Chapter 7

Asymptotic notation

Asymptotic notation is a tool for describing the behavior of functions on large values, which is used extensively in the analysis of algorithms.

7.1 Definitions

$O(f(n))$ A function $g(n)$ is in $O(f(n))$ ("**big O** of $f(n)$") if there exist constants $c > 0$ and N such that $|g(n)| \leq c|f(n)|$ for all $n > N$.

$\Omega(f(n))$ A function $g(n)$ is in $\Omega(f(n))$ ("**big Omega** of $f(n)$") if there exist constants $c > 0$ and N such that $|g(n)| \geq c|f(n)|$ for all $n > N$.

$\Theta(f(n))$ A function $g(n)$ is in $\Theta(f(n))$ ("**big Theta** of $f(n)$") if there exist constants $c_1 > 0$, $c_2 > 0$, and N such that $c_1|f(n)| \leq |g(n)| \leq c_2|f(n)|$ for all $n > N$. This is equivalent to saying that $g(n)$ is in both $O(f(n))$ and $\Omega(f(n))$.

$o(f(n))$ A function $g(n)$ is in $o(f(n))$ ("**little o** of $f(n)$") if for *every* $c > 0$ there exists an N such that $|g(n)| \leq c|f(n)|$ for all $n > N$. This is equivalent to saying that $\lim_{n\to\infty} g(n)/f(n) = 0$.

$\omega(f(n))$ A function $g(n)$ is in $\omega(f(n)$ ("**little omega** of $f(n)$") if for *every* $c > 0$ there exists an N such that $|g(n)| \geq c|f(n)|$ for all $n > N$. This is equivalent to saying that $\lim_{n\to\infty}|g(n)|/|f(n)|$ diverges to infinity.

7.2 Motivating the definitions

Why would we use this notation?

- Constant factors vary from one machine to another. The c factor hides this. If we can show that an algorithm runs in $O(n^2)$ time, we can be confident that it will continue to run in $O(n^2)$ time no matter how fast (or how slow) our computers get in the future.
- For the N threshold, there are several excuses:
  - Any problem can theoretically be made to run in $O(1)$ time for any finite subset of the possible inputs (e.g. all inputs expressible in 50 MB or less), by prefacing the main part of the algorithm with a very large table lookup. So it's meaningless to talk about the relative performance of different algorithms for bounded inputs.
  - If $f(n) > 0$ for all n, then we can get rid of N (or set it to zero) by making c large enough. But some functions $f(n)$ take on zero—or undefined—values for interesting n (e.g., $f(n) = n^2$ is zero when n is zero, and $f(n) = \log n$ is undefined for $n = 0$ and zero for $n = 1$). Allowing the minimum N lets us write $O(n^2)$ or $O(\log n)$ for classes of functions that we would otherwise have to write more awkwardly as something like $O(n^2 + 1)$ or $O(\log(n + 2))$.
  - Putting the $n > N$ rule in has a natural connection with the definition of a limit, where the limit as n goes to infinity of $g(n)$ is defined to be x if for each $\epsilon > 0$ there is an N such that $|g(n) - x| < \epsilon$ for all $n > N$. Among other things, this permits the limit test that says $g(n) = O(f(n))$ if the $\lim_{n\to\infty} \frac{g(n)}{f(n)}$ exists and is finite.

7.3 Proving asymptotic bounds

Most of the time when we use asymptotic notation, we compute bounds using stock theorems like $O(f(n)) + O(g(n)) = O(\max(f(n), g(n))$ or $O(cf(n)) = O(f(n))$. But sometimes we need to unravel the definitions to see whether a given function fits in a given class, or to prove these utility theorems to begin with. So let's do some examples of how this works.

Theorem 7.3.1. *The function n is in $O(n^3)$.*

Proof. We must find c, N such that for all $n > N$, $|n| \leq c|n^3|$. Since n^3 is much bigger than n for most values of n, we'll pick c to be something convenient to work with, like 1. So now we need to choose N so that for all $n > N$, $|n| \leq |n^3|$. It is not the case that $|n| \leq |n^3|$ for all n (try plotting

n vs n^3 for $n < 1$) but if we let $N = 1$, then we have $n > 1$, and we just need to massage this into $n^3 \geq n$. There are a couple of ways to do this, but the quickest is probably to observe that squaring and multiplying by n (a positive quantity) are both increasing functions, which means that from $n > 1$ we can derive $n^2 > 1^2 = 1$ and then $n^2 \cdot n = n^3 > 1 \cdot n = n$. □

