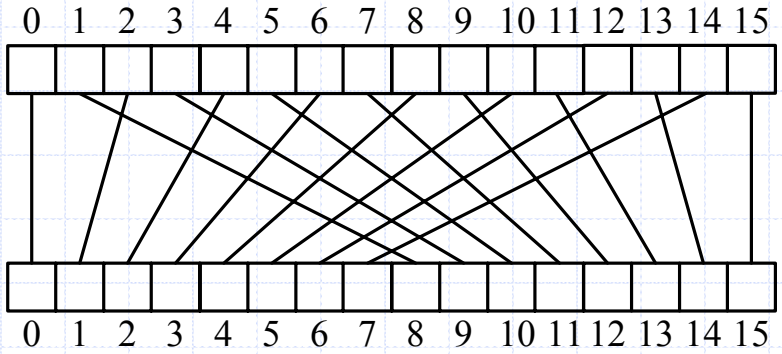
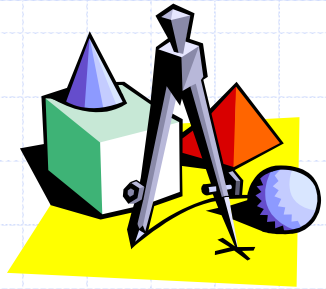
# The Fast Fourier Transform

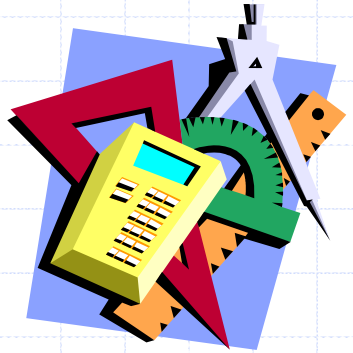


# Outline and Reading



- Polynomial Multiplication Problem
- Primitive Roots of Unity (§10.4.1)
- The Discrete Fourier Transform (§10.4.2)
- The FFT Algorithm (§10.4.3)
- Integer Multiplication (§10.4.4)
- Java FFT Integer Multiplication (§10.5)

# Polynomials



- Polynomial:

$$p(x) = 5 + 2x + 8x^2 + 3x^3 + 4x^4$$

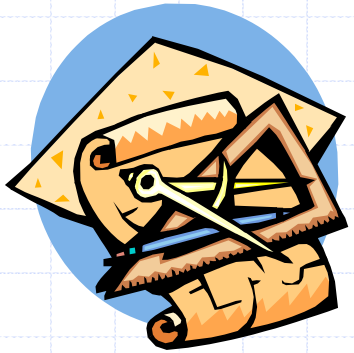
- In general,

$$p(x) = \sum_{i=0}^{n-1} a_i x^i$$

or

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

# Polynomial Evaluation



- **Horner's Rule:**
  - Given coefficients $(a_0,a_1,a_2,\ldots,a_{n-1})$, defining polynomial

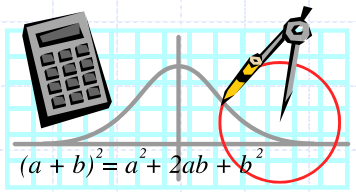
$$p(x)=\sum_{i=0}^{n-1}a_i x^i$$

  - Given x, we can evaluate p(x) in O(n) time using the equation

$$p(x)=a_0+x(a_1+x(a_2+\cdots+x(a_{n-2}+xa_{n-1})\cdots))$$

- **Eval**(A,x): [Where $A=(a_0,a_1,a_2,\ldots,a_{n-1})$]
  - If n=1, then return $a_0$
  - Else,
    - Let $A'=(a_1,a_2,\ldots,a_{n-1})$ [assume this can be done in constant time]
    - return $a_0$+x***Eval**(A',x)

# Polynomial Multiplication Problem




- Given coefficients $(a_0,a_1,a_2,\ldots,a_{n-1})$ and $(b_0,b_1,b_2,\ldots,b_{n-1})$ defining two polynomials, p() and q(), and number x, compute p(x)q(x).

