CS294-5 Great Algorithms **Spring 2006**

Lecture 4: 1.30.06

Lecturer: Karp *Scribes: Norm Aleks & Jason Wolfe*

News Flash

A (randomized) polynomial-time simplex algorithm, which has been sought after for many years, has finally been discovered. When it comes time to write a final report, the this might be a good topic to consider. For details, see J.A. Kelner and D.A. Spielman, "A randomized polynomial simplex algorithm for linear programming."

1 Signal analysis using the FFT

Let $F(t) : \mathbb{Z} \to \mathbb{R}$ be a real-valued periodic function with period $N = 2^k$. Then, F is completely determined by $F(0), F(1), \ldots F(N-1)$. We want to perform a Fourier analysis of F, expressing it as a superposition of sinusoids of periods $N, N/2, N/3, \ldots 1$:

$$F(t) = \sum_{j=1}^{N} c_j \cdot \cos(\frac{2\pi jt}{N} + a_j),$$

for some $c_j, a_j \in \mathbb{R}$.

More generally, if we allow $F(t) : \mathbb{Z} \to \mathbb{C}$ to take on complex values (but still require periodicity with period $N = 2^k$), we can instead express it as a superposition of complex exponentials of the form

$$A \cdot e^{i(\frac{2\pi jt}{N} + \theta)},$$

where $A \in \mathbb{C}$ and $\theta \in \mathbb{R}$. Recall that functions of this form describe a type of simple harmonic motion, rotation around the origin (of the complex plane) with constant radius and angular speed. Furthermore, the projection of this motion onto the real axis is a sinusoid (by Euler's formula, $e^{i\theta} = \cos(\theta) + i \cdot \sin(\theta)$):

$$A \cdot \cos(\frac{2\pi jt}{N} + \theta).$$

Now, defining $\omega = e^{\frac{2\pi i}{N}}$ as a simple Nth root of unity, we can represent a simple harmonic motion of period N/j as simply $c * \omega^{jt}$, where the angle θ has been absorbed into the complex constant c. Under this alternative representation, F can be expressed as a function of its Fourier coefficients as follows:

$$F(t) = \sum_{j=0}^{N-1} c_j \cdot \omega^{jt}.$$

This is just a linear transformation from c_j to $F(t)$, which can be rewritten in matrix-vector form as

$$\begin{bmatrix} F(0) \\ F(1) \\ \vdots \\ F(N-1) \end{bmatrix} = \left(\omega^{jt}\right) * \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{N-1} \end{bmatrix},$$

where $\left(\omega^{jt}\right)$ is shorthand for

$$\begin{bmatrix} \omega^{0\cdot 0} & \omega^{1\cdot 0} & \dots & \omega^{(N-1)\cdot 0} \\ \omega^{0\cdot 1} & \omega^{1\cdot 1} & \dots & \omega^{(N-1)\cdot 1} \\ \vdots & \vdots & \ddots & \vdots \\ \omega^{0\cdot (N-1)} & \omega^{1\cdot (N-1)} & \dots & \omega^{(N-1)\cdot (N-1)} \end{bmatrix}.$$

Hence,

$$\begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{N-1} \end{bmatrix} = \left(\omega^{jt}\right)^{-1} * \begin{bmatrix} F(0) \\ F(1) \\ \vdots \\ F(N-1) \end{bmatrix}.$$

In the last lecture, we saw that $\left(\omega^{jt}\right)^{-1} = \frac{1}{N}\left(\omega^{-jt}\right)$. Thus, the Fourier coefficients of F can be computed in only $O(N \log N)$ complex arithmetic operations using the FFT. For the special case when F is real-valued, we have instead

$$F(t) = \sum_{j=0}^{N-1} Re(c_j \cdot \omega^{jt}),$$

where $Re(x)$ stands for the real part of x. Despite the fact that in general the c_j are complex numbers, it can be easily verified that $Re(c_j \cdot w^{jt})$ is a sinusoid. The proof of this fact is left as an exercise to the reader.

