Introduction to Algorithms

6.046J/18.410J

Prof. Shafi Goldwasser

LECTURE 3

Fast Fourier Transform
Fast Fourier Transform

- Revolutionary Impact on Signal Processing
  
- Gives rise to Fast multiplication of polynomials
  
- Gives rise to Fast multiplication of integers
The Complexity of Computing on Polynomials

Evaluation: $O(n)$ using Horner’s rule

$$A(x) = a_{n-1}x^{n-1} + \ldots a_1 x + a_0$$

Degree $n-1$ Polynomials

Addition: $O(n)$

$$A(x) + B(x) = \sum_{i=0}^{n-1} (a_i + b_i)x^i = \sum_{i=0}^{n-1} c_i x^i$$

Multiplication (convolution): naïve $O(n^2)$.

$$A(x) * B(x) = \sum_{i=0}^{n-1} \left( \sum_{j=0}^{i} a_j b_{i-j} \right) x^i = \sum_{i=0}^{n-1} c_i x^i$$
Plan: Polynomial Multiplication

- Last time using Divide and Conquer developed an $O(n^{1+\varepsilon})$ algorithm for $0<\varepsilon<1$

- **Today:** using FFT develop an $O(n \log n)$ algorithm for polynomial multiplication

First Idea: Use an alternative Representation of polynomials
Alternative Representation of Polynomials: Point-Value Representation.

**Theorem** A polynomial of degree \( t \) is uniquely determined by (and can be interpolated from) its value at \( t+1 \) distinct points (generalizes 2 points determine a line etc.).

**Corollary:** An alternative representation to listing the coefficients \((a_0...a_{n-1})\) of an \( n-1 \) degree polynomial \( A \), is to give the value of the polynomial in any \( n \) distinct points \((x_i, A(x_i))\) for \( i=1,..,n\).
Why Use Alternative Representation of

**Theorem** A polynomial of degree $t$ is uniquely determined by (and can be interpolated from) its value at $t+1$ distinct points (2 points determine a line etc..)

Why : Polynomial Multiplication (convolution) in the point-value representation is more efficient.
How Compute on Polynomials using the Point-Value Representation

**Addition:** \( C=A+B \) is of degree \( n \) \( \Rightarrow \) need to specify it at \( n \) points

\[
C: (x_0, y_0 + y'_0), (x_1, y_1 + y'_1), \ldots, (x_{n-1}, y_{n-1} + y'_{n-1})
\]

Complexity: \( O(n) \)

**Multiplication (Convolution):** \( C=A*B \) is of degree \( 2n-2 \) \( \Rightarrow \) need to specify \( C \)'s value at \( 2n-1 \) points.

Given \( A \)'s values \( (y_0, y_1, \ldots, y_{2n-1}) \) at \( 2n-1 \) points \( x_0, x_1, \ldots, x_{2n-1} \) &

B's values \( (y'_0, y'_1, \ldots, y'_{2n-1}) \) at \( 2n-1 \) points

Set \( C \)'s values \( (y_0 y'_0, y_1 y'_1, \ldots, y_{2n-1} y'_{2n-1}) \) at \( 2n-1 \) points

Complexity: \( 2n-1=O(n) \) multiplications !!!
How Compute on Polynomials using the Point-Value Representation

A: \((x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\)
B: \((x_0, y'_0), (x_1, y'_1), \ldots, (x_{n-1}, y'_{n-1})\)

2 Degree n-1 Polynomials Evaluated at the same points \(x_0, x_1, \ldots, x_{n-1}\)

Addition: \(C = A + B\) is of degree \(n\) \(\Rightarrow\) need to specify it at \(n\) points

\(C: (x_0, y_0 + y'_0), (x_1, y_1 + y'_1), \ldots, (x_{n-1}, y_{n-1} + y'_{n-1})\)

Complexity: \(O(n)\)

Multiplication(Convolution): \(C = A \times B\) is of degree \((2n - 2)\) \(\Rightarrow\) need to specify \(C\)'s value at \((2n - 1)\) points.