- Horner's rule doesn't help, since

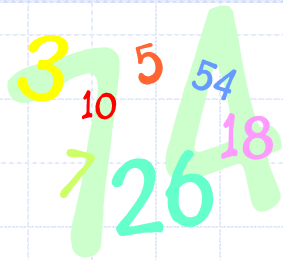
$$p(x)q(x) = \sum_{i=0}^{n-1} c_i x^i$$

where

$$c_i = \sum_{j=0}^{i} a_j b_{i-j}$$

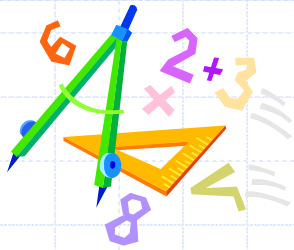
- A straightforward evaluation would take $O(n^2)$ time. The "magical" FFT will do it in O(n log n) time.

# Polynomial Interpolation & Polynomial Multiplication




- Given a set of n points in the plane with distinct x-coordinates, there is **exactly one** (n-1)-degree polynomial going through all these points.
- Alternate approach to computing p(x)q(x):
  - Calculate p() on 2n x-values, $x_0,x_1,\ldots,x_{2n-1}$.
  - Calculate q() on the same 2n x values.
  - Find the (2n-1)-degree polynomial that goes through the points $\{(x_0,p(x_0)q(x_0)), (x_1,p(x_1)q(x_1)), \ldots, (x_{2n-1},p(x_{2n-1})q(x_{2n-1}))\}$.
- Unfortunately, a straightforward evaluation would still take $O(n^2)$ time, as we would need to apply an O(n)-time Horner's Rule evaluation to 2n different points.
- The "magical" FFT will do it in O(n log n) time, by picking 2n points that are easy to evaluate…

# Primitive Roots of Unity



- 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 1:
  - $Z^*_{11}$:

| x | x^2 | x^3 | x^4 | x^5 | x^6 | x^7 | x^8 | x^9 | x^10 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 4 | 8 | 5 | 10 | 9 | 7 | 3 | 6 | 1 |
| 3 | 9 | 5 | 4 | 1 | 3 | 9 | 5 | 4 | 1 |
| 4 | 5 | 9 | 3 | 1 | 4 | 5 | 9 | 3 | 1 |
| 5 | 3 | 4 | 9 | 1 | 5 | 3 | 4 | 9 | 1 |
| 6 | 3 | 7 | 9 | 10 | 5 | 8 | 4 | 2 | 1 |
| 7 | 5 | 2 | 3 | 10 | 4 | 6 | 9 | 8 | 1 |
| 8 | 9 | 6 | 4 | 10 | 3 | 2 | 5 | 7 | 1 |
| 9 | 4 | 3 | 5 | 1 | 9 | 4 | 3 | 5 | 1 |
| 10 | 1 | 10 | 1 | 10 | 1 | 10 | 1 | 10 | 1 |

  - 2, 6, 7, 8 are 10-th roots of unity in $Z^*_{11}$
  - $2^2=4$, $6^2=3$, $7^2=5$, $8^2=9$ are 5-th roots of unity in $Z^*_{11}$
  - $2^{-1}=6$, $3^{-1}=4$, $4^{-1}=3$, $5^{-1}=9$, $6^{-1}=2$, $7^{-1}=8$, $8^{-1}=7$, $9^{-1}=5$
- Example 2: The complex number $e^{2\pi i/n}$ is a primitive n-th root of unity, where $i = \sqrt{-1}$

# Properties of Primitive Roots of Unity

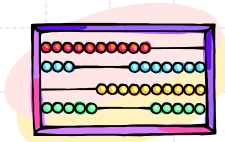
- **Inverse Property:** If $\omega$ is a primitive root of unity, then $\omega^{-1}=\omega^{n-1}$
  - Proof: $\omega\omega^{n-1}=\omega^{n}=1$
- **Cancellation Property:** For non-zero -n<k<n, $\sum_{j=0}^{n-1}\omega^{kj}=0$
  - Proof:

$$\sum_{j=0}^{n-1}\omega^{kj}=\frac{(\omega^{k})^{n}-1}{\omega^{k}-1}=\frac{(\omega^{n})^{k}-1}{\omega^{k}-1}=\frac{(1)^{k}-1}{\omega^{k}-1}=\frac{1-1}{\omega^{k}-1}=0$$