Theorem 7.3.2. *The function n^3 is not in $O(n)$.*

Proof. Here we need to negate the definition of $O(n)$, a process that turns all existential quantifiers into universal quantifiers and vice versa. So what we need to show is that for all $c > 0$ and N, there exists some $n > N$ for which $|n^3|$ is not less than $c|n|$. So fix some such $c > 0$ and N. We must find an $n > N$ for which $n^3 > cn$. Solving for n in this inequality gives $n > c^{1/2}$; so setting $n > \max(N, c^{1/2})$ finishes the proof. □

Theorem 7.3.3. *If $f_1(n)$ is in $O(g(n))$ and $f_2(n)$ is in $O(g(n))$, then $f_1(n) + f_2(n)$ is in $O(g(n))$.*

Proof. Since $f_1(n)$ is in $O(g(n))$, there exist constants c_1, N_1 such that for all $n > N_1$, $|f_1(n)| < c|g(n)|$. Similarly there exist c_2, N_2 such that for all $n > N_2$, $|f_2(n)| < c|g(n)|$.

To show $f_1(n) + f_2(n)$ in $O(g(n))$, we must find constants c and N such that for all $n > N$, $|f_1(n) + f_2(n)| < c|g(n)|$. Let's let $c = c_1 + c_2$. Then if n is greater than $\max(N_1, N_2)$, it is greater than both N_1 and N_2, so we can add together $|f_1| < c_1|g|$ and $|f_2| < c_2|g|$ to get $|f_1 + f_2| \leq |f_1| + |f_2| < (c_1 + c_2)|g| = c|g|$. □

7.4 General principles for dealing with asymptotic notation

7.4.1 Remember the difference between big-O, big-Ω, and big-Θ

- Use big-O when you have an upper bound on a function, e.g. the zoo never got more than $O(1)$ new gorillas per year, so there were at most $O(t)$ gorillas at the zoo in year t.

- Use big-Ω when you have a lower bound on a function, e.g. every year the zoo got at least one new gorilla, so there were at least $\Omega(t)$ gorillas at the zoo in year t.

- Use big-Θ when you know the function exactly to within a constant-factor error, e.g. every year the zoo got exactly five new gorillas, so there were $\Theta(t)$ gorillas at the zoo in year t.

For the others, use little-o and ω when one function becomes vanishingly small relative to the other, e.g. new gorillas arrived rarely and with declining frequency, so there were $o(t)$ gorillas at the zoo in year t. These are not used as much as big-O, big-Ω, and big-Θ in the algorithms literature.

7.4.2 Simplify your asymptotic terms as much as possible

- $O(f(n)) + O(g(n)) = O(f(n))$ when $g(n) = O(f(n))$. If you have an expression of the form $O(f(n) + g(n))$, you can almost always rewrite it as $O(f(n))$ or $O(g(n))$ depending on which is bigger. The same goes for Ω and Θ.

- $O(cf(n)) = O(f(n))$ if c is a constant. You should never have a constant inside a big O. This includes bases for logarithms: since $\log_a x = \log_b x / \log_b a$, you can always rewrite $O(\lg n)$, $O(\ln n)$, or $O(\log_{1.4467712} n)$ as just $O(\log n)$.

- But watch out for exponents and products: $O(3^n n^{3.1178} \log^{1/3} n)$ is already as simple as it can be.

7.4.3 Use limits (may require calculus)

If you are confused whether e.g. $\log n$ is $O(n)$, try computing the limit as n goes to infinity of $\frac{\log n}{n}$, and see if it converges to a constant (zero is OK). The general rule is that $f(n)$ is $O(g(n)$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ exists.[1]

You may need to use L'Hôpital's Rule to evaluate such limits if they aren't obvious. This says that

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

when $f(n)$ and $g(n)$ both diverge to infinity or both converge to zero. Here f' and g' are the derivatives of f and g with respect to n; see §H.2.

[1] Note that this is a sufficient but not necessary condition. For example, the function $f(n)$ that is 1 when n is even and 2 when n is odd is $O(1)$, but $\lim_{n\to\infty} \frac{f(n)}{1}$ doesn't exist.

7.5 Asymptotic notation and summations

Algorithms often involve loops, where the cost of the loop is the sum of the costs of each iteration. When we are looking for an asymptotic cost, we don't need to compute an exact value for this sum, but can instead use an approximation that is accurate up to constant factors. This can make our life much easier.

Here's my usual strategy for computing sums in asymptotic form:

7.5.1 Pull out constant factors

Pull as many constant factors out as you can (where constant in this case means anything that does not involve the summation index). Example: $\sum_{i=1}^{n} \frac{n}{i} = n \sum_{i=1}^{n} \frac{1}{i} = nH_n = \Theta(n \log n)$. (See harmonic series below.)

7.5.2 Bound using a known sum

See if it's bounded above or below by some other sum whose solution you already know. Here are some good sums to try (some of these previously appeared in §6.4).

