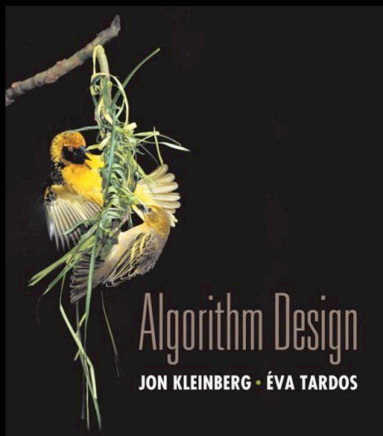


How to Multiply

integers, matrices, and polynomials

Slides by Kevin Wayne.

Complex Multiplication

Complex multiplication. $(a + bi) (c + di) = x + yi$.

Grade-school. $x = a \cdot c - b \cdot d$, $y = b \cdot c + a \cdot d$.

4 multiplications, 2 additions

Gauss. $x = a \cdot c - b \cdot d$, $y = (a + b) \cdot (c + d) - a \cdot c - b \cdot d$.

3 multiplications, 5 additions

Remark. Improvement if no hardware multiply.

5.5 Integer Multiplication

Integer Arithmetic

Add. Given two n-bit integers a and b, compute a + b.
Grade-school. $\Theta(n)$ bit operations.

Multiply. Given two n-bit integers a and b, compute a × b.
Grade-school. $\Theta(n^2)$ bit operations.

1	1	1	1	1	1	0	1	
	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1
1	0	1	0	1	0	0	1	0

Add

Multiply

```
                1 1 0 1 0 1 0 1
              * 0 1 1 1 1 1 0 1
                1 1 0 1 0 1 0 1
              0 0 0 0 0 0 0 0 0
            1 1 0 1 0 1 0 1 0
          1 1 0 1 0 1 0 1 0
        1 1 0 1 0 1 0 1 0
      1 1 0 1 0 1 0 1 0
    1 1 0 1 0 1 0 1 0
  0 0 0 0 0 0 0 0 0
0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 1
```

Divide-and-Conquer Multiplication: Warmup

To multiply two n-bit integers:
- Multiply four $\frac{1}{2}$n-bit integers.
- Add two $\frac{1}{2}$n-bit integers, and shift to obtain result.

$$\begin{aligned} x &= 2^{n/2} \cdot x_1 + x_0 \\ y &= 2^{n/2} \cdot y_1 + y_0 \\ xy &= \left(2^{n/2} \cdot x_1 + x_0\right)\left(2^{n/2} \cdot y_1 + y_0\right) = 2^n \cdot x_1 y_1 + 2^{n/2} \cdot \left(x_1 y_0 + x_0 y_1\right) + x_0 y_0 \end{aligned}$$

Ex. $\mathtt{x} = \underbrace{\mathtt{1000}}_{\mathtt{x_1}}\underbrace{\mathtt{1101}}_{\mathtt{x_0}}$ $\mathtt{y} = \underbrace{\mathtt{1110}}_{\mathtt{y_1}}\underbrace{\mathtt{0001}}_{\mathtt{y_0}}$

$$\mathrm{T}(n) = \underbrace{4T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n)}_{\text{add, shift}} \Rightarrow \mathrm{T}(n) = \Theta(n^2)$$

Karatsuba Multiplication

To multiply two n-bit integers:
- Add two $\frac{1}{2}$n bit integers.
- Multiply three $\frac{1}{2}$n-bit integers.
- Add, subtract, and shift $\frac{1}{2}$n-bit integers to obtain result.

$$\begin{aligned} x &= 2^{n/2} \cdot x_1 + x_0 \\ y &= 2^{n/2} \cdot y_1 + y_0 \\ xy &= 2^n \cdot x_1 y_1 + 2^{n/2} \cdot \left(x_1 y_0 + x_0 y_1\right) + x_0 y_0 \\ &= 2^n \cdot \underset{1}{x_1 y_1} + 2^{n/2} \cdot \left(\underset{2}{(x_1 + x_0)(y_1 + y_0)} - \underset{1}{x_1 y_1} - \underset{3}{x_0 y_0}\right) + \underset{3}{x_0 y_0} \end{aligned}$$

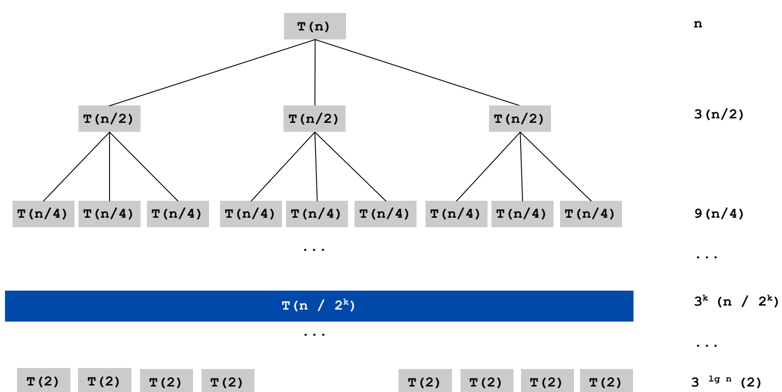
Theorem. [Karatsuba-Ofman, 1962] Can multiply two n-bit integers in $O(n^{1.585})$ bit operations.

