Lecture 3: Divide and Conquer: Fast Fourier Transform

- Polynomial Operations vs. Representations
- Divide and Conquer Algorithm
- Collapsing Samples / Roots of Unity
- FFT, IFFT, and Polynomial Multiplication

Polynomial operations and representation

A polynomial $A(x)$ can be written in the following forms:

$$\begin{aligned} A(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} \\ &= \sum_{k=0}^{n_1} a_k x^k \\ &= \langle a_0, a_1, a_2, \ldots, a_{n-1} \rangle \quad \text{(coefficient vector)} \end{aligned}$$

The degree of A is $n-1$.

Operations on polynomials

There are three primary operations for polynomials.

1. **Evaluation**: Given a polynomial $A(x)$ and a number x_0, compute $A(x_0)$. This can be done in $O(n)$ time using $O(n)$ arithmetic operations via Horner's rule.

   - **Horner's Rule:** $A(x) = a_0 + x(a_1 + x(a_2 + \cdots x(a_{n-1}) \cdots))$. At each step, a sum is evaluated, then multiplied by x, before beginning the next step. Thus $O(n)$ multiplications and $O(n)$ additions are required.

2. **Addition:** Given two polynomials $A(x)$ and $B(x)$, compute $C(x) = A(x) + B(x)$ $(\forall x)$. This takes $O(n)$ time using basic arithmetic, because $c_k = a_k + b_k$.

3. **Multiplication:** Given two polynomials $A(x)$ and $B(x)$, compute $C(x) = A(x) \cdot B(x)$ $(\forall x)$. Then $c_k = \sum_{j=0}^{k} a_j b_{k-j}$ for $0 \leq k \leq 2(n-1)$, because the degree of the resulting polynomial is twice that of A or B. This multiplication is then equivalent to a convolution of the vectors A and reverse(B). The *convolution* is the inner product of all relative shifts, an operation also useful for smoothing etc. in digital signal processing.

   - Naive polynomial multiplication takes $O(n^2)$.
   - $O(n^{\lg 3})$ or even $O(n^{1+\varepsilon})$ $(\forall \varepsilon > 0)$ is possible via Strassen-like divide-and-conquer tricks.
   - Today, we will compute the product in $O(n \lg n)$ time via Fast Fourier Transform!

Representations of polynomials

First, consider the different representations of polynomials, and the time necessary to complete operations based on the representation.

There are 3 main representations to consider.

1. Coefficient vector with a monomial basis

2. Roots and a scale term

   - $A(x) = (x - r_0) \cdot (x - r_1) \cdot \cdots \cdot (x - r_{n-1}) \cdot c$
   - However, it is impossible to find exact roots with only basic arithmetic operations and kth root operations. Furthermore, addition is extremely hard with this representation, or even impossible. Multiplication simply requires roots to be concatenated, and evaluation can be completed in $O(n)$.

3. Samples: (x_0, y_0), (x_1, y_1), $\ldots$, (x_{n-1}, y_{n-1}) with $A(x_i) = y_i$ $(\forall i)$ and each x_i is distinct. These samples uniquely determine a degree $n - 1$ polynomial A, according to the Lagrange and Fundamental Theorem of Algebra. Addition and multiplication can be computed by adding and multiplying the y_i terms, assuming that the x_i's match. However, evaluation requires interpolation.

The runtimes for the representations and the operations is described in the table below, with algorithms for the operations versus the representations.

Algorithms vs.	**Representations**		
	Coefficients	**Roots**	**Samples**
Evaluation	$O(n)$	$O(n)$	$O(n^2)$
Addition	$O(n)$	∞	$O(n)$
Multiplication	$O(n^2)$	$O(n)$	$O(n)$

We combine the best of each representation by converting between coefficients and samples in $O(n \lg n)$ time.

How? Consider the polynomial in matrix form.

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

where V is the Vandermonde matrix with entries $v_{jk} = x_j^k$.

Then we can convert between coefficients and samples using the matrix vector product $V \cdot A$, which is equivalent to evaluation. This takes $O(n^2)$.

Similarly, we can samples to coefficients by solving $V \backslash Y$ (in MATLAB®notation). This takes $O(n^3)$ via Gaussian elimination, or $O(n^2)$ to compute $A = V^{-1} \cdot Y$, if V^{-1} is precomputed.

To do better than $\Theta(n^2)$ when converting between coefficients and samples, and vice versa, we need to choose special values for $x_0, x_1, \ldots, x_{n-1}$. Thus far, we have only made the assumption that the x_i values are distinct.

Divide and Conquer Algorithm

We can formulate polynomial multiplication as a divide and conquer algorithm with the following steps for a polynomial $A(x)$ ∀ x ∈ X.

1. Divide the polynomial A into its even and odd coefficients:

$$A_{even}(x) = \sum_{k=0}^{\lceil \frac{n}{2} - 1 \rceil} a_{2k} x^k = \langle a_0, a_2, a_4, \ldots \rangle$$

$$A_{odd}(x) = \sum_{k=0}^{\lfloor \frac{n}{2} - 1 \rfloor} a_{2k+1} x^k = \langle a_1, a_3, a_5, \ldots \rangle$$

2. Recursively conquer $A_{even}(y)$ for $y \in X^2$ and $A_{odd}(y)$ for $y \in X^2$, where $X^2 = \{x^2 \mid x \in X\}$.

3. Combine the terms. $A(x) = A_{even}(x^2) + x \cdot A_{odd}(x^2)$ for $x \in$ X.

However, the recurrences for this algorithm is

$$\begin{aligned} T(n, |X|) &= 2 \cdot T\left(\frac{n}{2}, |X|\right) + O(n + |X|) \\ &= O(n^2) \end{aligned}$$

which is no better than before.

