Lecture 2

Asymptotic Analysis and Recurrences

2.1 Overview

In this lecture we discuss the notion of asymptotic analysis and introduce O, Ω, Θ, and o notation. We then turn to the topic of recurrences, discussing several methods for solving them. Recurrences will come up in many of the algorithms we study, so it is useful to get a good intuition for them right at the start. In particular, we focus on divide-and-conquer style recurrences, which are the most common ones we will see.

Material in this lecture:

- Asymptotic notation: O, Ω, Θ, and o.
- Recurrences and how to solve them.
  - Solving by unrolling.
  - Solving with a guess and inductive proof.
  - Solving using a recursion tree.
  - A master formula.

2.2 Asymptotic analysis

When we consider an algorithm for some problem, in addition to knowing that it produces a correct solution, we will be especially interested in analyzing its running time. There are several aspects of running time that one could focus on. Our focus will be primarily on the question: "how does the running time *scale* with the size of the input?" This is called *asymptotic analysis*, and the idea is that we will ignore low-order terms and constant factors, focusing instead on the shape of the running time curve. We will typically use n to denote the size of the input, and $T(n)$ to denote the running time of our algorithm on an input of size n.

We begin by presenting some convenient definitions for performing this kind of analysis.

Definition 2.1 *$T(n) \in O(f(n))$ if there exist constants $c, n_0 > 0$ such that $T(n) \leq cf(n)$ for all $n > n_0$.*

Informally we can view this as "$T(n)$ is proportional to $f(n)$, or better, as n gets large." For example, $3n^2 + 17 \in O(n^2)$ and $3n^2 + 17 \in O(n^3)$. This notation is especially useful in discussing upper bounds on algorithms: for instance, we saw last time that Karatsuba multiplication took time $O(n^{\log_2 3})$.

Notice that $O(f(n))$ is a set of functions. Nonetheless, it is common practice to write $T(n) = O(f(n))$ to mean that $T(n) \in O(f(n))$: especially in conversation, it is more natural to say "$T(n)$ is $O(f(n))$" than to say "$T(n)$ is in $O(f(n))$". We will typically use this common practice, reverting to the correct set notation when this practice would cause confusion.

Definition 2.2 *$T(n) \in \Omega(f(n))$ if there exist constants $c, n_0 > 0$ such that $T(n) \geq cf(n)$ for all $n > n_0$.*

Informally we can view this as "$T(n)$ is proportional to $f(n)$, or worse, as n gets large." For example, $3n^2 - 2n \in \Omega(n^2)$. This notation is especially useful for lower bounds. In Chapter 5, for instance, we will prove that any comparison-based sorting algorithm must take time $\Omega(n \log n)$ in the worst case (or even on average).

Definition 2.3 *$T(n) \in \Theta(f(n))$ if $T(n) \in O(f(n))$ and $T(n) \in \Omega(f(n))$.*

Informally we can view this as "$T(n)$ is proportional to $f(n)$ as n gets large."

Definition 2.4 *$T(n) \in o(f(n))$ if for all constants $c > 0$, there exists $n_0 > 0$ such that $T(n) < cf(n)$ for all $n > n_0$.*

For example, last time we saw that we could indeed multiply two n-bit numbers in time $o(n^2)$ by the Karatsuba algorithm. Very informally, O is like $\leq$, Ω is like $\geq$, Θ is like $=$, and o is like $<$. There is also a similar notation ω that corresponds to $>$.

In terms of computing whether or not $T(n)$ belongs to one of these sets with respect to $f(n)$, a convenient way is to compute the limit:

$$\lim_{n \to \infty} \frac{T(n)}{f(n)}. \tag{2.1}$$

If the limit exists, then we can make the following statements:

- If the limit is 0, then $T(n) = o(f(n))$ and $T(n) = O(f(n))$.
- If the limit is a number greater than 0 (e.g., 17) then $T(n) = \Theta(f(n))$ (and $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$)
- If the limit is infinity, then $T(n) = \omega(f(n))$ and $T(n) = \Omega(f(n))$.