$$\mathrm{T}(n) \le \underbrace{T\left(\lfloor n/2 \rfloor\right) + T\left(\lceil n/2 \rceil\right) + T\left(1 + \lceil n/2 \rceil\right)}_{\text{recursive calls}} + \underbrace{\Theta(n)}_{\text{add, subtract, shift}} \Rightarrow \mathrm{T}(n) = O(n^{\log_2 3}) = O(n^{1.585})$$

Karatsuba: Recursion Tree

$$\mathrm{T}(n) = \begin{cases} 0 & \text{if } n = 1 \\ 3T(n/2) + n & \text{otherwise} \end{cases}$$

$$\mathrm{T}(n) = \sum_{k=0}^{\log_2 n} n \left(\tfrac{3}{2}\right)^k = \frac{\left(\tfrac{3}{2}\right)^{1+\log_2 n} - 1}{\tfrac{3}{2} - 1} = 3n^{\log_2 3} - 2$$



Fast Integer Division Too (!)

Integer division. Given two n-bit (or less) integers a and b, compute quotient q = a / b and remainder r = a mod b.

Fact. Complexity of integer division is same as multiplication.
To compute quotient q:
- Approximate x = 1 / b using Newton's method: $x_{i+1} = 2x_i - b\,x_i^2$ using fast multiplication
- After log n iterations, either q = ⌊a x⌋ or q = ⌈a x⌉.

Matrix Multiplication

Matrix Multiplication

Matrix multiplication. Given two n-by-n matrices A and B, compute $\underbrace{C = AB}_{c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}}$.

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

Ex.

$$\begin{bmatrix} .59 & .32 & .41 \\ .31 & .36 & .25 \\ .45 & .31 & .42 \end{bmatrix} = \begin{array}{c} i \\ \\ \\ \end{array} \begin{bmatrix} .70 & .20 & .10 \\ .30 & .60 & .10 \\ .50 & .10 & .40 \end{bmatrix} \times \begin{array}{c} \begin{array}{ccc} & j & \end{array} \\ \begin{bmatrix} .80 & .30 & .50 \\ .10 & .40 & .10 \\ .10 & .30 & .40 \end{bmatrix} \end{array}$$

Brute force. $\Theta(n^3)$ arithmetic operations.

Fundamental question. Can we improve upon brute force?

Matrix Multiplication: Warmup

Divide-and-conquer.
- Divide: partition A and B into $\frac{1}{2}$n-by-$\frac{1}{2}$n blocks.
- Conquer: multiply 8 $\frac{1}{2}$n-by-$\frac{1}{2}$n recursively.
- Combine: add appropriate products using 4 matrix additions.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= (A_{11} \times B_{11}) + (A_{12} \times B_{21}) \\ C_{12} &= (A_{11} \times B_{12}) + (A_{12} \times B_{22}) \\ C_{21} &= (A_{21} \times B_{11}) + (A_{22} \times B_{21}) \\ C_{22} &= (A_{21} \times B_{12}) + (A_{22} \times B_{22}) \end{aligned}$$

$$T(n) = \underbrace{8T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, form submatrices}} \Rightarrow T(n) = \Theta(n^3)$$

Matrix Multiplication: Key Idea

Key idea. multiply 2-by-2 block matrices with only 7 multiplications.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= P_5 + P_4 - P_2 + P_6 \\ C_{12} &= P_1 + P_2 \\ C_{21} &= P_3 + P_4 \\ C_{22} &= P_5 + P_1 - P_3 - P_7 \end{aligned}$$

$$\begin{aligned} P_1 &= A_{11} \times (B_{12} - B_{22}) \\ P_2 &= (A_{11} + A_{12}) \times B_{22} \\ P_3 &= (A_{21} + A_{22}) \times B_{11} \\ P_4 &= A_{22} \times (B_{21} - B_{11}) \\ P_5 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\ P_6 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}) \\ P_7 &= (A_{11} - A_{21}) \times (B_{11} + B_{12}) \end{aligned}$$

- 7 multiplications.
- 18 = 8 + 10 additions (and subtractions).

Fast Matrix Multiplication

Fast matrix multiplication. [Strassen, 1969]

- Divide: partition A and B into $\frac{1}{2}$n-by-$\frac{1}{2}$n blocks.
- Compute: 14 $\frac{1}{2}$n-by-$\frac{1}{2}$n matrices via 10 matrix additions.
- Conquer: multiply 7 pairs of $\frac{1}{2}$n-by-$\frac{1}{2}$n matrices recursively.
- Combine: 7 products into 4 terms using 8 matrix additions.

