2 Mathematical Background

Before we begin learning algorithmic techniques, we take a detour to give ourselves some necessary mathematical tools. First, we cover mathematical definitions of terms that are used later on in the book. By expanding your mathematical vocabulary you can be more precise and you can state or formulate problems more simply. Following that, we cover techniques for analysing the running time of an algorithm. After each major algorithm covered in this book we give an analysis of its running time as well as a proof of its correctness

2.1 Asymptotic Notation

In addition to correctness another important characteristic of a useful algorithm is its time and memory consumption. Time and memory are both valuable resources and there are important differences (even when both are abundant) in how we can use them.

How can you measure resource consumption? One way is to create a function that describes the usage in terms of some characteristic of the input. One commonly used characteristic of an input dataset is its size. For example, suppose an algorithm takes an input as an array of n integers. We can describe the time this algorithm takes as a function f written in terms of n. For example, we might write:

$$f(n) = n^2 + 3n + 14$$

where the value of $f(n)$ is some unit of time (in this discussion the main focus will be on time, but we could do the same for memory consumption). Rarely are the units of time actually in seconds, because that would depend on the machine itself, the system it's running, and its load. Instead, the units of time typically used are in terms of the number of some fundamental operation performed. For example, some fundamental operations we might care about are: the number of additions or multiplications needed; the number of element comparisons; the number of memory-location swaps performed; or the raw number of machine instructions executed. In general we might just refer to these fundamental operations performed as steps taken.

Is this a good approach to determine an algorithm's resource consumption? Yes and no. When two different algorithms are similar in time consumption a precise function might help to determine which algorithm is faster under given conditions. But in many cases it is either difficult or impossible to calculate an analytical description of the exact number of operations needed, especially when the algorithm performs operations conditionally on the values of its input. Instead, what really is important is not the precise time required to complete the function, but rather the degree that resource consumption changes depending

on its inputs. Concretely, consider these two functions, representing the computation time required for each size of input dataset:

$$f(n) = n^3 - 12n^2 + 20n + 110$$

$$g(n) = n^3 + n^2 + 5n + 5$$

They look quite different, but how do they behave? Let's look at a few plots of the function ($f(n)$ is in red, $g(n)$ in blue):

