CS 6363.005.19S Lecture 5–January 29, 2019

Main topics are #divide-and-conquer with #fast_Fourier_transforms.

Prelude

- Homework 1 is due Tuesday, February 5th. I hope you've at least looked at it by now! If you're having difficulties, please please come to office hours.

Polynomials

- Today, we're going to look another application of the divide-and-conquer paradigm that has to do with the manipulation of polynomials.
- A polynomial is a function of one variable made from additions, subtractions, and multiplications. It's usually represented as the sum of weighted powers of a (complex) variable x.

$$p(x) = \sum_{j=0}^{n} a_j x^j.$$

- Each number a_j is called a *coefficient* of the polynomial. The *degree* of the polynomial is the largest power of x with a non-zero coefficient. So the example polynomial here has degree *at most* n.
- We can represent any polynomial of degree at most n using an array P[0..n] where P[j] is the coefficient of x^j.
- The most common operations on polynomials are *evaluation*, *addition*, and *multiplication* which can be implemented with the following algorithms.

```
EVALUATE($P[0..n], x$):
    $X \leftarrow 1$    $\langle\langle X = x^j \rangle\rangle$
    $y \leftarrow 0$
    for $j \leftarrow 0$ to $n$
        $y \leftarrow y + P[j] \cdot X$
        $X \leftarrow X \cdot x$
    return $y$
```

```
ADD($P[0..n], Q[0..n]$):
    for $j \leftarrow 0$ to $n$
        $R[j] \leftarrow P[j] + Q[j]$
    return $R[0..n]$
```

```
MULTIPLY($P[0..n], Q[0..m]$):
    for $j \leftarrow 0$ to $n + m$
        $R[j] \leftarrow 0$
    for $j \leftarrow 0$ to $n$
        for $k \leftarrow 0$ to $m$
            $R[j+k] \leftarrow R[j+k] + P[j] \cdot Q[k]$
    return $R[0..n+m]$
```

- Notice how the product of degree n and m polynomials have degree at most m + n.
- For this lecture, I'm going to assume we can do arithmetic operations in constant time so we can focus on the number of them as we manipulate polynomials.

- Each of these algorithms is iterative, so we can easily determine their running time. Evaluation and Addition each take O(n) time. However, multiplication of two degree-n polynomials takes O(n^2) time!
- One way to solve the problem of slow multiplications is to represent the polynomials in a different form.
- One consequence of the Fundamental Theorem of Algebra is that we can uniquely represent any polynomial of degree n using n + 1 pairs {(x_0, y_0), (x_1, y_1), …, (x_n, y_n)} where p(x_j) = y_j. The polynomial *interpolates* these points.
- If we agree upon which samples x_j to use in advance, we can again represent any polynomial using n + 1 values.
- Adding and multiplying polynomials in O(n) time really easy in this form as long as you're using the same samples for both polynomials. Just add or multiply their sample values. For multiplication, we need to make sure we have m + n + 1 samples, though, so we can uniquely determine the resulting polynomial of degree m + n.
- Unfortunately, evaluation in this form is much more complicated. We can use a formula by Lagrange:

$$p(x) = \sum_{j=0}^{n-1} \left(\frac{y_j}{\prod_{k \neq j}(x_j - x_k)} \prod_{k \neq j}(x - x_k) \right)$$

- However, we now need to evaluate O(n) products for each of the O(n) summands in O(n^2) time total.
- So we have this table then **[ignore the middle row]**. Two of the three operations are easy in both representations, but not the same two operations!

representation	evaluate	add	multiply
coefficients	$O(n)$	$O(n)$	$\boldsymbol{O(n^2)}$
roots + scale	$O(n)$	$\boldsymbol{\infty}$	$O(n)$
samples	$\boldsymbol{O(n^2)}$	$O(n)$	$O(n)$

Converting Between Representations

- What would be useful is a quick way to convert back and forth between the coefficient and sample representations. Then, if we need to do the slow operation in one representation, we can convert, perform the now-faster operations, and then convert back.
- To make this part easier to describe, I'm going to assume n is a power of 2 and that our polynomial has degree at most n - 1. The reasons why will hopefully become clear in a little bit.
- So let's say we've chosen some sample positions {x_0, x_1, x_{n-1}}. We can compute each p(x_j) in O(n) time given coefficient form, so converting *to* sample form using the basic

algorithm takes O(n^2) time. Converting back is worse: if we're careful, we can use Lagrange's formula to do the conversion in O(n^3) time.

