CIS 320: Algorithms September 20, 2021

Lecture 6

Lecturer: Aaron Roth *Scribe: Aaron Roth*

Divide and Conquer III: Integer Multiplication

Today we'll study a problem you learned an algorithm for in elementary school, that you might not have thought of as an algorithm: integer multiplication. In addition to being another example of applying our divide-and-conquer methodology, we'll use this as an example to see that the "obvious" algorithm — i.e. the one you learned in elementary school — is not always optimal, and there can be room for big improvements via careful algorithmic thinking.

Definition 1 (Representation as decimals) *We represent an integer k using digits in the standard manner:*

$$d(k) = d_n \circ \ldots \circ d_2 \circ d_1$$

where $d_i \in \{0, 1, \ldots, 9\}$ and

$$k = I(d_n, \ldots, d_1) \equiv \sum_{i=1}^{n} d_i \cdot 10^{i-1}$$

Here $n = \theta(\log k)$ is the size of the representation of the integer k. When we talk about the running time of algorithms it will be in terms of n.

Definition 2 (n digit integer multiplication) *Given two n digit integers in decimal representation $x_n, \ldots, x_1$ and $y_n, \ldots, y_1$, return the integer $k = I(x_n, \ldots, x_1) \cdot I(y_n, \ldots, y_1)$.*

The elementary school algorithm for multiplication requires multiplying all pairs of digits $x_i \cdot y_j$ for $1 \leq i \leq n$ and $1 \leq j \leq n$, and so runs in time $O(n^2)$. We would like to do better.

We make a couple of simple observations: we can solve the *addition* problem on n digit integers in time $O(n)$ using the elementary school algorithm — it involves adding $x_i + y_i$ for each i, plus an additional constant amount of work per digit to handle "carries". It is also very easy to multiply by powers of 10, because this only requires appending 0s to the end of the number: we can multiply any integer by 10^m in time $O(m)$.

How can we take advantage of the fact that addition and multiplication by powers of 10 are easy, while still being able to "divide" our n bit multiplication problem into smaller multiplication problems? Suppose we divide the bit representation of our integers in half: Given an input $x = (x_n, \ldots, x_1)$, we write (assuming for convenience that n is even):

$$x^2 = (x_n, \ldots, x_{n/2+1}), x^1 = (x_{n/2}, \ldots, x_1)$$

Observe that $I(x) = 10^{n/2} \cdot I(x^2) + I(x^1)$. Observe that if we are given $I(x^2)$ and $I(x^1)$, we can obtain $I(x)$ with a single multiplication by a factor of 10, and a single addition of two $n/2$ bit numbers, which are operations we can complete in time $O(n)$. This gives us our first idea: Perhaps we can compute:

$$I(x) \cdot I(y) = \left(10^{n/2} \cdot I(x^2) + I(x^1)\right)\left(10^{n/2} \cdot I(y^2) + I(y^1)\right)$$

$$= 10^n I(x^2)I(y^2) + 10^{n/2} I(x^2)I(y^1) + 10^{n/2} I(x^1)I(y^2) + I(x^1)I(y^1)$$

In other words, we can reduce a single multiplication of two n-bit integers to *four* multiplications of $n/2$ bit integers, plus $O(n)$ time overhead in the form of additions and multiplication by multiples of 10. Before trying to do a formal analysis, lets try and do some back of the envelope math to see if this might be useful. It seems we are getting at an algorithm that will have a recurrence that looks like:

$$T(n) = 4 \cdot T(n/2) + c \cdot n$$

Unrolling this recurrance (and drawing out the tree), we can see that at the i'th level of the tree, there are 4^i sub-problems each of size $n/2^i$, and the depth of the recursion is $\log n$. Thus the total amount of computation that must be done to solve the problem in this way appears to be:

$$n \sum_{i=0}^{\log n} 2^i = n \cdot (n + n/2 + n/4 + \ldots + 1) \approx 2n^2$$

So it doesn't seem like proceeding down this route will buy us anything. But if we could do a litte better — say by reducing the problem to *three* smaller integer multiplications, perhaps we could get a win.