7.5.2.1 Geometric series

$$\sum_{i=0}^{n} x^i = \frac{1 - x^{n+1}}{1 - x} = \frac{x^{n+1} - 1}{x - 1}.$$

and

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1 - x}.$$

The way to recognize a geometric series is that the ratio between adjacent terms is constant. If you memorize the second formula, you can rederive the first one. If you're Gauss, you can skip memorizing the second formula.

A useful trick to remember for geometric series is that if x is a constant that is not exactly 1, the sum is always big-Theta of its largest term. So for example $\sum_{i=1}^{n} 2^i = \Theta(2^n)$ (the exact value is $2^{n+1} - 1$), and $\sum_{i=1}^{n} 2^{-i} = \Theta(1)$ (the exact value is $1 - 2^{-n}$).

If the ratio between terms equals 1, the formula doesn't work; instead, we have a constant series (see below).

7.5.2.2 Constant series

$$\sum_{i=1}^{n} 1 = n.$$

7.5.2.3 Arithmetic series

The simplest arithmetic series is

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

The way to remember this formula is that it's just n times the average value $(n+1)/2$. The way to recognize an arithmetic series is that the difference between adjacent terms is constant. The general arithmetic series is of the form

$$\begin{aligned}\sum_{i=1}^{n}(ai+b) &= \sum_{i=1}^{n} ai + \sum_{i=1}^{n} b \\ &= a \cdot \frac{n(n+1)}{2} + bn.\end{aligned}$$

Because the general series expands so easily to the simple series, it's usually not worth memorizing the general formula.

In asymptotic terms, every arithmetic series is $\Theta(n^2)$.

7.5.2.4 Harmonic series

$$\sum_{i=1}^{n} 1/i = H_n = \Theta(n \log n).$$

Can be rederived using the integral technique given below or by summing the last half of the series, so this is mostly useful to remember in case you run across H_n (the "n-th **harmonic number**").

The infinite sum $\sum_{i=1}^{\infty} 1/i$ diverges: even though it grows very slowly as i gets larger, adding enough terms will eventually exceed any constant bound.

The value of the more general infinite sum $\sum_{i=1}^{\infty} 1/i^s$ is called $\zeta(s)$, and ζ is called the **Riemann zeta function**. The harmonic series is the case where $s = 1$, and because it diverges, s, $\zeta(1)$ is undefined. However, $\zeta(s)$ is defined for $s > 1$. The exact value can be hard to compute,[2] but as long as s does not depend on n, it's $\Theta(1)$ for any fixed $s > 1$.

[2] Finding just the value of $\zeta(2) = \sum_{i=1}^{\infty} 1/i^2 = \pi^2/6$ was known as the **Basel problem**

7.5.3 Bound part of the sum

See if there's some part of the sum that you can bound. For example, $\sum_{i=1}^{n} i^3$ has a (painful) exact solution, or can be approximated by the integral trick described below, but it can very quickly be solved to within a constant factor by observing that $\sum_{i=1}^{n} i^3 \leq \sum_{i=1}^{n} n^3 = O(n^4)$ and $\sum_{i=1}^{n} i^3 \geq \sum_{i=n/2}^{n} i^3 \geq \sum_{i=n/2}^{n} (n/2)^3 = \Omega(n^4)$.

7.5.4 Integrate

Integrate. If $f(n)$ is non-decreasing and you know how to integrate it, then $\int_{a-1}^{b} f(x)\,dx \leq \sum_{i=a}^{b} f(i) \leq \int_{a}^{b+1} f(x)\,dx$, which is enough to get a big-Theta bound for almost all functions you are likely to encounter in algorithm analysis. If you don't know how to integrate, see §H.3.

7.5.5 Grouping terms

Try grouping terms together. For example, the standard trick for showing that the harmonic series is unbounded in the limit is to argue that $1 + 1/2 + 1/3 + 1/4 + 1/5 + 1/6 + 1/7 + 1/8 + \ldots \geq 1 + 1/2 + (1/4 + 1/4) + (1/8 + 1/8 + 1/8 + 1/8) + \ldots \geq 1 + 1/2 + 1/2 + 1/2 + \ldots$. I usually try everything else first, but sometimes this works if you get stuck.

Warning: Though it's always safe to reorder terms in a finite sum, bad things can happen if you reorder an infinite sum that includes both positive and negative terms.

7.5.6 An odd sum

One oddball sum that shows up occasionally but is hard to solve using any of the above techniques is $\sum_{i=1}^{n} a^i i$. If $a < 1$, this is $\Theta(1)$ (the exact formula for $\sum_{i=1}^{\infty} a^i i$ when $a < 1$ is $a/(1-a)^2$, which gives a constant upper bound for the sum stopping at n); if $a = 1$, it's just an arithmetic series; if $a > 1$, the largest term dominates and the sum is $\Theta(a^n n)$ (there is an exact formula, but it's ugly—if you just want to show it's $O(a^n n)$, the simplest approach is to bound the series $\sum_{i=0}^{n-1} a^{n-i}(n-i)$ by the geometric series $\sum_{i=0}^{n-1} a^{n-i} n \leq a^n n/(1-a^{-1}) = O(a^n n)$. I wouldn't bother memorizing this one provided you remember how to find it in these notes.