- But maybe we can do better by looking at the problem in a slightly different way.
- Let V be an n x n matrix where v_{jk} = x_j^k (indexing rows and columns from 0)

$$V = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix}.$$

- V is often called the *Vandermonde matrix*.
- The vector of coefficients a-> = (a_0, a_1, …, a_{n-1}) and sample *values* y-> = (y_0, y_1, …, y_{n-1}) are related by the matrix equation Va-> = y-> or

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix}.$$

- So we can easily compute y-> using a-> in O(n^2) time.
- Similarly, we can compute a-> given y-> by solving a system of linear equations. If we use an algorithm called Gaussian elimination, this takes O(n^3) time.
- But here's where we can get our first improvement. We chose our sample positions in advance, independent of the actual polynomial. In other words, they aren't part of the input to this conversion problem.
- So, we can also have V^{-1} figured out in advance. Now a-> = V^{-1} y->. We can convert the sample representation *with these already chosen samples* to the coefficient representation in O(n^2) time.
- But we can do even better by not only fixing the sample positions in advance, but by choosing them carefully and using divide-and-conquer.

Divide-and-Conquer

- Given polynomial p in coefficient form, let p_{even} be the even coefficients and p_{odd} be its odd coefficients.
- p(x) = p_{even}(x^2) + x p_{odd}(x^2). We can evaluate p(x) by recursively evaluating p_even and p_odd and doing O(1) additional arithmetic operations.
- Of course, we still have to look at every coefficient so this procedure still takes O(n) time per sample position.
- But what if we could choose our sample positions carefully so we had fewer distinct calls

to p_even and p_odd?

- We'll call a set X of n values *collapsing* if
    - X has one element or
    - X^2 = {x^2 | x in X} has exactly n / 2 elements and is itself collapsing.
- If we have a polynomial p of degree n - 1 and a collapsing set X of size n, we can compute {p(x) | x in X} by
    1. Recursively computing {p_even(x^2) | x in X} = {p_even(y) | y in X^2}
    2. Recursively computing {p_odd(x^2) | x in X} = {p_odd(y) | y in X^2}
    3. For each x in X, compute p(x) = p_even(x^2) + x p_odd(x^2)
- The recursive computations are on polynomials of degree at most n / 2 - 1 using n / 2 sample positions. Also it takes Theta(n) time to do step 3.
- So if T(n) is the time to do the conversion of degree n - 1 polynomials with n samples positions, then we just need to solve T(n) = 2T(n / 2) + n.
- Which we've already done. T(n) = Theta(n log n).

The Discrete Fourier Transform

- So if we work with a collapsing set of sample positions, then we get a Theta(n log n) time algorithm for converting from coefficient to sample representation.
- We can easily use the definition to derive a collapsing set. Let X_1 = {1} and let X_n = { ± sqrt{x} | x in X}. So
    - X_1 = {1}
    - X_2 = {1, -1}
    - X_4 = {1, -1, i, -i}
    - X_8 = {1 , -1, i, -i, sqrt{2}/2 + sqrt{2}/2 i, -sqrt{2}/2 - sqrt{2}/2 i, sqrt{2}/2 - sqrt{2}/2 i, -sqrt{2}/2 + sqrt{2}/2 i}
    - …
- For any n, the set X_n is called the *complex nth roots of unity*. They're the roots of x^n - 1 = 0. They are spaced evenly around the unit circle in the complex plane. And they are all powers of the *primitive or principle nth* root w_n = e^{2pi i / n} = cos {2pi / n} + i sin {2pi / n}.
- In particular, they all have the form w_n^k = e^{(2pi i / n) k} = cos (2pi / n * k) + i sin (2pi / n * k).
- Note there are exact n different nth roots of unity: w_n^k = w_n^{k mod n}.
- The *discrete Fourier transform* of the polynomial p's coefficient vector a-> = (a_0, a_1, …) is the list of sample values y-> = (y_0, y_1, …) we get by sampling at the nth roots of unity.
- In other words, given coefficients P[0..n-1], its discrete Fourier transform is the vector P^* [0.. n - 1] defined as

