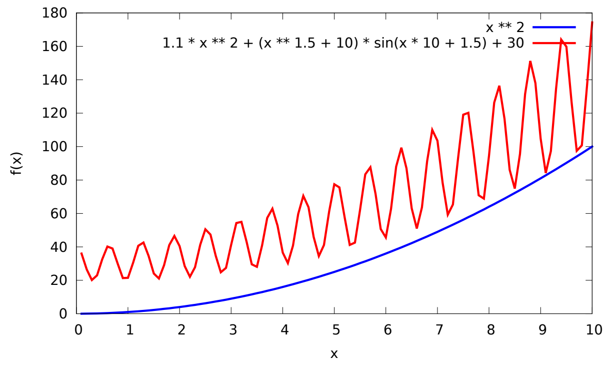
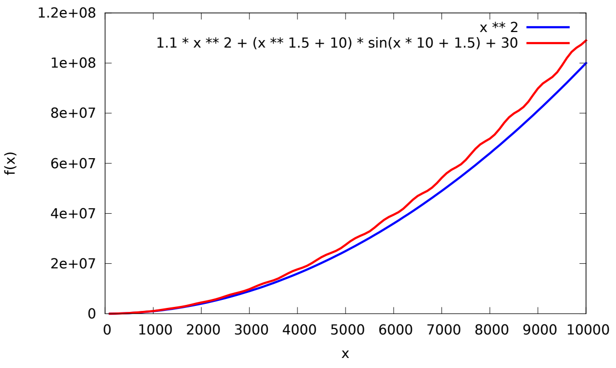
Asymptotic Complexity

These notes aim to help you build an intuitive understanding of asymptotic notation. They are a supplement to the material in the textbook, not a replacement for it.

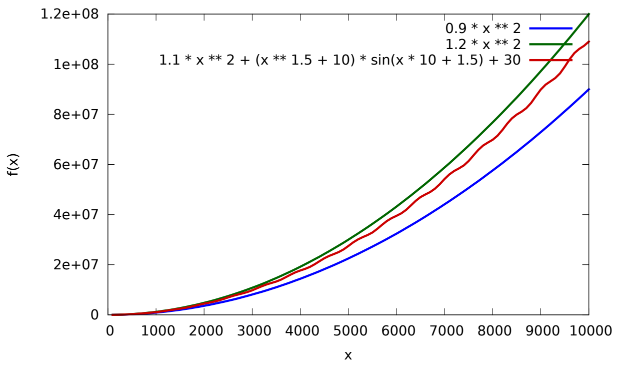
Informally, asymptotic notation takes a 10,000 feet view of the function's growth. For example, let's look at $f_1(x) = x^2$ and $f_2(x) = 1.1x^2 + (x^{1.9} + 10)\sin(10x + 1.5) + 30$. f_2 looks a lot more complex than f_1. For small values of x, the functions' graphs also look very different. However, if we increase the scale by 1000 times, we get a very different picture. It looks like, in the bigger picture of things, f_1 and f_2 aren't so different after all.





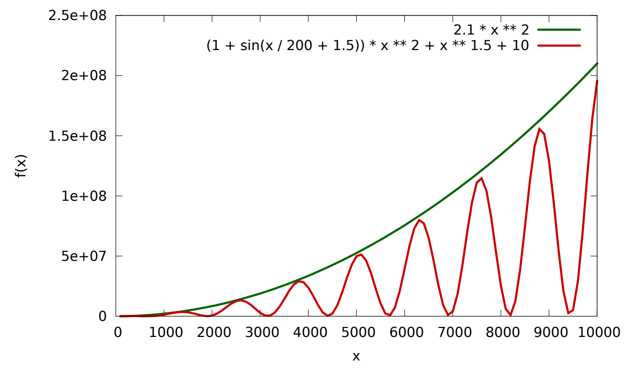
Asymptotic notation, also known as "big-Oh" notation, uses the symbols O, Θ and Ω. The notation, $f_2(x) = \Theta(x^2)$, is really misleading, because it makes it seem like $\Theta(x^2)$ is a function.

