CSE 202 NOTES FOR OCTOBER 8, 2002

Preface

The FFT was voted to be one of the "algorithms of the century" in a recent Science article. It has led to many improvements in lots of different fields, and can generally be considered just an all-around cool thing.

Multiplication and the FFT

Integer multiplication is a curious beast. Every algorithm we have thought of so far is $O(n^2)$, and that's no good. We will see an algorithm today that multiplies integers in $O(n \log n)$ time.

Let's ignore numbers for the moment and just think about polynomials. When we need to multiply two polynomials P_A and P_B, we need to multiply all terms together and then add like-degreed terms. This is obviously a long drawn-out process, taking $O(n^2)$ time. Of course, this is only true when we're talking about polynomials that are represented as sequences of coefficients; we know from algebra that any polynomial of degree n can be represented by $n+1$ points that lie on the polynomial curve. If we are given $2n$ of these points for each of the polynomials P_A and P_B, multiplication is relatively easy: simply multiply each of these points pairwise $(x_i, y_{iA}y_{iB})$ and you have the new polynomial. We need $2n$ points, clearly, because the product polynomial P_AP_B will be of degree $2n$, give or take. The problem is that if you are given a coefficient list and you want a coefficient list out the other end of this algorithm, then this fact doesn't help very much.

To relate this to the problem of multiplying integers, note that the number 10230 is the same as $1x^4 + 0x^3 + 2x^2 + 3x + 0$ where $x = 10$. Thus, if we can take two integers formulated in this polynomial fashion, turn them into "point-value representation", do a pairwise multiplication, and then convert them back to coefficient representation then we will have multiplied them. Of course, that doesn't seem like it's any easier than the naive shift-and-add methods, but let's just give it a whirl.

The basic idea is that we will need to create a coefficient representation for a polynomial, evaluate it at a number of points, do pairwise multiplication, and then convert it back to coefficients. We have some latitude in exactly which points we use to evaluate the polynomial—by choosing the complex roots of unity we end up with very nice evaluation properties. In fact, the evaluation of a polynomial of degree m at the m complex roots of unity is called the DFT, or Discrete Fourier Transform. Why this is, I don't know. I alluded to this earlier: the issues here run deep, my friend, whereas you probably do not.

The FFT, then, is a divide and conquer approach to evaluating a polynomial at the roots of unity.

To show the motivation for this idea from a mathematical standpoint, consider the following:

$$\begin{aligned} A &= r10^{n/2} + s \\ B &= t10^{n/2} + u \\ AB &= rt10^{n} + (ru + st)10^{n/2} + su \end{aligned}$$

In this process, $T(n) = 4T(n/2) + O(n) \in O(n^2)$. Let's construct a polynomial P_A from A such that $A = P_A(10^{n/2})$ and similarly for B. Multiply the polynomials, evaluate at $10^{n/2}$ and we'll have multiplied A and B. Consider what would happen if we did some sort of approach like Strassen's algorithm: multiply $(r+s)(t+u) = P_A(1)P_B(1)$. We could also choose 0 as an evaluation point, which gives us su, and we could also use -1 as a point, which gives us $(s-r)(u-t)$. We can get the middle coefficient of P_AP_B by taking $1/2P_AB(1) - 1/2P_AB(-1)$, and do something similar for the first and third coefficients. Here, $T(n) = 3T(n/2)+O(n)$, which falls into the bottom-heavy case and off the cuff we can say that $T(n) \in O(n^{\log_3 2}) = O(n^{\approx 1.7})$. Better than $O(n^2)$, but not great.

Let's do *reducto ad absurdum*: if we broke this into 2 pieces and it gave us some improvement, then breaking it into 3 pieces should give us an even better improvement. In this case, we define a P_A and P_B like this:

$$\begin{aligned} P_A &= a_2 10^{2n/3} + a_1 10^{n/3} + a_0 \\ P_B &= b_2 10^{2n/3} + b_1 10^{n/3} + b_0 \\ P_AB &= P_AP_B \\ P_AB(X) &= AB \text{ when } X = 10^{n/3} \end{aligned}$$

The process that we'll go through is:

Construction of the polynomial: Determining the a coefficients and b coefficients.

Evaluating the polynomial: Evaluate the new polynomials at a small number of points, multiply pointwise.

Interpolate: Based on the two steps above, figure out what the value is at the later point.

