Analysis: What remains is to analyze the running time of MergeSort. First let us consider the running time of the procedure `Merge(A, p, q, r)`. Let $n = r - p + 1$ denote the total length of both the left and right subarrays. What is the running time of Merge as a function of n? The algorithm contains four loops (none nested in the other). It is easy to see that each loop can be executed at most n times. (If you are a bit more careful you can actually see that all the while-loops together can only be executed n times in total, because each execution copies one new element to the array B, and B only has space for n elements.) Thus the running time to Merge n items is $\Theta(n)$. Let us write this without the asymptotic notation, simply as n. (We'll see later why we do this.)

Now, how do we describe the running time of the entire MergeSort algorithm? We will do this through the use of a *recurrence*, that is, a function that is defined recursively in terms of itself. To avoid circularity, the recurrence for a given value of n is defined in terms of values that are strictly smaller than n. Finally, a recurrence has some basis values (e.g. for $n = 1$), which are defined explicitly.

Let's see how to apply this to MergeSort. Let $T(n)$ denote the worst case running time of MergeSort on an array of length n. For concreteness we could count whatever we like: number of lines of pseudocode, number of comparisons, number of array accesses, since these will only differ by a constant factor. Since all of the real work is done in the Merge procedure, we will count the total time spent in the Merge procedure.

First observe that if we call MergeSort with a list containing a single element, then the running time is a constant. Since we are ignoring constant factors, we can just write $T(n) = 1$. When we call MergeSort with a list of length $n > 1$, e.g. `Merge(A, p, r)`, where $r-p+1 = n$, the algorithm first computes $q = \lfloor (p+r)/2 \rfloor$. The subarray $A[p..q]$, which contains $q - p + 1$ elements. You can verify (by some tedious floor-ceiling arithmetic, or simpler by just trying an odd example and an even example) that is of size $\lceil n/2 \rceil$. Thus the remaining subarray $A[q+1..r]$ has $\lfloor n/2 \rfloor$ elements in it. How long does it take to sort the left subarray? We do not know this, but because $\lceil n/2 \rceil < n$ for $n > 1$, we can express this as $T(\lceil n/2 \rceil)$. Similarly, we can express the time that it takes to sort the right subarray as $T(\lfloor n/2 \rfloor)$. Finally, to merge both sorted lists takes n time, by the comments made above. In conclusion we have

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

Lecture 7: Recurrences

(Tuesday, Feb 17, 1998)

Read: Chapt. 4 on recurrences. Skip Section 4.4.

Divide and Conquer and Recurrences: Last time we introduced divide-and-conquer as a basic technique for designing efficient algorithms. Recall that the basic steps in divide-and-conquer solution are (1) divide the problem into a small number of subproblems, (2) solve each subproblem recursively, and (3) combine the solutions to the subproblems to a global solution. We also described MergeSort, a sorting algorithm based on divide-and-conquer.

Because divide-and-conquer is an important design technique, and because it naturally gives rise to recursive algorithms, it is important to develop mathematical techniques for solving recurrences, either exactly or asymptotically. To do this, we introduced the notion of a *recurrence*, that is, a recursively defined function. Today we discuss a number of techniques for solving recurrences.

MergeSort Recurrence: Here is the recurrence we derived last time for MergeSort. Recall that $T(n)$ is the time to run MergeSort on a list of size n. We argued that if the list is of length 1, then the total sorting time is a constant $\Theta(1)$. If $n > 1$, then we must recursively sort two sublists, one of size $\lceil n/2 \rceil$ and the other of size $\lfloor n/2 \rfloor$, and the nonrecursive part took $\Theta(n)$ time for splitting the list (constant time) and merging the lists ($\Theta(n)$ time). Thus, the total running time for MergeSort could be described by the following recurrence:

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

Notice that we have dropped the $\Theta()$'s, replacing $\Theta(1)$ and $\Theta(n)$ by just 1 and n, respectively. This is done to make the recurrence more concrete. If we had wanted to be more precise, we could have replaced these with more exact functions, e.g., c_1 and $c_2 n$ for some constants c_1 and c_2. The analysis would have been a bit more complex, but we would arrive at the same asymptotic formula.

