Jelani Nelson (minilek@seas.harvard.edu) December 6, 2011

Berkeley Math Circle

Faster Multiplication

Today we'll see how to multiply two integers faster than how you learned in grade school. First, let's define the term *algorithm*.

Definition 1. *An* algorithm *is a well-defined procedure for performing some computational task.*

For example, below is an algorithm for the task of finding the maximum of a list of numbers.

Algorithm to find the maximum in a list of numbers

Input: $x_1, x_2, \ldots, x_n$

Output: The maximum value amongst $x_1, \ldots, x_n$

1. answer $\leftarrow -\infty$
2. for each number x in the input
   if $x >$ answer
   then answer $\leftarrow x$
3. **return** answer

Question 1: Give an algorithm for finding the *minimum* value in a list.

Of course, given some task, there may be multiple ways to complete that task. A usual goal in algorithms is to find the *most efficient* procedure for completing the task. In our case, "efficiency" is just the number of steps the algorithm takes, and the fewer steps our algorithm takes, the more efficient it is. Our max-finding algorithm takes n steps on an input list of n numbers.

Consider yet another problem. This time we're given a list of positive integers all less than 10 (there could be repetitions), and we want to find two of them that sum to 10.

Algorithm to find two inputs summing to 10

Input: $x_1, x_2, \ldots, x_n$

Output: "Yes" if there are two inputs summing to 10, and "No" otherwise

1. for $i = 1, \ldots, n$
   for $j = i + 1, \ldots, n$
   if $x_i + x_j = 10$ then **return** "Yes"
2. **return** "No"

Question 2: How many steps does the above algorithm take?

Question 3: Can you find a much more efficient algorithm in the case that n is large? (by large, I mean for example when $n > 30$?)

Recursion:

Recursion in computer science is solving a problem by solving simpler instantiations of the same problem. For example, here's a recursive max-finding algorithm.

Algorithm to find the maximum in a list of numbers

Input: $x_1, x_2, \ldots, x_n$

Output: The maximum value amongst $x_1, \ldots, x_n$

1. if $n = 1$ then **return** x_1
2. else
3. y $\leftarrow$ the maximum of the list $x_2, \ldots, x_n$ (computed recursively)
4. **return** the maximum of x_1 and y

If the list of numbers has size 1, then the maximum is just the only number in the list. Otherwise, we reduce the problem of max-finding in a list of size n to the *simpler* problem of max-finding in a list of size $n - 1$. This is recursion.

As another (classic) example, consider the *Fibonacci* sequence $1, 1, 3, 5, 8, 13, \ldots$. This sequence is defined by the 0th and 1st Fibonacci numbers both being 1, and subsequent Fibonacci numbers being the sum of the previous two.

That is, if F_i represents the ith Fibonacci number,

$$F_i = \begin{cases} 1 & \text{if } i = 0 \text{ or } i = 1 \\ F_{i-1} + F_{i-2} & \text{otherwise} \end{cases}$$

Now here is an example of using recursion to calculate the nth Fibonacci number. Note that the `fibonacci` function calls itself on smaller, i.e. simpler, inputs.

Algorithm to find the maximum in a list of numbers

Input: n

Output: F_n, the nth Fibonacci number

function **fibonacci**(n)

1. if $n < 2$ then **return** 1
2. else **return fibonacci**$(n-1)$ + **fibonacci**$(n-2)$

Integer Multiplication: Now that we know what algorithms and recursion are about, let's take a look at integer multiplication. We all learned how to multiply numbers with lots of digits in elementary school:

$$\begin{array}{r} 435 \\ 213 \\ \hline 1305 \\ 435 \\ 870 \\ \hline 92655 \end{array}$$

Question 4: Suppose each of the two integers we are multiplying is n digits long. Ignoring the time it takes to sum up all the intermediate calculations at the end, how many steps does integer multiplication take?

Now, a natural question to ask is: can we multiply numbers faster? After all, multiplication is some computational task, and what we learned in elementary school is just *one* algorithm for solving that computational task. Perhaps there's another algorithm which is much faster?