- **Reduction Property:** If w is a primitve (2n)-th root of unity, then $\omega^2$ is a primitive n-th root of unity.
  - Proof: If $1,\omega,\omega^2,\ldots,\omega^{2n-1}$ are all distinct, so are $1,\omega^2,(\omega^2)^2,\ldots,(\omega^2)^{n-1}$
- **Reflective Property:** If n is even, then $\omega^{n/2}=-1$.
  - Proof: By the cancellation property, for k=n/2:

$$0=\sum_{j=0}^{n-1}\omega^{(n/2)j}=\omega^{0}+\omega^{n/2}+\omega^{0}+\omega^{n/2}+\cdots+\omega^{0}+\omega^{n/2}=(n/2)(1+\omega^{n/2})$$

  - Corollary: $\omega^{k+n/2}=-\omega^{k}$.

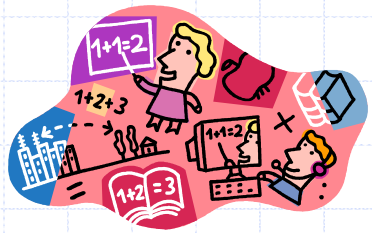


# The Discrete Fourier Transform

- Given coefficients $(a_0,a_1,a_2,\ldots,a_{n-1})$ for an (n-1)-degree polynomial p(x)
- The **Discrete Fourier Transform** is to evaluate p at the values
  - $1,\omega,\omega^2,\ldots,\omega^{n-1}$
  - We produce $(y_0,y_1,y_2,\ldots,y_{n-1})$, where $y_j=p(\omega^j)$
  - That is, $$y_j = \sum_{i=0}^{n-1} a_i \omega^{ij}$$
  - Matrix form: **y=Fa,** where **F**$[i,j]=\omega^{ij}$.

- The **Inverse Discrete Fourier Transform** recovers the coefficients of an (n-1)-degree polynomial given its values at $1,\omega,\omega^2,\ldots,\omega^{n-1}$
  - Matrix form: **a=F** $^{-1}$**y**, where **F** $^{-1}[i,j]=\omega^{-ij}/n$.

# Correctness of the inverse DFT




- The DFT and inverse DFT really are inverse operations
- Proof: Let $\mathbf{A=F^{-1}F}$. We want to show that $\mathbf{A=I}$, where