Given A’s values \((y_0, y_1, \ldots, y_{2n-1})\) at \(2n-1\) points
B’s values \((y'_0, y'_1, \ldots, y'_{2n-1})\) at \(2n-1\) points

Set C’s values \((y_0 y'_0, y_1 y'_1, \ldots, y_{2n-1} y'_{2n-1})\)

Multiplications rather than \(\text{Deg}^2 = O(n^2)\)

Complexity: \(2n - 1 = O(n)\) multiplications !!
Example: n=2

A(x): \[ A(0)=a_0, \quad A(1)=a_1+a_2+a_0, \quad A(2)=4a_2+2a_1+a_0 \]

B(x): \[ B(0)=b_0, \quad B(1)=b_1+b_2+b_0, \quad B(2)=4b_2+2b_1+b_0 \]

Point wise add:

A(x)+B(x): \[ A(0)+B(0), \quad A(1)+B(1), \quad A(2)+B(2) \]

Point wise Multiply:

A(x)*B(x): has deg 4, must be specified at deg+1=5 pts
Evaluate A: A(0), A(1), A(2), A(-1), A(-2)
Evaluate B: B(0), B(1), B(2), B(-1), B(-2)
Multiply \( A(i)*B(i) \) for \( i=0,1,2,-1,-2 \)

5=2deg+1
Multiplications
Rather than
9=deg^2 !!!
Are we done? Not so fast…

- **Need to obtain extra evaluations**: what’s the complexity of polynomial evaluation when its given in point/value rep. at \((x_i,y_i)\) \(i=1..n-1\) ?

\[
A(x) = \sum_{i=1}^{n} y_i \prod_{\substack{j=1 \atop j \neq i}}^{n} \frac{x - x_j}{x_i - x_j} \quad \text{O}(n^2)
\]

- **Input Polynomials was in Coefficient rep**: need to convert to point/value rep and back.
- Horner’s rule Evaluation of \(n\) pts. \(\text{O}(n^2)\)
- Lagrange interpolation \(\text{O}(n^2)\) [from above formula]
Can we get both fast Evaluation and Multiplication

<table>
<thead>
<tr>
<th>Representation</th>
<th>Multiplication</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coefficient</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Point-value</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

\[a_0, \ldots a_{n-1}, b_0, \ldots b_{n-1}\]

\[C(x_0) \ldots C(x_{2n-1})\]

\[A(x_0) \ldots A(x_{2n-1}), B(x_0) \ldots B(x_{2n-1})\]

**EVALUATION FFT** $O(n\log n)$

**INTERPOLATION Inverse FFT** $O(n\log n)$

**Coefficient Multiplication** $O(n^2)$

**Pt. Value Multiplication** $O(n)$
Overview of Plan

Let input polynomials

\[ A(x) = A_{n-1}x^{n-1} + \ldots + A_0 \]
\[ B(x) = B_{n-1}x^{n-1} + \ldots + B_0 \]

Convert polynomials \(A(x)\) and \(B(x)\) to point-Value representation by evaluating \(A(x)\) and \(B(x)\) at \(2n-1\) points

Multiply point wise and set \(C(x) = A(x) \times B(x)\)
At \(2n-1\) points \(x\)

Convert \((x, C(x))\) at \(2n-1\) points to coefficient representation of \(C\) by interpolation

\(O(n)\)
Disturbing Question

Q: How can we evaluate the polynomial in $2n-1$ points in $O(n \log n)$ time when evaluating the polynomial in 1 point takes $O(n)$?

