CS 473ug: Algorithms

Chandra Chekuri
chekuri@cs.uiuc.edu
3228 Siebel Center

University of Illinois, Urbana-Champaign

Fall 2007

Part I

Polynomials, Convolutions and FFT

Polynomials

Definition

A polynomial is a function of one variable built from additions, subtractions and multiplications (but no divisions).

$$p(x) = \sum_{j=0}^{n-1} a_j x^j$$

The numbers $a_0, a_1, \ldots, a_n$ are the coefficients of the polynomial. The degree is the highest power of x with a non-zero coefficient.

Example

$$p(x) = 3 - 4x + 5x^3$$

$a_0 = 3, a_1 = -4, a_2 = 0, a_3 = 5$ and $deg(p) = 3$

Polynomials

Definition

A polynomial is a function of one variable built from additions, subtractions and multiplications (but no divisions).

$$p(x) = \sum_{j=0}^{n-1} a_j x^j$$

The numbers $a_0, a_1, \ldots, a_n$ are the coefficients of the polynomial. The degree is the highest power of x with a non-zero coefficient.

Representation

Polynomials represented by vector $a = (a_0, a_1, \ldots a_{n-1})$ of coefficients.

Operations on Polynomials

Evaluate Given a polynomial p and a value x, compute $p(x)$

Operations on Polynomials

Evaluate Given a polynomial p and a value x, compute $p(x)$

Add Given polynomials p, q, compute polynomial $p + q$

Operations on Polynomials

Evaluate Given a polynomial p and a value x, compute $p(x)$

Add Given (representations of) polynomials p, q, compute (reprsentation of) polynomial $p + q$

Operations on Polynomials

Evaluate Given a polynomial p and a value x, compute $p(x)$

Add Given (representations of) polynomials p, q, compute (reprsentation of) polynomial $p + q$

Multiply Given (representation of) polynomials p, q, compute (representation of) polynomial $p \cdot q$.

Operations on Polynomials

Evaluate Given a polynomial p and a value x, compute $p(x)$

Add Given (representations of) polynomials p, q, compute (reprsentation of) polynomial $p + q$

Multiply Given (representation of) polynomials p, q, compute (representation of) polynomial $p \cdot q$.

Roots Given p find all *roots* of p.

Evaluation

Compute value of polynomial $a = (a_0, a_1, \ldots a_{n-1})$ at x

Evaluation

Compute value of polynomial $a = (a_0, a_1, \ldots a_{n-1})$ at x

```
power = 1
value = 0
for j = 0 to n − 1
    // invariant:  power = x^j
    value = value + a_j ·power
    power = power · x
end for
return value
```

Uses $2n$ multiplication and n additions

Evaluation

Compute value of polynomial $a = (a_0, a_1, \ldots a_{n-1})$ at x

```
power = 1
value = 0
for j = 0 to n − 1
    // invariant: power = x^j
    value = value + a_j ·power
    power = power · x
end for
return value
```

Uses $2n$ multiplication and n additions

Horner's rule can be used to cut the multiplications in half

$$a(x) = a_0 + x(a_1 + x(a_2 + \cdots + xa_{n-1}))$$

Evaluation: Numerical Issues

Question

How long does evaluation really take? $O(n)$ time?

Evaluation: Numerical Issues

Question

How long does evaluation really take? $O(n)$ time?

Size of x^n in terms of bits is $n \log x$ while size of x is only $\log x$. Thus, need to pay attention to size of numbers and multiplication complexity.

Ignore this issue for now. Can get around it for applications of interest.

Addition

Compute the sum of polynomials
$a = (a_0, a_1, \ldots a_{n-1})$ and $b = (b_0, b_1, \ldots b_{n-1})$

Addition

Compute the sum of polynomials
$a = (a_0, a_1, \ldots a_{n-1})$ and $b = (b_0, b_1, \ldots b_{n-1})$
$a + b = (a_0 + b_0, a_1 + b_1, \ldots a_{n-1} + b_{n-1})$. Takes $O(n)$ time.

Multiplication

Compute the product of polynomials
$a = (a_0, a_1, \ldots a_n)$ and $b = (b_0, b_1, \ldots b_m)$

Multiplication

Compute the product of polynomials
$a = (a_0, a_1, \ldots a_n)$ and $b = (b_0, b_1, \ldots b_m)$
Recall $a \cdot b = (c_0, c_1, \ldots c_{n+m})$ where

$$c_k = \sum_{i,j:\, i+j=k} a_i \cdot b_j$$

for j = 0 to n+m $c_j = 0$
for j = 0 to n
    for k = 0 to m
        $c_{j+k} = c_{j+k} + a_j \cdot b_k$
return $(c_0, c_1, \ldots c_{n+m})$

Takes $O(n^2)$ time.

Multiplication

Compute the product of polynomials
$a = (a_0, a_1, \ldots a_n)$ and $b = (b_0, b_1, \ldots b_m)$
Recall $a \cdot b = (c_0, c_1, \ldots c_{n+m})$ where

$$c_k = \sum_{i,j:\ i+j=k} a_i \cdot b_j$$

for j = 0 to n+m $c_j = 0$
for j = 0 to n
    for k = 0 to m
        $c_{j+k} = c_{j+k} + a_j \cdot b_k$
return $(c_0, c_1, \ldots c_{n+m})$

Takes $O(n^2)$ time. We will give a better algorithm!

Convolutions

Definition

The convolution of vectors $a = (a_0, a_1, \ldots a_n)$ and $b = (b_0, b_1, \ldots b_m)$ is the vector $c = (c_0, c_1, \ldots c_{n+m})$ where

$$c_k = \sum_{i,j:\ i+j=k} a_i \cdot b_j$$

Convolution of vectors a and b is denoted by $a * b$.