$$\boxed{P^*[j] := p(\omega_n^j) = \sum_{k=0}^{n-1} P[k] \cdot \omega_n^{jk}}$$

- Now observe, if n is even then w_{n/2}^{k} = e^{(2p i / (n/2)) k} = e^{(2pi / n} 2k} = w_{n}^{2k}. In other words, every (n/2)th root of unity is the square of an nth root of unity.
- But there are half as many (n/2)th roots of unity, so the nth roots of unity are collapsible.
- In particular, w_n^{k + n / 2} = -w_n^k, so w_n^k and w_n^{k + n / 2} square to the same thing.
- Because our set of sample positions is collapsible, we can compute the discrete Fourier transform in only O(n log n) time.
- This divide-and-conquer algorithm is called the *fast Fourier transform* and was popularized by Cooley and Tukey in 1965, even though it has a much longer history with many others describing some form of it including Gauss around 1805.
- The following pseudocode algorithm assumes n is a power of 2. If necessary, you can always pad any set of input coefficients with 0s to make that the case. Also, this particular form of the FFT is often called the radix-2 fast Fourier transform, because other divide-and-conquer strategies exist.

```
RADIX2FFT($P[0..n-1]$):
    if $n = 1$
        return $P$

    for $j \leftarrow 0$ to $n/2 - 1$
        $U[j] \leftarrow P[2j]$
        $V[j] \leftarrow P[2j+1]$

    $U^* \leftarrow$ RADIX2FFT($U[0..n/2-1]$)
    $V^* \leftarrow$ RADIX2FFT($V[0..n/2-1]$)

    $\omega_n \leftarrow \cos(\frac{2\pi}{n}) + i\sin(\frac{2\pi}{n})$
    $\omega \leftarrow 1$

    for $j \leftarrow 0$ to $n/2 - 1$
        $P^*[j]         \leftarrow U^*[j] + \omega \cdot V^*[j]$
        $P^*[j + n/2]   \leftarrow U^*[j] - \omega \cdot V^*[j]$
        $\omega \leftarrow \omega \cdot \omega_n$

    return $P^*[0..n-1]$
```

Inverting the FFT

- Now, our original motivation for converting to sample representation was so we could multiply polynomials quickly, and it would be good to get back to coefficient representation.
- Remember the Vandermonde matrix. Since we're using the complex nth roots of unity, it looks like

$$V = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)^2} \end{bmatrix}$$

- To go from samples y-> to coefficients a->, we need compute V^{-1} y-> ,but one can show the amazing fact V^{-1} = bar{V} / n. In other words, the (j, k) entry of V^{-1} is bar{w_n^k} / n = w_n^{-jk} / n.
- So inverting the discrete Fourier transform is the same as taking the discrete Fourier transform, except our roots of unity go around the unit circle in the opposite direction, and we need to divide our products by n.
- Here's what it looks like if we apply those changes directly to Radix2FFT.