What we need is to do a little algebra. Suppose we perform the following multiplication:

$$p \equiv (I(x^2) + I(x^1))(I(y^2) + I(y^1)) = I(x^2)I(y^2) + I(x^2)I(y^1) + I(x^1)I(y^2) + I(x^1)I(y^1)$$

If we also compute $I(x^2)I(y^2)$ and $I(x1)I(y1)$, can compute:

$$p - I(x^2)I(y^2) - I(x1)I(y1) = I(x^2)I(y^1) + I(x^1)I(y^2)$$

These three multiplications therefore give us the ability to compute every term we need to assemble $I(x) \cdot I(y)$! Recall:

$$\begin{aligned} I(x) \cdot I(y) &= 10^n I(x^2)I(y^2) + 10^{n/2} I(x^2)I(y^1) + 10^{n/2} I(x^1)I(y^2) + I(x^1)I(y^1) \\ &= 10^n I(x^2)I(y^2) + 10^{n/2} \left(p - I(x^2)I(y^2) - I(x^1)I(y^1)\right) + I(x^1)I(y^1) \end{aligned}$$

So we have a proposed algorithm:

Algorithm 1 RecursiveMultiply(k_1, k_2)

Let $x = d(k_1)$ and $y = d(k_2)$, the n digit representation of each integer.
if $n = 2$ **then**
  Return $k_1 \cdot k_2$
end if
Split $x = (x^2, x^1)$ and $y = (y^2, y^1)$ into two substrings of $n/2$ digits each.
Let $p =$ RecursiveMultiply$((I(x^2) + I(x^1)), (I(y^2) + I(y^1)))$
Let $a =$ RecursiveMultiply$(I(x^2), I(y^2))$ and $b =$ RecursiveMultiply$(I(x^1), I(y^1))$.
Return $10^n \cdot a + 10^{n/2} \cdot (p - a - b) + b$

We've already done most of the necessary calculations, but lets cross our t's and establish that the algorithm does what it is supposed to:

Theorem 3 *Given two n-digit integers k_1 and k_2 RecursiveMultiply(k_1, k_2) returns their product $k_1 \cdot k_2$.*

Proof Note that without loss of generality we can assume that n is a power of 2 (if it is not already, we can pad the integers with leading 0s until it is, which at most doubles n). We proceed by induction on n. In the base case when $n = 2$, by construction the algorithm returns $k_1 \cdot k_2$. In the inductive case, we assume that the claim is true for n digit integers, and prove that it is also true for $2n$ digit integers.

If x, y are $2n$ digits long, then x^2, x^1, y^2, y^1 are each n digits long. Hence by our inductive assumption $p = v(I(x^2) + I(x^1))(I(y^2) + I(y^1))$, $a = I(x^2) \cdot I(y^2)$ and $b = I(x^1) \cdot I(y^1)$. Thus using our previous calculations, the value we return is:

$$\begin{aligned} 10^n I(x^2)I(y^2) + 10^{n/2} \left(p - I(x^2)I(y^2) - I(x^1)I(y^1)\right) + I(x^1)I(y^1) &= I(x) \cdot I(y) \\ &= k_1 \cdot k_2 \end{aligned}$$

as desired. ■

Finally we can establish the running time of our algorithm and verify that we have improved over the elementary school algorithm:

Theorem 4 *Given two n digit integers, RecursiveMultiply runs in time $O(n^{1+\log(3/2)}) \leq O(n^{1.59})$.*

Proof RecursiveMultiply makes three recursive calls on inputs of length $n/2$, and then performs $c \cdot n$ additional work for some constant c. Thus we can write down a recurrence for its running time $T(n)$:

$$T(n) \leq 3T(n/2) + cn$$

We unroll the recursion tree (a picture is useful here). The recursion has depth $\log(n)$, and at level i, there are 3^i computations each of size $n_i = n/2^i$. Thus the total amount of work can be bounded by:

$$T(n) \leq cn \sum_{i=0}^{\log n} (3/2)^i$$

We recall that a geometric series of the form $\sum_{i=0}^{k} a^i$ has a closed form solution: $\left(\frac{1-a^{k+1}}{1-a}\right)$. Here we simply have a geometric series with $k = \log n$ and $a = 3/2$, and so we can solve:

$$T(n) \leq cn \cdot 2\left(\left(\frac{3}{2}\right)^{\log n+1} - 1\right) = O\left(n^{1+\log(3/2)}\right)$$

■

So we can do better than the elementary school algorithm! In fact, we can do even better, but the story is not entirely closed yet. In 2019, Harvey and van der Hoeven gave (following a long series of results getting gradual improvements) an algorithm for multiplying two n bit numbers in time $O(n \log n)$. This is conjectured to be optimal (but the conjecture is unproven).