Convolutions

Definition

The convolution of vectors $a = (a_0, a_1, \ldots a_n)$ and $b = (b_0, b_1, \ldots b_m)$ is the vector $c = (c_0, c_1, \ldots c_{n+m})$ where

$$c_k = \sum_{i,j:\, i+j=k} a_i \cdot b_j$$

Convolution of vectors a and b is denoted by $a * b$. In other words, the convolution is the coefficients of the product of the two polynomials

Applications: Signal Processing

- Let $a = (a_0, a_1, \ldots a_{n-1})$ be a sequence of measurements over time.

Applications: Signal Processing

- Let $a = (a_0, a_1, \ldots a_{n-1})$ be a sequence of measurements over time.
- To account for measurement errors or random fluctuations, measurements are "smoothed" by averaging each value with a weighted sum of its neighbors. For example, in Gaussian smoothing a_i is replaced by

$$a'_i = \frac{1}{Z} \sum_{j=i-k}^{i+k} a_j e^{-(j-i)^2}$$

Applications: Signal Processing

- Let $a = (a_0, a_1, \ldots a_{n-1})$ be a sequence of measurements over time.
- To account for measurement errors or random fluctuations, measurements are "smoothed" by averaging each value with a weighted sum of its neighbors. For example, in Gaussian smoothing a_i is replaced by

$$a'_i = \frac{1}{Z} \sum_{j=i-k}^{i+k} a_j e^{-(j-i)^2}$$

- Smoothing can be thought of as vector of weights $w = (w_{-k}, w_{-(k-1)}, \ldots, w_{-1}, w_0, w_1 \ldots w_k)$ used to average each entry as $a'_i = \sum_{s=-k}^{k} w_s a_{i+s}$

Applications: Signal Processing

- Let $a = (a_0, a_1, \ldots a_{n-1})$ be a sequence of measurements over time.
- To account for measurement errors or random fluctuations, measurements are "smoothed" by averaging each value with a weighted sum of its neighbors. For example, in Gaussian smoothing a_i is replaced by

$$a'_i = \frac{1}{Z} \sum_{j=i-k}^{i+k} a_j e^{-(j-i)^2}$$

- Smoothing can be thought of as vector of weights $w = (w_{-k}, w_{-(k-1)}, \ldots, w_{-1}, w_0, w_1 \ldots w_k)$ used to average each entry as $a'_i = \sum_{s=-k}^{k} w_s a_{i+s}$
- Taking $b = (b_0, b_1, \ldots b_{2k})$ to be $b_j = w_{k-j}$, $a' = a * b$

Historical Applications

- Gauss used convolutions in the 1800s to compute the path of asteroids from a finite number of equi-spaced observations.

Historical Applications

- Gauss used convolutions in the 1800s to compute the path of asteroids from a finite number of equi-spaced observations.
- In the mid-60s, Cooley and Tukey used it to detect Soviet nuclear tests by interpolating off-shore seismic readings

Many Applications

To mention a few:

- Signal and image processing (radar, MRI, astronomy, compression, ...)
- Statistics
- Multiplication of numbers
- Pattern matching

Revisiting Polynomial Representations

Representation

Polynomials represented by vector $a = (a_0, a_1, \ldots a_{n-1})$ of coefficients.

Revisiting Polynomial Representations

Representation

Polynomials represented by vector $a = (a_0, a_1, \ldots a_{n-1})$ of coefficients.

Question

Are there other ways to represent polynomials?

Revisiting Polynomial Representations

Representation

Polynomials represented by vector $a = (a_0, a_1, \ldots a_{n-1})$ of coefficients.

Question

Are there other ways to represent polynomials?

Root of a polynomial $p(x)$: r such that $p(r) = 0$. If $r_1, r_2, \ldots, r_{n-1}$ are roots then $p(x) = a_{n-1}(x - r_1)(x - r_2) \ldots (x - r_{n-1})$.

Revisiting Polynomial Representations

Representation

Polynomials represented by vector $a = (a_0, a_1, \ldots a_{n-1})$ of coefficients.

Question

Are there other ways to represent polynomials?

Root of a polynomial $p(x)$: r such that $p(r) = 0$. If $r_1, r_2, \ldots, r_{n-1}$ are roots then $p(x) = a_{n-1}(x - r_1)(x - r_2) \ldots (x - r_{n-1})$.

Theorem (Fundamental Theorem of Algebra)

Every polynomial $p(x)$ of degree d has exactly d roots $r_1, r_2, \ldots, r_d$ where the roots can be complex numbers and can be repeated.

Representing Polynomials by Roots

Representation

Polynomials represented by vector scale factor a_{n-1} and roots $r_1, r_2, \ldots, r_{n-1}$.

Representing Polynomials by Roots

Representation

Polynomials represented by vector scale factor a_{n-1} and roots $r_1, r_2, \ldots, r_{n-1}$.

- Evaluating p at a given x is easy. Why?

Representing Polynomials by Roots

Representation

Polynomials represented by vector scale factor a_{n-1} and roots $r_1, r_2, \ldots, r_{n-1}$.

- Evaluating p at a given x is easy. Why?
- Multiplication: given p, q with roots $r_1, \ldots, r_{n-1}$ and $s_1, \ldots, s_{m-1}$ the product $p \cdot q$ has roots $r_1, \ldots, r_{n-1}, s_1, \ldots, s_{m-1}$. Easy!

Representing Polynomials by Roots

Representation

Polynomials represented by vector scale factor a_{n-1} and roots $r_1, r_2, \ldots, r_{n-1}$.

- Evaluating p at a given x is easy. Why?
- Multiplication: given p, q with roots $r_1, \ldots, r_{n-1}$ and $s_1, \ldots, s_{m-1}$ the product $p \cdot q$ has roots $r_1, \ldots, r_{n-1}, s_1, \ldots, s_{m-1}$. Easy!
- Addition: requires $O(n^2)$ time??