We can do better if X is *collapsing*: either $|X| = 1$ (base case), or $|X^2| = \frac{|X|}{2}$ and X^2 is (recursively) collapsing. Then the recurrence is of the form

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n) = O(n \lg n).$$

Roots of Unity

Collapsing sets can be constructed via square roots. Each of the following collapsing sets is computing by taking all square roots of the previous set.

1. $\{1\}$

2. $\{1, -1\}$

3. $\{1, -1, i, -i\}$

4. $\{1, -1, \pm\frac{\sqrt{2}}{2}(1+i), \pm\frac{\sqrt{2}}{2}(-1+i)\}$, which lie on a unit circle

We can repeat this process and make our set larger and larger by finding more and more points on this circle. These points are called the nth roots of unity. Formally, the nth roots of unity are n x's such that $x^n = 1$. These points are uniformly spaced around the unit circle in the complex plane (including 1). These points are of the form $(\cos\theta, \sin\theta) = \cos\theta + i\sin\theta = e^{i\theta}$ by Euler's Formula, for $\theta = 0, \frac{1}{n}\tau, \frac{2}{n}\tau, \ldots, \frac{n-1}{n}\tau$ (where $\tau = 2\pi$).

The nth roots of unity where $n = 2^\ell$ form a collapsing set, because $(e^{i\theta})^2 = e^{i(2\theta)} = e^{i(2\theta \bmod \tau)}$. Therefore the even nth roots of unity are equivalent to the $\frac{n}{2}$nd roots of unity.

FFT, IFFT, and Polynomial Multiplication

We can take advantage of the nth roots of unity to improve the runtime of our polynomial multiplication algorithm. The basis for the algorithm is called the Discrete Fourier Transform (DFT).

The DFT allows the transformation between coefficients and samples, computing $A \to A^* = V \cdot A$ for $x_k = e^{i\tau k/n}$ where $n = 2^\ell$, where A is the set of coefficients and A^* is the resulting samples. The individual terms $a_j^* = \sum_{j=0}^{n-1} e^{i\tau jk/n} \cdot a_j$.

Fast Fourier Transform (FFT)

The FFT algorithm is an $O(n \lg n)$ divide and conquer algorithm for DFT, used by Gauss circa 1805, and popularized by Cooley and Turkey and 1965. Gauss used the algorithm to determine periodic asteroid orbits, while Cooley and Turkey used it to detect Soviet nuclear tests from offshore readings.

A practical implementation of FFT is FFTW, which was described by Frigo and Johnson at MIT. The algorithm is often implemented directly in hardware, for fixed n.

Inverse Discrete Fourier Transform

The Inverse Discrete Fourier Transform is an algorithm to return the coefficients of a polynomial from the multiplied samples. The transformation is of the form $A^* \to V^{-1} \cdot A^* = A$.

In order to compute this, we need to find V^{-1}, which in fact has a very nice structure.

Claim 1. *$V^{-1} = \frac{1}{n}\bar{V}$, where $\bar{V}$ is the complex conjugate of V.*[1]

[1] Recall the complex conjugate of $p + qi$ is $p - qi$.

Proof. We claim that $P = V \cdot \bar{V} = nI$:

$$\begin{aligned}
p_{jk} &= (\text{row } j \text{ of } V) \cdot (\text{col. } k \text{ of } \bar{V}) \\
&= \sum_{m=0}^{n-1} e^{ij\tau m/n} \overline{e^{ik\tau m/n}} \\
&= \sum_{m=0}^{n-1} e^{ij\tau m/n} e^{-ik\tau m/n} \\
&= \sum_{m=0}^{n-1} e^{i(j-k)\tau m/n}
\end{aligned}$$

Now if $j = k$, $p_{jk} = \sum_{m=0}^{n-1} = n$. Otherwise it forms a geometric series.

$$\begin{aligned}
p_{jk} = &= \sum_{m=0}^{n-1} (e^{i(j-k)\tau/n})^m \\
&= \frac{(e^{i\tau(j-k)/n})^n - 1}{e^{i\tau(j-k)/n} - 1} \\
&= 0
\end{aligned}$$

because $e^{i\tau} = 1$. Thus $V^{-1} = \frac{1}{n}\bar{V}$, because $V \cdot \bar{V} = nI$. □

This claim says that the Inverse Discrete Fourier Transform is equivalent to the Discrete Fourier Transform, but changing x_k from $e^{ik\tau/n}$ to its complex conjugate $e^{-ik\tau/n}$, and dividing the resulting vector by n. The algorithm for IFFT is analogous to that for FFT, and the result is an $O(n \lg n)$ algorithm for IDFT.

Fast Polynomial Multiplication

In order to compute the product of two polynomials A and B, we can perform the following steps.

1. Compute $A^* = FFT(A)$ and $B^* = FFT(B)$, which converts both A and B from coefficient vectors to a sample representation.

2. Compute $C^* = A^* \cdot B^*$ in sample representation in linear time by calculating $C_k^* = A_k^* \cdot B_k^*$ $(\forall k)$.

3. Compute $C = \text{IFFT}(C^*)$, which is a vector representation of our final solution.

Applications

Fourier (frequency) space many applications. The polynomial $A^* = FFT(A)$ is complex, and the amplitude $|a_k^*|$ represents the amplitude of the frequency-k signal, while $\arg(a_k^*)$ (the angle of the 2D vector) represents the phase shift of that signal. For example, this perspective is particularly useful for audio processing, as used by Adobe Audition, Audacity, etc.:

- High-pass filters zero out high frequencies
- Low-pass filters zero out low frequencies
- Pitch shifts shift the frequency vector
- Used in MP3 compression, etc.

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2015