For example, suppose $T(n) = 2n^3 + 100n^2 \log_2 n + 17$ and $f(n) = n^3$. The ratio of these is $2 + (100 \log_2 n)/n + 17/n^3$. In this limit, this goes to 2. Therefore, $T(n) = \Theta(f(n))$. Of course, it is possible that the limit doesn't exist — for instance if $T(n) = n(2 + \sin n)$ and $f(n) = n$ then the ratio oscillates between 1 and 3. In this case we would go back to the definitions to say that $T(n) = \Theta(n)$.

One convenient fact to know (which we just used in the paragraph above and you can prove by taking derivatives) is that for any constant k, $\lim_{n\to\infty}(\log n)^k/n = 0$. This implies, for instance, that $n\log n = o(n^{1.5})$ because $\lim_{n\to\infty}(n\log n)/n^{1.5} = \lim_{n\to\infty}(\log n)/\sqrt{n} = \lim_{n\to\infty}\sqrt{(\log n)^2/n} = 0$.

So, this notation gives us a language for talking about desired or achievable specifications. A typical use might be "we can prove that *any* algorithm for problem X must take $\Omega(n\log n)$ time in the worst case. My fancy algorithm takes time $O(n\log n)$. Therefore, my algorithm is asymptotically optimal."

2.3 Recurrences

We often are interested in algorithms expressed in a recursive way. When we analyze them, we get a recurrence: a description of the running time on an input of size n as a function of n *and* the running time on inputs of smaller sizes. Here are some examples:

Mergesort: To sort an array of size n, we sort the left half, sort right half, and then merge the two results. We can do the merge in linear time. So, if $T(n)$ denotes the running time on an input of size n, we end up with the recurrence $T(n) = 2T(n/2) + cn$.

Selection sort: In selection sort, we run through the array to find the smallest element. We put this in the leftmost position, and then recursively sort the remainder of the array. This gives us a recurrence $T(n) = cn + T(n-1)$.

Multiplication: Here we split each number into its left and right halves. We saw in the last lecture that the straightforward way to solve the subproblems gave us $T(n) = 4T(n/2) + cn$. However, rearranging terms in a clever way improved this to $T(n) = 3T(n/2) + cn$.

What about the base cases? In general, once the problem size gets down to a small constant, we can just use a brute force approach that takes some other constant amount of time. So, almost always we can say the base case is that $T(n) \leq c$ for all $n \leq n_0$, where n_0 is a constant we get to choose (like 17) and c is some other constant that depends on n_0.

What about the "integrality" issue? For instance, what if we want to use mergesort on an array with an odd number of elements — then the recurrence above is not technically correct. Luckily, this issue turns out almost never to matter, so we can ignore it. In the case of mergesort we can argue formally by using the fact that $T(n)$ is sandwiched between $T(n')$ and $T(n'')$ where n' is the next smaller power of 2 and n'' is the next larger power of 2, both of which differ by at most a constant factor from each other.

We now describe four methods for solving recurrences that are useful to know.

2.3.1 Solving by unrolling

Many times, the easiest way to solve a recurrence is to unroll it to get a summation. For example, unrolling the recurrence for selection sort gives us:

$$T(n) = cn + c(n-1) + c(n-2) + \ldots + c. \tag{2.2}$$

Since there are n terms and each one is at most cn, we can see that this summation is at most cn^2. Since the first $n/2$ terms are each at least $cn/2$, we can see that this summation is at least

$(n/2)(cn/2) = cn^2/4$. So, it is $\Theta(n^2)$. Similarly, a recurrence $T(n) = n^5 + T(n-1)$ unrolls to:

$$T(n) = n^5 + (n-1)^5 + (n-2)^5 + \ldots + 1^5, \tag{2.3}$$

which solves to $\Theta(n^6)$ using the same style of reasoning as before. Another convenient way to look at many summations of this form is to see them as an approximation to an integral. E.g., in this last case, the sum is at most the integral of $f(x) = x^5$ evaluated at $x = n + 1$.