Getting a feel: We could try to get a feeling for what this means by plugging in some values and expanding the definition.

$$\begin{array}{rcll} T(1) & = & 1 & \text{(by the basis.)} \\ T(2) & = & T(1) + T(1) + 2 = 1 + 1 + 2 = 4 & \\ T(3) & = & T(2) + T(1) + 3 = 4 + 1 + 3 = 8 & \\ T(4) & = & T(2) + T(2) + 4 = 4 + 4 + 4 = 12 & \\ T(5) & = & T(3) + T(2) + 5 = 8 + 4 + 5 = 17 & \\ \ldots & & & \\ T(8) & = & T(4) + T(4) + 8 = 12 + 12 + 8 = 32 & \\ \ldots & & & \\ T(16) & = & T(8) + T(8) + 16 = 32 + 32 + 16 = 80 & \\ \ldots & & & \\ T(32) & = & T(16) + T(16) + 32 = 80 + 80 + 32 = 192. & \end{array}$$

It's hard to see much of a pattern here, but here is a trick. Since the recurrence divides by 2 each time, let's consider powers of 2, since the function will behave most regularly for these values. If we consider the ratios $T(n)/n$ for powers of 2 and interesting pattern emerges:

$$\begin{array}{rclcrcl} T(1)/1 & = & 1 & \qquad & T(8)/8 & = & 4 \\ T(2)/2 & = & 2 & & T(16)/16 & = & 5 \\ T(4)/4 & = & 3 & & T(32)/32 & = & 6. \end{array}$$

This suggests that for powers of 2, $T(n)/n = (\lg n) + 1$, or equivalently, $T(n) = (n \lg n) + n$ which is $\Theta(n \log n)$. This is not a proof, but at least it provides us with a starting point.

Logarithms in Θ-notation: Notice that I have broken away from my usual convention of say $\lg n$ and just said $\log n$ inside the $\Theta()$. The reason is that the base really does not matter when it is inside the Θ. Recall the change of base formula:

$$\log_b n = \frac{\log_a n}{\log_a b}.$$

If a and b are constants the $\log_a b$ is a constant. Consequently $\log_b n$ and $\log_a n$ differ only by a constant factor. Thus, inside the $\Theta()$ we do not need to differentiate between them. Henceforth, I will not be fussy about the bases of logarithms if asymptotic results are sufficient.

Eliminating Floors and Ceilings: One of the nasty things about recurrences is that floors and ceilings are a pain to deal with. So whenever it is reasonable to do so, we will just forget about them, and make whatever simplifying assumptions we like about n to make things work out. For this case, we will make the simplifying assumption that n is a power of 2. Notice that this means that our analysis will only be correct for a very limited (but infinitely large) set of values of n, but it turns out that as long as the algorithm doesn't act significantly different for powers of 2 versus other numbers, the asymptotic analysis will hold for all n as well. So let us restate our recurrence under this assumption:

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

Verification through Induction: We have just generated a guess for the solution to our recurrence. Let's see if we can verify its correctness formally. The proof will be by strong induction on n. Because n is limited to powers of 2, we cannot do the usual n to $n+1$ proof (because if n is a power of 2, $n+1$ will generally not be a power of 2). Instead we use strong induction.

Claim: For all $n \geq 1$, n a power of 2, $T(n) = (n \lg n) + n$.

Proof: (By strong induction on n.)

Basis case: ($n = 1$) In this case $T(1) = 1$ by definition and the formula gives $1 \lg 1 + 1 = 1$, which matches.

Induction step: Let $n > 1$, and assume that the formula $T(n') = (n' \lg n') + n'$, holds whenever $n' < n$. We want to prove the formula holds for n itself. To do this, we need to express $T(n)$ in terms of smaller values. To do this, we apply the definition:

$$T(n) = 2T(n/2) + n.$$

Now, $n/2 < n$, so we can apply the induction hypothesis here, yielding $T(n/2) = (n/2)\lg(n/2) + (n/2)$. Plugging this in gives

$$\begin{aligned} T(n) &= 2((n/2)\lg(n/2) + (n/2)) + n \\ &= (n \lg(n/2) + n) + n \\ &= n(\lg n - \lg 2) + 2n \\ &= (n \lg n - n) + 2n \\ &= n \lg n + n, \end{aligned}$$

which is exactly what we want to prove.