$$A[i,j] = \frac{1}{n}\sum_{k=0}^{n-1}\omega^{-ki}\omega^{kj}$$

- If i=j, then

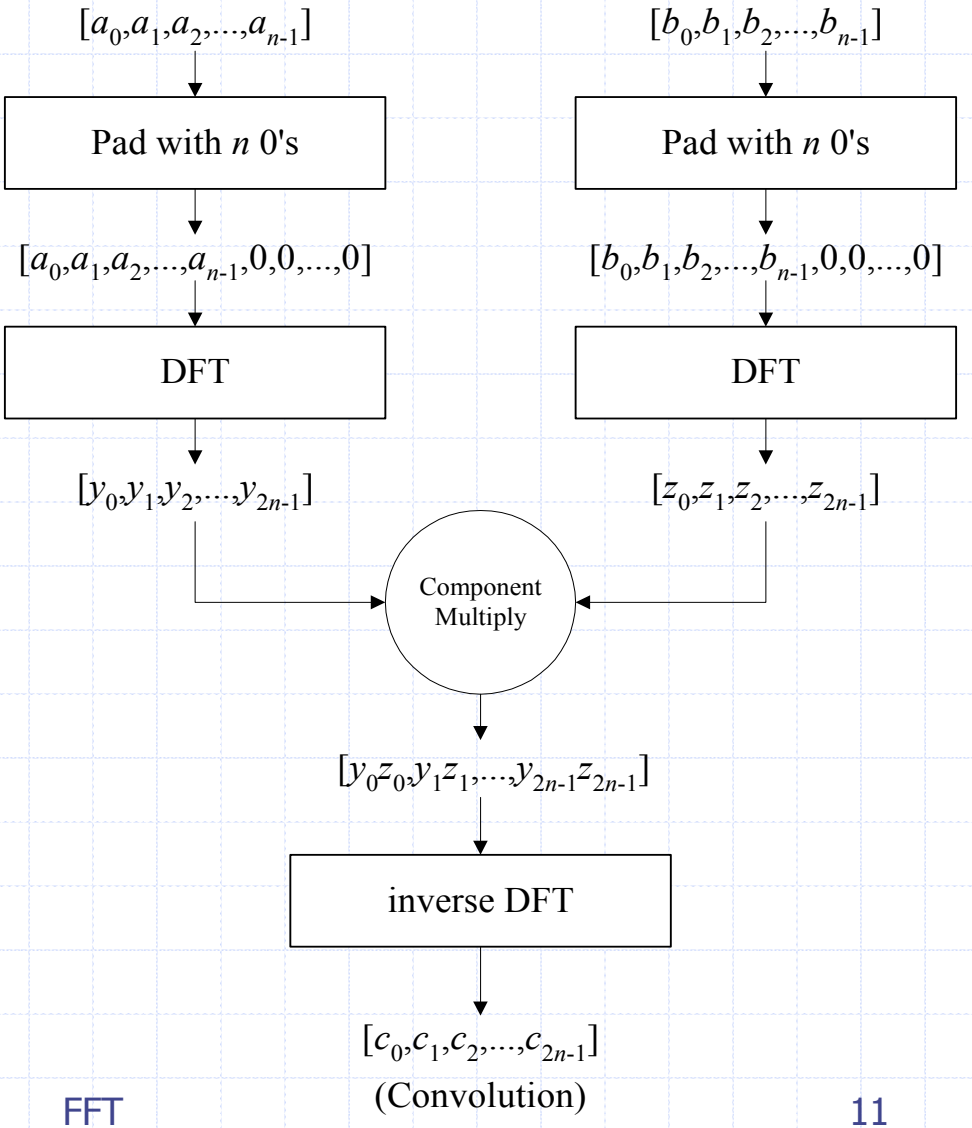
$$A[i,i] = \frac{1}{n}\sum_{k=0}^{n-1}\omega^{-ki}\omega^{ki} = \frac{1}{n}\sum_{k=0}^{n-1}\omega^{0} = \frac{1}{n}n = 1$$

- If i and j are different, then

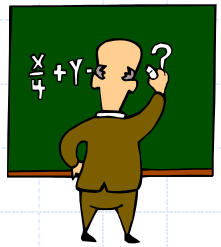
$$A[i,j] = \frac{1}{n}\sum_{k=0}^{n-1}\omega^{(j-i)k} = 0 \qquad \text{(by Cancellation Property)}$$

# Convolution

- The DFT and the inverse DFT can be used to multiply two polynomials
- So we can get the coefficients of the product polynomial quickly if we can compute the DFT (and its inverse) quickly…



# The Fast Fourier Transform



- The FFT is an efficient algorithm for computing the DFT
- The FFT is based on the divide-and-conquer paradigm:
  - If n is even, we can divide a polynomial

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

into two polynomials

$$\begin{aligned} p^{\text{even}}(x) &= a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1} \\ p^{\text{odd}}(x) &= a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1} \end{aligned}$$

and we can write

$$p(x) = p^{\text{even}}(x^2) + x p^{\text{odd}}(x^2).$$

# The FFT Algorithm

**Algorithm** FFT$(\mathbf{a}, \omega)$:

***Input:*** An $n$-length coefficient vector $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}]$ and a primitive $n$th root of unity $\omega$, where $n$ is a power of 2

