We have this messy summation to solve though. First observe that the value n remains constant throughout the sum, and so we can pull it out front. Also note that we can write $3^i/4^i$ and $(3/4)^i$.

$$T(n) = n^{\log_4 3} + n \sum_{i=0}^{(\log_4 n)-1} \left(\frac{3}{4}\right)^i .$$

Note that this is a geometric series. We may apply the formula for the geometric series, which gave in an earlier lecture. For $x \neq 1$:

$$\sum_{i=0}^{m} x^i = \frac{x^{m+1}-1}{x-1}.$$

In this case $x = 3/4$ and $m = \log_4 n - 1$. We get

$$T(n) = n^{\log_4 3} + n \frac{(3/4)^{\log_4 n} - 1}{(3/4) - 1}.$$

Applying our favorite log identity once more to the expression in the numerator (with $a = 3/4$ and $b = 4$) we get

$$(3/4)^{\log_4 n} \;=\; n^{\log_4 (3/4)} \;=\; n^{(\log_4 3 - \log_4 4)} \;=\; n^{(\log_4 3 - 1)} \;=\; \frac{n^{\log_4 3}}{n}.$$

If we plug this back in, we have

$$\begin{aligned}
T(n) &= n^{\log_4 3} + n \frac{\frac{n^{\log_4 3}}{n} - 1}{(3/4) - 1} \\
&= n^{\log_4 3} + \frac{n^{\log_4 3} - n}{-1/4} \\
&= n^{\log_4 3} - 4(n^{\log_4 3} - n) \\
&= n^{\log_4 3} + 4(n - n^{\log_4 3}) \\
&= 4n - 3n^{\log_4 3}.
\end{aligned}$$

So the final result (at last!) is

$$T(n) = 4n - 3n^{\log_4 3} \approx 4n - 3n^{0.79} \in \Theta(n).$$

It is interesting to note the unusual exponent of $\log_4 3 \approx 0.79$. We have seen that two nested loops typically leads to $\Theta(n^2)$ time, and three nested loops typically leads to $\Theta(n^3)$ time, so it seems remarkable that we could generate a strange exponent like 0.79 as part of a running time. However, as we shall see, this is often the case in divide-and-conquer recurrences.

Lecture 8: More on Recurrences

(Thursday, Feb 19, 1998)

Read: Chapt. 4 on recurrences, skip Section 4.4.

Recap: Last time we discussed recurrences, that is, functions that are defined recursively. We discussed their importance in analyzing divide-and-conquer algorithms. We also discussed two methods for solving recurrences, namely guess-and-verify (by induction), and iteration. These are both very powerful methods, but they are quite “mechanical”, and it is difficult to get a quick and intuitive sense of what is going on in the recurrence. Today we will discuss two more techniques for solving recurrences. The first provides a way of visualizing recurrences and the second, called the Master Theorem, is a method of solving many recurrences that arise in divide-and-conquer applications.

Visualizing Recurrences Using the Recursion Tree: Iteration is a very powerful technique for solving recurrences. But, it is easy to get lost in all the symbolic manipulations and lose sight of what is going on. Here is a nice way to visualize what is going on in iteration. We can describe any recurrence in terms of a tree, where each expansion of the recurrence takes us one level deeper in the tree.

Recall that the recurrence for MergeSort (which we simplified by assuming that n is a power of 2, and hence could drop the floors and ceilings)

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2T(n/2) + n & \text{otherwise.} \end{cases}$$

Suppose that we draw the recursion tree for MergeSort, but each time we merge two lists, we label that node of the tree with the time it takes to perform the associated (nonrecursive) merge. Recall that to merge two lists of size $m/2$ to a list of size m takes $\Theta(m)$ time, which we will just write as m. Below is an illustration of the resulting recursion tree.