Key Insight: Choose points $x_1 \ldots x_{2n-1}$ carefully s.t. simultaneous evaluations of at these points are cheap.
Plan for Multiplication in $O(n \log n)$

On Input: $A$ and $B$ in coefficient form, let $m=2n-1$

1. Choose $m$ special points $x_1 \ldots x_m$ (how?)

2. **FFT**: Evaluate $A$ and $B$ in special points

3. Compute $C(x_0) \ldots C(x_m)$ where $C(x_i) = A(x_i)B(x_i)$

4. **Inverse FFT**: Interpolate to get coefficients of $C$. 

\[
\begin{align*}
\mathbf{a}_0, \ldots, \mathbf{a}_{n-1} \\
\mathbf{b}_0, \ldots, \mathbf{b}_{n-1} \\
\mathbf{c}_0, \ldots, \mathbf{c}_{n-1}
\end{align*}
\]
General Lesson: sometime it may be useful to look for different representations where operations are more efficient
FFT: Divide and Conquer
(assume n even)

Divide $A(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1}$

into $A_{\text{EVEN}}(y)$ and $A_{\text{ODD}}(y)$ polynomials of degree $n/2-1$

- $A_{\text{EVEN}}(y) = a_0 + a_2 y + a_4 y^2 + \ldots + a_{n-2} y^{n/2-1}$ \hspace{1cm} \text{EVEN coeff}
- $A_{\text{ODD}}(y) = a_1 + a_3 y + a_5 y^2 + \ldots + a_{n-1} y^{n/2-1}$ \hspace{1cm} \text{ODD coeff}

Fact: $A(x) = A_{\text{EVEN}}(x^2) + x \ A_{\text{ODD}}(x^2)$

Our Goal: Evaluate $A(x_j) = A_{\text{EVEN}}((x_j)^2) + x_j \ A_{\text{ODD}}((x_j)^2)$
at n-1 points $x_0 \ldots x_{n-1}$ in $O(n \log n)$
FFT: DIVIDE and CONQUER

• Trial 1:
  – Divide \( A(x) \) into \( A^{\text{even}}(y) \) and \( A^{\text{odd}}(y) \)
  – Conquer: Recursively evaluate \( A^{\text{even}}((x_j)^2) \) and \( A^{\text{odd}}((x_j)^2) \) for \( j=0 \ldots n-1 \)
  - Combine Compute \( A(x_j) = A^{\text{even}}((x_j)^2) + x_j \cdot A^{\text{odd}}((x_j)^2) \) for \( j=0 \ldots n-1 \)

What is the complexity?

• The two recursive sub-problems evaluate a polynomial with degree \( n/2 \) coefficients at \( n \) points

• The four next level sub-problems evaluate a polynomial with \( n/4 \) coefficients, still at \( n \) points

• At the bottom of the recursion, we have \( n \) sub-problems, each evaluating a polynomial with 1 coefficient at \( n \) points
This takes \( n^2 \) time!
FFT: DIVIDE and CONQUER

• Trial 1:
  - Divide $A(x)$ into $A^{\text{even}}(y)$ and $A^{\text{odd}}(y)$
  - Conquer: Recursively evaluate $A^{\text{even}}((x_j)^2)$ and $A^{\text{odd}}((x_j)^2)$ for $j=0\ldots n-1$
  - Combine Compute $A(x_j) = A^{\text{even}}((x_j)^2) + x_j A^{\text{odd}}((x_j)^2)$ for $j=0\ldots n-1$

What is the complexity?

• The two recursive sub-problems evaluate a polynomial with degree $n/2$ coefficients at $n$ points.

• The four next level sub-problems evaluate a polynomial with $n/4$ coefficients, still at $n$ points.

• At the bottom of the recursion, we have $n$ sub-problems, each evaluating a polynomial with 1 coefficient at $n$ points.

This takes $n^2$ time!

we need to reduce the number of points we evaluate the polynomials at in recursive calls
Reducing the number of points in rec calls?

Idea 2: Choose the n points as $\pm x_0, \ldots \pm x_{n/2-1}$

Observation: $A(x) = A^{EVEN}(x^2) + x A^{ODD}(x^2)$

$A(-x) = A^{EVEN}((-x)^2) - x A^{ODD}((-x)^2)$

So, can get the value of A in 2 points $\pm x$ for the price of 1 recursive call.

Progress: Reduced evaluating polynomial of degree n-1 on n points to evaluating 2 polynomials of degree $n/2-1$ on $n/2$ points

Problem: Can we continue recursively?