***Output:*** A vector $\mathbf{y}$ of values of the polynomial for $\mathbf{a}$ at the $n$th roots of unity

**if** $n = 1$ **then**
  **return** $\mathbf{y} = \mathbf{a}$.
$x \leftarrow \omega^0$ {$x$ will store powers of $\omega$, so initially $x = 1$.}
{Divide Step, which separates even and odd indices}
$\mathbf{a}^{\text{even}} \leftarrow [a_0, a_2, a_4, \ldots, a_{n-2}]$
$\mathbf{a}^{\text{odd}} \leftarrow [a_1, a_3, a_5, \ldots, a_{n-1}]$
{Recursive Calls, with $\omega^2$ as $(n/2)$th root of unity, by the reduction property}
$\mathbf{y}^{\text{even}} \leftarrow \text{FFT}(\mathbf{a}^{\text{even}}, \omega^2)$
$\mathbf{y}^{\text{odd}} \leftarrow \text{FFT}(\mathbf{a}^{\text{odd}}, \omega^2)$
{Combine Step, using $x = \omega^i$}
**for** $i \leftarrow 0$ to $n/2 - 1$ **do**
  $y_i \leftarrow y_i^{\text{even}} + x \cdot y_i^{\text{odd}}$
  $y_{i+n/2} \leftarrow y_i^{\text{even}} - x \cdot y_i^{\text{odd}}$ {Uses reflective property}
  $x \leftarrow x \cdot \omega$
**return** $\mathbf{y}$

The running time is O(n log n). [inverse FFT is similar]

# Multiplying Big Integers

- Given N-bit integers I and J, compute IJ.
- Assume: we can multiply words of O(log N) bits in constant time.
- Setup: Find a prime p=cn+1 that can be represented in one word, and set m=(log p)/3, so that we can view I and J as n-length vectors of m-bit words.
- Finding a primitive root of unity.
  - Find a generator x of $Z^*_p$.
  - Then $\omega = x^c$ is a primitive n-th root of unity in $Z^*_p$ (arithmetic is mod p)
- Apply convolution and FFT algorithm to compute the convolution C of the vector representations of I and J.
- Then compute $K = \sum_{i=0}^{n-1} c_i 2^{mi}$
- K is a vector representing IJ, and takes O(n log n) time to compute.

# Java Example: Multiplying Big Integers

- Setup: Define BigInt class, and include essential parameters, including the prime, P, and primitive root of unity, OMEGA.

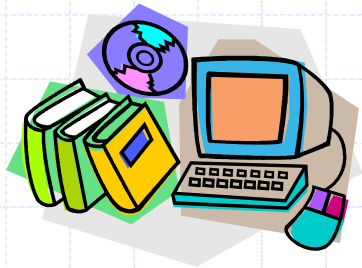
```
import java.lang.*;
import java.math.*;
import java.util.*;

public class BigInt {
  protected int signum=0;                  // neg = -1, 0 = 0, pos = 1
  protected int[] mag;                     // magnitude in little-endian format
  public final static int MAXN=134217728;  // Maximum value for n
  public final static int ENTRYSIZE= 10;   // Bits per entry in mag
  protected final static long P=2013265921; // The prime 15*2^{27}+1
  protected final static int OMEGA=440564289; // Root of unity 31^{15} mod P
  protected final static int TWOINV=1006632961; // 2^{-1} mod P
```

# Java Integer Multiply Method

- Use convolution to multiply two big integers, this and val:

```
public BigInt multiply(BigInt val) {
  int n = makePowerOfTwo(Math.max(mag.length,val.mag.length))*2;
  int signResult = signum * val.signum;
  int[] A = padWithZeros(mag,n);   // copies mag into A padded w/ 0's
  int[] B = padWithZeros(val.mag,n); // copies val.mag into B padded w/ 0's
  int[] root = rootsOfUnity(n);      // creates all n roots of unity
  int[] C = new int[n];              // result array for A*B
  int[] AF = new int[n];             // result array for FFT of A
  int[] BF = new int[n];             // result array for FFT of B
  FFT(A,root,n,0,AF);
  FFT(B,root,n,0,BF);
  for (int i=0; i<n; i++)
    AF[i] = (int)(((long)AF[i]*(long)BF[i]) % P);   // Component multiply
  reverseRoots(root);                // Reverse roots to create inverse roots
  inverseFFT(AF,root,n,0,C);         // Leaves inverse FFT result in C
  propagateCarries(C);               // Convert C to right no. bits per entry
  return new BigInt(signResult,C);
  }
```

