CMPSCI611: **Three Divide-and-Conquer Examples** Lecture 2

Last lecture we presented and analyzed *Mergesort*, a simple divide-and-conquer algorithm. We then stated and proved the *Master Theorem*, which gives the big-O running time for divide-and-conquer algorithms with recurrences of a particular form:

If $T(n)$ satisfies $T(n) = aT(n/b) + \Theta(n^\alpha)$ and $T(n) = O(1)$ for $n \leq c$, we let $\beta = \log_b a$ and then:

- If $\alpha > \beta$, $T(n) = \Theta(n^\alpha)$
- If $\alpha = \beta$, $T(n) = \Theta(n^\alpha \log n)$
- If $\alpha < \beta$, $T(n) = \Theta(n^\beta)$

Mergesort had a recurrence of $T(n) = 2T(n/2) + \Theta(n)$, and thus a solution of $T(n) = \Theta(n \log n)$. Today we will see examples of all three cases of the Master Theorem.

Note that when we have a recurrence of this form, with a Θ bound on the time to integrate the solutions, the result is a Θ bound on the worst-case time taken by *this algorithm*. It provides only a big-O bound on the worst-case time needed to *solve the problem*, since there might be a different, faster algorithm.

Sometimes our analysis gives us only a big-O bound on the time to integrate the solutions, but the Master Theorem then still gives us a big-O result.

To get a Θ bound on the time for the *problem*, we need a *lower bound argument*. For the sorting problem, with the right assumptions, we have a matching lower bound, but usually we don't have one.

Matrix Multiplication: If A and B are each n by n matrices over any *ring*, their *product* $C = AB$ is defined by the rule:

$$C_{i,j} = \sum_k A_{i,k} B_{k,j}$$

There is an algorithm to compute C that should be obvious from this definition (too obvious to require code). For each of the n^2 entries of C, for each k, we multiply $A_{i,k}$ by $B_{k,j}$, requiring n scalar multiplications. Then we add up n products for each entry of C. All in all we use exactly n^3 multiplications and $\Theta(n^3)$ (actually $n^3 - n^2$) additions, for $\Theta(n^3)$ total operations.

We can't hope for fewer than $2n^2$ operations because each entry of A or B might affect the answer and must be looked at in the worst case. Surprisingly, though, we can do better than the standard algorithm!

Before we present subcubic matrix multiplication, we will warm up with a related but simpler problem,

Multiplying Integers: Our input is two n-bit integers $a = \Sigma_i\, a_i 2^i$ and $b = \Sigma_i\, b_i 2^i$, given as sequences of bits. Our output is the integer $c = ab$, given in the same format. Note that c has at most $2n$ bits.

Note: Most of the integers we see in daily life may be multiplied in a single step. But the algorithm we are about to present can be adapted to multiply integers stored in n *words* of fixed size, in a format like that used for the `BigNum` class in Java. For the time being we will count how many *bit operations* we use to multiply – if we have words of $O(1)$ bits each a word operation is only an abbreviation for $O(1)$ bit operations anyway.

Adding integers is easy in $O(n)$ bit operations.

The **standard algorithm** computes $c = \Sigma_{i,j}\, a_i b_j 2^{i+j}$ by taking the AND of each a_i with each b_j, shifting the result into position, and adding the results in $\Theta(n^2)$ total time.

Divide and Conquer? If we write $a = a_1 2^{n/2} + a_2$ and $b = b_1 2^{n/2} + b_2$, then we can express ab in terms of these $n/2$-bit numbers as

$$a_1 b_1 2^n + (a_1 b_2 + a_2 b_1) 2^{n/2} + a_2 b_2.$$

We can thus multiply the n-bit numbers by carrying out four multiplications of $n/2$-bit numbers, plus a constant number of additions. Our recurrence is $T(n) = 4T(n/2) + \Theta(n)$, fitting the Master Theorem with $a = 4$ and $b = 2$. But since $\beta = \log_2 4 = 2$, this algorithm also takes $\Theta(n^2)$ time, the same as the standard algorithm.