One approach we can try is the *divide and conquer* method. Divide and conquer is a strategy based on dividing up the input into smaller pieces, solving the problem on the smaller pieces *recursively*, then combining the result to get the answer for the full input.

So, let's say we're trying to multiply a `list` of digits a by another `list` of digits b, each of length n (in base 10). For the sake of simplifying all future discussion, let's assume n is a perfect power of 2 (if not, we can pad both a and b by 0s at their beginnings until their lengths *are* powers of 2, and doing this at most doubles n). Let a_{high} represent the first half of the digits of a (the left-most half), and let a_{low} represent the right half of digits. Then, treating a as an integer, $a = a_{\text{high}} \times 10^{n/2} + a_{\text{low}}$. Doing similarly for b, this means that

$$\begin{aligned} a \times b &= (a_{\text{high}} \times 10^{n/2} + a_{\text{low}}) \times (b_{\text{high}} \times 10^{n/2} + b_{\text{low}}) \\ &= a_{\text{high}} \times b_{\text{high}} \times 10^n + (a_{\text{high}} \times b_{\text{low}} + a_{\text{low}} \times b_{\text{high}}) \times 10^{n/2} + a_{\text{low}} \times b_{\text{low}} \end{aligned}$$

In other words, to multiply two n-digit numbers, we just need to multiply four pairs of $n/2$-digit numbers, append either $n/2$ or n zeroes to some of our results (this is what multiplying by a power of 10 does), then add up the results. Appending n zeroes or adding two $n/2$-digit numbers both take at most n steps. When $n = 1$, we can just do the multiplication in 1 step (we've memorized our multiplication tables). Thus, if $T(n)$ is the running time to multiply two n-digit numbers, then we know

$$T(n) \leq \begin{cases} 1 & \text{if } n = 1 \\ 4 \cdot T(n/2) + 5n & \text{otherwise} \end{cases}$$

As we'll see in class, this means the running time is at most $5n^2$.

Karatsuba's algorithm Anatolii Alexeevitch Karatsuba in 1960 found a way to make the divide-and-conquer approach work for speeding up integer multiplication. The story goes that Andrey Kolmogorov, a giant of probability theory and other areas of mathematics, had a conjecture from 1956 stating that it is impossible to multiply two n-digit numbers significantly faster than the standard method. In 1960 Kolmogorov told many scientists his conjecture at a seminar at Moscow State University, and Karatsuba, then in the audience, went home and disproved Kolmogorov's conjecture in exactly one week[1]. Let's now cover the method he came up with.

Let $X = a_{\text{high}} \times b_{\text{high}}$, $Y = a_{\text{low}} \times b_{\text{low}}$, and $Z = (a_{\text{high}} + a_{\text{low}}) \times (b_{\text{high}} + b_{\text{low}})$. Then

$$a \times b = X \times 10^n + (Z - X - Y) \times 10^{n/2} + Y.$$

Thus, now, to multiply two n-digit numbers we only need to multiply *three* pairs of $n/2$-digit numbers. This gives the recurrence

$$T(n) \leq \begin{cases} 1 & \text{if } n = 1 \\ 3 \cdot T(n/2) + 6.5n & \text{otherwise} \end{cases}.$$

Question 5: Give an upper bound for how fast Karatsuba's algorithm takes to multiply two n-digit numbers. Remember that if $x \neq 1$,

$$\sum_{i=0}^{t} x^i = (x^{t+1} - 1)/(x - 1).$$

Question 6: The Toom-3 algorithm which we won't cover today splits each number into 3 pieces of equal size and does only 5 recursive multiplications along with some number of shifts and additions to combine the results. How fast is the Toom-3 algorithm? How many multiplications would the "obvious" method need after splitting into 3 pieces, and what would be its running time?

In fact it is possible to get integer multiplication algorithms with running times just a bit larger than $n \log n$ using what's known as the *Fast Fourier Transform*, but we will not cover these methods in this lecture.

[1] See A. A. Karatsuba. The complexity of computations. Proceedings of the Steklov Institute of Mathematics, Vol. 211, pp. 169–183, 1995.