# CSE 421
# Algorithms

Richard Anderson

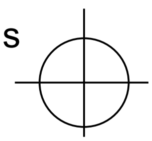
Lecture 15

Fast Fourier Transform

## FFT, Convolution and Polynomial Multiplication

- FFT: O(n log n) algorithm
  - Evaluate a polynomial of degree n at n points in O(n log n) time
- Polynomial Multiplication: O(n log n) time

## Complex Analysis



- Polar coordinates: $re^{\theta i}$
- $e^{\theta i} = \cos\theta + i \sin\theta$
- a is an $n^{th}$ root of unity if $a^n = 1$
- Square roots of unity: +1, -1
- Fourth roots of unity: +1, -1, i, -i
  - Eighth roots of unity: +1, -1, i, -i, $\beta + i\beta$, $\beta - i\beta$, $-\beta + i\beta$, $-\beta - i\beta$ where $\beta$ = sqrt(2)

## $e^{2\pi ki/n}$

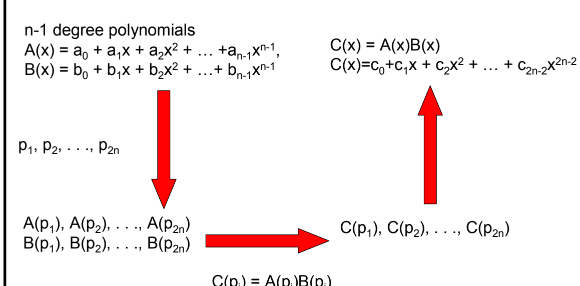
- $e^{2\pi i} = 1$
- $e^{\pi i} = -1$
- $n^{th}$ roots of unity: $e^{2\pi ki/n}$ for k = 0 …n-1
- Notation: $\omega_{k,n} = e^{2\pi ki/n}$
- Interesting fact:

  $1 + \omega_{k,n} + \omega^2_{k,n} + \omega^3_{k,n} + \ldots + \omega^{n-1}_{k,n} = 0$ for k != 0

## FFT Overview

- Polynomial interpolation
  - Given n+1 points $(x_i, y_i)$, there is a unique polynomial P of degree at most n which satisfies $P(x_i) = y_i$

## Polynomial Multiplication



## FFT

- Polynomial $A(x) = a_0 + a_1x + . . . + a_{n-1}x^{n-1}$
- Compute $A(\omega_{j,n})$ for $j = 0, . . ., n-1$
- For simplicity, n is a power of 2

## Useful trick

$A(x) = a_0 + a_1x + a_2x^2 + a_3x^3 +. . . + a_{n-1}x^{n-1}$

$A_{even}(x) = a_0 + a_2x + a_4x^2 + . . . + a_{n-2}x^{(n-2)/2}$

$A_{odd}(x) = a_1+ a_3x + a_5x^2 + …+ a_{n-1}x^{(n-2)/2}$

Show: $A(x) = A_{even}(x^2) + x A_{odd}(x^2)$

## Lemma: $\omega^2_{j,2n} = \omega_{j,n}$

Squares of $2n^{th}$ roots of unity are $n^{th}$ roots of unity

## FFT Algorithm

// Evaluate the $2n-1^{th}$ degree polynomial A at
// $\omega_{0,2n}, \omega_{1,2n}, \omega_{2,2n}, . . ., \omega_{2n-1,2n}$
FFT(A, 2n)
  Recursively compute FFT($A_{even}$, n)
  Recursively compute FFT($A_{odd}$, n)

  for j = 0 to 2n-1
    $A(\omega_{j,2n}) = A_{even}(\omega^2_{j,2n}) + \omega_{j,2n}A_{odd}(\omega^2_{j,2n})$

## Polynomial Multiplication

- $n-1^{th}$ degree polynomials A and B
- Evaluate A and B at $\omega_{0,2n}, \omega_{1,2n}, . . ., \omega_{2n-1,2n}$
- Compute $C(\omega_{j,2n})$ for j = 0 to 2n -1
- We know the value of a $2n-2^{th}$ degree polynomial at 2n points – this determines a unique polynomial, we just need to determine the coefficients

## Now the magic happens . . .

$C(x) = c_0 + c_1x + c_2x^2 + … + c_{2n-1}x^{2n-1}$
(we want to compute the $c_i$'s)
Let $d_j = C(\omega_{j,2n})$
$D(x) = d_0 + d_1x + d_2x^2 + … + d_{2n-1}x^{2n-1}$
Evaluate D(x) at the $2n^{th}$ roots of unity

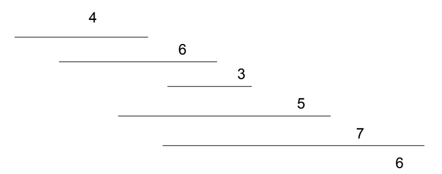
$D(\omega_{j,2n})$ = [see text for details] $= 2nc_{2n-j}$

## Polynomial Interpolation

- Build polynomial from the values of C at the $2n^{th}$ roots of unity
- Evaluate this polynomial at the $2n^{th}$ roots of unity

## Dynamic Programming

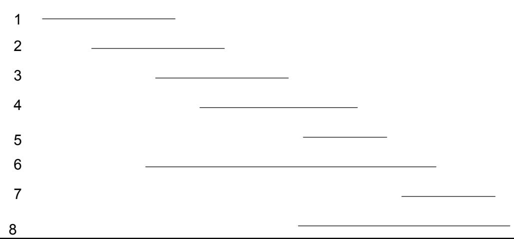
- Weighted Interval Scheduling
- Given a collection of intervals $I_1,\ldots,I_n$ with weights $w_1,\ldots,w_n$, choose a maximum weight set of non-overlapping intervals




## Recursive Algorithm

Intervals sorted by finish time

p[i] is the index of the last interval which finishes before i starts




## Optimality Condition

- Opt[j] is the maximum weight independent set of intervals $I_1, I_2, \ldots, I_j$

## Algorithm

MaxValue(j) =
  if j = 0 return 0
  else
    return max( MaxValue(j-1),
        $w_j$ + MaxValue(p[ j ]))

## Run time

- What is the worst case run time of MaxValue
- Design a worst case input

# A better algorithm

M[ j ] initialized to -1 before the first recursive call for all j

```
MaxValue(j) =
   if j = 0 return 0;
   else if M[ j ] != -1 return M[ j ];
   else
        M[ j ] = max(MaxValue(j-1),w_j + MaxValue(p[ j ]));
        return M[ j ];
```