CSE 101 Notes

Large Integer Multiplication Using Divide-and-Conquer

The "grade school" algorithm for multiplying n-digit integers x and y multiplies every digit in x by every digit in y, yielding a $\Theta(n^2)$ algorithm. We want to do better using a divide-and-conquer method. We assume the integers are represented in binary.

For the "divide" step, we split each binary integer into two roughly equal-sized parts, each containing about half of the bits of the original: one part contains the high order bits, and the other part contains the low order bits. Specifically, if x and y have n bits, we let $m = \lfloor \frac{n}{2} \rfloor$ and divide x into its low-order m bits (x_2), and its high-order $n - m$ bits (x_1); y_1 and y_2 are defined similarly. These are given by $x_1 = x/2^m$, $x_2 = x\%2^m$, and similarly for y. These operations can be implemented by zeroing x's low-order bits and shifting the remaining bits down to get x_1, and by simply zeroing x's high-order bits to get x_2. This shifting and zeroing of bits can be done in linear time in the number of bits, so this "divide" step takes time linear in n.

Note that this decomposition is such that we have:

$$x = 2^m x_1 + x_2$$

and

$$y = 2^m y_1 + y_2$$

So that:

$$xy = x_2 y_2 + 2^m (x_1 y_2 + y_1 x_2) + 2^{2m} x_1 y_1$$

This specifies the product xy that we want, in terms of products of smaller integers - these smaller products can be computed with recursive calls. The above formula for xy makes it look as though we need to do four recursive calls (one each for $x_1 y_1$, $x_2 y_1$, $y_2 x_1$, and $x_2 y_2$) - but if we write down that straightforward divide-and-conquer algorithm we get a $\Theta(n^2)$ method that is no better than the "grade school" algorithm. But, we can do better than this.

First, do two recursive calls to compute $x_1 y_1$ and $x_2 y_2$. Now, note that

$$(x_1 + x_2)(y_1 + y_2) = x_1 y_2 + y_1 x_2 + x_1 y_1 + x_2 y_2$$

so that we can compute this product with one additional multiply, and then get the remaining term we need:

$$x_1 y_2 + y_1 x_2 = (x_1 + x_2)(y_1 + y_2) - x_1 y_1 - x_2 y_2$$

without doing any additional multiplies. That is, we have done three multiplies (plus some additions, which only take linear time in the number of bits), and we have obtained all the information we need to compute xy according to the formula above. This is given in detail in the algorithm below.

MUL(x, y)

1. $n \leftarrow \max(\text{numdigits}(x), \text{numdigits}(y))$
2. $m \leftarrow \lfloor \frac{n}{2} \rfloor$
3. $x_1 \leftarrow x/2^m$
4. $x_2 \leftarrow x\%2^m$
5. $y_1 \leftarrow y/2^m$
6. $y_2 \leftarrow y\%2^m$
7. $a \leftarrow \text{MUL}(x_1, y_1)$
8. $b \leftarrow \text{MUL}(x_2, y_2)$
9. $c \leftarrow \text{MUL}(x_1 + x_2, y_1 + y_2) - a - b$
10. **return** $b + c2^m + a2^{2m}$

The complete running time analysis is a bit tricky, so let's do a rough analysis first to get some idea of the running time. Roughly, for integers of size n, we decompose them into integers of size $\frac{n}{2}$ and do 3 recursive calls plus a linear amount of extra work (using linear-time shift, bit-zeroing, and addition operations as discussed above). Calling our approximately analyzed time $A(n)$, this would give

$$A(n) = 3A(\frac{n}{2}) + \Theta(n).$$

Using the master theorem for recurrences (page 62 of the text): we have $b = 2$, $a = 3$, and $f(n)$ is $\Theta(n)$. So, $\log_b(a) = \log_2(3)$. Since $\log_2(3)$ is about 1.58, we have that $f(n) = \Theta(n^{1.0})$ is $O(n^{\log_b(a)-\epsilon})$ for (picking a somewhat arbitrary value) $\epsilon = 0.01$, and therefore we are in case 1 of the master theorem, and

$$A(n) = \Theta(n^{log_2(3)}).$$

Unfortunately, when computing a sum of two $\frac{n}{2}$ bit integers, such as $x_1 + x_2$, carry may result in a sum that has $\frac{n}{2} + 1$ bits rather than just $\frac{n}{2}$. Thus, some subproblem sizes are a little larger than the above analysis suggests. Simplify the analysis by assuming that ALL subproblems are of size $\frac{n}{2} + 1$ (which doesn't change the asymptotic running time, and which can be easily achieved by padding subproblems with an extra leading zero bit, if necessary). Then our actual running time is

$$T(n) = 3T(\frac{n}{2} + 1) + \Theta(n).$$

We solve this using an intermediate function $T'(n)$, which we define to be equal to $T(n+2)$. Now,

$$\begin{aligned} T'(n) &= T(n+2) \\ &= 3T(\frac{n+2}{2} + 1) + \Theta(n) \\ &= 3T(\frac{n}{2} + 2) + \Theta(n) \\ &= 3T'(\frac{n}{2}) + \Theta(n). \end{aligned}$$

Taking the first and last lines of this, we have

$$T'(n) = 3T'(\frac{n}{2}) + \Theta(n),$$

so $T'(n) = \Theta(n^{log_2(3)})$ by the analysis in the above paragraph. $T(n) = T'(n-2)$ by the definition of T', so

$$T(n) = \Theta((n-2)^{log_2(3)}) = \Theta(n^{\log_2(3)}) \approx \Theta(n^{1.58}).$$