2.1 Asymptotic Analysis

Throughout the course we will use $O(\cdot)$, $\Omega(\cdot)$, and $\Theta(\cdot)$ notation in order to "hide" constants. This is called *asymptotic* notation – you should have seen it in data structures (and possibly MFCS / discrete math), but we'll do a quick refresher to make sure that everyone is on the same page. In addition to making bounds simpler and easier to compare, asymptotic notation and analysis also forces us to focus on how algorithms scale. While for small inputs easy algorithms with bad bounds might be reasonable, at scale it is not the constants that matter, it is the asymptotics. This is particularly true now that we are in the "big data" era – when considering these huge problems, the constants are completely dominated by the asymptotics.

The following definition is the most basic and important:

Definition 2.1.1 *$g(n) \in O(f(n))$ if there exist constants $c, n_0 > 0$ such that $g(n) \leq c \cdot f(n)$ for all $n > n_0$.*

While technically $O(f(n))$ is a set (hence the $\in$ notation), we will usually say that "$g(n)$ is $O(f(n))$" or that $g(n) = O(f(n))$.

Examples:

- $2n^2 + 27 = O(n^2)$: set $n_0 = 6$ and $c = 3$
- $2n^2 + 27 = O(n^3)$: same values, or $n_0 = 4$ and $c = 1$
- $n^3 + 2000n^2 + 2000n = O(n^3)$: set $n_0 = 10000$ and $c = 2$

Let's see an example proof of the first of these examples.

Theorem 2.1.2 $2n^2 + 27 = O(n^2)$

Proof: Let's set $c = 3$. Now suppose that $2n^2 + 27 > cn^2 = 3n^2$. Then simple algebra implies that $n^2 < 27$, and thus $n < 6$. So $2n^2 + 27 \leq 3n^2$ for all $n \geq 6$. So if we set $n_0 = 6$, we have proved that $2n^2 + 27 \leq cn^2$ for all $n > n_0$. Thus we have proved that $2n^2 + 27 = O(n^2)$. ■

This notation is particularly useful for giving upper bounds on the running times of algorithms, but note that it's notation about *functions*, not about running times – we can also use it to give upper bounds on space usage, approximation ratio, etc. The important thing to remember is that it expresses an *upper* bound.

The natural complement is $\Omega(\cdot)$ notation:

Definition 2.1.3 *$g(n) \in \Omega(f(n))$ if there exist constants $c, n_0 > 0$ such that $g(n) \geq c \cdot f(n)$ for all $n > n_0$.*

This notation lets us provide a *lower* bound on functions as they grow. Examples:

- $2n^2 + 27 = \Omega(n^2)$: set $n_0 = 1$ and $c = 1$
- $2n^2 + 27 = \Omega(n)$: set $n_0 = 1$ and $c = 1$
- $\frac{1}{100}n^3 - 1000n^2 = \Omega(n^3)$: set $n_0 = 1000000$ and $c = 1/1000$

While most of this class is about upper bounds, we will occasionally discuss lower bounds as well so this is very useful notation to know.

The combination of these two is $\Theta(\cdot)$ notation:

Definition 2.1.4 *$g(n) \in \Theta(f(n))$ if $g(n) \in O(f(n))$ and $g(n) \in \Omega(f(n))$.*

Note that it is not necessary for the proof that $g(n) = O(f(n))$ to use the same constants n_0, c as the proof that $g(n) = \Omega(f(n))$!

Two more useful pieces of notation are the strict versions of O and Ω:

Definition 2.1.5 *$g(n) \in o(f(n))$ if for every constant $c > 0$ there exists a constant $n_0 > 0$ such that $g(n) < c \cdot f(n)$ for all $n > n_0$.*

Definition 2.1.6 *$g(n) \in \omega(f(n))$ if for every constant $c > 0$ there exists a constant $n_0 > 0$ such that $g(n) > c \cdot f(n)$ for all $n > n_0$.*

Examples:

- $2n^2 + 27 = o(n^2 \log n)$
- $2n^2 + 27 = \omega(n)$

2.2 Recurrence Relations

A number of the algorithms we will see are recursive, in which case it is very natural to express their running time through a recurrence relation. We saw this last lecture with Karatsuba's algorithm for multiplication and Strassen's algorithm for matrix multiplication. A more familiar set of examples might be from sorting:

- **Selection Sort.** Recall that selection sort works by finding the smallest currently unsorted element, putting it just after the set of sorted elements, and repeating. Since it takes $O(n)$ time to find the smallest element of an n-element array or list, this means that the running time is $T(n) = T(n-1) + cn$ for some constant c (where the $T(n-1)$ is because we recurse on the remaining unsorted elements).
- **Mergesort.** Recall that mergesort recursively sorts the left half of the array, then the right half, and then merges the two sorted halves. Clearly merging takes $O(n)$ time, so the overall running time is $T(n) = 2T(n/2) + cn$.