---

and took 90 years to solve. When s can be a complex number, showing that $\zeta(s) = 0$ only if s is a negative even integer or of the form $1/2 + ai$ is the **Riemann hypothesis** and has not yet been proven or disproven since it was first proposed in 1859. You will not be asked to solve either of these problems in this class.

7.5.7 Final notes

In practice, almost every sum you are likely to encounter in algorithm analysis will be of the form $\sum_{i=1}^{n} f(n)$ where $f(n)$ is exponential (so that it's bounded by a geometric series and the largest term dominates) or polynomial (so that $f(n/2) = \Theta(f(n))$) and the sum is $\Theta(nf(n))$ using the $\sum_{i=n/2}^{n} f(n) = \Omega(nf(n))$ lower bound).

Graham *et al.* [GKP94] spend a lot of time on computing sums exactly. The most generally useful technique for doing this is to use generating functions (see §11.3).

7.6 Variations in notation

As with many tools in mathematics, you may see some differences in how asymptotic notation is defined and used.

7.6.1 Absolute values

Some authors leave out the absolute values. For example, Biggs [Big02] defines $f(n)$ as being in $O(g(n))$ if $f(n) \leq cg(n)$ for sufficiently large n. If $f(n)$ and $g(n)$ are non-negative, this is not an unreasonable definition. But it produces odd results if either can be negative: for example, by this definition, $-n^{1000}$ is in $O(n^2)$. Some authors define O, Ω, and Θ *only* for non-negative functions, avoiding this problem.

The most common definition (which we will use) says that $f(n)$ is in $O(g(n))$ if $|f(n)| \leq c|g(n)|$ for sufficiently large n; by this definition $-n^{1000}$ is *not* in $O(n^2)$, though it is in $O(n^{1000})$. This definition was designed for error terms in asymptotic expansions of functions, where the error term might represent a positive or negative error.

7.6.2 Abusing the equals sign

Formally, we can think of $O(g(n))$ as a predicate on functions, which is true of all functions $f(n)$ that satisfy $f(n) \leq cg(n)$ for some c and sufficiently large n. This requires writing that n^2 is $O(n^2)$ where most computer scientists or mathematicians would just write $n^2 = O(n^2)$. Making sense of the latter statement involves a standard convention that is mildly painful to define formally but that greatly simplifies asymptotic analyses.

Let's take a statement like the following:

$$O(n^2) + O(n^3) + 1 = O(n^3).$$

What we want this to mean is that the left-hand side can be replaced by the right-hand side without causing trouble. To make this work formally, we define the statement as meaning that for any f in $O(n^2)$ and any g in $O(n^3)$, there exists an h in $O(n^3)$ such that $f(n) + g(n) + 1 = h(n)$.

In general, any appearance of O, Ω, or Θ on the left-hand side gets a universal quantifier (for all) and any appearance of O, Ω, or Θ on the right-hand side gets an existential quantifier (there exists). So

$$f(n) + o(f(n)) = \Theta(f(n))$$

means that for any g in $o(f(n))$, there exists an h in $\Theta(f(n))$ such that $f(n) + g(n) = h(n)$, and

$$O(f(n)) + O(g(n)) + 1 = O(\max(f(n), g(n))) + 1$$

means that for any r in $O(f(n))$ and s in $O(g(n))$, there exists t in $O(\max(f(n), g(n))$ such that $r(n) + s(n) + 1 = t(n) + 1$.

The nice thing about this definition is that as long as you are careful about the direction the equals sign goes in, you can treat these complicated pseudo-equations like ordinary equations. For example, since $O(n^2)+O(n^3) = O(n^3)$, we can write

$$\begin{aligned}\frac{n^2}{2} + \frac{n(n+1)(n+2)}{6} &= O(n^2) + O(n^3) \\ &= O(n^3),\end{aligned}$$

which is much simpler than what it would look like if we had to talk about particular functions being elements of particular sets of functions.

This is an example of **abuse of notation**, the practice of redefining some standard bit of notation (in this case, equations) to make calculation easier. It's generally a safe practice as long as everybody understands what is happening. But beware of applying facts about unabused equations to the abused ones. Just because $O(n^2) = O(n^3)$ doesn't mean $O(n^3) = O(n^2)$—the big-O equations are not reversible the way ordinary equations are.

More discussion of this can be found in [Fer08, §10.4] and [GKP94, Chapter 9].