in O(g(n))$ and $s(n) \in O(t(n))$, then $f(n) + s(n) \in O(g(n) + t(n))$.

We can multiply big-O expressions: if $f(n) \in O(g(n))$ and $s(n) \in O(t(n))$, then $f(n)s(n) \in O(g(n)t(n))$.

We can do both the above with plain functions as well: if $f(n) \in O(g(n))$, then for any function $h(n) \geq 0$, we have $f(n) + h(n) \in O(g(n) + h(n))$ and $f(n)h(n) \in O(g(n)h(n))$.

For any integers $p, q \geq 0$, if $p \leq q$, then $n^p \in O(n^q)$.

All of the above properties follow straightforwardly from the definition of big-O.

Exercise: prove them.

Combining the above properties, we can analyze any polynomial: for example,

$$5n^4 + 3n^3 - n + 100 \in O(n^4)$$

The general procedure is that we first drop all terms except the one with highest degree. (We can justify dropping them with the same proof as above: we bound these lower-degree terms by a multiple of the highest-degree one.) The highest-degree term must have a positive coefficient, since we assumed the function as a whole was nonnegative for large n; we drop this coefficient (set it to 1) since we can absorb any scalar multiple into c.

The same idea works for any sum of terms, not just polynomials. That is, we only need to keep the largest term, and we can simplify by dropping any positive coefficient on this term.

Log and exp

After polynomials, the most common functions used with big-O are logarithms and exponentials. Logarithms tend to arise in divide-and-conquer algorithms; for example, many sorting algorithms take $O(n \log n)$ operations since they work by recursively splitting a list into two sublists. Exponentials tend to arise from brute-force strategies: for example, there are 2^n bit strings of length n, so if we iterate over all of them, we will take time at least $O(2^n)$.

The base of a logarithm doesn't matter in big-O, since changing the base results only in a constant factor difference.

The function $\log n$ grows very slowly: it is $O(n^k)$ for any $k > 0$, even fractional k. In fact, even $\log^d n$ is $O(n^k)$.

Because $\log^d n$ grows so slowly, there's a common variant of big-O that tries to ignore

log factors instead of just constants: we say $f \in \tilde{O}(g)$ when we can scale g by $c(1 + \log^d n)$ to beat f, instead of just by c as in plain big-O.

Exponential functions bring a different wrinkle to big-O notation: they grow so quickly that we stop being able to ignore some constant factors that we might want to ignore. For example, 3^n is not in $O(2^n)$: the constant factor is in the base of the exponent, which is beyond what big-O can hide. For this reason, we will often see expressions like $2^{O(n)}$. This means exactly what it looks like: it's the set of functions of the form $2^{g(n)}$ where $g \in O(n)$.