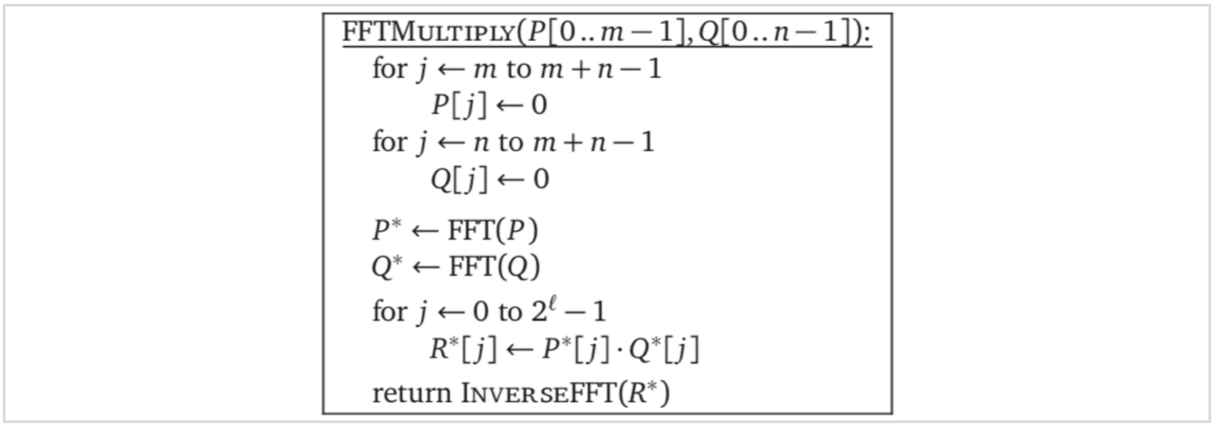
```
INVERSEFFT($P^*[0..n-1]$):
    $P[0..n-1] \leftarrow \mathrm{FFT}(P^*)$
    for $j \leftarrow 0$ to $n-1$
        $P^*[j] \leftarrow \overline{P^*[j]}/n$
    return $P[0..n-1]$
```

```
INVERSERADIX2FFT($P^*[0..n-1]$):
    if $n = 1$
        return $P$

    for $j \leftarrow 0$ to $n/2-1$
        $U^*[j] \leftarrow P^*[2j]$
        $V^*[j] \leftarrow P^*[2j+1]$

    $U \leftarrow$ INVERSERADIX2FFT($U^*[0..n/2-1]$)
    $V \leftarrow$ INVERSERADIX2FFT($V^*[0..n/2-1]$)

    $\overline{\omega_n} \leftarrow \cos(\frac{2\pi}{n}) - i\sin(\frac{2\pi}{n})$
    $\overline{\omega} \leftarrow 1$

    for $j \leftarrow 0$ to $n/2-1$
        $P[j]       \leftarrow (U[j] + \overline{\omega} \cdot V[j])/2$
        $P[j+n/2] \leftarrow (U[j] - \overline{\omega} \cdot V[j])/2$
        $\overline{\omega} \leftarrow \overline{\omega} \cdot \overline{\omega_n}$

    return $P[0..n-1]$
```

- Those / 2's are there because we only divided by (n / 2) during the recursive calls.

Fast Polynomial Multiplication

- Finally, how do we do the full job of multiplying two polynomials in coefficient representation?
- We pad them with enough 0's to make sure we have a power of 2 for the number of coefficients and to make sure we have enough samples for the multiplication. Then we do an FFT to both, multiply their sample representations, and invert the result.



```
FFTMultiply($P[0..m-1], Q[0..n-1]$):
    for $j \leftarrow m$ to $m+n-1$
        $P[j] \leftarrow 0$
    for $j \leftarrow n$ to $m+n-1$
        $Q[j] \leftarrow 0$
    $P^* \leftarrow$ FFT($P$)
    $Q^* \leftarrow$ FFT($Q$)
    for $j \leftarrow 0$ to $2^\ell - 1$
        $R^*[j] \leftarrow P^*[j] \cdot Q^*[j]$
    return InverseFFT($R^*$)
```

- Now, this recursive algorithm is not the only way to implement FFTs. The texts go into more detail. In particular, they are often implemented in hardware as circuits with very pretty diagrams.
- Also, these ideas are not exclusively used for multiplying polynomials. In particular, they're most often used now for signal processing as they let you go from a sample based representation of signals to one based on the weighted sum of simple functions like sinusoids.