Representing Polynomials by Roots

Representation

Polynomials represented by vector scale factor a_{n-1} and roots $r_1, r_2, \ldots, r_{n-1}$.

- Evaluating p at a given x is easy. Why?
- Multiplication: given p, q with roots $r_1, \ldots, r_{n-1}$ and $s_1, \ldots, s_{m-1}$ the product $p \cdot q$ has roots $r_1, \ldots, r_{n-1}, s_1, \ldots, s_{m-1}$. Easy!
- Addition: requires $O(n^2)$ time??
- Given coefficient representation, how do we go to root representation?? No finite algorithm because of potential for irrational roots.

Representing Polynomials by Samples

Let p be a polynomial of degree $n-1$.
Pick n distinct *samples* $x_0, x_1, x_2, \ldots, x_{n-1}$
Let $y_0 = p(x_0), y_1 = p(x_1), \ldots, y_{n-1} = p(x_{n-1})$.

Representation

Polynomials represented by $(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})$.

Representing Polynomials by Samples

Let p be a polynomial of degree $n-1$.
Pick n distinct *samples* $x_0, x_1, x_2, \ldots, x_{n-1}$
Let $y_0 = p(x_0), y_1 = p(x_1), \ldots, y_{n-1} = p(x_{n-1})$.

Representation

Polynomials represented by $(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})$.

Is the above a valid representation?

Representing Polynomials by Samples

Let p be a polynomial of degree $n-1$.
Pick n distinct *samples* $x_0, x_1, x_2, \ldots, x_{n-1}$
Let $y_0 = p(x_0), y_1 = p(x_1), \ldots, y_{n-1} = p(x_{n-1})$.

Representation

Polynomials represented by $(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})$.

Is the above a valid representation? Why do we use $2n$ numbers instead of n numbers for coefficient and root representation?

Sample Representation

Theorem

Given a list $\{(x_0, y_0), (x_1, y_1), \ldots (x_{n-1}, y_{n-1})\}$ *there is exactly one polynomial* p *of degree* $n-1$ *such that* $p(x_j) = y_j$ *for* $j = 0, 1, \ldots, n-1$.

Sample Representation

Theorem

Given a list $\{(x_0, y_0), (x_1, y_1), \ldots (x_{n-1}, y_{n-1})\}$ *there is exactly one polynomial* p *of degree* $n-1$ *such that* $p(x_j) = y_j$ *for* $j = 0, 1, \ldots, n-1$.

So representation is valid.

Sample Representation

Theorem

Given a list $\{(x_0, y_0), (x_1, y_1), \ldots (x_{n-1}, y_{n-1})\}$ *there is exactly one polynomial* p *of degree* $n-1$ *such that* $p(x_j) = y_j$ *for* $j = 0, 1, \ldots, n-1$.

So representation is valid.
Can use same $x_0, x_1, \ldots, x_{n-1}$ for all polynomials of degree $n-1$!
No need to store them explicitly! Need only n numbers $y_0, y_1, \ldots, y_{n-1}$.

Sample Representation

Theorem

Given a list $\{(x_0, y_0), (x_1, y_1), \ldots (x_{n-1}, y_{n-1})\}$ there is exactly one polynomial p of degree $n-1$ such that $p(x_j) = y_j$ for $j = 0, 1, \ldots, n-1$.

So representation is valid.

Can use same $x_0, x_1, \ldots, x_{n-1}$ for all polynomials of degree $n-1$! No need to store them explicitly! Need only n numbers $y_0, y_1, \ldots, y_{n-1}$.

Lagrange interpolation formula: Given $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$ the following polynomial p satisfies $p(x_j) = y_j$ for $j = 0, 1, 2, \ldots, n-1$.

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

Lagrange Interpolation

For $n = 3$

$$p(x) = y_0 \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} + y_1 \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} + y_2 \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}$$

Easy to verify that $p(x_j) = y_j$! Thus there exists one polynomial of degree $n - 1$ that interpolates the values $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$.

Lagrange Interpolation

For $n = 3$

$$p(x) = y_0 \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} + y_1 \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} + y_2 \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}$$

Easy to verify that $p(x_j) = y_j$! Thus there exists one polynomial of degree $n-1$ that interpolates the values $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$.

Can there be two distinct polynomials?

Lagrange Interpolation

For $n = 3$

$$p(x) = y_0 \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} + y_1 \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} + y_2 \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}$$

Easy to verify that $p(x_j) = y_j$! Thus there exists one polynomial of degree $n - 1$ that interpolates the values $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$.

Can there be two distinct polynomials?

No! Use Fundamental Theorem of Algebra to prove it — exercise.

Addition and Multiplication with Sample Representation

- Let $\{(x_0, y_0), (x_1, y_1), \ldots (x_{n-1}, y_{n-1})\}$ and $\{(x_0, y_0'), (x_1, y_1'), \ldots (x_{n-1}, y_{n-1}')\}$ be representations of two polynomials of degree $n - 1$

Addition and Multiplication with Sample Representation

- Let $\{(x_0, y_0), (x_1, y_1), \ldots (x_{n-1}, y_{n-1})\}$ and $\{(x_0, y'_0), (x_1, y'_1), \ldots (x_{n-1}, y'_{n-1})\}$ be representations of two polynomials of degree $n-1$
- $a + b$ can be represented by $\{(x_0, (y_0 + y'_0)), (x_1, (y_1 + y'_1)), \ldots (x_{n-1}, (y_{n-1} + y'_{n-1}))\}$
  - Thus, can be computed in $O(n)$ time

Addition and Multiplication with Sample Representation

- Let $\{(x_0, y_0), (x_1, y_1), \ldots (x_{n-1}, y_{n-1})\}$ and $\{(x_0, y'_0), (x_1, y'_1), \ldots (x_{n-1}, y'_{n-1})\}$ be representations of two polynomials of degree $n-1$
- $a + b$ can be represented by $\{(x_0, (y_0 + y'_0)), (x_1, (y_1 + y'_1)), \ldots (x_{n-1}, (y_{n-1} + y'_{n-1}))\}$
  - Thus, can be computed in $O(n)$ time
- $a \cdot b$ can be evaluated at n samples $\{(x_0, (y_0 \cdot y'_0)), (x_1, (y_1 \cdot y'_1)), \ldots (x_{n-1}, (y_{n-1} \cdot y'_{n-1}))\}$
  - Can be computed in $O(n)$ time!