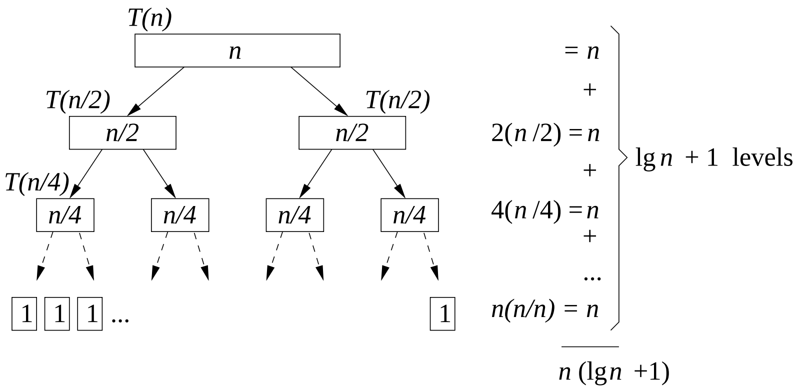

Figure 5: Using the recursion tree to visualize a recurrence.

Observe that the total work at the topmost level of the recursion is $\Theta(n)$ (or just n for short). At the second level we have two merges, each taking $n/2$ time, for a total of $2(n/2) = n$. At the third level we have 4 merges, each taking $n/4$ time, for a total of $4(n/4) = n$. This continues until the bottommost level of the tree. Since the tree exactly $\lg n + 1$ levels $(0, 1, 2, \ldots, \lg n)$, and each level contributes a total of n time, the total running time is $n(\lg n + 1) = n \lg n + n$. This is exactly what we got by the iteration method.

This can be used for a number of simple recurrences. For example, let's try it on the following recurrence. The tree is illustrated below.

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 3T(n/2) + n^2 & \text{otherwise.} \end{cases}$$

Again, we label each node with the amount of work at that level. In this case the work for $T(m)$ is m^2. For the top level (or 0th level) the work is n^2. At level 1 we have three nodes whose work is $(n/2)^2$ each, for a total of $3(n/2)^2$. This can be written as $n^2(3/4)$. At the level 2 the work is $9(n/4)^2$, which can be written as $n^2(9/16)$. In general it is easy to extrapolate to see that at the level i, we have 3^i nodes, each involving $(n/2^i)^2$ work, for a total of $3^i(n/2^i)^2 = n^2(3/4)^i$.

This leads to the following summation. Note that we have not determined where the tree bottoms out, so we have left off the upper bound on the sum.

$$T(n) = n^2 \sum_{i=0}^{?} \left(\frac{3}{4}\right)^i .$$

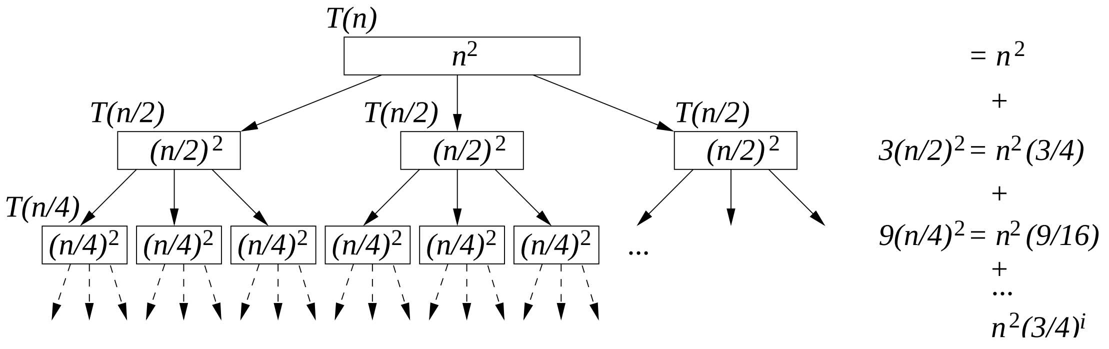

Figure 6: Another recursion tree example.

If all we wanted was an asymptotic expression, then are essentially done at this point. Why? The summation is a geometric series, and the base $(3/4)$ is less than 1. This means that this series converges to some nonzero constant (even if we ran the sum out to ∞). Thus the running time is $\Theta(n^2)$.