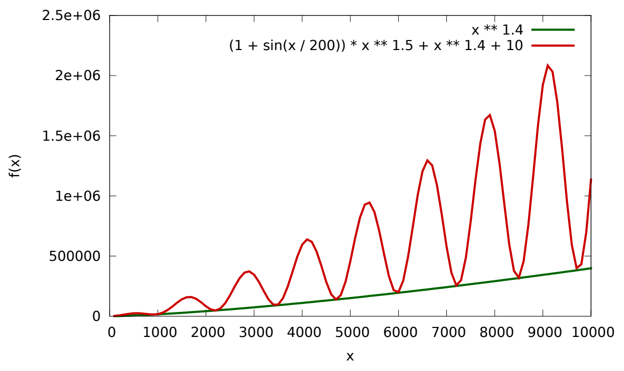
$f(x) = \Theta(x^2)$ implies both a lower bound and an upper bound, as the graph below shows. The graph shows a visual proof that $f_2(x) = \Theta(x^2)$, by showing that it's bounded from above by $1.2x^2$, and that it is bounded from below by $0.9x^2$. These two functions only differ from the function inside the Θ by a constant factor.



Θ "constrains" a function both from above and from below. O only makes a statement about the upper bound of a function, and Ω only makes a statement about the function's lower bound.

The left graph below plots $g(x) = (1 + \sin(\frac{x}{200} + 1.5))x^2 + x^{1.5} + 10$, and proves visually that $g(x) = O(x^2)$. Note that we cannot say $g(x) = \Theta(x^2)$, because the coefficient of x^2 becomes 0 for some values of x, so $g(x)$ is only bounded from below by $x^{1.5}$ ($g(x) = \Omega(x^{1.5})$).





The right graph above shows $h(x) = (1 + \sin(\frac{x}{200} + 1.5))x^{1.5} + x^{1.4} + 10$, and proves visually that $h(x) = \Omega(x^{1.4})$. $h(x) \neq \Theta(x^{1.4})$ because, for most values of x, the coefficient of $x^{1.5}$ is positive, so the $x^{1.5}$ term dominates the function's value.

Asymptotic Drowning

In asymptotic notation, we can reduce complex functions involving logarithms, using the following rules.

- $\log(n^{100}) = 100\log(n) = \Theta(\log(n))$ - constant exponents don't matter
- $\log_{\log(5)}(n) = \frac{\log(n)}{\log(5)} = \Theta(\log(n))$ - constant bases don't matter

Don't confuse an exponential inside a logarithm with a logarithm inside an exponential. For example:

- $n^{\log(5)}$ cannot be simplified; we can get some loose bounds for it by observing that $\log(5) \approx 2.3219\ldots$, so $n^2 \leq n^{\log(5)} \leq n^3$

The rules were extracted from the textbook, which also contains useful rules for other functions that we'll meet in 6.006, such as polynomials.

Exercises

Compute simple, tight asymptotic bounds for $f(n)$, where $f(n)$ is the following:

- 10^{80} (the number of atoms in the universe)
- $\log_{\ln 5}(\log^{\log 100} n)$
- $(20n)^7$
- $5^{\log(3)}n^3 + 10^{80}n^2 + \log(3)n^{3.1} + 6006$
- $\log \binom{n}{\frac{n}{2}}$ (*hint*: use Stirling's approximation, $n! \approx \sqrt{2\pi n}(\frac{N}{e})^N$)

Solutions

- 10^{80} (the number of atoms in the universe) is $\Theta(1)$ because there is no n in it, so it's a constant (although a mighty big one)
- $\log_{\ln 5}(\log^{\log 100} n)$ is $\Theta(\log(\log(n)))$ after applying the properties of logarithms
- $(20n)^7$ is $\Theta(n^7)$, because $(20n)^7 = 20^7 \times n^7 = \Theta(n^7)$ (20^7 is a big constant factor)
- $5^{\log(3)}n^3 + 10^{80}n^2 + \log(3)n^{3.1} + 6006$ is $\Theta(n^{3.1})$. Eliminate the constant factors to obtain $\Theta(n^3) + \Theta(n^2) + \Theta(n^{3.1}) + \Theta(1)$, then observe that $\Theta(n^{3.1})$ dominates the sum.
- $\log \binom{n}{\frac{n}{2}}$ is $O(n)$. Use the binomial identity to obtain $\binom{n}{\frac{n}{2}} = \frac{n!}{(\frac{n}{2}!)^2}$, then apply Stirling's approximation and the logarithm properties.