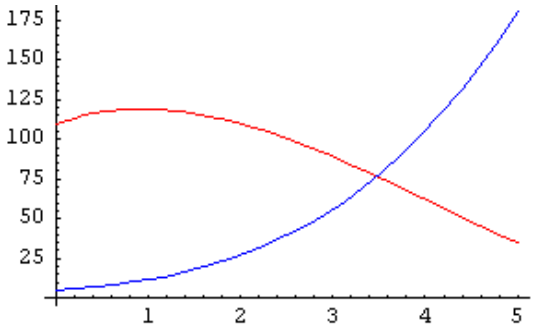

Figure 1 Plot of f and g, in range 0 to 5

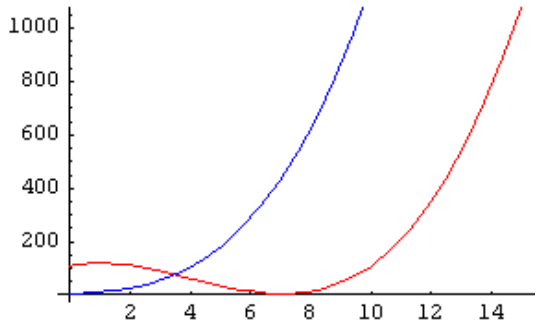

Figure 2 Plot of f and g, in range 0 to 15

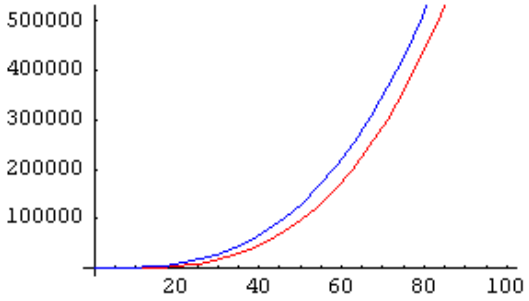

Figure 3 Plot of f and g, in range 0 to 100

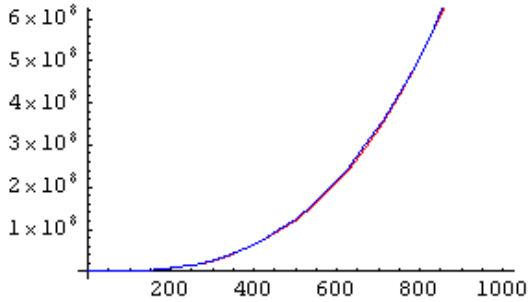

Figure 4 Plot of f and g, in range 0 to 1000

In the first, very-limited plot the curves appear somewhat different. In the second plot they start going in sort of the same way, in the third there is only a very small difference, and at last they are virtually identical. In fact, they approach n^3, the dominant term. As n gets larger, the other terms become much less significant in comparison to n^3.

As you can see, modifying a polynomial-time algorithm's low-order coefficients doesn't help much. What really matters is the highest-order coefficient. This is why we've adopted a notation for this kind of analysis. We say that:

$$f(n) = n^3 - 12n^2 + 20n + 110 = O(n^3)$$

We ignore the low-order terms. We can say that:

$$O(\log n) \leq O(\sqrt{n}) \leq O(n) \leq O(n \log n) \leq O(n^2) \leq O(n^3) \leq O(2^n)$$

This gives us a way to more easily compare algorithms with each other. Running an insertion sort on n elements takes steps on the order of $O(n^2)$. Merge sort sorts in $O(n \log n)$ steps. Therefore, once the input dataset is large enough, merge sort is faster than insertion sort.

In general, we write

$$f(n) = O(g(n))$$

when

$$\exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n).$$

That is, $f(n) = O(g(n))$ holds if and only if there exists some constants c and n_0 such that for all $n > n_0$ $f(n)$ is positive and less than or equal to $cg(n)$.

Note that the equal sign used in this notation describes a relationship between $f(n)$ and $g(n)$ instead of reflecting a true equality. In light of this, some define Big-O in terms of a set, stating that:

$$f(n) \in O(g(n))$$

when

$$f(n) \in \{f(n) : \exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)\}.$$

Big-O notation is only an upper bound; these two are both true:

$$n^3 = O(n^4)$$

$$n^4 = O(n^4)$$

If we use the equal sign as an equality we can get very strange results, such as:

$$n^3 = n^4$$

which is obviously nonsense. This is why the set-definition is handy. You can avoid these things by thinking of the equal sign as a one-way equality, i.e.:

$$n^3 = O(n^4)$$

does not imply

$$O(n^4) = n^3$$

Always keep the O on the right hand side.

2.1.1 Big Omega

Sometimes, we want more than an upper bound on the behavior of a certain function. Big Omega provides a lower bound. In general, we say that

$$f(n) = \Omega(g(n))$$

when

$$\exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : 0 \leq c \cdot g(n) \leq f(n).$$

i.e. $f(n) = \Omega(g(n))$ if and only if there exist constants c and n_0 such that for all $n > n_0$ f(n) is positive and **greater** than or equal to cg(n).

So, for example, we can say that

$n^2 - 2n = \Omega(n^2)$, (c=1/2, n_0=4) or

$n^2 - 2n = \Omega(n)$, (c=1, n_0=3),

but it is false to claim that

$$n^2 - 2n = \Omega(n^3).$$

2.1.2 Big Theta

When a given function is both O(g(n)) and Ω(g(n)), we say it is Θ(g(n)), and we have a tight bound on the function. A function f(n) is Θ(g(n)) when

$$\exists c_1 > 0, \exists c_2 > 0, \exists n_0 > 0, \forall n \geq n_0 : 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n),$$

but most of the time, when we're trying to prove that a given $f(n) = \Theta(g(n))$, instead of using this definition, we just show that it is both O(g(n)) and Ω(g(n)).