2 Fast integer multiplication

Suppose we have two n-bit numbers, u and v, which we wish to multiply to obtain a $2n$-bit product. The primitive operations we will count in determining time complexity are those available on a basic computer: shifts and 2-input *or, and,* and *not.*

2.1 The schoolbook method

The obvious "school book" method, adapted to base 2, requires $\Theta(n^2)$ operations (a constant number of *or*'s and *and*'s per pair of digits in u and v, plus n shifts). For example, multiplying $(10110)_2 \times (11101)_2$:

					1	0	1	1	0
				×	1	1	1	0	1
					1	0	1	1	0
			1	0	1	1	0		
		1	0	1	1	0			
	1	0	1	1	0				
1	0	0	1	1	1	1	1	1	0

2.2 The Karatsuba method

In 1962 the Russian mathematician Karatsuba described a faster method that recursively splits terms into high-order and low-order parts, then uses a trick to find the split numbers' product with three recursive calls rather than four. Again using u and v, let $u = \mathbf{u}_1 \cdot 2^{n/2} + \mathbf{u}_0$ and $v = \mathbf{v}_1 \cdot 2^{n/2} + \mathbf{v}_0$. Now

$$uv = \mathbf{u}_1\mathbf{v}_1 \cdot 2^n + (\mathbf{u}_1\mathbf{v}_0 + \mathbf{u}_0\mathbf{v}_1) \cdot 2^{n/2} + \mathbf{u}_0\mathbf{v}_0 \quad (1)$$

which apparently has four multiplications—a wash if multiplication is $\mathrm{O}(n^2)$, since the terms are half as long. The trick is that by rearranging

$$(\mathbf{u}_1 + \mathbf{u}_0)(\mathbf{v}_1 + \mathbf{v}_0) = \mathbf{u}_1\mathbf{v}_1 + \mathbf{u}_1\mathbf{v}_0 + \mathbf{u}_0\mathbf{v}_1 + \mathbf{u}_0\mathbf{v}_0$$

into

$$\mathbf{u}_1\mathbf{v}_0 + \mathbf{u}_0\mathbf{v}_1 = (\mathbf{u}_1 + \mathbf{u}_0)(\mathbf{v}_0 + \mathbf{v}_1) - \mathbf{u}_1\mathbf{v}_1 - \mathbf{u}_0\mathbf{v}_0$$

we express the middle part of Equation 1, including its two multiplications, in terms of Equation 1's left and right terms (which we already had to calculate), one new multiplication, and four additions. Thus the total cost of calculating Equation 1 is two shifts, six additions, and three multiplications, which we perform as recursive calls to the procedure. Letting $T(n)$ be the number of operations required to multiply two n-bit numbers,

$$\begin{aligned} T(n) &= \mathrm{O}(n) + 3T(n/2) \\ &= \mathrm{O}(n^{log_2 3}) \end{aligned}$$

2.3 Schönhage and Strassen's method

Now that we've used the Fast Fourier Transform to multiply polynomials in $\mathrm{O}(n \log n)$, it is natural to think of integers as a special case of polynomials and use the FFT to multiply them. In 1971, Schönhage and Strassen[1] described a method for doing this efficiently.

As before, let the factors u and v be n-digit numbers. We will partition each into 2^k l-bit blocks, perform an FFT on the resulting 2^k-value vectors (which, in signal processing terms, would be in the time domain), do the multiplication in the "frequency domain," and then interpolate to give a $2n$-bit answer. The choice of k and l turns out to be important for efficiency and accuracy, as will be described below; for now, we require that $2n \leq 2^k l \leq 4n$ and note that for 8,192-bit inputs on current hardware, $k = 11$ and $l = 8$ are good choices.

2.3.1 Notation

As stated, u and v are our input numbers, each n bits long. We partition each into 2^k l-bit blocks, padded with leading zeros as necessary. For convenience, define $K = 2^k$ and $L = 2^l$. Let us now refer to the padded, partitioned u and v as vectors of K base-L digits, $\mathbf{u}_0 \ldots \mathbf{u}_{K-1}$ and $\mathbf{v}_0 \ldots \mathbf{v}_{K-1}$, where $\mathbf{u}_0$ and $\mathbf{v}_0$ represent the least significant bits. Now

$$u = \sum_{j=0}^{K-1} \mathbf{u}_j L^j \qquad \text{and} \qquad v = \sum_{j=0}^{K-1} \mathbf{v}_j L^j$$