The Iteration Method: The above method of "guessing" a solution and verifying through induction works fine as long as your recurrence is simple enough that you can come up with a good guess. But if the recurrence is at all messy, there may not be a simple formula. The following method is quite powerful. When it works, it allows you to convert a recurrence into a summation. By in large, summations are easier to solve than recurrences (and if nothing else, you can usually approximate them by integrals).

The method is called *iteration*. Let's start expanding out the definition until we see a pattern developing. We first write out the definition $T(n) = 2T(n/2) + n$. This has a recursive formula inside $T(n/2)$ which we can expand, by filling in the definition but this time with the argument $n/2$ rather than n. Plugging in we get $T(n) = 2(2T(n/4) + n/2) + n$. We then simplify and repeat. Here is what we get when we repeat this.

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2(2T(n/4) + n/2) + n \;=\; 4T(n/4) + n + n \\ &= 4(2T(n/8) + n/4) + n + n \;=\; 8T(n/8) + n + n + n \\ &= 8(2T(n/16) + n/8) + n + n + n \;=\; 16T(n/16) + n + n + n + n \\ &= \ldots \end{aligned}$$

At this point we can see a general pattern emerging.

$$\begin{aligned} T(n) &= 2^k T(n/(2^k)) + (n + n + \cdots + n) \qquad (k \text{ times}) \\ &= 2^k T(n/(2^k)) + kn. \end{aligned}$$

Now, we have generated alot of equations, but we still haven't gotten anywhere, because we need to get rid of the $T()$ from the right-hand side. Here's how we can do that. We *know* that $T(1) = 1$. Thus, let us select k to be a value which forces the term $n/(2^k) = 1$. This means that $n = 2^k$, implying that $k = \lg n$. If we substitute this value of k into the equation we get

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

In simplifying this, we have made use of the formula from the first homework, $a^{\log_b n} = n^{\log_b a}$, where $a = b = 2$. Thus we have arrived at the same conclusion, but this time no guesswork was involved.

The Iteration Method (a Messier Example): That one may have been a bit too easy to see the general form. Let's try a messier recurrence:

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 3T(n/4) + n & \text{otherwise.} \end{cases}$$

To avoid problems with floors and ceilings, we'll make the simplifying assumption here that n is a power of 4. As before, the idea is to repeatedly apply the definition, until a pattern emerges.

$$\begin{aligned} T(n) &= 3T(n/4) + n \\ &= 3(3T(n/16) + n/4) + n \;=\; 9T(n/16) + 3(n/4) + n \\ &= 9(3T(n/64) + n/16) + 3(n/4) + n \;=\; 27T(n/64) + 9(n/16) + 3(n/4) + n \\ &= \ldots \\ &= 3^k T\left(\frac{n}{4^k}\right) + 3^{k-1}(n/4^{k-1}) + \cdots + 9(n/16) + 3(n/4) + n \\ &= 3^k T\left(\frac{n}{4^k}\right) + \sum_{i=0}^{k-1} \frac{3^i}{4^i} n. \end{aligned}$$

As before, we have the recursive term $T(n/4^k)$ still floating around. To get rid of it we recall that we know the value of $T(1)$, and so we set $n/4^k = 1$ implying that $4^k = n$, that is, $k = \log_4 n$. So, plugging this value in for k we get:

$$\begin{aligned} T(n) &= 3^{\log_4 n} T(1) + \sum_{i=0}^{(\log_4 n)-1} \frac{3^i}{4^i} n \\ &= n^{\log_4 3} + \sum_{i=0}^{(\log_4 n)-1} \frac{3^i}{4^i} n. \end{aligned}$$

Again, in the last step we used the formula $a^{\log_b n} = n^{\log_b a}$ where $a = 3$ and $b = 4$, and the fact that $T(1) = 1$. (Why did we write it this way? This emphasizes that the function is of the form n^c for some constant c.) By the way, $\log_4 3 = 0.7925\ldots \approx 0.79$, so $n^{\log_4 3} \approx n^{0.79}$.

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.