Lecture 13

Numerical algorithms: Today we'll cover algorithms for various numerical problems: integer multiplication, powering, and solving linear systems of equations.

Integer Multiplication: We all learned how to multiply numbers with lots of digits when we were kids:

$$\begin{array}{r} 435 \\ 213 \\ \hline 1305 \\ 435 \\ 870 \\ \hline 92655 \end{array}$$

Suppose the integers were given in input as lists of their digits, from left to right. Then the above method corresponds to the following code.

```
# takes two lists of digits representing integers a,b and outputs a
# list of digits corresponding to their product
def integerMultiply(a, b):
    # we will put the result in c
    c = [0]*(len(a) + len(b))
    for i in xrange(len(b)):
        atB = len(b) - i - 1
        # multiply b[atB] by a and put it in t
        t = [0]*(len(a) + 1)
        # keep track of the carry digit
        carry = 0
        for j in xrange(len(a)):
            atA = len(a) - j - 1
            product = b[atB]*a[atA] + carry
            t[len(t) - 1 - j] = product % 10
            carry = product / 10
        t[0] = carry
        # now add t, shifted over by i to the left, to c
        carry = 0
        for j in xrange(len(t)):
            sum = c[len(c) - 1 - j - i] + t[len(t) - 1 - j] + carry
            c[len(c) - 1 - j - i] = sum % 10
            carry = sum / 10
    # strip away the leading zeroes from c before returning it
```

```
at = 0
while at<len(c) and c[at]==0:
    at += 1
if at == len(c):
    return [0]
else:
    return c[at:]
```

In fact, the above code is more memory-efficient than the grade school way of multiplying we learned, since it adds to the result after processing every digit in b rather than waiting to process all digits then add up everything at the end. If both numbers have n digits, the running time of the above implementation is $\Theta(n^2)$ (two nested `for` loops each taking n steps), and the memory usage is $\Theta(n)$.

How can we speed this up? Let's try the *divide and conquer* method. Divide and conquer is a strategy based on dividing up the input into smaller pieces, solving the problem on the smaller pieces, then combining the result to get the answer for the full input. We did this for example with `mergeSort`: to sort a list we divided it into two smaller pieces, recursively solved (i.e. sorted) the smaller pieces, then merged to get the result for the overall input.

So, let's say we're trying to multiply a `list` of digits a by another `list` of digits b, each of length n. For the sake of simplifying all future discussion, let's assume n is a perfect power of 2 (if not, we can pad both a and b by 0s at their beginnings until their lengths *are* powers of 2, and doing this at most doubles n). Let a_{high} represent the first half of the digits of a, i.e. $a[:n/2]$, and let a_{low} represent $a[n/2:]$. Then, treating a as an integer, $a = a_{\text{high}} \cdot 10^{n/2} + a_{\text{low}}$. Doing similarly for b, this means that

$$\begin{aligned} a \times b &= (a_{\text{high}} \cdot 10^{n/2} + a_{\text{low}}) \times (b_{\text{high}} \cdot 10^{n/2} + b_{\text{low}}) \\ &= a_{\text{high}} \cdot b_{\text{high}} \cdot 10^n + (a_{\text{high}} \cdot b_{\text{low}} + a_{\text{low}} \cdot b_{\text{high}}) \cdot 10^{n/2} + a_{\text{low}} \cdot b_{\text{low}} \end{aligned}$$

In other words, to multiply two n-digit numbers, we just need to multiply four pairs of $n/2$-digit numbers, shift some results over by either $n/2$ or n (this is what multiplying by a power of 10 does), then add up the results. Shifting n-digit numbers over and adding them takes $\Theta(n)$ time. When $n = 1$, we can just do the multiplication in constant time. Thus, if $T(n)$ is the running time to multiply two n-digit numbers, we have the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 4 \cdot T(n/2) + \Theta(n) & \text{otherwise} \end{cases}$$

