Recursive Algorithms II

Margaret M. Fleck

29 March 2010

This lecture wraps up our discussion of algorithm analysis (section 4.4 and 7.1 of Rosen).

1 Announcements

Quiz should be graded soon. More information as soon as I have clear information about how close to done it is.

Homework 8 is posted, due this Friday. Reminder that honors homework 2 is due next Monday (the 5th).

The second midterm will be a week from Wednesday (7 April). 7-9pm in 141 Wohlers, just like the first midterm. Study materials will be posted very soon. If you need special arrangements (e.g. extra time), contact me and (if necessary) book a room at DRES.

Our final exam will be 7-10pm on Friday May 7th. The conflict final will be Monday the 10th, 1:30-4:30pm. As the end of term approaches, watch for further instructions (e.g. because the main exam involves two rooms in different buildings).

2 Recap

Last lecture, we started talking about mergesort, a sorting algorithm whose big-O running time is $O(n \log n)$, i.e. better than the $O(n^2)$ running time of bubblesort and insertion sort. This is, in fact, the best big-O running time you can have for a sorting algorithm.

The idea behind mergesort is to divide the (big) input list into two half-size lists. We keep dividing in half until we have a large collection of very

small lists. In our pseudo-code these base-case lists will be length 1. Real implementations often stop with lists of length 2 or even slightly longer, because it is easy to write code to directly sort very short lists.

Once the tiny lists are all sorted (which is automatically true for lists of length 1), then we start merging them back together. We first merge the tiny lists in pairs to make lists twice as long. We keep merging pairs together until we get one big (sorted) list.

[quick demo of the algorithm for merging two sorted lists]

For sorting small to medium lists of numbers, the $O(n^2)$ quicksort algorithm typically runs fastest, because it has an average running time of $O(n \log n)$ and very good constants. Mergesort's constants are only so-so. Mergesort is interesting because it's has $O(n \log n)$ worst-case running time and is easy to analyze. Also, because the key merge step accesses its three lists in storage order, variations of this technique are used when sorting data that is stored in places that are slow to access. In the Bad Old Days, "slow storage" was usually magnetic tapes. These days, it's often places that are somewhere else on the internet.

3 Pseudo-code for mergesort

The pseudocode for mergesort looks like:

1. mergesort($L = a_1, a_2, \ldots, a_n$: list of real numbers)
    2. if $(n = 1)$ then return L
    3. else
        4. $m = \lfloor n/2 \rfloor$
        5. $L_1 = a_1, a_2, \ldots, a_m$
        6. $L_2 = a_{m+1}, a_{m+2}, \ldots, a_n$
        7. return merge(mergesort(L_1),mergesort(L_2))

And the pseudocode for the merge function might look like:

1. merge(L_1,L_2: sorted lists of real numbers)
    2. O = emptylist
    3. while (L_1 or L_2 is not empty)

4. if (L_1 is empty)
    5. add smallest(L_2) to the end of O
6. else if (L_2 is empty)
    7. add smallest(L_1) to the end of O
8. else if (smallest(L_1) < smallest(L_2))
    9. add smallest(L_1) to the end of O
10. else add smallest(L_2) to the end of O
11. return O

We assume that the lists are sorted in decreasing order, so that the smallest element of a list is always at the start of the list. For any of the standard ways to implement a list in real programming languages, this means that merge is easy to build and runs efficiently.

We've written mergesort so that it returns the new sorted list. This is different from our code for bubble sort and insertion sort, which rearranged the values within the input array. All three algorithms can be written in either style: I've chosen the one that's most natural for each algorithm.

4 Analysis of mergesort

Now, let's do a big-O analysis of mergesort's running time. We need to start by looking at the merge function.

For merge, a good way to measure the size of the input n is that n is the sum of the lengths of the two input arrays. Or, equivalently, n is the length of the output array.

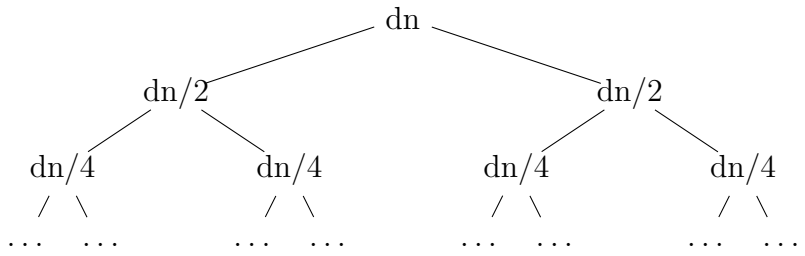
The merge function has one big loop. Since the operations within the loop all take constant time, we just need to figure out how many times the loop runs. Each time the loop runs, we move one number from L_1 or L_2 onto the output list O. n numbers have to be moved, in total. So the loop must run n times. So merge takes $O(n)$ (aka linear) time.

In this analysis, we have a resource: the numbers in the list. This resource has a known size and is being consumed at some known rate. Analyzing the consumption of a resource is another strategy for figuring out how many times a critical loop might run.

To analyze mergesort, suppose that its running time is $T(n)$, where n is the length of the input array. Then we can write the following recurrence for T, where c and d are constants.

- $T(1) = c$
- $T(n) = 2T(n/2) + dn$

This recurrence has the following tree:



The tree has $O(\log n)$ non-leaf levels and the work at each level sums up to dn. So the work from the non-leaf nodes sums up to $O(n \log n)$. In addition, there are n leaf nodes (aka base cases for the recursive function), each of which involves c work. So the total running time is $O(n \log n) + cn$ which is just $O(n \log n)$.