Addition and Multiplication with Sample Representation

- Let $\{(x_0, y_0), (x_1, y_1), \ldots (x_{n-1}, y_{n-1})\}$ and $\{(x_0, y'_0), (x_1, y'_1), \ldots (x_{n-1}, y'_{n-1})\}$ be representations of two polynomials of degree $n-1$
- $a + b$ can be represented by $\{(x_0, (y_0 + y'_0)), (x_1, (y_1 + y'_1)), \ldots (x_{n-1}, (y_{n-1} + y'_{n-1}))\}$
  - Thus, can be computed in $O(n)$ time
- $a \cdot b$ can be evaluated at n samples $\{(x_0, (y_0 \cdot y'_0)), (x_1, (y_1 \cdot y'_1)), \ldots (x_{n-1}, (y_{n-1} \cdot y'_{n-1}))\}$
  - Can be computed in $O(n)$ time!

But what if p, q are given in coefficient form? Convolution requires p, q to be in coefficient form.

Coefficient representation to Sample representation

Given p as $(a_0, a_1, \ldots, a_{n-1})$ can we obtain a sample representation $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$ quickly? Also can we *invert* the representation quickly?

Coefficient representation to Sample representation

Given p as $(a_0, a_1, \ldots, a_{n-1})$ can we obtain a sample representation $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$ quickly? Also can we *invert* the representation quickly?

- Suppose we choose $x_0, x_1, \ldots, x_{n-1}$ arbitrarily.
- Take $O(n)$ time to evaluate $y_j = p(x_j)$ given $(a_0, \ldots, a_{n-1})$.
- Total time is $\Omega(n^2)$!
- Inversion via Lagrange interpolation also $\Omega(n^2)$.

Key Idea

Can choose $x_0, x_1, \ldots, x_{n-1}$ carefully!

Total time to evaluate $p(x_0), p(x_1), \ldots, p(x_{n-1})$ should be better than evaluating each separately.

Key Idea

Can choose $x_0, x_1, \ldots, x_{n-1}$ carefully!

Total time to evaluate $p(x_0), p(x_1), \ldots, p(x_{n-1})$ should be better than evaluating each separately.

How do we choose $x_0, x_1, \ldots, x_{n-1}$ to save work?

A Simple Start

$$a(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \ldots + a_{n-1} x^{n-1}$$

Assume n is a power of 2 for rest of the discussion.

Observation: $(-x)^{2j} = x^{2j}$. Can we exploit this?

A Simple Start

$$a(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \ldots + a_{n-1} x^{n-1}$$

Assume n is a power of 2 for rest of the discussion.

Observation: $(-x)^{2j} = x^{2j}$. Can we exploit this?

Example

$$3 + 4x + 6x^2 + 2x^3 + x^4 + 10x^5 = (3 + 6x^2 + x^4) + x(4 + 2x^2 + 10x^4)$$

A Simple Start

$$a(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \ldots + a_{n-1} x^{n-1}$$

Assume n is a power of 2 for rest of the discussion.

Observation: $(-x)^{2j} = x^{2j}$. Can we exploit this?

Example

$$3 + 4x + 6x^2 + 2x^3 + x^4 + 10x^5 = (3 + 6x^2 + x^4) + x(4 + 2x^2 + 10x^4)$$

If we have $a(x)$ then easy to also compute $a(-x)$!

Odd and Even Decomposition

- Let $a = (a_0, a_1, \ldots a_{n-1})$ be a polynomial.
- Let $a_{\text{odd}} = (a_1, a_3, a_5, \ldots)$ be the $n/2$ degree polynomial defined by the odd coefficients; so

$$a_{\text{odd}}(x) = a_1 + a_3 x + a_5 x^2 + \cdots$$

- Let $a_{\text{even}} = (a_0, a_2, a_4, \ldots)$ be the $n/2$ degree polynomial defined by the even coefficients.
- Observe

$$a(x) = a_{\text{even}}(x^2) + x a_{\text{odd}}(x^2)$$

- Thus, evaluating a at x can be reduced to evaluating lower degree polynomials plus constantly many arithmetic operations.

Exploiting Odd-Even Decomposition

$$a(x) = a_{\text{even}}(x^2) + x a_{\text{odd}}(x^2)$$

- Choose n samples

  $x_0, x_1, x_2, \ldots, x_{n/2-1}, -x_0, -x_1, \ldots, -x_{n/2-1}$

Exploiting Odd-Even Decomposition

$$a(x) = a_{\mathrm{even}}(x^2) + x a_{\mathrm{odd}}(x^2)$$

- Choose n samples
  $x_0, x_1, x_2, \ldots, x_{n/2-1}, -x_0, -x_1, \ldots, -x_{n/2-1}$
- Sufficient to evaluate a_{even} and a_{odd} at $x_0^2, x_1^2, x_2^2, \ldots, x_{n/2-1}^2$!
  Pluse $O(n)$ work gives $a(x)$ for all n samples!