2.1.3 Little-O and Omega

When the asymptotic bound is not tight, we can express this by saying that $f(n) = o(g(n))$ or $f(n) = \omega(g(n))$. The definitions are:

f(n) is o(g(n)) iff $\forall c > 0, \exists n_0 > 0, \forall n \geq n_0 : 0 \leq f(n) < c \cdot g(n)$ and

f(n) is ω(g(n)) iff $\forall c > 0, \exists n_0 > 0, \forall n \geq n_0 : 0 \leq c \cdot g(n) < f(n)$.

Note that a function f is in o(g(n)) when for any coefficient of g, g eventually gets larger than f, while for O(g(n)), there only has to exist a single coefficient for which g eventually gets at least as big as f.

[TODO: define what T(n,m) = O(f(n,m)) means. That is, when the running time of an algorithm has two dependent variables. Ex, a graph with n nodes and m edges. It's important to get the quantifiers correct!]

2.2 Algorithm Analysis: Solving Recurrence Equations

Merge sort of n elements: $T(n) = 2 * T(n/2) + c(n)$ This describes one iteration of the merge sort: the problem space n is reduced to two halves ($2 * T(n/2)$), and then merged back together at the end of all the recursive calls ($c(n)$). This notation system is the bread and butter of algorithm analysis, so get used to it.

There are some theorems you can use to estimate the big Oh time for a function if its recurrence equation fits a certain pattern.

[TODO: write this section]

2.2.1 Substitution method

Formulate a guess about the big Oh time of your equation. Then use proof by induction to prove the guess is correct.

[TODO: write this section]

2.2.2 Summations

[TODO: show the closed forms of commonly needed summations and prove them]

2.2.3 Draw the Tree and Table

This is really just a way of getting an intelligent guess. You still have to go back to the substitution method in order to prove the big Oh time.

[TODO: write this section]

2.2.4 The Master Theorem

Consider a recurrence equation that fits the following formula:

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^k)$$

for $a \geq 1$, $b > 1$ and $k \geq 0$. Here, a is the number of recursive calls made per call to the function, n is the input size, b is how much smaller the input gets, and k is the polynomial order of an operation that occurs each time the function is called (except for the base cases). For example, in the merge sort algorithm covered later, we have

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

because two subproblems are called for each non-base case iteration, and the size of the array is divided in half each time. The $O(n)$ at the end is the "conquer" part of this divide and conquer algorithm: it takes linear time to merge the results from the two recursive calls into the final result.

Thinking of the recursive calls of T as forming a tree, there are three possible cases to determine where most of the algorithm is spending its time ("most" in this sense is concerned with its asymptotic behaviour):

1. the tree can be **top heavy**, and most time is spent during the initial calls near the root;
2. the tree can have a **steady state**, where time is spread evenly; or
3. the tree can be **bottom heavy**, and most time is spent in the calls near the leaves

Depending upon which of these three states the tree is in T will have different complexities:

The Master Theorem

Given $T(n) = aT\left(\frac{n}{b}\right) + O(n^k)$ for $a \geq 1$, $b > 1$ and $k \geq 0$:

- If $a < b^k$, then $T(n) = O(n^k)$ (top heavy)
- If $a = b^k$, then $T(n) = O(n^k \cdot \log n)$ (steady state)
- If $a > b^k$, then $T(n) = O(n^{\log_b a})$ (bottom heavy)

For the merge sort example above, where

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

we have

$$a = 2, b = 2, k = 1 \Longrightarrow b^k = 2$$

thus, $a = b^k$ and so this is also in the "steady state": By the master theorem, the complexity of merge sort is thus

$$T(n) = O(n^1 \log n) = O(n \log n)$$

2.3 Amortized Analysis

[Start with an adjacency list representation of a graph and show two nested for loops: one for each node n, and nested inside that one loop for each edge e. If there are n nodes and m edges, this could lead you to say the loop takes O(nm) time. However, only once could the innerloop take that long, and a tighter bound is O(n+m).]