If we draw out the recursion tree for T, labeling each node with how much work needs to be done there, the root does at most Cn work for some constant C. It then has 4 children, each doing $Cn/2$ work. Each of those children then again has 4 children, each doing $Cn/4$ work. This goes on for $\log_2 n$ levels of the tree, at which point n is finally 1. Thus the total work is $\sum_{i=0}^{\log_2 n} 4^i \cdot (Cn/2^i) = Cn \cdot \sum_{i=0}^{\log_2 n} 2^i$. Recalling that $\sum_{i=0}^{t} x^i = (x^{t+1} - 1)/(x - 1)$ for $x \neq 1$, this means that the total work is $Cn \cdot (2^{1+\log_2 n} - 1) = 2Cn^2 - Cn = \Theta(n^2)$. In other words, the obvious recursive divide-and-conquer approach for this problem has no benefit over the way we learned to solve the problem in grade school.

Karatsuba's algorithm Anatolii Alexeevitch Karatsuba in 1960 found a way to make the divide-and-conquer approach work for speeding up integer multiplication. The story goes that Andrey Kolmogorov, a giant of probability theory and other areas of mathematics, had a conjecture from 1956 stating that it is impossible to multiply two n-digit numbers in faster than $\Omega(n^2)$ time. In 1960 Kolmogorov told many scientists his conjecture at a seminar at Moscow State University, and Karatsuba, then in the audience, went home and disproved Kolmogorov's conjecture in exactly one week[1]. Let's now cover the method he came up with.

Let $X = a_{\text{high}} \cdot b_{\text{high}}$, $Y = a_{\text{low}} \cdot b_{\text{low}}$, and $Z = (a_{\text{high}} + a_{\text{low}}) \cdot (b_{\text{high}} + b_{\text{low}})$. Then

$$a \times b = X \cdot 10^n + (Z - X - Y) \cdot 10^{n/2} + Y.$$

Thus, now, to multiply two n-digit numbers we only need to multiply *three* pairs of $n/2$-digit numbers. This gives the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 3 \cdot T(n/2) + \Theta(n) & \text{otherwise} \end{cases}.$$

Now if we draw out the recursion tree for T, labeling each node with how much work needs to be done there, the root does at most Cn work for some constant C. It then has 3 children, each doing $Cn/2$ work. Each of those children then again has 3 children, each doing $Cn/4$ work. This goes on for $\log_2 n$ levels of the tree, at which point n is finally 1. Thus the total work is $\sum_{i=0}^{\log_2 n} 3^i \cdot (Cn/2^i) = Cn \cdot \sum_{i=0}^{\log_2 n} (3/2)^i$. Recalling that $\sum_{i=0}^{t} x^i = (x^{t+1} - 1)/(x - 1)$ for $x \neq 1$, this means that the total work is $Cn \cdot ((3/2)^{1+\log_2 n} - 1) = 1.5Cn^{\log_2 3} - Cn = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$.

In fact it is possible to get integer multiplication algorithms with running times just a bit larger than $n \log n$ using what's known as the *Fast Fourier Transform*, but we will not cover these methods in this course.

Powering: Another important numerical algorithm is that for *powering*. Suppose we have two integers a, n and would like to compute a^n. The obvious way would be something like the following:

```
def power(a, n):
    ans = 1
    while n > 0:
        ans *= a
        n -= 1
    return ans
```

The above method does n integer multiplications in order to compute a^n. A faster method is given below, which is known as the method of *repeated squaring*.

```
def power(a, n):
    if n == 0:
        return 1
```

[1] See A. A. Karatsuba. The complexity of computations. Proceedings of the Steklov Institute of Mathematics, Vol. 211, pp. 169–183, 1995.

```
b = power(a, n/2)
b *= b
if n % 2 == 1:
    b *= a
return b
```

That is, to raise a to the nth power, we just need to raise it to the $\lfloor n/2 \rfloor$th power then square the result (and multiply in an extra factor of a if n was odd). The repeated squaring method performs at most $2 \log_2 n = O(\log_2 n)$ integer multiplications to compute a^n.