Recurrences

Recurrences show up when trying to analyze the running time of divide-and-conquer algorithms. In a nutshell, divide-and-conquer is a general approach that suggests breaking down big problems into many smaller sub-problems that are manageable to solve, and then combining the solutions for the small sub-problems to obtain the solution to the bigger problem.

1. **Divide** Break a problem into smaller sub-problems
2. **Conquer** Really small subproblems are easy
3. **Profit** Combine answers to sub-problems

Sample Recurrence

Binary search is the canonical example of divide and conquer. If I would be looking for the word "algorithms" in a physical dictionary (we'll need to stop using this example when e-books rule the world), I would open the book right in the middle, see that the first word on the left page is "minotaur", and conclude that it's safe to ignore the right half of the book. Then I would split the left half of the dictionary into two, and perhaps I would see the word "gargantuan". Again, I know I can ignore all the pages to the right of "gargantuan", so I'd focus on the left half of the pages. Eventually, I will find my word, and my search will take much less time than it would if I would go through each page individually.

Binary search maps to the divide-and-conquer paradigm as follows. Suppose I am looking for a number x in a sorted array of numbers $A[1 \ldots n]$

1. **Divide** Compare $A[\frac{n}{2}]$ with x. If the numbers are equal, report success. If $x < A[\frac{n}{2}]$, recurse (repeat the process focusing) on $A[1 \ldots \frac{n}{2} - 1]$. Otherwise, recurse on $A[\frac{n}{2} + 1 \ldots n]$.
2. **Conquer** If the array A is empty, x cannot be in it, so I can report failure.

3. **Profit** If I have found x, the problem is obviously solved. However, if I have not found x in the half of the array that I recursed on, I need to convince myself that I'm not missing out by ignoring the other half of the array, and declare that x is not in A, despite the fact that I completely ignored half of its elements.

Let $T(n)$ be the time it takes to find x in $A[1 \ldots n]$, or give up. For simplicity, let's focus on the case where x does not exist in A. Convince yourself that this uncovers the worst-case runnning time for binary search.

Assuming that making a guess takes a constant amount of time, we can write the following recurrence for $T(n)$.

$$\begin{aligned} T(n) &= T(\frac{n}{2}) + \Theta(1) \\ T(1) &= \Theta(1) \end{aligned}$$

Remember, from the section on asymptotics, that any constant is $\Theta(1)$, so we could have used more base cases, if that would have been convenient for us. We could even say that $T(1)$ up to $T(1000)$ are $\Theta(1)$, and our recurrence only works for $n > 1000$.

Let's aim to guess the closed form formula for $T(n)$. Since we're making a guess, we'll be a bit sloppy. $\Theta(1)$ is a constant, so we can rewrite the recurrence as $T(n) = T(\frac{n}{2}) + c$. The advantage of using c is that we can resist the urge of adding up the $\Theta(1)$ terms. Then we'll expand it a few times:

$$\begin{aligned} T(n) &= T(\frac{n}{2}) + c \\ T(n) &= T(\frac{n}{4}) + c + c \\ T(n) &= T(\frac{n}{8}) + c + c + c \\ T(n) &= T(\frac{n}{16}) + c + c + c + c \\ &\vdots \\ T(n) &= T(\frac{n}{2^i}) + i \times c \end{aligned}$$

We can stop expanding when we hit a base case, so we want to set i such that $\frac{n}{2^i} = 1$, so that $T(\frac{n}{2^i}) = \Theta(1)$. we get $T(n) = \Theta(1) + i\Theta(1)$. Using $i = \log(n)$ we obtain $T(n) = \Theta(\log(n))$.