Technically we also need to have a base case. When n is constant it is almost always obvious (and is certainly true of all of our examples) that a trivial brute-force algorithm takes constant time. So

we will essentially always be able to say that $T(n) \leq c$ for all $n \leq n_0$, where n_0 is some constant and c is a constant which may depend on n_0.

You should have seen recurrence relations (and how to solve them) in MFCS / discrete math, but as with asymptotic notation, we'll give a quick refresher.

2.2.1 Guess-and-check (induction)

This works if you already have a good idea of the right answer, or are just exploring possibilities trying to get some intuition. Suppose we are given a recurrence relation like

$$\begin{aligned} T(n) &= 3T(n/3) + n \\ T(1) &= 1. \end{aligned}$$

Suppose that we're only trying to prove an upper bound on $T(n)$. Maybe our first guess is that $T(n) \leq cn$. In order to check this (inductively), we would assume that it's true for $n' < n$, and try to prove that it stays true for n. More formally, we get that

$$T(n) = 3T(n/3) + n \leq 3cn/3 + n = (c+1)n$$

This is not enough! You might think that it's enough since we started with $T(n') = O(n)$ and ended with $T(n) = O(n)$, but the constant changed! So it wasn't a constant after all. This is one of the major examples of a common phenomenon: sometimes constants matter and sometimes they don't, and it's important to recognize the difference. A constant changing in a proof by induction matters, since it means that it wasn't a constant to begin with.

So it's not true that $T(n) \leq cn$. What would be a better guess? Note that our "constant" went up by 1 when n went up by a factor of 3. What function has this behavior? Answer: $\log_3 n$. So let's try to prove inductively that $T(n) \leq n \log_3 n$. Except that in order to make this work with the base case, we'll try for $T(n) \leq n \log_3(3n)$

$$\begin{aligned} T(n) &= 3T(n/3) + n \leq 3(n/3)\log_3(n) + n = n\log_3(n) + n \\ &= n(\log_3(n) + \log_3 3) = n\log_3(3n). \end{aligned}$$

2.2.2 Unrolling

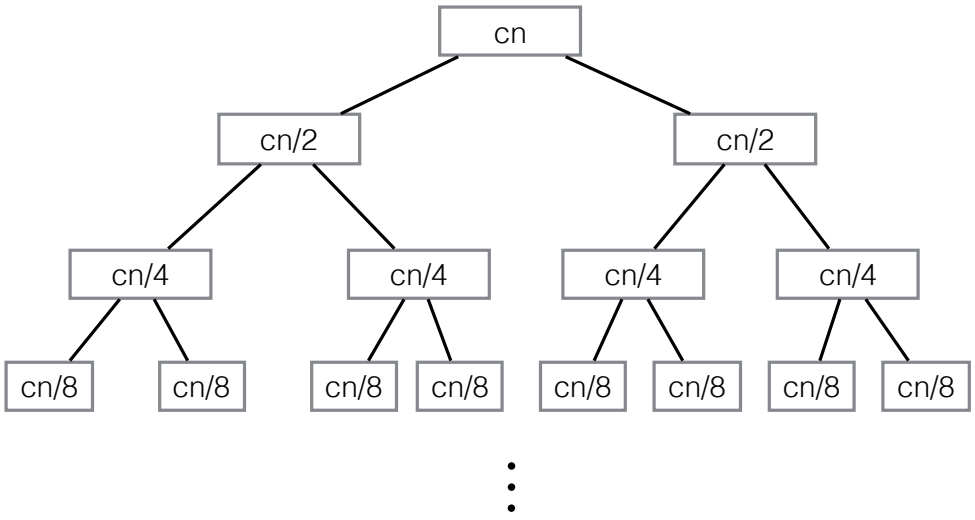
Guess-and-check is a reasonable approach for some situations, but can also be a bit tricky if you don't have good intuition to start with. For simple recurrence relations, a good place to start is "unrolling". This is particularly useful on recurrence relations like selection sort ($T(n) = T(n-1) + cn$) where there is only a single recursive call. Like the name suggests, unrolling simply involves writing out the recurrence. In the case of selection sort, for example, we get

$$T(n) = cn + c(n-1) + c(n-2) + \cdots + c$$

There are n terms each of which is at most cn, so $T(n) = O(n^2)$. Conversely, there are at least $n/2$ terms which are each at least $cn/2$, so $T(n) \geq cn^2/4 = \Omega(n^2)$. Thus $T(n) = \Theta(n^2)$.

