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: As we saw in a previous lecture, 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 $T(n) = cn + T(n-1)$.

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.

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:

$$T(n) = cn + c(n-1) + c(n-2) + \ldots + c.$$  \hspace{1cm} (1)
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:

$$T(n) = n^5 + (n-1)^5 + (n-2)^5 + \ldots + 1^5,$$

which solves to $\Theta(n^6)$ using the same style of reasoning as before. In particular, there are $n$ terms each of which is at most $n^5$ so the sum is at most $n^6$, and the top $n/2$ terms are each at least $(n/2)^5$ so the sum is at least $(n/2)^6$. Another convenient way to look at many summations of this form is to see them as approximations to an integral. E.g., in this last case, the sum is at least the integral of $f(x) = x^5$ evaluated from 0 to $n$, and at most the integral of $f(x) = x^5$ evaluated from 1 to $n+1$. So, the sum lies in the range $[\frac{1}{6}n^6, \frac{1}{6}(n+1)^6]$.

**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

$$T(n) = 7T(n/7) + n,$$  

$$T(1) = 0.$$  

We might first try a solution of $T(n) \leq cn$ for some $c > 0$. We would then assume it holds true inductively for $n' < n$ (the base case is obviously true) and plug in to our recurrence (using $n' = n/7$) to get:

$$T(n) \leq 7(cn/7) + n = cn + n = (c + 1)n.$$

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. So, we have a new guess of

$$T(n) \leq n \log_7(n).$$

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

$$T(n) \leq 7[(n/7) \log_7(n/7)] + n = n \log_7(n/7) + n = n \log_7(n) - n + n = n \log_7(n).$$
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!

**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:

\[
T(n) = aT(n/b) + cn^k \\
T(1) = c,
\]

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 (7) is given below.\(^2\)

![Recursion Tree Diagram](image)

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:

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

---

\(^2\)This tree has branching factor \( a \).
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:

\[
 cn^k \left[ 1 + r + r^2 + r^3 + \ldots + r^{\log_b n} \right]
\]

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 (8) 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 (8) are equal to 1, so the result is \( cn^k(\log_b n + 1) \in \Theta(n^k \log n) \).

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

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

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

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

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

\[
 T(n) \in \Theta \left( a^{\log_b n} \right).
\]

We can simplify this further using \( a^{\log_b n} = b^{(\log_b a)(\log_b n)} = n^{\log_b a} \) to get:

\[
 T(n) \in \Theta \left( n^{\log_b a} \right).
\]

Note that Case 3 is what we will use 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 1** The recurrence

\[
 T(n) = aT(n/b) + cn^k
\]

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

\[
 T(n) \in \Theta(n^k) \text{ if } a < b^k
\]

\[
 T(n) \in \Theta(n^k \log n) \text{ if } a = b^k
\]

\[
 T(n) \in \Theta(n^{\log_b a}) \text{ if } a > b^k
\]
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.