A Trick: There are three coefficients we need to compute, and we originally thought of four binary products we could use, along with additions, to get them. But these four binary products are not linearly independent. It turns out that if we compute just three binary products:

$$\begin{aligned} P_1 &= a_1 b_1 \\ P_2 &= a_2 b_2 \\ P_3 &= (a_1 + a_2)(b_1 + b_2) \end{aligned}$$

then the three coefficients that we want are P_1, $P_3 - P_1 - P_2$, and P_2. These *three* multiplications of $n/2$-bit numbers, together with a constant number of additions (a larger constant number, note), allow us to multiply the n-bit numbers. The new recurrence is:

$$T(n) = 3T(n/2) + \Theta(n)$$

which has solution $T(n) = \Theta(n^{\log_2 3}) = \Theta(n^{1.59\ldots})$.

There is a way to use the Fast Fourier Transform algorithm from next lecture to multiply two n-bit numbers even faster, in $\Theta(n \log n \log\log n)$ time. The obvious lower bound on the time needed is only $\Omega(n)$, the time needed to read all the input.

Back To Matrix Mulitiplication:

Just as we could write an n-bit integer as a shifted sum of two $n/2$-bit integers, we can write an n by n matrix as a 2 by 2 matrix whose *entries* are $n/2$ by $n/2$ matrices.

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

$$B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$AB = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

Simple Divide-and-Conquer:

There are *eight* products of $n/2$ by $n/2$ matrices, plus a constant number of additions of such matrices. (Note that two n by n matrices may be added in $\Theta(n^2)$ time.)

The recurrence here is $T(n) = 8T(n/2) + \Theta(n^2)$, which has solution $T(n) = \Theta(n^3)$ because $\beta = \log_2 8 = 3$. So this is no better than the standard algorithm.

Strassen's Trick:

We want four particular sums of products of the eight $n/2$ by $n/2$ matrices. There are a variety of ways we might multiply one sum of matrices by another, beyond the eight simple combinations we just used. Strassen was able to find a set of *seven* products of sums that could be used to form all four necessary coefficients:

$$\begin{aligned}
P_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
P_2 &= (A_{21} + A_{22})(B_{11}) \\
P_3 &= (A_{11})(B_{12} - B_{22}) \\
P_4 &= (A_{22})(-B_{11} + B_{21}) \\
P_5 &= (A_{11} + A_{12})(B_{22}) \\
P_6 &= (-A_{11} + A_{21})(B_{11} + B_{12}) \\
P_7 &= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}$$

The product matrix is expressed as follows:

$$AB = \begin{pmatrix} P_1 + P_4 - P_5 + P_7 & P_3 + P_5 \\ P_2 + P_4 & P_1 - P_2 + P_3 + P_6 \end{pmatrix}$$

The new recurrence is $T(n) = 7T(n/2)+\Theta(n^2)$. The new β is $\log_2 7 = 2.81\ldots$, so the solution of $T(n^{2.81\ldots})$ is indeed subcubic. There are more recent algorithms that are (at least theoretically) even faster than Strassen – the current champion is by Coppersmith and Winograd (1990) and takes time $O(n^{2.376\ldots})$.

CLRS, Chapter 28, has more on how Strassen may have gone about finding his solution.

Is Strassen's algorithm practical? At *some* value of n, it ought to be, because a function that is $o(n^3)$ will eventually overtaken by one that is $\Theta(n^3)$ as n increases. When this happens depends, of course, on the exact constants and lower-order terms in the two time functions. Further complications come from details of the implementation. For example, copies from one location to another *within the RAM* of the computer are always much faster than those that involve the external memory. Once matrices get too big to fit in RAM, things slow down dramatically.

As the notes say, "different sources quote quite different numbers for this tradeoff point, ranging from $n = 8$ to $n = 100$". Of course, when we use Strassen we will actually use a hybrid algorithm that uses Strassen's trick to subdivide the matrices *only until* they are small enough that the standard algorithm is faster. Similarly, a Mergesort algorithm might use another general sorting algorithm or even special-purpose code on sufficiently small lists.