In this new shape it's clear that uv can be calculated as a convolution of coefficients, and is a candidate for calculation with the FFT:

$$uv = \sum_{j=0}^{K-1} \mathbf{w}_j L^j \qquad \text{where} \qquad \mathbf{w}_j = \sum_{k=0}^{j} \mathbf{u}_k \mathbf{v}_{j-k}$$

2.3.2 Calculating the K roots of unity

To calculate the FFT we first need to calculate the values of the K roots of unity. Our earlier choice of K as a power of two makes this easier: starting with $\omega_K^{2^k} = 1$, we can progress through $\omega_K^{2^{k-1}} = -1$, $\omega_K^{2^{k-2}} = i$, etc., taking square roots repeatedly until we reach ω_K^1. The magnitude of every ω^n is one, so both ω_K^{n+1} and ω_K^n lie on the unit circle in the complex plane; since multiplying complex numbers adds their angles, ω_K^n's angle is half ω_K^{n+1}'s. Call ω_K^{n+1}'s angle 2α and remember that $e^{i2\alpha} = \cos 2\alpha + i \sin 2\alpha$; what we must find, then, is $e^{i\alpha} = \cos\alpha + i \sin\alpha$. From high school trigonometry, $\cos 2\alpha = \cos^2\alpha - \sin^2\alpha = 2\cos^2\alpha - 1$, giving

[1] Arnold Schönhage and Volker Strassen, "Schnelle Multiplikation großer Zahlen," *Computing* 7 (1971), 281–292.

$$\cos\alpha = \sqrt{\frac{\cos 2\alpha + 1}{2}} \qquad \text{and} \qquad \sin\alpha = \sqrt{1 - \cos^2\alpha}$$

Since we are in the first quadrant from i on, we always take the positive square root.

When this process is complete we will have generated all of ω_K^K, $\omega_K^{K/2}$, $\omega_K^{K/4}$, ..., ω_K^1. We can now use these values to generate a table of *all* powers of ω_K up to ω_K^{K-1} at a cost of $\mathrm{O}(K)$ multiplications, because each ω_K^n is the product of some $\omega_K^{r^2}$ and another ω_K^m. For example, $\omega_K^{11} = \omega_K^8 \times \omega_K^2 \times \omega_K^1$ and would be calculated as $\omega_K^{10} \times \omega_K^1$; in the prior round, we would have calculated $\omega_K^{10} = \omega_K^8 \times \omega_K^2$.

2.3.3 Calculating uv

Using our table of the K roots of unity, we now use the methods we've already described to calculate the complex-valued FFT's of u and v, multiply corresponding magnitudes to find the point-value version of uv, and finally interpolate back to $\mathbf{w}$, the coefficient representation of uv. The process requires $\mathrm{O}(Kk) = \mathrm{O}(n \log n)$ multiplications, or $\mathrm{O}(n \log n\, T(m))$ bit operations where $T(m)$ is the number of bit operations required to multiply m-bit numbers.

With the $\mathbf{w}_j$ in hand, we are now ready to calculate uv as we described in Section 2.3.1:

$$uv = \sum_{j=0}^{K-1} \mathbf{w}_j \cdot 2^{lj} \tag{2}$$

What is the time complexity of this operation? Though we calculated the $\mathbf{w}_j$ via a Fourier transform, the values are the same we'd have reached by convolving $\mathbf{u}$ and $\mathbf{v}$:

$$\mathbf{w}_j = \sum_{k=0}^{j} \mathbf{u}_k \mathbf{v}_{j-k}$$

The $\mathbf{u}_n$ and $\mathbf{v}_n$ are l-bit numbers and $j \le 2^k$, so the length of any $\mathbf{w}_n \le 2l + k$. The multiplication by a power of two in Equation 2 is a simple shift, so we are left with a series of additions in an integer 2^k bits long, with any given column of bits up to $2l + k$ bits high; the final cost is $\mathrm{O}(2^k(k + l))$, or $\mathrm{O}(n)$ under our initial condition that $2n \le 2^k l \le 4n$.