# Java FFT in $Z^*_p$



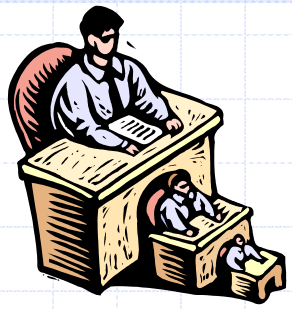
```
public static void FFT(int[] A, int[] root, int n, int base, int[] Y) {
  int prod;
  if (n==1) {
    Y[base] = A[base];
    return;
    }
  inverseShuffle(A,n,base);     // inverse shuffle to separate evens and odds
  FFT(A,root,n/2,base,Y);       // results in Y[base] to Y[base+n/2-1]
  FFT(A,root,n/2,base+n/2,Y); // results in Y[base+n/2] to Y[base+n-1]
  int j = A.length/n;
  for (int i=0; i<n/2; i++) {
    prod = (int)(((long)root[i*j]*Y[base+n/2+i]) % P);
    Y[base+n/2+i] = (int)(((long)Y[base+i] + P - prod) % P);
    Y[base+i] = (int)(((long)Y[base+i] + prod) % P);
    }
}
public static void inverseFFT(int[] A, int[] root, int n, int base, int[] Y) {
  int inverseN = modInverse(n);       // n^{-1}
  FFT(A,root,n,base,Y);
  for (int i=0; i<n; i++)
    Y[i] = (int)(((long)Y[i]*inverseN) % P);
}
```

# Support Methods for Java FFT in $Z^*_p$

```
protected static int modInverse(int n) { // assumes n is power of two
  int result = 1;
  for (long twoPower = 1; twoPower < n; twoPower *= 2)
    result = (int)(((long)result*TWOINV) % P);
  return result;
}
protected static void inverseShuffle(int[] A, int n, int base) {
  int shift;
  int[] sp = new int[n];
  for (int i=0; i<n/2; i++) { // Unshuffle A into the scratch space
    shift = base + 2*i;
    sp[i] = A[shift];          // an even index
    sp[i+n/2] = A[shift+1];  // an odd index
    }
  for (int i=0; i<n; i++)
    A[base+i] = sp[i];         // copy back to A
}
```

```
protected static int[] rootsOfUnity(int n) { //assumes n is power of 2
  int t = MAXN;
  int nthroot = OMEGA;
  for (int t = MAXN; t>n; t /= 2)  // Find prim. nth root of unity
    nthroot = (int)(((long)nthroot*nthroot) % P);
  int[] roots = new int[n];
  int r = 1;              // r will run through all nth roots of unity
  for (int i=0; i<n; i++) {
    roots[i] = r;
    r = (int)(((long)r*nthroot) % P);
    }
  return roots;
}
protected static void propagateCarries(int[] A) {
  int i, carry;
  carry = 0;
  for (i=0; i<A.length; i++) {
    A[i] = A[i] + carry;
    carry = A[i] >>> ENTRYSIZE;
    A[i] = A[i] – (carry << ENTRYSIZE);
    }
  }
```

# Non-recursive FFT



- There is also a non-recursive version of the FFT
  - Performs the FFT in place
  - Precomputes all roots of unity
  - Performs a cumulative collection of shuffles on A and on B prior to the FFT, which amounts to assigning the value at index i to the index bit-reverse(i).
- The code is a bit more complex, but the running time is faster by a constant, due to improved overhead

# Experimental Results

- Log-log scale shows traditional multiply runs in $O(n^2)$ time, while FFT versions are almost linear