Today we will see two divide-and-conquer approaches for fast multiplication. We will see both how multiply two n-bit integers faster than the grade school $O(n^2)$ method you learned, and also we will see how to multiply two $n \times n$ matrices faster than the naïve cubic time multiplication.

1 Karatsuba Multiplication

Say we want to multiply two n-bit numbers: for example, 41×42 (or, in binary, 101001×101010). According to the definition of what it means to multiply, what we are looking for is the result of adding 41 to itself 42 times (or vice versa). You could imagine actually computing the answer that way (i.e., performing 41 additions), which would be correct but not particularly efficient. If we used this approach to multiply two n-bit numbers, we would be making $\Theta(2^n)$ additions. This is exponential in n even without counting the number of steps needed to perform each addition. And, in general, exponential is bad.[1] A better way to multiply is to do what we learned in grade school:

```
       101001 = 41
x      101010 = 42
-------------
      1010010
    101001
+ 101001
-------------
  11010111010 = 1722
```

More formally, we scan the second number right to left, and every time we see a 1, we add a copy of the first number, shifted by the appropriate number of bits, to our total. Each addition takes $O(n)$ time, and we perform at most n additions, which means the total running time here is $O(n^2)$. So, this is a simple example where even though the problem is defined "algorithmically", using the definition is not the best way of solving the problem.

Is the above method the fastest way to multiply two numbers? It turns out it is not. Here is a faster method called Karatsuba Multiplication, discovered by Anatoli Karatsuba, in Russia, in 1962. In this approach, we take the two numbers X and Y and split them each into their most-significant half and their least-significant half:

$X = 2^{n/2}A + B$ | A | B |

$Y = 2^{n/2}C + D$ | C | D |

We can now write the product of X and Y as

$$XY = 2^n AC + 2^{n/2}BC + 2^{n/2}AD + BD. \quad (1)$$

This does not yet seem so useful: if we use (1) as a recursive multiplication algorithm, we need to perform four $n/2$-bit multiplications, three shifts, and three $O(n)$-bit additions. If we use $T(n)$ to

[1] This is reminiscent of an exponential-time sorting algorithm I once saw in Prolog. The code just contains the definition of what it means to sort the input — namely, to produce a permutation of the input in which all elements are in ascending order. When handed directly to the interpreter, it results in an algorithm that examines all $n!$ permutations of the given input list until it finds one that is in the right order.

denote the running time to multiply two n-bit numbers by this method, this gives us a recurrence of

$$T(n) \quad = \quad 4T(n/2) + cn, \tag{2}$$

for some constant c. (The cn term reflects the time to perform the additions and shifts.) This recurrence solves to $O(n^2)$, so we do not seem to have made any progress. (In recitation we will review how to solve recurrences like this — see Appendix A.)

However, we can take the formula in (1) and rewrite it as follows:

$$(2^n - 2^{n/2})AC + 2^{n/2}(A+B)(C+D) + (1-2^{n/2})BD. \tag{3}$$

It is not hard to see — you just need to multiply it out — that the formula in (3) is equivalent to the expression in (1). The new formula looks more complicated, but, it results in only *three* multiplications of size $n/2$, plus a constant number of shifts and additions. So, the resulting recurrence is

$$T(n) \quad = \quad 3T(n/2) + c'n, \tag{4}$$

for some constant c'. This recurrence solves to $O(n^{\log_2 3}) \approx O(n^{1.585})$.

Is *this* method the fastest possible? Again it turns out that one can do better. The first improvements of this were based on splitting the two n-bit numbers into mor than 2 pieces. The Toom-3 algorithm splits numbers into 3 parts and reduces 9 multiplications to 5. Solving the recurrence, this runs in $O(n^{(\log_2 5)/(\log_2 3)})$ time, which is $O(n^{1.46})$ time. More generally, the Toom-k algorithm splits numbers into k parts and runs in $O(c(k)n^{(\log_2(2k-1))/(\log_2 k)})$, where $c(k)$ is a rapidly growing function of k. One way of optimizing this gives $O(n \cdot 2^{\sqrt{2\log_2 n}} \log n)$ time. Note that $2^{\sqrt{2\log_2 n}}$ grows slower than n^ϵ for any constant $\epsilon > 0$.