Need to choose $\pm x_0, \ldots \pm x_{n/2-1}$ so that $(x_0)^2, \ldots (x_{n/2-1})^2$ are $n/4$ minus-plus pairs, but how can we: they are all positive!

UNLESS we use complex numbers
Example of Complex Numbers with the right properties.

Let $i = \sqrt{-1}$

$$e^{i/8} = e^{\frac{i}{8}}$$

is the 8th root of unity
Magic Points: Complex Roots of Unity

\( w_n = e^{2\pi i/n} \) is the primitive \( n \)-th root of unity, where \( i=\sqrt{-1} \) meaning \( (w_n)^n=1 \), but \( (w_n)^j \neq 1 \) for \( 0<j<n \), The other roots of unity are: \( (\omega_n)^j \) for \( j=0,\ldots,n-1 \)

**Halving Lemma:** if \( n>0 \) is even, then the squares of the \( n \)-th roots of unity are the \( n/2 \)-th roots of unity, i.e.,
\( (\omega_n^j)^2 = \omega_{n/2}^j \) for \( j=0\ldots n-1 \)

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

Therefore, there are only \( n/2 \) distinct elements of the form \( (\omega_n^j)^2 \)
\{ \( (\omega_n^0)^2 \), \( \ldots \), \( (\omega_n^{n/2-1})^2 \), \( (\omega_n^{n/2})^2 \), \ldots, \( (\omega_n^{n-1})^2 \) \} = \{ \( \omega_{n/2}^0 \), \ldots, \( \omega_{n/2}^{n-1} \) \}

Furthermore, \( n \)-th roots are plus-minus paired: for even \( n \)
\( (w_n)^{n/2} = -1 \) \( \Rightarrow \) \( w^{j+n/2} = -w^j \) for \( j=0\ldots n/2 \)
FFT: the final algorithm

On input polynomial $A(x)$ to be evaluated at points $\{\omega_n^0, \ldots, \omega_n^{n-1}\}$

**FFT**
Divide $A$ into $A^{\text{EVEN}}$ and $A^{\text{ODD}}$

**Conquer:** Recursively call FFT on polynomials $A^{\text{EVEN}}(y)$ and $A^{\text{ODD}}(y)$ to be evaluated at points $P=\{(\omega_n^0)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n-1})^2\} = \{\omega_{n/2}^0, \ldots, \omega_{n/2}^{n/2-1}\}$

**Combine** Return for $j=0\ldots n/2-1$

$$A(\omega_n^j) = A^{\text{EVEN}}((\omega_n^j)^2) + \omega_n^j A^{\text{ODD}}((\omega_n^j)^2)$$
$$A(\omega_n^{j+n/2}) = A^{\text{EVEN}}((\omega_n^j)^2) - \omega_n^j A^{\text{ODD}}((\omega_n^j)^2)$$
FFT: Code

FFT(A,n)

1. If n=1, return A(1)

2. Divide A into $A^{\text{EVEN}}$ and $A^{\text{ODD}}$

3. Call FFT($A^{\text{EVEN}}(y), n/2$) and FFT($A^{\text{ODD}}(y), n/2$)

3. Compute $A(\omega_n^j) = A^{\text{EVEN}}((\omega_n^j)^2) + \omega_n^j A^{\text{ODD}}((\omega_n^j)^2)$

$A(\omega_n^{j+n/2}) = A^{\text{EVEN}}((\omega_n^j)^2) - \omega_n^j A^{\text{ODD}}((\omega_n^j)^2)$

for $j=0\ldots n/2-1$

Return $A(\omega_n^0) A(\omega_n^1) \ldots A(\omega_n^{n-1})$
FFT: Analysis

FFT(A,n)