Recurrence Traps

One potential pitfall in a proof involving big-O notation is the fact that the notation hides information about the constants involved. To illustrate this problem, we shall prove that the function $f(n) = n$ is $O(1)$. The proof is by induction, with $n = 1$ as the base case. Clearly, $f(1) = 1$ is a constant, so we can say that $f(1) = O(1)$. The inductive step quickly follows: assume $f(x) = O(1)$ for all $x < n$.

$$\begin{aligned} f(n) &= n \\ &= n-1+1 \\ &= f(n-1)+1 \\ &= O(1)+1 \\ &= O(1) \end{aligned}$$

What went wrong here? The problem here occurred in the inductive step. By definition, $f(n)$ is $O(1)$ if and only if there exists some constant c such that for sufficiently large values of n, $f(n) < c$. This constant c must be the *same* for all values of n. When we assume that $f(n-1)$ is $O(1)$, the constant used is not the same as the constant used when we prove that that $f(n)$ is $O(1)$. The big-O notation hides the fact that the constant increases with every step of the inductive process, and is therefore not actually constant.

To avoid this trap, it is generally a good idea to avoid the use of big-O notation in proofs by induction. It is usually a good policy to replace all uses of big-O and big-Ω with their definitions, picking fixed variables to represent the constants involved. If we attempt to do this in the proof that $f(n)$ is $O(1)$, our inductive assumption becomes $f(x) < c$ for all $x < n$. Therefore, the attempted proof that $f(n) < c$ becomes:

$$\begin{aligned} f(n) &= n \\ &= n-1+1 \\ &= f(n-1)+1 \\ &< c+1. \end{aligned}$$

This lets us avoid the error.

2-D Peak Finding: Algorithm 5

After class, several students asked about a variant of the peak-finding algorithm presented in Lecture 1. In Lecture 1, the basic algorithm is:

1. Find the location (r, c) that has the maximum value of in the middle column.
2. Look at the values at locations $(r, c-1)$ and $(r, c+1)$. If neither value is greater than the value at (r, c), then it is a peak, so return it. Otherwise, recurse into one of the halves containing a greater value.

The suggested modification to the algorithm is:

1. Treat the middle column as a one-dimensional peak problem, and find a one-dimensional peak (r, c) using the algorithm given in lecture.

2. Look at the values at locations $(r, c-1)$ and $(r, c+1)$. If neither value is greater than the value at (r, c), then it is a peak, so return it. Otherwise, recurse into one of the halves containing a greater value.

This algorithm has been implemented in the same framework as the existing algorithms. The implementation can be found in the support files for this recitation, and can be run in much the same way. You are encouraged to try it out on your own.

We begin by analyzing the efficiency of the new algorithm. Let $T(m, n)$ denote the runtime of the algorithm when run on a matrix with m rows and n columns. The number of elements in the middle column is m, so the time required to find a one-dimensional peak is $O(\log m)$. Checking the two-dimensional neighbors of the one-dimensional peak requires $O(1)$ time. The recursive call reduces the number of columns to at most $\lfloor n/2 \rfloor$, but does not change the number of rows. Therefore, we may write the following recurrence relation for the runtime of the algorithm:

$$T(m, n) = O(1) + O(\log m) + T(m, \lfloor n/2 \rfloor).$$

Intuitively, the number of rows in the problem does not change over time, so the cost per recursive call is always $O(1) + O(\log m)$. The number of columns n is halved at every step, so the number of recursive calls is at most $O(1 + \log n)$. So we may guess a bound of $O((1 + \log m)(1 + \log n))$. To show this bound more formally, we must first rewrite the recurrence relation using constants $c_1, c_2 > 0$, instead of big-O notation:

$$T(m, n) \leq c_1 + c_2 \log m + T(m, \lfloor n/2 \rfloor).$$

We now want to show that $T(m, n) \leq c_3(1+\log m)(1+\log n)$. We will show this by induction. Assume that this is true for all $n < k$. We wish to show that this is also true for $n = k$. We may perform the following substitution:

$$\begin{aligned} T(m, k) &\leq c_1 + c_2 \log m + T(m, \lfloor k/2 \rfloor) \\ &\leq c_1 + c_2 \log m + c_3(1 + \log m)(1 + \log\lfloor k/2 \rfloor) \\ &\leq c_1 + c_2 \log m + c_3(1 + \log m)(1 + \log k - \log 2) \\ &\leq c_1 + c_2 \log m + c_3(1 + \log m)(1 + \log k) - c_3(1 + \log m)\log 2 \\ &\leq c_3(1 + \log m)(1 + \log k) + (c_1 + c_2 \log m - c_3(1 + \log m)\log 2) \end{aligned}$$

So as long as it is possible to set c_3 sufficiently high to make $(c_1 + c_2 \log m - c_3(1+\log m)\log 2) \leq 0$, we know that we will have proved the inductive step. To ensure that this is true, it is sufficient to set $c_3 = (c_1 + c_2)/\log 2$, resulting in the following inequality:

$$\begin{aligned} c_1 + c_2 \log m - c_3(1 + \log m)\log 2 &= c_1 + c_2 \log m - ((c_1 + c_2)/\log 2) \cdot (1 + \log m)\log 2 \\ &= c_1 + c_2 \log m - (c_1 + c_2)(1 + \log m) \\ &= c_1 + c_2 \log m - c_1 - c_1 \log m - c_2 - c_2 \log m \\ &= -c_1 \log m - c_2 \\ &\leq 0 \end{aligned}$$

Hence, the runtime of the algorithm is $O((1 + \log m)(1 + \log n))$. This is faster than the algorithm seen in Lecture 1, but is it also correct?

If the algorithm were incorrect, how would we find a counterexample for it? We might begin by looking at the differences between the algorithm presented in Lecture 1, which is known to be correct, and the algorithm that we are studying. In the first step, both algorithms look at the central column; one finds the maximum, and one finds a peak. A good technique for constructing a counterexample would be to begin with a matrix in which the 1D peak found by the algorithm would not be the maximum of the central column. To that end, we will start with a 5×5 matrix with a central column satisfying that property:

$$\begin{array}{ccccc}
? & ? & 0 & ? & ? \\
? & ? & 2 & ? & ? \\
? & ? & 1 & ? & ? \\
? & ? & 0 & ? & ? \\
? & ? & 7 & ? & ?
\end{array}$$

The one-dimensional peak-finding algorithm will examine the very center of the matrix, and will recurse on the upper half (because its neighbor to the south is strictly smaller). Therefore, it will find the one-dimensional peak 2. If 2 is a two-dimensional peak, the algorithm will (correctly) return it. So if we want to construct a counterexample, 2 must have a neighbor that is strictly greater. The algorithm must pick a half to recurse on; for simplicity, we force it to recurse on the left:

$$\begin{array}{ccccc}
? & ? & 0 & 0 & 0 \\
? & 3 & 2 & 0 & 0 \\
? & ? & 1 & 0 & 0 \\
? & ? & 0 & 0 & 0 \\
? & ? & 7 & 0 & 0
\end{array}$$

This will cause the algorithm to examine only the two left-most columns of the matrix. Is it possible to ensure that there are no peaks in those columns? Because there is a large value adjacent to this region, it is:

$$\begin{array}{ccccc}
1 & 2 & 0 & 0 & 0 \\
2 & 3 & 2 & 0 & 0 \\
3 & 4 & 1 & 0 & 0 \\
4 & 5 & 0 & 0 & 0 \\
5 & 6 & 7 & 0 & 0
\end{array}$$

Hence, we have an example where the algorithm will recurse into a subproblem that does not contain any two-dimensional peaks. As a result, the algorithm cannot return a peak, and must therefore be incorrect. This counterexample may be found in the file `counter.py` in the code distributed with these recitation notes.