2.3.2 Solving by guess and inductive proof

Another good way to solve recurrences is to make a guess and then prove the guess correct inductively. Or if we get into trouble proving our guess correct (e.g., because it was wrong), often this will give us clues as to a better guess. For example, say we have the recurrence

$$T(n) = 7T(n/7) + n, \tag{2.4}$$
$$T(1) = 1. \tag{2.5}$$

We might first try a solution of $T(n) \leq cn$, where $c \geq 1$ to handle the base case. We would then assume it holds true inductively for $n' < n$ and plug in to our recurrence (using $n' = n/7$) to get:

$$\begin{aligned} T(n) &\leq 7(cn/7) + n \\ &= cn + n \\ &= (c+1)n. \end{aligned}$$

Unfortunately, this isn't what we wanted: our multiplier "c" went up by 1 when n went up by a factor of 7. In other words, our multiplier is acting like $\log_7(n)$. So, let's make a new guess using a multiplier of this form — or $\log_7(7n)$ to get the base case of $n = 1$ right. So, we have a new guess of

$$T(n) \leq n \log_7(7n). \tag{2.6}$$

If we assume this holds true inductively for $n' < n$, then we get:

$$\begin{aligned} T(n) &\leq 7[(n/7)\log_7(n)] + n \\ &= n \log_7(n) + n \\ &= n \log_7(7n). \end{aligned} \tag{2.7}$$

So, we have verified our guess.

It is important in this type of proof to be careful. For instance, one could be lulled into thinking that our initial guess of cn was correct by reasoning "we assumed $T(n/7)$ was $\Theta(n/7)$ and got $T(n) = \Theta(n)$". The problem is that the constants changed (c turned into $c + 1$) so they really weren't constant after all!

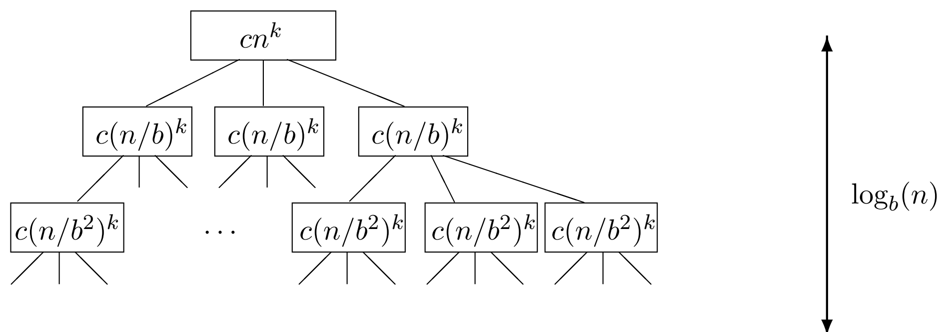
2.3.3 Recursion trees, stacking bricks, and a Master Formula

The final method we examine, which is especially good for divide-and-conquer style recurrences, is the use of a recursion tree. We will use this to method to produce a simple "master formula" that can be applied to many recurrences of this form.

Consider the following type of recurrence:

$$\begin{aligned} T(n) &= aT(n/b) + cn^k \\ T(1) &= c, \end{aligned} \tag{2.8}$$

for positive constants a, b, c, and k. This recurrence corresponds to the time spent by an algorithm that does cn^k work up front, and then divides the problem into a pieces of size n/b, solving each one recursively. For instance, mergesort, Karatsuba multiplication, and Strassen's algorithm all fit this mold. A *recursion tree* is just a tree that represents this process, where each node contains inside it the work done up front and then has one child for each recursive call. The leaves of the tree are the base cases of the recursion. A tree for the recurrence (2.8) is given below.[1]



To compute the result of the recurrence, we simply need to add up all the values in the tree. We can do this by adding them up level by level. The top level has value cn^k, the next level sums to $ca(n/b)^k$, the next level sums to $ca^2(n/b^2)^k$, and so on. The depth of the tree (the number of levels not including the root) is $\log_b(n)$. Therefore, we get a summation of:

$$cn^k \left[1 + a/b^k + (a/b^k)^2 + (a/b^k)^3 + ... + (a/b^k)^{\log_b n}\right] \tag{2.9}$$