1. If \( n=1 \), return \( A(1) \)
2. Divide \( A \) into \( A^{\text{EVEN}} \) and \( A^{\text{ODD}} \)
3. Call \( FFT(A^{\text{EVEN}}(y),n/2) \) and \( FFT(A^{\text{ODD}}(y),n/2) \)

\[
A(\omega_n^j) = A^{\text{EVEN}}( (\omega_n^j)^2 ) + \omega_n^j A^{\text{ODD}}( (\omega_n^j)^2 )
\]
\[
A(\omega_n^{j+n/2}) = A^{\text{EVEN}}( (\omega_n^j)^2 ) - \omega_n^j A^{\text{ODD}}( (\omega_n^j)^2 )
\]
for \( j=0\ldots n/2-1 \)

Return \( A(\omega_n^0) A(\omega_n^1) \ldots A(\omega_n^{n-1}) \)

Time: \( T(n)=2T(n/2)+O(n) \implies T(n)=O(n \log n) \)
Putting Together: Polynomial multiplication (Convolution) via FFT

Given two polynomials $A(x)$, $B(x)$ of degree $n-1$, compute $C(x) = A(x) \times B(x)$

- I.e., compute $c_i = \sum_{j=0}^{i} a_j b_{i-j}$, for $i=0..2n-1$

Algorithm (use special points):

1. Evaluate $A(\omega_{2n}^0), \ldots, A(\omega_{2n}^{2n-1})$ and $B(\omega_{2n}^0), \ldots, B(\omega_{2n}^{2n-1})$
2. For $i=0 \ldots 2n-1$, compute $C(\omega_{2n}^j) = A(\omega_{2n}^j) \times B(\omega_{2n}^j)$
3. Interpolate $C(x)$ from $C(\omega_{2n}^0), \ldots, C(\omega_{2n}^{2n-1})$

Still Missing: Inverse FFT
Matrix formulation of FFT

- FFT: evaluate $y_j = \sum_k a_k \omega_n^{jk}$ for $j=0\ldots n-1$
  where $\omega_n = e^{2\pi i/n}$

- Implicitly represent this as a matrix-vector product: $y = V*a$

\[
\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} \omega_n^0 & \omega_n^0 & \omega_n^0 \\ \omega_n^0 & \omega_n^1 & \omega_n^{n-1} \\ \vdots & \vdots & \vdots \\ \omega_n^0 & \omega_n^{n-1} & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix}
\]

- $V$ is called a Vandermonde or Fourier matrix and its invertible
Inverse Fourier Transform (IFT)

- Forward transform: \( y = V^*a \)
- Inverse transform: \( a = V^{-1}y \) where \( V^{-1} \) is an inverse of \( V \)

- Amazing coincidence: \( V^{-1} \) looks “almost” like \( V \), except that \( \omega_n^{jk} \) is replaced by \( 1/n \omega_n^{-jk} \) AND \( w_n^{-1} \) is also a primitive root of unity

\[
\begin{bmatrix}
  a_0 \\
a_1 \\
  . \\
  . \\
a_{n-1}
\end{bmatrix} = \frac{1}{n} \begin{bmatrix}
  \omega_n^0 & \omega_n^0 & \omega_n^0 \\
  \omega_n^0 & \omega_n^{-1} & \omega_n^{-(n-1)} \\
  . & . & . \\
  \omega_n^0 & \omega_n^{-(n-1)} & \omega_n^{-(n-1)(n-1)}
\end{bmatrix} \begin{bmatrix}
y_0 \\
y_1 \\
  . \\
  . \\
y_{n-1}
\end{bmatrix}
\]

Can use essentially the same code \( \text{FFT}(y_0 \ldots y_{n-1}) \) for inverse FFT!
Final Analysis: Polynomial multiplication (Convolution) via FFT

Task: given two polynomials $A(x), B(x)$ of degree $n-1$, compute $C(x)=A(x)\times B(x)$

Algorithm (use special points):
1. Evaluate $A(\omega_{2n^0}), \ldots, A(\omega_{2n^{2n-1}})$ and $B(\omega_{2n^0}), \ldots, B(\omega_{2n^{2n-1}})$
2. For $i=0 \ldots 2n-1$, compute $C(\omega_{2n^j})= A(\omega_{2n^j}) \times B(\omega_{2n^j})$
3. Interpolate $C(x)$ from $C(\omega_{2n^0}), \ldots, C(\omega_{2n^{2n-1}})$