Exploiting Odd-Even Decomposition

$$a(x) = a_{\text{even}}(x^2) + x a_{\text{odd}}(x^2)$$

- Choose n samples
  $x_0, x_1, x_2, \ldots, x_{n/2-1}, -x_0, -x_1, \ldots, -x_{n/2-1}$
- Sufficient to evaluate a_{even} and a_{odd} at $x_0^2, x_1^2, x_2^2, \ldots, x_{n/2-1}^2$! Pluse $O(n)$ work gives $a(x)$ for all n samples!
- Suppose we can make this work recursively. Then

$$T(n) = 2T(n/2) + O(n) \text{ which implies } T(n) = O(n \log n)$$

Can we recurse?

- n samples $x_0, x_1, x_2, \ldots, x_{n/2-1}, -x_0, -x_1, \ldots, -x_{n/2-1}$
- Next step in recursion $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$

Can we recurse?

- n samples $x_0, x_1, x_2, \ldots, x_{n/2-1}, -x_0, -x_1, \ldots, -x_{n/2-1}$
- Next step in recursion $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$
- To continue recursion, we need

$$\{x_0^2, x_1^2, \ldots, x_{n/2-1}^2\} = \{z_0, z_1, \ldots, z_{n/4-1}, -z_0, -z_1, \ldots, -z_{n/4-1}\}$$

Can we recurse?

- n samples $x_0, x_1, x_2, \ldots, x_{n/2-1}, -x_0, -x_1, \ldots, -x_{n/2-1}$
- Next step in recursion $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$
- To continue recursion, we need

$$\{x_0^2, x_1^2, \ldots, x_{n/2-1}^2\} = \{z_0, z_1, \ldots, z_{n/4-1}, -z_0, -z_1, \ldots, -z_{n/4-1}\}$$

- If $z_0 = x_0^2$ and say $-z_0 = x_j^2$ then $x_0 = \sqrt{-1}x_j$! That is $x_0 = ix_j$ where i is the imaginary number.

Can we recurse?

- n samples $x_0, x_1, x_2, \ldots, x_{n/2-1}, -x_0, -x_1, \ldots, -x_{n/2-1}$
- Next step in recursion $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$
- To continue recursion, we need

$$\{x_0^2, x_1^2, \ldots, x_{n/2-1}^2\} = \{z_0, z_1, \ldots, z_{n/4-1}, -z_0, -z_1, \ldots, -z_{n/4-1}\}$$

- If $z_0 = x_0^2$ and say $-z_0 = x_j^2$ then $x_0 = \sqrt{-1}x_j$! That is $x_0 = ix_j$ where i is the imaginary number.
- Can continue recursion but need to go to *complex* numbers.

Complex Numbers

Notation

For the rest of lecture, i stands for $\sqrt{-1}$

Definition

Complex numbers are points lying in the complex plane represented as

Complex Numbers

Notation

For the rest of lecture, i stands for $\sqrt{-1}$

Definition

Complex numbers are points lying in the complex plane represented as

Cartesian $a + ib$

Complex Numbers

Notation

For the rest of lecture, i stands for $\sqrt{-1}$

Definition

Complex numbers are points lying in the complex plane represented as

Cartesian $a + ib$

Polar $re^{\theta i}$

Complex Numbers

Notation

For the rest of lecture, i stands for $\sqrt{-1}$

Definition

Complex numbers are points lying in the complex plane represented as

Cartesian $a + ib = \sqrt{a^2 + b^2}e^{(\arctan(b/a))i}$

Polar $re^{\theta i}$

Complex Numbers

Notation

For the rest of lecture, i stands for $\sqrt{-1}$

Definition

Complex numbers are points lying in the complex plane represented as

Cartesian $a + ib = \sqrt{a^2 + b^2} e^{(\arctan(b/a))i}$

Polar $re^{\theta i} = r(\cos\theta + i\sin\theta)$

Complex Numbers

Notation

For the rest of lecture, i stands for $\sqrt{-1}$

Definition

Complex numbers are points lying in the complex plane represented as

Cartesian $a + ib = \sqrt{a^2 + b^2}e^{(\arctan(b/a))i}$

Polar $re^{\theta i} = r(\cos\theta + i\sin\theta)$

Thus, $e^{\pi i} = -1$ and $e^{2\pi i} = 1$.

Power Series for Functions

What is e^z when z is a real number? When z is a complex number?

$$e^z = 1 + z/1! + z^2/2! + \ldots + z^j/j! + \ldots$$

Therefore

$$\begin{aligned} e^{i\theta} &= 1 + i\theta/1! + (i\theta)^2/2! + (i\theta)^3/3! + \ldots \\ &= (1 - \theta^2/2! + \theta^4/4! - \ldots +) + i(\theta - \theta^3/3! + \ldots +) \\ &= \cos\theta + i\sin\theta \end{aligned}$$

Complex Roots of Unity

What are the roots of the polynomial $x^k - 1$?

- Clearly 1 is a root.

Complex Roots of Unity

What are the roots of the polynomial $x^k - 1$?

- Clearly 1 is a root.
- Suppose $re^{\theta i}$ is a root then $r^k e^{k\theta i} = 1$ which implies that $r = 1$ and $k\theta = 2\pi$ since $e^{k\theta i} = \cos(k\theta) + i\sin(k\theta) = 1$

Complex Roots of Unity

What are the roots of the polynomial $x^k - 1$?

- Clearly 1 is a root.
- Suppose $re^{\theta i}$ is a root then $r^k e^{k\theta i} = 1$ which implies that $r = 1$ and $k\theta = 2\pi$ since $e^{k\theta i} = \cos(k\theta) + i\sin(k\theta) = 1$
- Let $\omega_k = e^{2\pi i/k}$. The roots are $1 = \omega_k^0, \omega_k^2, \ldots, \omega_k^{k-1}$ where $w_k^j = e^{2\pi ji/k}$.