5 Tower of Hanoi

The Tower of Hanoi puzzle was invented by the French mathematician Edouard Lucas in 1883. It consists of three pegs and a set of disks of graduated size that fit on them. The disks start out in order on one peg. The goal is to rebuild the ordered tower on another peg without ever placing a disk on top of a smaller disk.

Show the game and its solution using the animation at:

`http://www.mazeworks.com/hanoi/`

The best way to understand the solution is recursively. Suppose that we know how to move k disks from one peg to another peg, using a third temporary-storage peg. To move $k + 1$ disks from peg A to peg B using a third peg C, we first move the top k disks from A to C using B as temporary

storage. Then we move the biggest disk from A to B. Then we move the other k disks from C to B, using A as temporary storage.

So our recursive solver looks like

1. hanoi(A,B,C: pegs, $d_1, d_2 \ldots d_n$: disks)
    2. if $(n = 1)$ move $d_1 = d_n$ from A to B.
    3. else
        4. hanoi(A,C,B,$d_1, d_2, \ldots d_{n-1}$)
        5. move d_n from A to B.
        6. hanoi(C,B,A,$d_1, d_2, \ldots d_{n-1}$)

The function hanoi breaks up a problem of size n into two problems of size $n-1$. Alert! Warning bells! This can't be good: the sub-problems aren't much smaller than the original problem!

Anyway, hanoi breaks up a problem of size n into two problems of size $n-1$. Other than the two recursive calls, it does only a constant amount of work. So the running time $T(n)$ for the function hanoi would be given by the recurrence (where c and d are constants):

- $T(1) = c$
- $T(n) = 2T(n-1) + d$

If we unroll this recurrence, we get

$$\begin{aligned} T(n) &= 2T(n-1) + d \\ &= 2 \cdot 2(T(n-2) + d) + d \\ &= 2 \cdot 2(2(T(n-3) + d) + d) + d \\ &= 2^3 T(n-3) + 2^2 d + 2d + d \\ &= 2^k T(n-k) + d \sum_{i=0}^{k-1} 2^i \end{aligned}$$

We'll hit the base case when $k = n - 1$. So

$$
\begin{aligned}
T(n) &= 2^k T(n-k) + d\sum_{i=0}^{k-1} 2^i \\
&= 2^{n-1}c + d\sum_{i=0}^{n-2} 2^i \\
&= 2^{n-1}c + d(2^{n-1}-1) \\
&= 2^{n-1}c + 2^{n-1}d - d) \\
&= O(2^n)
\end{aligned}
$$

6 Multiplying big integers

Suppose we want to multiply two integers. Back in the Bad Old Days of slow computers, we would need to multiply moderate-sized integers digit-by-digit. These days, we typically have a CPU that can multiply two moderate-sized numbers (e.g. 16-bit, 32-bit) as a single operation. But some applications (e.g. in cryptography) involve multiplying very, very large integers. Each very long integer must then be broken up into a sequence of 16-bit or 32-bit integers.

Let's suppose that we are breaking up each integer into individual digits, and that we're working in base 2. The recursive multiplication algorithm divides each n-digit input number into two $n/2$-digit halves. Specifically, suppose that our inputs numbers are x and y and they each have $2m$ digits. We can then divide them up as

$$x = x_1 2^m + x_0$$
$$y = y_1 2^m + y_0$$

If we multiply x by y in the obvious way, we get

$$xy = A2^{2m} + B2^m + C$$

where $A = x_1 y_1$, $B = x_0 y_1 + x_1 y_0$, and $C = x_0 y_0$. Set up this way, computing xy requires multiplying four numbers with half the number of digits.

So, if you just count the number of multiplications, the running time of this naive method has a recurrence

$$T(n) = 4T(n/2) + O(n)$$

If you solve this recurrence with $T(1)$ being a constant, you find that $T(n) = O(n^2)$. (Unrolling works well for finding this solution.)

In this analysis, we worry primarily about the multiplications and treat other arithmetic as being fast, aka $O(n)$. To understand why this is reasonable, first notice that adding two numbers requires time proportional to the number of digits in them. Multiplying by 2^m is also fast, because it just requires left-shifting the bits in the numbers. If you haven't done much with binary numbers yet (and perhaps don't know what it means to shift them), then it's just like multiplying by 10^m when working with (normal) base-10 numbers. To multiply by 10^m, you just have to add m zeros to the end of the number.

However, we can rewrite our algebra for computing B as follows

$$B = (x_1 + x_0)(y_1 + y_0) - A - C$$

So we can compute B with only one multiplication. So, if we use this formula for B, the running time of multiplication has the recurrence:

$$P(n) = 3P(n/2) + O(n)$$

It's not obvious that we've gained anything substantial, but we have. If we build a recursion tree for P, we discover that the kth level of the tree contains 3^k problems, each involving $n\frac{1}{2^k}$ work. The tree height is $\log_2(n)$.

So each level requires $n(\frac{3}{2})^k$ work. If you add up this up for all levels, you get a summation that is dominated by the term for the bottom level of the tree: $n(\frac{3}{2})^{\log_2 n}$ work. If you mess with this expression a bit, using facts about logarithms, you find that it's $O(n^{\log_2 3})$ which is approximately $O(n^{1.585})$.

So this trick, due to Anatolii Karatsuba, has improved our algorithm's speed from $O(n^2)$ to $O(n^{1.585})$ with essentially no change in the constants. If $n = 2^10 = 1024$, then the naive algorithm requires $(2^10)^2 = 1,048,576$ multiplications, whereas Katatsuba's method requires $3^{10} = 59,049$ multiplications. So this is a noticable improvement and the difference will widen as n increases.

There are actually other integer multiplication algorithms with even faster running times, e.g. Schoöhage-Strassen's method takes $O(n \log n \log\log n)$ time. But these methods are more involved.