Analysis
- Steps 1, 3: $O(n \log n)$, thanks to FFT
- Step 2: $O(n)$
- Total: $O(n \log n)$
Fast Fourier Transform [Origin]

Discrete Fourier transform (DFT) maps a vector \([a_0 \ldots a_{n-1}]\) into \([y_0 \ldots y_{n-1}]\), where

\[ y_j = \sum_{k=0}^{n-1} a_k (w_n^j)^k \quad \text{for } w_n = e^{2\pi i/n} = \cos(2\pi i/n) + is\sin(2\pi i/n) \]

Interpretation

- \([a_0 \ldots a_{n-1}]\): sample of signal in time domain
- \([y_0 \ldots y_{n-1}]\): correlation of signal with sine wave of frequencies \(j=0..n-1\)

Fast Fourier Transform (FFT)

\(O(n \log n)\) divide and conquer algorithm for DFT
Applying to Integer Multiplication

**Input**: 2 n-bit integers $a$ and $b$ grade school algorithm
Seek $O(n \log n \log \log n)$ multiplication algorithm

**Idea 1**: Divide integers into $k$-bit blocks

$$a = A_{k-1} \cdot 2^{(k-1)n/k} + \ldots + A_1 \cdot 2^{n/k} + A_0$$

$$b = B_{k-1} \cdot 2^{(k-1)n/k} + \ldots + B_1 \cdot 2^{n/k} + B_0$$

where $A_i$, $B_i$ are n/k bit integers

Represent Integers as **polynomials**
Reduce integer multiplication to **polynomial multiplication**
How to Represent Integers as Polynomials

Write

\[ a = A_{k-1} 2^{(k-1)n/k} + A_{k-1} 2^{n/k} + A_0 \]
\[ b = B_{k-1} 2^{(k-1)n/k} + B_{k-1} 2^{n/k} + B_0 \]

Define polynomials

\[ A(x) = A_{k-1} x^{k-1} + A_{k-1} x + A_0 \]
\[ B(x) = B_{k-1} x^{(k-1)n} + B_{k-1} x + B_0 \]

Fact: Let \( x = 2^{n/k} \). Then \( a = A(x), b = B(x) \)

Strategy to compute \( a \times b \):

- Multiply polynomials to get \( C = A \times B \)
- Evaluate polynomial \( C(x) \) at \( x = 2^{n/3} \). Then \( C(2^{n/3}) = a \times b \)
Overview of Plan for integer multiplication of 2 n-bit numbers

1. On input integers a, b write (k-1)-degree polynomials A(x) and B (x)

2. Evaluate A(x) and B(x) in O(2k-1) points

3. Multiply in Pt/Value representation of C(x) =A(x) *B(x) in O(2k-1) points

4. Interpolate C(x) from O(2k-1) pt/value pairs

5. Evaluate C(x) at x=2^{n/k}

Poly Multiplication Of (2k-1) Degree Polys
Analysis without using FFT’s:

– We can obtain a sequence of asymptotically faster integer multiplication algorithms by splitting the inputs into larger $k$.

– Splitting $A$ and $B$ into $k$ equal parts, in step 3, we perform $O(2^k - 1)$ inner multiplications of numbers of size $O(n/k)$, leading to recurrence $T(n) = (2^k - 1)T(n/k) + \text{Cost[steps 2,4,5]}$.

– The naïve (without using FFT) cost of steps 2 and 4 is to perform $O(k^2)$ multiplications of $O(n/k)$ bit integers with $O(\log k)$ bit integers (see Horner’s rule and Lagrange interpolation formulas), resulting in $O(kn\log k)$ complexity. To evaluate the resulting polynomial at step 5 is $O(n)$.

    Ultimately, $T(n) = (2^k - 1)T(n/k) + O(kn\log k)$.

– Claim[Toom]: $\forall \varepsilon > 0 \exists c$ s.t for $k = O(c^{\sqrt{n/k}})$, $T(n) = O(n^{(1+\varepsilon)})$.