To get a more exact result, observe that the recursion bottoms out when we get down to single items, and since the sizes of the inputs are cut by half at each level, it is not hard to see that the final level is level $\lg n$. (It is easy to be off by ± 1 here, but this sort of small error will not affect the asymptotic result. In this case we happen to be right.) So, we can plug in $\lg n$ for the "?" in the above summation.

$$T(n) = n^2 \sum_{i=0}^{\lg n} \left(\frac{3}{4}\right)^i .$$

If we wanted to get a more exact answer, we could plug the summation into the formula for the geometric series and simplify. This would lead to an expression like

$$T(n) = n^2 \frac{(3/4)^{\lg n+1} - 1}{(3/4) - 1}.$$

This will take some work to simplify, but at this point it is all just tedious algebra to get the formula into simple form. (This sort of algebraic is typical of algorithm analysis, so be sure that you follow each step.)

$$\begin{aligned}
T(n) &= n^2 \frac{(3/4)^{\lg n+1} - 1}{(3/4) - 1} \;=\; -4n^2((3/4)^{\lg n+1} - 1) \\
&= 4n^2(1 - (3/4)^{\lg n+1}) \;=\; 4n^2(1 - (3/4)(3/4)^{\lg n}) \\
&= 4n^2(1 - (3/4)n^{\lg(3/4)}) \;=\; 4n^2(1 - (3/4)n^{\lg 3 - \lg 4}) \\
&= 4n^2(1 - (3/4)n^{\lg 3 - 2}) \;=\; 4n^2(1 - (3/4)(n^{\lg 3}/n^2)) \\
&= 4n^2 - 3n^{\lg 3}.
\end{aligned}$$

Note that $\lg 3 \approx 1.58$, so the whole expression is $\Theta(n^2)$.

In conclusion, the technique of drawing the recursion tree is a somewhat more visual way of analyzing summations, but it is really equivalent to the method of iteration.

(Simplified) Master Theorem: If you analyze many divide-and-conquer algorithms, you will see that the same general type of recurrence keeps popping up. In general you are breaking a problem into a subproblems, where each subproblem is roughly a factor of $1/b$ of the original problem size, and the time it takes to do the splitting and combining on an input of size n is $\Theta(n^k)$. For example, in MergeSort, $a = 2$, $b = 2$, and $k = 1$.

Rather than doing every such recurrence from scratch, can we just come up with a general solution? The answer is that you can if all you need is an asymptotic expression. This result is called the *Master Theorem*, because it can be used to "master" so many different types of recurrence. Our text gives a fairly complicated version of this theorem. We will give a simpler version, which is general enough for most typical situations. In cases where this doesn't apply, try the one from the book. If the one from the book doesn't apply, then you will probably need iteration, or some other technique.

Theorem: (Simplified Master Theorem) Let $a \geq 1$, $b > 1$ be constants and let $T(n)$ be the recurrence

$$T(n) = aT(n/b) + n^k,$$

defined for $n \geq 0$. (As usual let us assume that n is a power of b. The basis case, $T(1)$ can be any constant value.) Then

Case 1: if $a > b^k$ then $T(n) \in \Theta(n^{\log_b a})$.
Case 2: if $a = b^k$ then $T(n) \in \Theta(n^k \log n)$.
Case 3: if $a < b^k$ then $T(n) \in \Theta(n^k)$.

Using this version of the Master Theorem we can see that in the MergeSort recurrence $a = 2$, $b = 2$, and $k = 1$. Thus, $a = b^k$ $(2 = 2^1)$ and so Case 2 applies. From this we have $T(n) \in \Theta(n \log n)$.

In the recurrence above, $T(n) = 3T(n/2) + n^2$, we have $a = 3$, $b = 2$ and $k = 2$. We have $a < b^k$ $(3 < 2^2)$ in this case, and so Case 3 applies. From this we have $T(n) \in \Theta(n^2)$.

Finally, consider the recurrence $T(n) = 4T(n/3) + n$, in which we have $a = 4$, $b = 3$ and $k = 1$. In this case we have $a > b^k$ $(4 > 3^1)$, and so Case 1 applies. From this we have $T(n) \in \Theta(n^{\log_3 4}) \approx \Theta(n^{1.26})$. This may seem to be a rather strange running time (a non-integer exponent), but this not uncommon for many divide-and-conquer solutions.

There are many recurrences that cannot be put into this form. For example, if the splitting and combining steps involve sorting, we might have seen a recurrence of the form

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2T(n/2) + n \log n & \text{otherwise.} \end{cases}$$

This solves to $T(n) = \Theta(n \log^2 n)$, but the Master Theorem (neither this form nor the one in CLR) will tell you this. However, iteration works just fine here.