The Closest Pair Problem:

Our final example of a divide-and-conquer algorithm comes from *computational geometry*. Suppose that we are given n points in a plane, each given by a pair of real (actually floating-point) numbers. We want to find which of the $\binom{n}{2}$ *pairs* of points has the shortest distance between them.

We can do this in $\Theta(n^2)$ time by computing the distance for each of the pairs and taking the minimum of all these distances. There is a clear $\Omega(n)$ lower bound on the time needed, because if we never look at one of the points it might turn out to be very close to another point.

We'll use a recursive algorithm to solve this problem in time $O(n \log n)$. The base case of $n \leq 2$ is easy, as we have at most one pair to check.

Dividing the Problem in Half:

The basic idea will be to draw a vertical line that has $n/2$ points on each side of it. Once we do this, we can apply the algorithm recursively to find the closest pair on each side. But there are a number of issues left after we have this idea:

1. How do we find the correct vertical line, and how long does it take?
2. What if the closest pair involves one point on each side of the line? We don't want to check all the pairs that cross the line, because there are $n^2/4$ of them!

The first problem is not so bad – we sort the points by x-coordinate, taking $\Theta(n \log n)$ time, and pick the median coordinate as the position of our line. Any points *exactly on* this line can be arbitrarily assigned to one side or the other to make the sides have exactly $n/2$ points each (again we assume that n is a power of two).

Checking Pairs that Cross the Line:

The key insight is that we *only care* about a pair that crosses the line if its distance is smaller than the smallest distance that *doesn't* cross the line. Let δ_L be the smallest distance found by the recursive call to the left, and δ_R the smallest distance found on the right. Let δ be the minimum of these two. The only points that could be part of *interesting* line-crossing pairs are those within δ of the line.

We can find this set of points, called P_M, easily (in $O(n)$ time) from the list of points sorted by x-coordinate. Of course, there is no reason why most or all of the n points might not be in P_M, so we need a better idea than just checking all the pairs of points in P_M.

But a point in P_M can be close to another point in P_M *only if they are close in their y-coordinate* as well as in their x-coordinate. If we sort P_M by y-coordinate, we only have to check the distance of each point to the *next few* points in the y list. This is because points in P_M on the same side of the vertical line are guaranteed to be at least δ apart, so only so many can fit into the 2δ by δ box in P_M just above any given point x. (We only check *above* x because we will cover the pairs where x is the higher point when we check the box above the other point.)

There's a diagram on page 16 of the printed notes that shows (or would show, if the captions were readable) that there might be as many as *seven* points in this box to check for distance to x.

This means that even if there are $\Theta(n)$ points in P_M, once we sort them we have to check only $O(1)$ pairs for each, so we can do this in $O(n)$ time.

What's the recurrence resulting from this?

To find the closest pair, we:

1. Sorted the points by x-coordinate – $O(n \log n)$ time
2. Found δ_L and δ_R – $2T(n/2)$ time
3. Found P_M – $O(n)$ time
4. Sorted P_M by y-coordinate – $O(n \log n)$ time, and
5. Checked each point in P_M against its close neighbors – $O(n)$ time.

This gives us $T(n) = 2T(n/2) + O(n \log n)$, which is *not* a case of the Master Theorem but (as you'll show on HW#1) has a solution of $T(n) = O(n \log^2 n)$. This is much better than $O(n^2)$, but we can actually do a little better.

If we sort the entire set of points *once* by y-coordinate at the beginning of the algorithm, this costs us $O(n \log n)$ time up front. But then we can sort any *subset* of the points in $O(n)$ time, by moving down the sorted list and picking out the points we want. This gives us:

$$T(n) = 2T(n/2) + \Theta(n)$$

which we know has solution $T(n) = \Theta(n \log n)$. Adding in the time for the presort keeps us at $\Theta(n \log n)$. This is actually the best possible running time for this problem – the lower bound is the sort of thing discussed in the computational geometry course taught by Profs. Brock and Streinu.