More on the Roots of Unity

Observations

- $\omega_k^j = \omega_k^{j \bmod k}$
- $\omega_k = \omega_{jk}^j$; thus, every kth root is also a jkth root.
- $\sum_{s=0}^{k-1}(\omega_k^j)^s = (1 + \omega_k^j + \omega_k^{2j} + \ldots + \omega_k^{j(k-1)}) = 0$ for $j \neq 0$

More on the Roots of Unity

Observations

- $\omega_k^j = \omega_k^{j \bmod k}$
- $\omega_k = \omega_{jk}^j$; thus, every kth root is also a jkth root.
- $\sum_{s=0}^{k-1} (\omega_k^j)^s = (1 + \omega_k^j + \omega_k^{2j} + \ldots + \omega_k^{j(k-1)}) = 0$ for $j \neq 0$
  - ω_k^j is root of $x^k - 1 = (x-1)(x^{k-1} + x^{k-2} + \ldots + 1)$
  - Thus, ω_k^j is root of $(x^{k-1} + x^{k-2} + \ldots + 1)$

Back to Recursive Idea

- Let $a = (a_0, a_1, \ldots a_{n-1})$ be a polynomial. Assume n is a power of 2.

Back to Recursive Idea

- Let $a = (a_0, a_1, \ldots a_{n-1})$ be a polynomial. Assume n is a power of 2.
- Recall

$$a(x) = a_{\mathrm{even}}(x^2) + x a_{\mathrm{odd}}(x^2)$$

Back to Recursive Idea

- Let $a = (a_0, a_1, \ldots a_{n-1})$ be a polynomial. Assume n is a power of 2.
- Recall
$$a(x) = a_{\text{even}}(x^2) + x a_{\text{odd}}(x^2)$$
- Choose $x_0, x_1, \ldots, x_{n-1}$ to be n'th roots of unity. That is $x_j = \omega_n^j$.

Evaluating at roots of unity

- Problem: Evaluate degree $n-1$ polynomial $a = (a_0, a_1, \ldots a_{n-1})$ on the nth roots of unity

Evaluating at roots of unity

- Problem: Evaluate degree $n-1$ polynomial $a = (a_0, a_1, \ldots a_{n-1})$ on the nth roots of unity
- Recall
$$a(\omega_n^j) = a_{\mathrm{even}}((\omega_n^j)^2) + \omega_n^j a_{\mathrm{odd}}((\omega_n^j)^2)$$

Evaluating at roots of unity

- Problem: Evaluate degree $n-1$ polynomial $a = (a_0, a_1, \ldots a_{n-1})$ on the nth roots of unity
- Recall
$$a(\omega_n^j) = a_{\mathrm{even}}((\omega_n^j)^2) + \omega_n^j a_{\mathrm{odd}}((\omega_n^j)^2)$$
- Observe that $(\omega_n^j)^2 = \omega_{n/2}^j$

Evaluating at roots of unity

- Problem: Evaluate degree $n-1$ polynomial $a = (a_0, a_1, \ldots a_{n-1})$ on the nth roots of unity
- Recall
$$a(\omega_n^j) = a_{\mathrm{even}}((\omega_n^j)^2) + \omega_n^j a_{\mathrm{odd}}((\omega_n^j)^2)$$
- Observe that $(\omega_n^j)^2 = \omega_{n/2}^j$
- Thus, evaluating a on the nth roots of unity, can be accomplished by evaluating a_{even} and a_{odd} on the $n/2$ roots of unity!

Divide and Conquer Evaluation

Evaluation Problem

Evaluate $n-1$-degree polynomial on nth roots of unity

`Construct polynomials` a_{even} `and` a_{odd}
`Recurisvely evaluate` a_{even} `and` a_{odd} `on the` $n/2$`th roots of unity`
`Compute` $a(\omega_n^j)$ `as` $a_{\mathrm{even}}(\omega_{n/2}^j) + \omega_n^j a_{\mathrm{odd}}(\omega_{n/2}^j)$

Let $T(n)$ denote the running time of evaluating an $n-1$-degree polynomial on the nth roots of unity. Then,

$$T(n) \leq 2T(n/2) + O(n) = O(n \log n)$$

Discrete Fourier Transform

Definition

Given a polynomial $a = (a_0, a_1, \ldots, a_{n-1})$ the *Discrete Fourier Transform* of a is the vector $a' = (a'_0, a'_1, \ldots, a'_{n-1})$ where $a'_j = a(\omega_n^j)$ for $0 \leq j < n$.

a' is a sample representation of a for n'th roots of unity.

We have shown that a' can be computed from a in $O(n \log n)$ time. This divide and conquer *algorithm* is called the *Fast Fourier Transform* (FFT).

Back to Convolutions and Polynomial Multiplication

Convolutions

Compute convolution $c = (c_0, c_1, \ldots, c_{2n-2})$ of $a = (a_0, a_1, \ldots a_{n-1})$ and $b = (b_0, b_1, \ldots b_{n-1})$

1. Compute values of a and b at some n sample points.
2. Compute sample representation of product. That is $c' = (a'_0 b'_0, a'_1 b'_1, \ldots, a'_{n-1} b'_{n-1})$.
3. Compute coefficients of unique polynomial associated with sample representation of product. That is compute c from c'.

Back to Convolutions and Polynomial Multiplication

Convolutions

Compute convolution $c = (c_0, c_1, \ldots, c_{2n-2})$ of $a = (a_0, a_1, \ldots a_{n-1})$ and $b = (b_0, b_1, \ldots b_{n-1})$

1. Compute values of a and b at the nth roots of unity.
2. Compute sample representation of product. That is $c' = (a'_0 b'_0, a'_1 b'_1, \ldots, a'_{n-1} b'_{n-1})$.
3. Compute coefficients of unique polynomial associated with sample representation of product. That is compute c from c'.

How can be compute c from c'? We only have n sample points and c' has $2n - 1$ coefficients!