2.2.3 Recursion Tree

This is the technique I always use, and is essentially a generalization of unrolling. It consists of drawing out the recursion tree, and analyzing it level by level. Let's do two examples. First, let's analyze the mergesort recursion $T(n) = 2T(n/2) + cn$. Let's draw the tree. Each box represent the amount of non-recursive work done in one recursive call, i.e., the cn part of relation for the appropriate n. Then we have a child for each recursive call.

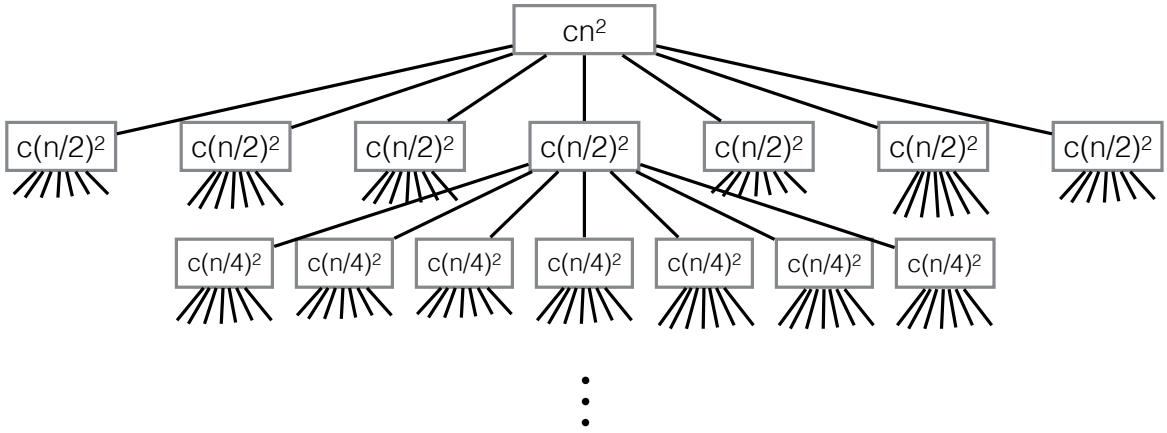


The total value of $T(n)$ is the sum of the value of every node in the tree. We can analyze it level by level. The first level clearly contributed cn. The second level contributes $2c(n/2) = cn$. The third level contributes $4c(n/4) = cn$. In general, level i contributes $2^{i-1}c(n/2^{i-1}) = cn$. This bottoms out when $i = \log n + 1$, since then we have $T(1) = c$. Since there are $\log n + 1$ levels, the total value of $T(n)$ is $cn(\log n + 1) = \Theta(n \log n)$.

Now let's analyze the recurrence that we got last time for Strassen's algorithm:

$$T(n) = 7T(n/2) + cn^2.$$

As before, we first draw the tree.



Now the value of the first level is cn^2, the value of the second level is $7c(n/2)^2 = 7cn^2/4$, the value of the third level is $7^2c(n/2^2)^2$, etc. So the value of level i is $7^{i-1}c(n/2^{i-1})^2 = (7/4)^{i-1}cn^2$. When we sum over all $\log n + 1$ levels, we get a total cost of

$$T(n) = \sum_{i=1}^{\log n+1} \left(\frac{7}{4}\right)^{i-1} cn^2 = cn^2 \sum_{i=1}^{\log n+1} \left(\frac{7}{4}\right)^{i-1}$$

This is dominated by the last term (if you don't see why, try pulling out a factor of $(7/4)^{\log n-1}$ and testing whether the remaining sequence converges). Thus overall we get that