The method of course does not only work for raising integers to integer powers. We could also use it, for example, to raise matrices to integer powers. Recall that an $n \times m$ matrix is a 2D-grid of numbers, with n rows and m columns. The definition of multiplication of matrices is that if A is $n \times m$ and B is $m \times p$, then AB is $n \times p$ where the (i, j)th entry of AB is equal to $\sum_{k=1}^{m} A_{i,k} \cdot B_{k,j}$. Let F_i be the ith Fibonacci number. Consider the following matrix product:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_i \\ F_{i-1} \end{bmatrix} = \begin{bmatrix} F_i + F_{i-1} \\ F_i \end{bmatrix}.$$

If we let A be the matrix on the lefthand side above, then

$$A^n \cdot \begin{bmatrix} F_1 \\ F_0 \end{bmatrix} = \begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix}$$

Thus we can compute the nth Fibonacci number in just $O(\log_2 n)$ integer multiplications by doing repeated squaring to calculate A^{n-1} then multiplying the result by a 2×1 matrix then reading off the result.

Gaussian Elimination: The last thing we'll cover today is solving systems of linear equations. The most well known algorithm for doing this is called *Gaussian Elimination*, named after the 19th century mathematician Carl Friedrich Gauss, though it was named after him by mistake. The algorithm appeared roughly 2000 years ago in the 19th century Chinese math book "The Nine Chapters on the Mathematical Art", and was rediscovered in Europe by Isaac Newton (one of the creators of calculus) in the late 1600s.

So what is the method? Suppose we have n variables $x_1, \ldots, x_n$. We also have n equations, the ith of which is of the form $\sum_{j=1}^{n} A_{i,j} x_j = b_i$. We can write down this linear system of equations in matrix form. A is an $n \times n$ matrix, where the ith row contains all the coefficients $A_{i,j}$. x is an $n \times 1$ matrix (usually called a vector) where the ith entry is x_i. Then b is a length-n vector with its ith entry being b_i, and we would like to find an x such that $Ax = b$ is true.

Let A_i denote the ith row of i. The key to Gaussian elimination is just noting that if $A_i x = b_i$ and $A_j x = b_j$ are both true, then so is any linear combination, i.e. $(c_1 A_i + c_2 A_j)x = c_1 b_i + c_2 b_j$ is also true. Also, if A was a diagonal matrix (all the entries under the top-left to bottom-right diagonal in A are 0), we could easily tell what the x_i should be. So, we keep doing row simplification on the matrix A until it becomes diagonal, then read off the answer for x.

Here's an example of a diagonal matrix:

$$\begin{bmatrix} 1 & 1 & 0 & 10 \\ 0 & -4 & 2 & -1 \\ 0 & 0 & 7 & 12 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We now give the code.

```
def gaussianElimination(A, b):
    n = len(A)
    x = [0]*n
    for k in xrange(n):
        # try to make the kth row of A only have entries from k onward
        # but first, find a row that has some non-zero entry in the
        # kth column, and swap it to make it the kth row
        pivot = k;
        for i in xrange(k, n):
            if A[i][k] != 0:
                pivot = i
                break
        if pivot != k:
            b[k],b[pivot] = b[pivot],b[k]
            for i in xrange(k, n):
                A[k][i],A[pivot][i] = A[pivot][i],A[k][i]
        # now change the ith row of A to zero out its kth column by
        # adding a multiple of the kth row to it
        for i in xrange(k+1, n):
            factor = -A[i][k] / A[k][k]
            b[i] = b[i] + factor*b[k]
            for j in xrange(k, n):
                A[i][j] = A[i][j] + factor * A[k][j]
    # now A is diagonal and its easy to solve for x
    for ii in xrange(n):
        i = n - ii - 1
        x[i] = b[i]
        for j in xrange(i+1, n):
            x[i] = x[i] - A[i][j]*x[j]
        x[i] = x[i] / A[i][i]
    return x
```