Later, Karp discovered a way to use the so-called Fast Fourier Transform (FFT) to multiply two n-bit numbers in time $O(n \log^2 n)$. Schönhage and Strassen in 1971 improved this to $O(n \log n \log\log n)$, which was until recently the asymptotically fastest algorithm known. Fürer in 2007 improved this by replacing the $\log\log n$ term with $2^{O(\log^* n)}$, where $\log^* n$ is a very slowly growing function discussed in Lecture 6. And very recently, Harvey and van der Hoeven achieved $O(n \log n)$ time! We will discuss the FFT later on in this course.

Actually, the kind of analysis we have been doing really is meaningful only for very large numbers. On a computer, if you are multiplying numbers that fit into the word size, you would do this in hardware that has gates working in parallel. So instead of looking at sequential running time, in this case we would want to examine the size and depth of the circuit used, for instance. This points out that, in fact, there are different kinds of specifications that can be important in different settings.

2 Strassen's algorithm for matrix multiplication

It turns out the same basic divide-and-conquer approach of Karatsuba's algorithm can be used to speed up matrix multiplication as well. To be clear, we will now be considering a computational model where individual elements in the matrices are viewed as "small" and can be added or multiplied in constant time. In particular, to multiply two n-by-n matrices in the usual way (we take the ith row of the first matrix and compute its dot-product with the jth column of the second matrix in order to produce the entry ij in the output) takes time $O(n^3)$. If one breaks down each n by n

matrix into four $n/2$ by $n/2$ matrices, then the standard method can be thought of as performing eight $n/2$-by-$n/2$ multiplications and four additions as follows:

$$\begin{array}{|c|c|}\hline A & B \\ \hline C & D \\ \hline \end{array} \times \begin{array}{|c|c|}\hline E & F \\ \hline G & H \\ \hline \end{array} = \begin{array}{|c|c|}\hline AE+BG & AF+BH \\ \hline CE+DG & CF+DH \\ \hline \end{array}$$

Strassen noticed that, as in Karatsuba's algorithm, one can cleverly rearrange the computation to involve only *seven* $n/2$-by-$n/2$ multiplications (and 14 additions).

In particular, the quantities that one computes recursively are $q_1 = (A+D)(E+H)$, $q_2 = D(G-E)$, $q_3 = (B-D)(G+H)$, $q_4 = (A+B)H$, $q_5 = (C+D)E$, $q_6 = A(F-H)$, and $q_7 = (C-A)(E+F)$. The upper-left quadrant of the solution is $q_1 + q_2 + q_3 - q_4$, the upper-right is $q_4 + q_6$, the lower-left is $q_2 + q_5$, and the lower right is $q_1 - q_5 + q_6 + q_7$. (feel free to check!).

Since adding two n-by-n matrices takes time $O(n^2)$, this results in a recurrence of

$$T(n) \quad = \quad 7T(n/2) + cn^2. \tag{5}$$

This recurrence solves to a running time of just $O(n^{\log_2 7}) \approx O(n^{2.81})$ for Strassen's algorithm. According to Manuel Blum, Strassen said that when coming up with his algorithm, he first tried to solve the problem mod 2. Solving mod 2 makes the problem easier because you only need to keep track of the parity of each entry, and in particular, addition is the same as subtraction. One he figured out the solution mod 2, he was then able to make it work in general.

Matrix multiplication is especially important in scientific computation. Strassen's algorithm has more overhead than standard method, but it is the preferred method on many modern computers for even modestly large matrices. Asymptotically, the best matrix multiply algorithm known for a long time was by Coppersmith and Winograd and has time $O(n^{2.376})$, but is not practical. Recently, there have been several improvements, including those by Stothers and Williams which achieved $O(n^{2.3736})$ and $O(n^{2.3727})$ time, respectively.

Nobody knows if it is possible to do better — the FFT approach doesn't seem to carry over.