

In lecture on Thursday, we saw a divide-and-conquer algorithm, due to Karatsuba, that multiplies two  $n$ -digit integers using  $O(n^{\lg 3})$  single-digit additions, subtractions, and multiplications. In this lab we'll look at some extensions and applications of this algorithm.

1. Describe an algorithm to compute the product of an  $n$ -digit number and an  $m$ -digit number, where  $m < n$ , in  $O(m^{\lg 3-1}n)$  time.

**Solution:** Split the larger number into  $\lceil n/m \rceil$  chunks, each with  $m$  digits. Multiply the smaller number by each chunk in  $O(m^{\lg 3})$  time using Karatsuba's algorithm, and then add the resulting partial products with appropriate shifts.

```

SKEWMULTIPLY( $x[0..m-1], y[0..n-1]$ ):
   $prod \leftarrow 0$ 
   $offset \leftarrow 0$ 
  for  $i \leftarrow 0$  to  $\lceil n/m \rceil - 1$ 
     $chunk \leftarrow y[i \cdot m .. (i+1) \cdot m - 1]$ 
     $prod \leftarrow prod + \text{MULTIPLY}(x, chunk) \cdot 10^{i \cdot m}$ 
  return  $prod$ 

```

Each call to MULTIPLY requires  $O(m^{\lg 3})$  time, and all other work within a single iteration of the main loop requires  $O(m)$  time. Thus, the overall running time of the algorithm is  $O(1) + \lceil n/m \rceil \cdot O(m^{\lg 3}) = O(m^{\lg 3-1}n)$  as required.

This is the standard method for multiplying a large integer by a single “digit” integer *written in base  $10^m$* , but with each single-“digit” multiplication implemented using Karatsuba's algorithm. ■

2. Describe an algorithm to compute the decimal representation of  $2^n$  in  $O(n^{\lg 3})$  time.

[Hint: Repeated squaring. The standard algorithm that computes one decimal digit at a time requires  $\Theta(n^2)$  time.]

**Solution:** We compute  $2^n$  via repeated squaring, following Piñgala's 2100-year-old recurrence:

$$2^n = \begin{cases} 1 & \text{if } n = 0 \\ (2^{n/2})^2 & \text{if } n > 0 \text{ is even} \\ 2 \cdot (2^{\lfloor n/2 \rfloor})^2 & \text{if } n \text{ is odd} \end{cases}$$

We use Karatsuba's algorithm to implement decimal multiplication for each square.

```

TwoToThe(n):
  if n = 0
    return 1
  m ← ⌊n/2⌋
  z ← TwoToThe(m)    <<recurse!>>
  z ← MULTIPLY(z, z) <<Karatsuba>>
  if n is odd
    z ← ADD(z, z)
  return z

```

The running time of this algorithm satisfies the recurrence  $T(n) = T(\lfloor n/2 \rfloor) + O(n^{\lg 3})$ . We can safely ignore the floor in the recursive argument. The recursion tree for this algorithm is just a path; the work done at recursion depth  $i$  is  $O((n/2^i)^{\lg 3}) = O(n^{\lg 3}/3^i)$ . Thus, the levels sums form a descending geometric series, which is dominated by the work at level 0, so the total running time is at most  $O(n^{\lg 3})$ . ■

3. Describe a divide-and-conquer algorithm to compute the decimal representation of an arbitrary  $n$ -bit binary number in  $O(n^{\lg 3})$  time.

[Hint: Let  $x = a \cdot 2^{n/2} + b$ . Watch out for an extra log factor in the running time.]

**Solution:** Following the hint, we break the input  $x$  into two smaller numbers by setting  $x = a \cdot 2^{n/2} + b$ ; recursively convert  $a$  and  $b$  into decimal; convert  $2^{n/2}$  into decimal using the solution to problem 2; multiply  $a$  and  $2^{n/2}$  using Karatsuba's algorithm; and finally add the product to  $b$  to get the final result.

```

DECIMAL( $x[0..n-1]$ ):
  if  $n < 100$ 
    use brute force
   $m \leftarrow \lceil n/2 \rceil$ 
   $a \leftarrow x[m..n-1]$ 
   $b \leftarrow x[0..m-1]$ 
  return ADD(MULTIPLY(DECIMAL( $a$ ), TwoToThe( $m$ )), DECIMAL( $b$ ))

```

The running time of this algorithm satisfies the recurrence  $T(n) = 2T(n/2) + O(n^{\lg 3})$ ; the  $O(n^{\lg 3})$  term includes the running times of both MULTIPLY and TwoToThe (as well as the final linear-time addition).