To help us understand this, let's define $r = a/b^k$. Notice that r is a *constant*, since a, b, and k are constants. For instance, for Strassen's algorithm $r = 7/2^2$, and for mergesort $r = 2/2 = 1$. Using our definition of r, our summation simplifies to:

$$cn^k \left[1 + r + r^2 + r^3 + ... + r^{\log_b n}\right] \tag{2.10}$$

We can now evaluate three cases:

Case 1: $r < 1$. In this case, the sum is a convergent series. Even if we imagine the series going to infinity, we still get that the sum $1 + r + r^2 + \ldots = 1/(1 - r)$. So, we can upper-bound formula (2.9) by $cn^k/(1-r)$, and lower bound it by just the first term cn^k. Since r and c are constants, this solves to $\Theta(n^k)$.

Case 2: $r = 1$. In this case, all terms in the summation (2.9) are equal to 1, so the result is $cn^k(\log_b n + 1) \in \Theta(n^k \log n)$.

---

[1] This tree has branching factor a.

Case 3: $r > 1$. In this case, the last term of the summation dominates. We can see this by pulling it out, giving us:

$$cn^k r^{\log_b n}\left[(1/r)^{\log_b n} + \ldots + 1/r + 1\right] \tag{2.11}$$

Since $1/r < 1$, we can now use the same reasoning as in Case 1: the summation is at most $1/(1-1/r)$ which is a constant. Therefore, we have

$$T(n) \quad \in \quad \Theta\left(n^k (a/b^k)^{\log_b n}\right).$$

We can simplify this formula by noticing that $b^{k \log_b n} = n^k$, so we are left with

$$T(n) \quad \in \quad \Theta\left(a^{\log_b n}\right). \tag{2.12}$$

We can simplify this further by swapping the "a" and the "n" to get:

$$T(n) \quad \in \quad \Theta\left(n^{\log_b a}\right). \tag{2.13}$$

(Take $\log_b$ of (2.12) and (2.13) to convince yourself this is legal!!)

Note that Case 3 is what we used for Karatsuba multiplication ($a = 3, b = 2, k = 1$) and Strassen's algorithm ($a = 7, b = 2, k = 2$).

Combining the three cases above gives us the following "master theorem".

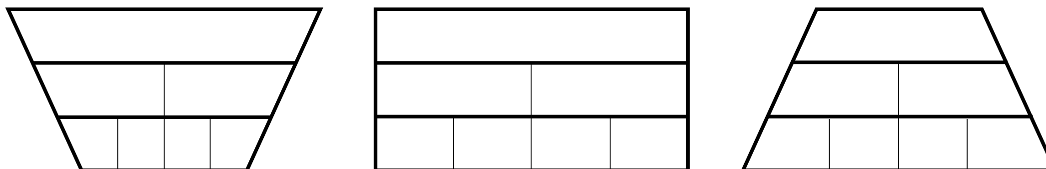
Theorem 2.1 *The recurrence*

$$\begin{aligned} T(n) &= aT(n/b) + cn^k \\ T(1) &= c, \end{aligned}$$

where a, b, c, and k are all constants, solves to:

$$\begin{aligned} T(n) &\in \Theta(n^k) \text{ if } a < b^k \\ T(n) &\in \Theta(n^k \log n) \text{ if } a = b^k \\ T(n) &\in \Theta(n^{\log_b a}) \text{ if } a > b^k \end{aligned}$$

A nice intuitive way to think of the computation above is to think of each node in the recursion tree as a brick of height 1 and width equal to the value inside it. Our goal is now to compute the area of the stack. Depending on whether we are in Case 1, 2, or 3, the picture then looks like one of the following:



In the first case, the area is dominated by the top brick; in the second case, all levels provide an equal contribution, and in the last case, the area is dominated by the bottom level.