Recursion Trees Revisited: The recursion trees offer some intuition about why it is that there are three cases in the Master Theorem. Generally speaking the question is where is most of the work done: at the top of the tree (the root level), at the bottom of the tree (the leaf level), or spread equally throughout the entire tree.

For example, in the MergeSort recurrence (which corresponds to Case 2 in the Master Theorem) every level of the recursion tree provides the same total work, namely n. For this reason the total work is equal to this value times the height of the tree, namely $\Theta(\log n)$, for a total of $\Theta(n \log n)$.

Next consider the earlier recurrence $T(n) = 3T(n/2) + n^2$ (which corresponds to Case 3 in the Master Theorem). In this instance most of the work was concentrated at the root of the tree. Each level of the tree provided a smaller fraction of work. By the nature of the geometric series, it did not matter how many levels the tree had at all. Even with an infinite number of levels, the geometric series that result will converge to a constant value. This is an important observation to make. A common way to design the most efficient divide-and-conquer algorithms is to try to arrange the recursion so that most of the work is done at the root, and at each successive level of the tree the work at this level reduces (by some constant factor). As long as this is the case, Case 3 will apply.

Finally, in the recurrence $T(n) = 4T(n/3) + n$ (which corresponds to Case 1), most of the work is done at the leaf level of the recursion tree. This can be seen if you perform iteration on this recurrence, the resulting summation is

$$n \sum_{i=0}^{\log_3 n} \left(\frac{4}{3}\right)^i.$$

(You might try this to see if you get the same result.) Since $4/3 > 1$, as we go deeper into the levels of the tree, that is deeper into the summation, the terms are growing successively larger. The largest contribution will be from the leaf level.

Lecture 9: Medians and Selection

(Tuesday, Feb 24, 1998)

Read: Todays material is covered in Sections 10.2 and 10.3. You are not responsible for the randomized analysis of Section 10.2. Our presentation of the partitioning algorithm and analysis are somewhat different from the ones in the book.

Selection: In the last couple of lectures we have discussed recurrences and the divide-and-conquer method of solving problems. Today we will give a rather surprising (and very tricky) algorithm which shows the power of these techniques.

The problem that we will consider is very easy to state, but surprisingly difficult to solve optimally. Suppose that you are given a set of n numbers. Define the *rank* of an element to be one plus the number of elements that are smaller than this element. Since duplicate elements make our life more complex (by creating multiple elements of the same rank), we will make the simplifying assumption that all the elements are distinct for now. It will be easy to get around this assumption later. Thus, the rank of an element is its final position if the set is sorted. The minimum is of rank 1 and the maximum is of rank n.

Of particular interest in statistics is the *median*. If n is odd then the median is defined to be the element of rank $(n + 1)/2$. When n is even there are two natural choices, namely the elements of ranks $n/2$ and $(n/2) + 1$. In statistics it is common to return the average of these two elements. We will define the median to be either of these elements.

Medians are useful as measures of the *central tendency* of a set, especially when the distribution of values is highly skewed. For example, the median income in a community is likely to be more meaningful measure of the central tendency than the average is, since if Bill Gates lives in your community then his gigantic income may significantly bias the average, whereas it cannot have a significant influence on the median. They are also useful, since in divide-and-conquer applications, it is often desirable to partition a set about its median value, into two sets of roughly equal size. Today we will focus on the following generalization, called the *selection problem*.

Selection: Given a set A of n distinct numbers and an integer k, $1 \le k \le n$, output the element of A of rank k.

The selection problem can easily be solved in $\Theta(n \log n)$ time, simply by sorting the numbers of A, and then returning $A[k]$. The question is whether it is possible to do better. In particular, is it possible to solve this problem in $\Theta(n)$ time? We will see that the answer is yes, and the solution is far from obvious.

The Sieve Technique: The reason for introducing this algorithm is that it illustrates a very important special case of divide-and-conquer, which I call the *sieve technique*. We think of divide-and-conquer as breaking the problem into a small number of smaller subproblems, which are then solved recursively. The sieve technique is a special case, where the number of subproblems is just 1.