The recursion tree for this algorithm is a binary tree, with  $2^i$  nodes at recursion depth  $i$ . Each recursive call at depth  $i$  converts an  $n/2^i$ -bit binary number to decimal; the non-recursive work at the corresponding node of the recursion tree is  $O((n/2^i)^{\lg 3}) = O(n^{\lg 3}/3^i)$ . Thus, the total work at depth  $i$  is  $2^i \cdot O(n^{\lg 3}/3^i) = O(n^{\lg 3}/(3/2)^i)$ . The level sums define a descending geometric series, which is dominated by its largest term  $O(n^{\lg 3})$ .

Notice that if we had converted  $2^{n/2}$  to decimal *recursively* instead of calling TwoToThe, the recurrence would have been  $T(n) = 3T(n/2) + O(n^{\lg 3})$ . Every level of this recursion tree has the same sum, so the overall running time would be  $O(n^{\lg 3} \log n)$ . ■

**Think about later:**

- \*4. Suppose we can multiply two  $n$ -digit numbers in  $O(M(n))$  time. Describe an algorithm to compute the decimal representation of an arbitrary  $n$ -bit binary number in  $O(M(n) \log n)$  time.

**Solution:** We modify the solutions of problems 2 and 3 to use the faster multiplication algorithm instead of Karatsuba's algorithm. Let  $T_2(n)$  and  $T_3(n)$  denote the running times of `TwoToThe` and `DECIMAL`, respectively. We need to solve the recurrences

$$T_2(n) = T_2(n/2) + O(M(n)) \quad \text{and} \quad T_3(n) = 2T_3(n/2) + T_2(n) + O(M(n)).$$

But how can we do that when we don't know  $M(n)$ ?

For the moment, consider the special case where  $M(n) = O(n^c)$  for some constant  $c > 0$ . Since any algorithm to multiply two  $n$ -digit numbers must *read* all  $n$  digits, we have  $M(n) = \Omega(n)$ , and therefore  $c \geq 1$ . On the other hand, the grade-school lattice algorithm implies  $M(n) = O(n^2)$ , so we can safely assume  $c \leq 2$ . With this assumption, the recursion tree method implies

$$\begin{aligned} T_2(n) &= T_2(n/2) + O(n^c) &\implies T_2(n) &= O(n^c) \\ T_3(n) &= 2T_3(n/2) + O(n^c) &\implies T_3(n) &= \begin{cases} O(n \log n) & \text{if } c = 1, \\ O(n^c) & \text{if } c > 1. \end{cases} \end{aligned}$$

So in this case, we have  $T_3(n) = O(M(n) \log n)$  as required.

In general,  $M(n)$  is not necessarily a simple polynomial; however, we can effectively **ignore** any sub-polynomial noise using the following trick. Suppose we can write  $M(n) = n^c \cdot \mu(n)$  for some constant  $c \geq 1$  and some arbitrary non-decreasing function  $\mu(n)$ .<sup>a</sup>

To solve the recurrence  $T_2(n) = T_2(n/2) + O(M(n))$ , we define a new function  $\tilde{T}_2(n) = T_2(n)/\mu(n)$ . Then we have

$$\tilde{T}_2(n) = \frac{T_2(n/2)}{\mu(n)} + \frac{O(M(n))}{\mu(n)} \leq \frac{T_2(n/2)}{\mu(n/2)} + \frac{O(M(n))}{\mu(n)} = \tilde{T}_2(n/2) + O(n^c).$$

Here we used the inequality  $\mu(n) \geq \mu(n/2)$ ; this is the only fact about  $\mu$  that we actually need. The recursion tree method implies  $\tilde{T}_2(n) \leq O(n^c)$ , and therefore  $T_2(n) \leq O(n^c) \cdot \mu(n) = O(M(n))$ .

Similarly, to solve the recurrence  $T_3(n) = 2T_3(n/2) + O(M(n))$ , we define  $\tilde{T}_3(n) = T_3(n)/\mu(n)$ , which gives us the recurrence  $\tilde{T}_3(n) \leq 2\tilde{T}_3(n/2) + O(n^c)$ . The recursion tree method implies

$$\tilde{T}_3(n) \leq \begin{cases} O(n \log n) & \text{if } c = 1, \\ O(n^c) & \text{if } c > 1. \end{cases}$$

In both cases, we have  $\tilde{T}_3(n) = O(n^c \log n)$ , which implies  $T_3(n) = O(M(n) \log n)$ . ■

<sup>a</sup>A very recent multiplication algorithm based on fast Fourier transforms runs in  $O(n \log n)$  time, so we can safely assume that  $c = 1$  and  $\mu(n) = \log n$ . But our solution doesn't use that fact.