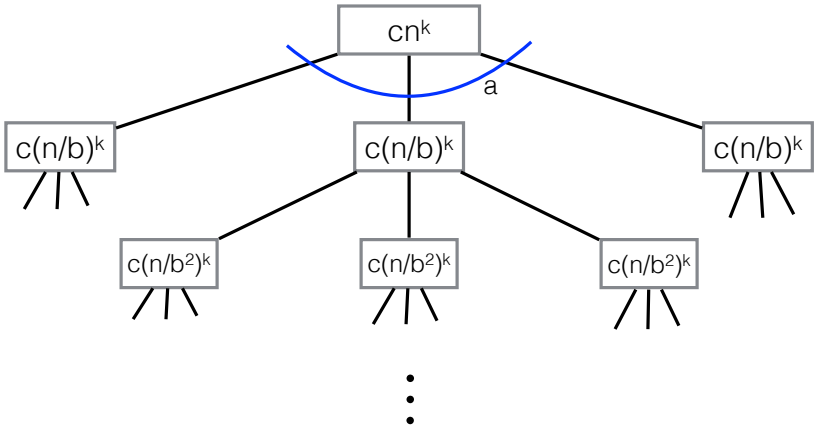
$$T(n) = O(n^2(7/4)^{\log n}) = O(n^2 n^{\log(7/4)}) = O(n^2 n^{\log 7-2}) = O(n^{\log 7})$$

2.2.4 Master theorem

Let's use the recursion tree to prove a more general theorem. Suppose we have a recursion of the form

$$T(n) = aT(n/b) + cn^k$$
$$T(1) = c$$

where a, b, c, and k are all constants with $a \geq 1$, $b > 1$, $c > 0$, and $k \geq 0$. When we draw the recursion tree, we get the following:



The first level has value cn^k, the second level has value $ac(n/b)^k$, the third level has value $a^2c(n/b^2)^k$, etc. So level i has value $cn^k(a/b^k)^{i-1}$. Clearly the total number of levels is $\log_b n + 1$.

To simplify notation, let's let $\alpha = a/b^k$. With this notation, we get that

$$T(n) = cn^k \left(1 + \alpha + \alpha^2 + \cdots + \alpha^{\log_b n}\right)$$

We now have three cases, depending on the value of α.

- **Case 1:** $\alpha = 1$**.** In this case the summation is equal to $\log_b n$, so $T(n) = \Theta(n^k \log n)$
- **Case 2:** $\alpha < 1$**.** In this case, the infinite summation $\sum_{i=0}^{\infty} \alpha^i$ is convergent, and moreover $\sum_{i=0}^{\infty} \alpha^i = 1/(1-\alpha)$. Thus $T(n) < cn^k(1/(1-\alpha))$. On the other hand, just from the first term we know that $T(n) > cn^k$. Since α is a constant, these imply that $T(n) = \Theta(n^k)$. Less formally, in this case the top level dominates the tree.
- **Case 3:** $\alpha > 1$**.** This is the trickiest case, but it isn't too bad. Looking at the summation, we can pull out the largest factor and then end up with a convergent sequence, just like in the second case:

$$\begin{aligned}\left(1 + \alpha + \alpha^2 + \cdots + \alpha^{\log_b n}\right) &= \alpha^{\log_b n}\left(1 + 1/\alpha + 1/\alpha^2 + \cdots + 1/\alpha^{\log_b n}\right) \\ &\leq \alpha^{\log_b n}\frac{1}{1-(1/\alpha)} = O(\alpha^{\log_b n})\end{aligned}$$

  In other words, the lowest level dominates in this case. So overall we get that

$$T(n) = \Theta(n^k \alpha^{\log_b n}) = \Theta(n^k (a/b^k)^{\log_b n})$$

  Now note that $b^{k \log_b n} = n^k$, and thus

$$T(n) = \Theta(a^{\log_b n}) = \Theta(n^{\log_b a}).$$

We can put these together to get the following theorem, which is sometimes called the "master theorem".

Theorem 2.2.1 *The recurrence*

$$T(n) = aT(n/b) + cn^k$$
$$T(1) = c$$

where a, b, c, and k are constants with $a \geq 1$, $b > 1$, $c > 0$, and $k \geq 0$, is equal to

$$T(n) = \Theta(n^k) \text{ if } a < b^k,$$
$$T(n) = \Theta(n^k \log n) \text{ if } a = b^k,$$
$$T(n) = \Theta(n^{\log_b a}) \text{ if } a > b^k.$$