So far, we don't have a good way of picking the small number of points that you evaluate at, but that doesn't matter in light of a larger problem: when the degrees get to near 10^x for some value of x, then this just turns into a shift-and-add approach. In the case above, P_AP_B will be a quartic polynomial, so we'll need 5 points. Try $\{0, 1, -1, 2, -2\}$. Again, we can see the expression that corresponds to $P_AB(1)$ is $(a_2 + a_1 + a_0)(b_2 + b_1 + b_0)$, and if we did something similar to the other evaluation points we could do some algebra, but let's not get carried away here. The point is that each subproblem consists of about $n/3$ digits, and there are 5 of them. Evaluations are shifts and adds, which takes $O(n)$ time, so we end up with $T(n) = 5T(n/3) + O(n)$ which works out to $T(n) \in O(n^{\log_3 5} \in O(n^{3/2})$ $(n\sqrt{n})$. If 3 groups of $n/3$ gave us an improvement, try k groups of n/k. In this case, the recurrence relation works out to be $T(n) = (2k-1)T(n/k) + O(n)$ which is still bottom heavy since $k < 2k-1$ for $k > 1$. By the master theorem, $T(k) \in O(n^{\log_k (2k-1)})$, and $n^{\log_k (2k-1)} < n^{\log_k 2k} = n^{\frac{\log 2 + \log k}{\log k}}$, and we end up with the result that for large k, this multiplication procedure is in time $n^{1+\epsilon}$ where ϵ is an arbitrarily small number (provided k is arbitrarily large).

Step back for a moment — what we've done is parameterize the problem of multiplying n digit numbers (or polynomials, if you will) at k points. Let's choose $k = n$. In this case $A = a_{n-1}10^{n-1} + \cdots a_0$, and similarly for B. If $n > 10$, though, we suddenly have a problem: we have to evaluate the $P_A B$ polynomial at $n = 10$, which is exactly what we're trying to evaluate by evaluating the $P_A B$ polynomial at $n = 10$. Hmm. That's no good. Oh! Try using the rational numbers! Hmm, that'll give us the same problem, but sort of in reverse.

Let's try complex numbers. There are many properties to the roots of unity that I won't go into—they are in the book in chapter 35, and well worth the read. Class time ran out quickly, but the algorithm from the book is listed in algorithm 1.

```
RECURSIVE-FFT($a$: array of numbers
$n \leftarrow$ length($a$)
**if** $n = 1$ **then**
    return $a$;
**end if**
$\omega_n \leftarrow e^{2\pi i/n}$
$\omega \leftarrow 1$
$a^{[0]} \leftarrow (a_0, a_2, \ldots, a_{n-2})$
$a^{[1]} \leftarrow (a_1, a_3, \ldots, a_{n-1})$
$y^{[0]} \leftarrow$ RECURSIVE-FFT($a^{[0]}$)
$y^{[1]} \leftarrow$ RECURSIVE-FFT($a^{[1]}$)
**for** $k \leftarrow 0$ upto $n/2 - 1$ **do**
    $y_{k+n/2} \leftarrow y_k \leftarrow y_k^{[0]} + \omega y_k^{[1]}$
    $\omega \leftarrow \omega\omega_n$
**end for**
return $y$;
```

This algorithm divides a polynomial P into two pieces: P_{even} and P_{odd}. The idea is that $P_{\text{even}} = a_0 + a_2 x + a_4 x^2 \ldots$, and similarly for the odd polynomial, and $P_A(x) = xP_{A,\text{odd}}(x^2) + P_{A,\text{even}}(x^2)$. The reason that this works is that $\omega_{n+j}^2 = \omega_1^{2n+2j} = \omega_1^{2j} = \omega_j^2$, because of the properties of ω (if you haven't guessed, ω_j is the j-th complex n-th root of unity. What algorithm 1 is doing is dividing the polynomials into two parts, evaluating the subparts, and then stitching them back together through Horner's rule. Note that the property of ωs listed above is referred to as the "Halving lemma", and makes the problem feasible through divide and conquer methods. The halving lemma is listed on page 832 of CLRS. Read it.

Because the Fourier Transform is its own inverse, by a suitable substitution of variables the same algorithm can change the point-value representation back to the coefficient representation. To really understand this you have to understand more about Fourier transforms than I do. I used them a lot in college, but how this all magically works out like this is completely mystifying to me.

Note that in the book, in the lecture, and in the notes for both the book and the lecture, I have been really vague and handwavy: I have not shown that a polynomial evaluated at the complex roots of unity is the same as the Fourier Transform of the original polynomial! This is deliberately avoided, since this isn't a course in analysis. It would be a good thing to learn why all of this works out at a deeper level, but for the purposes of multiplication we can just claim this as proof by authority (our teacher told us so). Anyway, divide and conquer once again saves the day.

□