– Q: why can’t we increase $k$ arbitrarily, in the above?

– A: the cost of naïve evaluation and interpolation will quickly exceed the savings due to [step 3].
Using FFT to do evaluation and interpolation

The obvious next step is to use FFT and IFTT to do evaluation and interpolation steps.

Q: Since one may view steps 2-4 as polynomial multiplication of 2 $O(k)$ degree polynomials, can use the $O(k \log k)$ bound for polynomial multiplication algorithm derived earlier?

A: NO! Earlier, we only counted the number of arithmetic operations on intermediate coefficients and evaluation points, assuming each is $O(1)$ cost. Now, we must account for the cost of operations on the coefficients and the evaluation points as a function of $n$.

A complication that comes up in applying FFT directly to multiplying integers is in the choice of special evaluation points on which multiplication is efficient. Different FFT based integer multiplication algorithms, map the coefficients of $A$ and $B$ to different rings which have primitive roots of order $O(k)$ on which doing multiplication is efficient.
Other Primitive Roots of Unity : Modular Arithmetic

• The Schonhage-Strassen algorithm proposed to avoid complex numbers and use modular arithmetic mod \((2^n')+1\). where \(n' = 2^n\). Such modular arithmetic gives way to special evaluation points as follows.

• Recall, a number \(\omega\) is a **primitive n-th root of unity**, for \(n > 1\), if \(\omega^n = 1\), the numbers 1, \(\omega\), \(\omega^2\), ..., \(\omega^{n-1}\) are all distinct.

• Example:
  
  Z\(^*\)\(\mathbb{Z}\)\(_{11}\):

<table>
<thead>
<tr>
<th>(x)</th>
<th>(x^2)</th>
<th>(x^3)</th>
<th>(x^4)</th>
<th>(x^5)</th>
<th>(x^6)</th>
<th>(x^7)</th>
<th>(x^8)</th>
<th>(x^9)</th>
<th>(x^{10})</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
<td>5</td>
<td>10</td>
<td>9</td>
<td>7</td>
<td>3</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>9</td>
<td>5</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>9</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>9</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>5</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>10</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>6</td>
<td>4</td>
<td>10</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>9</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

– 2, 6, 7, 8 are 10-th roots of unity in \(Z\(^*\)\(\mathbb{Z}\)\(_{11}\)
– \(2^2=4, 6^2=3, 7^2=5, 8^2=9\) are 5-th roots of unity in \(Z\(^*\)\(\mathbb{Z}\)\(_{11}\)
– \(2^{-1}=6, 3^{-1}=4, 4^{-1}=3, 5^{-1}=9, 6^{-1}=2, 7^{-1}=8, 8^{-1}=7, 9^{-1}=5\)
“Rough-Rough” Analysis Using FFT

Let $T(n)$ = time to multiply $n$-bit integers assuming $O(1)$ cost for multiplying $O(\log n)$ bit integers as a base case.

Let $T_1(k) = \text{time for FFT (and IFTT) on } k \text{ coefficients each size } O(n/k)$

Using $k$-bit evaluation points. Set $k=n/k$.

Write input integers $a, b$ as $\deg(2k-1)$ poly $A$ and $B$

Evaluate by FFT $A$ and $B$ in $O(2k-1)$ points

Set $C(x) = A(x) * B(x)$ in $O(2k-1)$ points

Interpolate $C$ by IFFT from $O(2k-1)$ points

Evaluate $C(x)$ at $x=2^{n/k}$

Then $T(n) \approx (2k-1)T(n/k) + O(n \log n) = O((n \log n)(\log \log n))$
Conclusions and final remarks

- FFT computes Fourier Transform in $O(n \log n)$ time
- Very efficient implementations, e.g. FFTW (developed at MIT)

- Is it optimal? Nobody knows…

- However, can do better if signal has only $k$ “large” frequencies, $k << n$
  - $O(k \log^{O(1)} n)$ time
  - Time sub-linear in $n$