Introduction to Algorithms
6.046J/18.401

Lecture 17
Prof. Piotr Indyk
Fast Fourier Transform

- **Discrete Fourier Transform (DFT):**
  - Given: coefficients of a polynomial
    \[ a(x) = a_0 + a_1 x + \ldots + a_{n-1} x^{n-1} \]
  - Goal: compute \( a(\omega_n^0), a(\omega_n^1), \ldots, a(\omega_n^{n-1}) \),
    \( \omega_n \) is the “principal n-th root of unity”
- **Challenge:** Perform DFT in \( O(n \log n) \) time.
Motivation I: 6.003

- FFT is **essential** for digital signal processing
  - $a_0, a_1, \ldots, a_{n-1}$: signal in the “time domain”
  - $a(\omega_n^0), a(\omega_n^1) \ldots a(\omega_n^{n-1})$: signal in the “frequency domain”
  - FFT enables quick conversion from one domain to the other

- Used in Compact Disks, Digital Cameras, Synthesizers, etc, etc.
Example application: SETI

- Searching For Extraterrestrial Intelligence (SETI):

  “At each drift rate, the client searches for signals at one or more bandwidths between 0.075 and 1,221 Hz. This is accomplished by using FFTs of length $2^n (n = 3, 4, ..., 17)$ to transform the data into a number of time-ordered power spectra.”
FFT

- Very elaborate implementations (e.g., FFTW, “the Fastest Fourier Transform in the West”, done at MIT)
- Hardware implementations
Motivation II: Computer Science

• We will see how to multiply two polynomials in $O(n \log n)$ time using FFT
• Multiplication of polynomials $\rightarrow$ mult. of (large) integers - cryptography
• Also: pattern matching, etc.
DFT

• Recall: want $a(\omega_n^0), a(\omega_n^1) \ldots a(\omega_n^{n-1})$

• $\omega_n$ is the “principal $n$-th root of unity, i.e., for $j=0\ldots n-1$ we have $(\omega_n^j)^n=1$.

• We will work in the field of complex numbers where

$$\omega_n = e^{2\pi i/n} = \cos(2\pi/n) + i \sin(2\pi/n)$$

• $\omega_n$ is indeed the principal $n$-th root of unity:

$$(\omega_n^j)^n = e^{2\pi ij} = \cos(2\pi j) + i \sin(2\pi j) = 1$$
Halving Lemma

• If \( n > 0 \) is even, then the squares of the \( n \) complex \( n \)-th roots of unity are the \( n/2 \) complex \( (n/2) \)-th roots of unity, i.e.:

\[
\{ (\omega_n^0)^2, \ldots, (\omega_n^{n-1})^2 \} = \{ \omega_{n/2}^0, \ldots, \omega_{n/2}^{n/2-1} \}
\]

• Proof: \((\omega_n^j)^2 = e^{2\pi ij/n} = e^{2\pi ij/(n/2)} = \omega_{n/2}^j\)
FFT

- Divide-and-conquer algorithm
- “Split” \( a(x) \) into \( a^{[0]}(x) \) and \( a^{[1]}(x) \):
  \[
  a^{[0]}(x) = a_0 + a_2 x + \ldots + a_{n-2} x^{n/2-1}
  \]
  \[
  a^{[1]}(x) = a_1 + a_3 x + \ldots + a_{n-1} x^{n/2-1}
  \]
- Therefore
  \[
  a^{[0]}(x^2) + x a^{[1]}(x^2) = a(x)
  \]
FFT: the algorithm

- Recall we need to evaluate the polynomial $a$ at points $\{\omega_n^0, \ldots, \omega_n^{n-1}\}$
- Suffices to
  - Evaluate polynomials $a^{[0]}$ and $a^{[1]}$ at points $\{(\omega_n^0)^2, \ldots, (\omega_n^{n-1})^2\} = P$
  - Compute $a(\omega_n^j) = a^{[0]}((\omega_n^j)^2) + \omega_n^j a^{[1]}((\omega_n^j)^2)$
- However, $P = \{\omega_{n/2}^0, \ldots, \omega_{n/2}^{n/2-1}\}$, $|P| = n/2$
- Thus, we just need to recursively evaluate two polynomials with degree $n/2-1$ at $n/2$ points!
- Time: $T(n) = 2T(n/2) + O(n) \rightarrow T(n) = O(n \log n)$
Comments

- We assumed that $n$ is a power of 2
- This is **NOT** without loss of generality
Inverse DFT

- Given: the values $a(\omega_n^0), a(\omega_n^1) \ldots a(\omega_n^{n-1})$, denoted by $y_0, y_1, \ldots, y_{n-1}$.
- Goal: compute the coefficients $a_0, a_1, \ldots, a_{n-1}$
- Algorithm:
  - “Observe” that $a_j = y((\omega_n^{-1})^j)$, $y(x)$ is a polynomial with coefficients $y_0, \ldots, y_{n-1}$ (see CLRS for proof)
  - Run FFT
Polynomial multiplication

**Input:** \( a(x) = a_0 + a_1 x + \ldots + a_{n-1} x^{n-1} \),
\( b(x) = b_0 + b_1 x + \ldots + b_{n-1} x^{n-1} \),

**Output:** \( c(x) = a(x) \times b(x) = c_0 + c_1 x + \ldots + c_{2n-2} x^{2n-2} \),
\( c_i = a_0 b_i + a_1 b_{i-1} + \ldots + a_{i-1} b_1 + a_i b_0 \)

How to solve it in \( O(n \log n) \) time?
FFT-based algorithm

- Extend $a, b$ to degree $2n-2$ (by adding 0’s)
- Compute $a(\omega_{2n}^0)\ldots a(\omega_{2n}^{2n-2})$ and $b(\omega_{2n}^0)\ldots b(\omega_{2n}^{2n-2})$ (via FFT)
- Compute $c(\omega_{2n}^j)= a(\omega_{2n}^j) * b(\omega_{2n}^j)$, $j=0\ldots2n-2$
- Compute $c_0, c_1, \ldots, c_{2n-2}$ (via inverse FFT)
- Same time as FFT
Uniqueness of c

- Can show (CLRS) that if we fix the values of a \((d-1)\)-degree polynomial at \(d\) different points, then the polynomial is unique
- E.g., there is only one line passing through 2 points
- Therefore, the algorithm is correct