Notes for Lecture 2

Today we see another algorithm based on the divide-and-conquer methodology, namely a fast (although not the fastest known) algorithm for multiplication of large integers. We then state and prove the Master Theorem, a general result that shows how to solve the recursive equations that come up in the analysis of divide-and-conquer algorithms.

1 Fast integer multiplication

We want to design an algorithm to multiply arbitrarily big integers. We assume that large integers are stored as sequences of digits in an appropriate base. So a large integer A in base b is represented as a sequence $a_0, a_1, \ldots, a_{n-1}$, where

$$A = a_0 + ba_1 + \cdots + b^{n-1}a_{n-1}$$

For example, in base 10, the number 346,523,456 would be represented as the sequence $(6, 5, 4, 3, 2, 5, 6, 4, 3)$. In practice one wants to pick the base so that your computer can compute sums and products of numbers up to $b - 1$ with one machine language operation. So typically you would choose $b = 2^{64}$. The algorithm that we describe works in any basis. We will refer to base 10 because then it is easier to give examples.

Before we get to multiplication, you should think about how to implement the elementary-school algorithm for addition, and verify that you can add an n-digit number and an m-digit number in $O(n + m)$, since the algorithm performs only a constant number of operations per digit.

If you think about how the elementary-school algorithm for multiplication works, you can see that it takes $O(mn)$ time to multiply an n-digit number by an m-digit number, so that the running time is $O(n^2)$ if the numbers have the same number of digits.

In trying to apply a divide-and-conquer approach, we need to think of a useful to split our data (sequences of digits) into smaller data sets. Then it makes sense to think of

$$A = A_L + 10^{n/2}A_M$$

where

$$A_L = \sum_{i=0}^{n/2-1} 10^i a_i$$

contains the least significant digits of A and

$$A_M = \sum_{i=0}^{n/2-1} 10^i a_{i+n/2}$$

contains the most significant digits. For example we might write

$$1789 = 89 + 10^2 \cdot 17$$

splitting a four-digit number into two two-digit numbers.

If we do the same for B, writing $B = B_L + 10^{n/2}B_M$, we see that we want to compute

$$A \cdot B = (A_L + 10^{n/2}A_M) \cdot (B_L + 10^{n/2}B_M) = A_L B_L + 10^{n/2}(A_L B_M + A_M B_L) + 10^n A_M B_M$$

This means that up to multiplying by a multiple of the base (which means just adding zeroes to our list of digits) and computing three sums (each taking $O(n)$ time), we have reduced our problem to performing four multiplications over $\frac{n}{2}$-digit numbers. This means that a recursive computation using the above formula yields a divide-and-conquer algorithm whose running time obeys the equation

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

which we solved in class opening it up and bounding the resulting exponential sum (we do not repeat the calculation because it is subsumed by the Master Theorem that we prove next). Unfortunately the recursion solves to $O(n^2)$, not faster than the naive elementary-school algorithm. (Indeed, we are doing the same operations as that algorithm, and just organizing them differently.)

It turns out, however, that there is something smarter that we can do: we want to compute the three integers

- $A_L \cdot B_L$
- $A_L \cdot B_M + A_M \cdot B_L$
- $A_M \cdot B_M$

One way to compute them is to do four products and one sum, but we can also do that with only *three* multiplications (and four sums/substractions). The point is that

$$A_L \cdot B_M + A_M \cdot B_L = (A_L + A_M) \cdot (B_L + B_M) - A_L \cdot B_L - A_M \cdot B_M$$

So that, after we compute the three products (and two sums) necessary to compute $A_L \cdot B_L$, $A_M \cdot B_M$, and $(A_L + A_M) \cdot (B_L + B_M)$, we can compute $A \cdot B$ with only $O(n)$ additional work. This leads to a divide-and-conquer algorithm whose running time satisfies

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

which solves to $T(n) = O(n^{\log_2 3})$, where $\log_2 3$ is about 1.58.

2 The Master Theorem

In the examples that we have seen so far, and in other important examples, divide-and-conquer algorithms lead to running times described by recursive equations that look like

$$T(n) = aT(n/b) + O(n^d)$$

where a is the number of recursive calls, n/b is the size of the input of each recursive calls, and $O(n^d)$ is the time for preparing the recursive calls and combining the results.

THEOREM 1
The recursion

$$T(n) = aT(n/b) + O(n^d)$$

solves to

- $O(n^d)$ *if* $d > \log_b a$
- $O(n^d \log n)$ *if* $d = \log_b a$
- $O(n^{\log_b a})$

That is, your running time is order of whichever is larger of n^d and $n^{\log_b a}$. If the two bounds are the same, then you pay an extra factor of the order of $\log n$.

The proof is not difficult, but it relies on the following observation. If $c < 1$ is a fixed constant, then the series

$$1 + c + c^2 + \cdots + c^k + \cdots$$

converges (to $1/(1-c)$), and so there is a fixed constant that upper bounds any partial sum.

$$1 + c + c^2 + \ldots + c^k < \frac{1}{1-c}$$

On the other hand, if $c > 1$, then we have the upper bound

$$1 + c + c^2 + \ldots + c^k = \frac{c^{k+1} - 1}{c - 1} < c^k \cdot \frac{c}{c-1}$$

Finally, not very interestingly, we have that if $c = 1$

$$1 + c + c^2 + \ldots + c^k = k + 1$$

Thus, no matter the value of c, we have a way to bound the sum of powers of c.

Now let us go back to our recursion, and, after scaling time so that we have

$$T(n) = aT(n/b) + n^d$$

we get

$$\begin{aligned}
T(n) &= n^d + aT(n/b) \\
&= n^d + a\frac{n^d}{b^d} + a^2 T(n/b^2) \\
&= n^d + a\frac{n^d}{b^d} + a^2\frac{n^d}{b^{2d}} + a^3 T(n/b^3) \\
&\vdots \\
&= n^d + a\frac{n^d}{b^d} + a^2\frac{n^d}{b^{2d}} + \cdots + a^{k-1}\frac{n^d}{b^{(k-1)d}} + a^k T(n/b^k)
\end{aligned}$$

The above expression is true for every k, and if we instantiate it to the special case $k = \log_b n$, then the last term $T(n/b^k) = T(1) = 1$, so we have

$$T(n) = \sum_{i=0}^{k} a^i \frac{n^d}{b^{id}} = n^d \sum_{i=0}^{k} \left(\frac{a}{b^d}\right)^i$$

So we have that if $a/b^d < 1$ (equivalent to the condition that $\log_b a < d$) the partial sum is bounded by a constant, and so

$$T(n) = O(n^d)$$

If $a/b^d = 1$, then $T(n) = kn^d$ and so

$$T(n) = O(n^d \log n)$$

Finally, if $a/b^d > 1$, then the sum is bounded, within a constant factor, by the last term, and so $T(n) = O(n^d a^k / b^{dk})$. Let us understand such bound a bit better. We have:

$$n^d / b^{dk} = 1$$

and

$$a^k = a^{\log_b n} = n^{\log_b a}$$

so that the running time is

$$T(n) = O(n^{\log_b a})$$