Convolutions and Polynomial Multiplication

Convolutions

Compute convolution $c = (c_0, c_1, \ldots, c_{2n-2})$ of $a = (a_0, a_1, \ldots a_{n-1})$ and $b = (b_0, b_1, \ldots b_{n-1})$

1. Pad a with n zeroes to make it a $(2n-1)$ degree polynomial $a = (a_0, a_1, \ldots, a_{n-1}, a_n, a_{n+1}, \ldots, a_{2n-1})$. Similarly for b.
2. Compute values of a and b at some $2n$ sample points.
3. Compute sample representation of product. That is $c' = (a'_0 b'_0, a'_1 b'_1, \ldots, a'_{n-1} b'_{n-1}, \ldots, a'_{2n-1} b'_{2n-1})$.
4. Compute coefficients of unique polynomial associated with sample representation of product. That is compute c from c'.

Convolutions and Polynomial Multiplication

Convolutions

Compute convolution $c = (c_0, c_1, \ldots, c_{2n-2})$ of $a = (a_0, a_1, \ldots a_{n-1})$ and $b = (b_0, b_1, \ldots b_{n-1})$

1. Pad a with n zeroes to make it a $(2n-1)$ degree polynomial $a = (a_0, a_1, \ldots, a_{n-1}, a_n, a_{n+1}, \ldots, a_{2n-1})$. Similarly for b.
2. Compute values of a and b at the $2n$th roots of unity.
3. Compute sample representation of product. That is $c' = (a'_0 b'_0, a'_1 b'_1, \ldots, a'_{n-1} b'_{n-1}, \ldots, a'_{2n-1} b'_{2n-1})$.
4. Compute coefficients of unique polynomial associated with sample representation of product. That is compute c from c'.

Convolutions and Polynomial Multiplication

Convolutions

Compute convolution $c = (c_0, c_1, \ldots, c_{2n-2})$ of $a = (a_0, a_1, \ldots a_{n-1})$ and $b = (b_0, b_1, \ldots b_{n-1})$

1. Pad a with n zeroes to make it a $(2n-1)$ degree polynomial $a = (a_0, a_1, \ldots, a_{n-1}, a_n, a_{n+1}, \ldots, a_{2n-1})$. Similarly for b.
2. Compute values of a and b at the $2n$th roots of unity.
3. Compute sample representation of product. That is $c' = (a'_0 b'_0, a'_1 b'_1, \ldots, a'_{n-1} b'_{n-1}, \ldots, a'_{2n-1} b'_{2n-1})$.
4. Compute coefficients of unique polynomial associated with sample representation of product. That is compute c from c'.

- Step 1 takes $O(n \log n)$ using divide and conquer algorithm

Convolutions and Polynomial Multiplication

Convolutions

Compute convolution $c = (c_0, c_1, \ldots, c_{2n-2})$ of $a = (a_0, a_1, \ldots a_{n-1})$ and $b = (b_0, b_1, \ldots b_{n-1})$

1. Pad a with n zeroes to make it a $(2n-1)$ degree polynomial $a = (a_0, a_1, \ldots, a_{n-1}, a_n, a_{n+1}, \ldots, a_{2n-1})$. Similarly for b.
2. Compute values of a and b at the $2n$th roots of unity.
3. Compute sample representation of product. That is $c' = (a'_0 b'_0, a'_1 b'_1, \ldots, a'_{n-1} b'_{n-1}, \ldots, a'_{2n-1} b'_{2n-1})$.
4. Compute coefficients of unique polynomial associated with sample representation of product. That is compute c from c'.

- Step 1 takes $O(n \log n)$ using divide and conquer algorithm
- Step 2 takes $O(n)$ time

Convolutions and Polynomial Multiplication

Convolutions

Compute convolution $c = (c_0, c_1, \ldots, c_{2n-2})$ of $a = (a_0, a_1, \ldots a_{n-1})$ and $b = (b_0, b_1, \ldots b_{n-1})$

1. Pad a with n zeroes to make it a $(2n-1)$ degree polynomial $a = (a_0, a_1, \ldots, a_{n-1}, a_n, a_{n+1}, \ldots, a_{2n-1})$. Similarly for b.
2. Compute values of a and b at the $2n$th roots of unity.
3. Compute sample representation of product. That is $c' = (a'_0 b'_0, a'_1 b'_1, \ldots, a'_{n-1} b'_{n-1}, \ldots, a'_{2n-1} b'_{2n-1})$.
4. Compute coefficients of unique polynomial associated with sample representation of product. That is compute c from c'.

- Step 1 takes $O(n \log n)$ using divide and conquer algorithm
- Step 2 takes $O(n)$ time
- Step 3??

Inverse Fourier Transform

Input Given the evaluation of a $2n-1$-degree polynomial c on the $2n$th roots of unity

Goal Compute the coefficients of c

Inverse Fourier Transform

Input Given the evaluation of a $2n-1$-degree polynomial c on the $2n$th roots of unity

Goal Compute the coefficients of c

We saw that a' can be computed from a in $O(n \log n)$ time. Can we compute a from a' in $O(n \log n)$ time?

A Matrix Point of View

$a'_0 = a(x_0), a'_1 = a(x_1), \ldots, a'_{n-1} = a(x_{n-1})$ where $x_j = \omega_n^j$.
Let $\omega_n^1 = e^{2\pi/n} = \omega$.