2.3.4 How we choose k and l

Though both our math operations and our table of ω_K^n have finite precision, we can still calculate uv exactly by making approximate computations and then rounding results to the nearest integer. An error analysis, which we do not perform here, tells us that for correct answers after rounding we need to keep m bits of precision where $m \ge 3k + 2l + \log k + \frac{7}{2}$. Thus our earlier choice of $k = 11$ and $l = 8$: this combination allows $n = 8,192$ and requires $m \ge 56$, meaning that 64-bit precision suffices to multiply 8,192-bit integers.

2.3.5 Time complexity with arbitrary-length inputs

As noted in the previous section, 64-bit precision is adequate if we limit our inputs to $< 2^{8192}$; on current hardware we may then be able to do the math in hardware, and the total algorithm's complexity is $\mathrm{O}(n \log n)$ hardware operations. But suppose we have longer inputs or no hardware multiply—in that case we can use the algorithm recursively. Using the time estimates above and letting $T(n)$ be the number of operations required to multiply n-bit numbers, we get $T(n) = \mathrm{O}(n) + \mathrm{O}(KkT(m))$. If $k = l$ then $Kk = n$ and $m \le 6 \log n$. Therefore,

$$T(n) = \mathrm{O}(nT(6 \log n))$$

It follows that, for some c,

$$T(n) \le cnT(c \log n)$$

which unwinds to

$$T(n) \leq cn(c \log n)(c \log \log n) \ldots$$

With a more careful analysis of a variant of this algorithm, Schönhage and Strassen achieved a bound of $O(n \log n \log \log n)$ on the number of steps to multiply n-digit numbers.

3 Introduction to stable matching

Consider the problem of matching n men with n women[2] in a 1-1 fashion, such that their preferences are respected as much as possible. Assume that the men are named $A, B, C, \ldots$ and the women are named $a, b, c, \ldots$ (or vice-versa, if you prefer), and every individual expresses his/her preferences as a total order over the members of the opposite sex.

Now, consider an arbitrary 1-1 matching between the men and women. Such a matching exhibits *instability* if there exist two individuals, who we will call A and b, who prefer one another to their current partners. For example, perhaps A is paired with a and b is paired with B in the matching, but A prefers b to a and b prefers A to B. We call a matching *stable* if does not exhibit instability, and thus no individual has any incentive to leave his/her current partner.

In fact, a simple procedure called the *proposal algorithm* exists to efficiently construct a stable matching given a set of preferences:

1. At each step, an unmatched man proposes to the first woman in his preference ordering who has not yet rejected him.
2. When a woman receives a proposal from a man, she *rejects* it iff she is currently paired with a man whom she prefers. Otherwise, she *tentatively accepts* the proposal.
3. The algorithm terminates and the partnerships become final when all n men and women have tentative partners.

This algorithm always terminates with a stable matching. Furthermore, the resulting matching will be optimal from the point of view of the proposers (here the men), in that each proposer ends up with the best partner possible in *any* stable matching. Proofs of these facts are left to the reader.

The operation of the proposal algorithm can be best illustrated by an example. If the men and women have the following preferences (listed best-first):

	1	2	3	4
A	b	c	a	d
B	a	b	d	c
C	c	d	a	b
D	b	a	c	d

	1	2	3	4
a	D	B	A	C
b	D	A	C	B
c	D	C	A	B
d	D	A	B	C

,

then the algorithm will proceed as follows:

- Initially, all men and women are unmatched.
- A proposes to his first choice b, who tentatively accepts since she is currently unmatched.
- B proposes to his first choice a, who tentatively accepts since she is unmatched.
- C proposes to his first choice c, who tentatively accepts since she is unmatched.
- D proposes to his first choice b. Since b likes D better than her current partner A, she rejects A and tentatively accepts the proposal of D.
- A, now back on the market, proposes to his next choice c. Since c prefers her current partner C to A, she rejects his proposal.

[2] Or, more practically, matching n medical students to n slots at k medical schools

- A proposes to his next choice a. Since a prefers her current partner B to A, she rejects his proposal.
- A proposes to his last choice d, who tentatively accepts since she is unmatched.
- Since all men and women are paired, the algorithm terminates, returning the current matching.

The reader can easily verify that this matching reached by the proposal algorithm is, in fact, stable.

It is important to note that the proposal algorithm only works for constructing *bipartite matchings* between disjoint sets of individuals. In fact, in the related *stable roommates* problem where individuals express their preferences over all other individuals, it may be the case that *no stable matchings* exist at all.