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_j & x_j^2 & \ldots & x_j^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \ldots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_j \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} a'_0 \\ a'_1 \\ \vdots \\ a'_j \\ \vdots \\ a'_{n-1} \end{bmatrix}$$

A Matrix Point of View

$a'_0 = a(x_0), a'_1 = a(x_1), \ldots, a'_{n-1} = a(x_{n-1})$ where $x_j = \omega_n^j$.
Let $\omega_n^1 = e^{2\pi/n} = \omega$.

$$\begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega & \omega^2 & \ldots & \omega^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^j & \omega^{2j} & \ldots & \omega^{j(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \ldots & \omega^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_j \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} a'_0 \\ a'_1 \\ \vdots \\ a'_j \\ \vdots \\ a'_{n-1} \end{bmatrix}$$

Inverting the Matrix

$$\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_j \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega & \omega^2 & \ldots & \omega^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^j & \omega^{2j} & \ldots & \omega^{j(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \ldots & \omega^{(n-1)(n-1)} \end{bmatrix}^{-1} \begin{bmatrix} a'_0 \\ a'_1 \\ \vdots \\ a'_j \\ \vdots \\ a'_{n-1} \end{bmatrix}$$

Inverting the Matrix

$$\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^j & \omega^{2j} & \cdots & \omega^{j(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix}^{-1} = \frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \cdots & \omega^{-(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-j} & \omega^{-2j} & \cdots & \omega^{-j(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \cdots & \omega^{-(n-1)(n-1)} \end{bmatrix}$$

Replace ω by ω^{-1} which is also a root of unity!
Inverse matrix is simply a permutation of the original matrix modulo scale factor $1/n$.

Why does it work?

Can check using simple algebra $VV^{-1} = I$ where V is the orignal matrix and I is the $n \times n$ identity matrix.

$$(1, \omega^j, \omega^{2j}, \ldots, \omega^{j(n-1)}) \cdot (1, \omega^{-k}, \omega^{-2k}, \ldots, \omega^{-k(n-1)}) = \sum_{s=0}^{n-1} \omega^{(j-k)s}$$

Note that ω^{j-k} is a n'th root of unity. If $j = k$ then sum is n, otherwise by previous observation sum is 0.

Rows of matrix V (and hence also those of V^{-1}) are *orthogonal*. Thus $a' = Va$ can be thought of a transforming the vector a into a new Fourier basis with basis vectors corresponding to rows of V.

Inverse Fourier Transform

Input Given the evaluation of a $2n-1$-degree polynomial c on the $2n$th roots of unity

Goal Compute the coefficients of c

Inverse Fourier Transform

Input Given the evaluation of a $2n-1$-degree polynomial c on the $2n$th roots of unity

Goal Compute the coefficients of c

We saw that a' can be computed from a in $O(n \log n)$ time. Can we compute a from a' in $O(n \log n)$ time?

Yes! $a = V^{-1}a$ which is simply a permuted and scaled version of DFT. Hence can be computed in $O(n \log n)$ time.

Convolutions Once More

Convolutions

Compute convolution of $a = (a_0, a_1, \ldots a_{n-1})$ and $b = (b_0, b_1, \ldots b_{n-1})$

1. Compute values of a and b at the $2n$th roots of unity
2. Compute sample representation c' of product $c = a \cdot b$
3. Compute c from c' using inverse Fourier transform.

Convolutions Once More

Convolutions

Compute convolution of $a = (a_0, a_1, \ldots a_{n-1})$ and $b = (b_0, b_1, \ldots b_{n-1})$

1. Compute values of a and b at the $2n$th roots of unity
2. Compute sample representation c' of product $c = a \cdot b$
3. Compute c from c' using inverse Fourier transform.

Convolutions Once More

Convolutions

Compute convolution of $a = (a_0, a_1, \ldots a_{n-1})$ and $b = (b_0, b_1, \ldots b_{n-1})$

1. Compute values of a and b at the $2n$th roots of unity
2. Compute sample representation c' of product $c = a \cdot b$
3. Compute c from c' using inverse Fourier transform.

- Step 1 takes $O(n \log n)$ using two FFTs

Convolutions Once More

Convolutions

Compute convolution of $a = (a_0, a_1, \ldots a_{n-1})$ and $b = (b_0, b_1, \ldots b_{n-1})$

1. Compute values of a and b at the $2n$th roots of unity
2. Compute sample representation c' of product $c = a \cdot b$
3. Compute c from c' using inverse Fourier transform.

- Step 1 takes $O(n \log n)$ using two FFTs
- Step 2 takes $O(n)$ time

Convolutions Once More

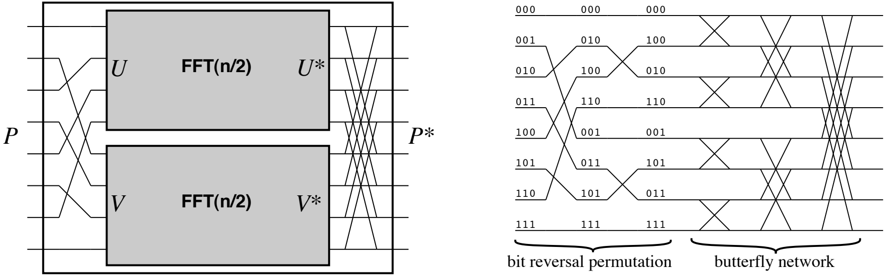
Convolutions

Compute convolution of $a = (a_0, a_1, \ldots a_{n-1})$ and $b = (b_0, b_1, \ldots b_{n-1})$

1. Compute values of a and b at the $2n$th roots of unity
2. Compute sample representation c' of product $c = a \cdot b$
3. Compute c from c' using inverse Fourier transform.

- Step 1 takes $O(n \log n)$ using two FFTs
- Step 2 takes $O(n)$ time
- Step 3 takes $O(n \log n)$ using one FFT

FFT Circuit



The recursive structure of the FFT algorithm.

Numerical Issues

- As noted earlier evaluating a polynomial a at a point x makes numbers big
- Are we cheating when we say $O(n \log n)$ algorithm for convolution?
- Can get around numerical issues — work in finite fields and avoid numbers growing too big.
- Outside the scope of class.