Analysis.

- Assume n is a power of 2.
- T(n) = # arithmetic operations.

$$T(n) = \underbrace{7T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, subtract}} \Rightarrow T(n) = \Theta(n^{\log_2 7}) = O(n^{2.81})$$

Fast Matrix Multiplication in Practice

Implementation issues.

- Sparsity.
- Caching effects.
- Numerical stability.
- Odd matrix dimensions.
- Crossover to classical algorithm around n = 128.

Common misperception: "Strassen is only a theoretical curiosity."

- Advanced Computation Group at Apple Computer reports 8x speedup on G4 Velocity Engine when n ≈ 2,500.
- Range of instances where it's useful is a subject of controversy.

Remark. Can "Strassenize" Ax=b, determinant, eigenvalues, ….

Fast Matrix Multiplication in Theory

Q. Multiply two 2-by-2 matrices with 7 scalar multiplications?
A. Yes! [Strassen, 1969] $\Theta(n^{\log_2 7}) = O(n^{2.807})$

Q. Multiply two 2-by-2 matrices with 6 scalar multiplications?
A. Impossible. [Hopcroft and Kerr, 1971] $\Theta(n^{\log_2 6}) = O(n^{2.59})$

Q. Two 3-by-3 matrices with 21 scalar multiplications?
A. Also impossible. $\Theta(n^{\log_3 21}) = O(n^{2.77})$

Begun, the decimal wars have. [Pan, Bini et al, Schönhage, …]

- Two 20-by-20 matrices with 4,460 scalar multiplications. $O(n^{2.805})$
- Two 48-by-48 matrices with 47,217 scalar multiplications. $O(n^{2.7801})$
- A year later. $O(n^{2.7799})$
- December, 1979. $O(n^{2.521813})$
- January, 1980. $O(n^{2.521801})$

Fast Matrix Multiplication in Theory

Best known. $O(n^{2.376})$ [Coppersmith-Winograd, 1987]

Conjecture. $O(n^{2+\varepsilon})$ for any $\varepsilon > 0$.

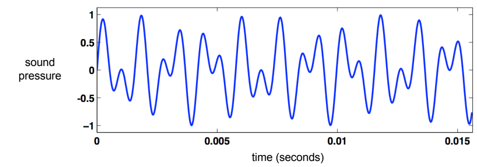
Caveat. Theoretical improvements to Strassen are progressively less practical.

5.6 Convolution and FFT

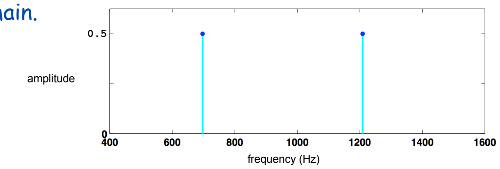
Time Domain vs. Frequency Domain

Signal. [touch tone button 1] $y(t) \;=\; \frac{1}{2}\sin(2\pi \cdot 697\; t) \;+\; \frac{1}{2}\sin(2\pi \cdot 1209\; t)$

Time domain.



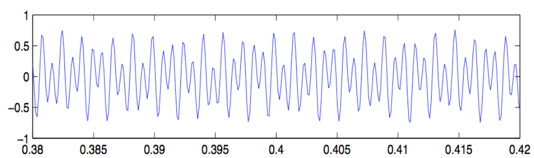
Frequency domain.



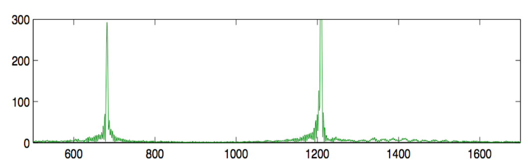
Reference: Cleve Moler, Numerical Computing with MATLAB

Time Domain vs. Frequency Domain

Signal. [recording, 8192 samples per second]



Magnitude of discrete Fourier transform.



Reference: Cleve Moler, Numerical Computing with MATLAB

Fast Fourier Transform

FFT. Fast way to convert between time-domain and frequency-domain.

Alternate viewpoint. Fast way to add, multiply, and evaluate polynomials.

we take this approach

> If you speed up any nontrivial algorithm by a factor of a million or so the world will beat a path towards finding useful applications for it. -Numerical Recipes

Fast Fourier Transform: Applications

Applications.

- Optics, acoustics, quantum physics, telecommunications, radar, control systems, signal processing, speech recognition, data compression, image processing, seismology, mass spectrometry…
- Digital media. [DVD, JPEG, MP3, H.264]
- Medical diagnostics. [MRI, CT, PET scans, ultrasound]
- Numerical solutions to Poisson's equation.
- Shor's quantum factoring algorithm.
- …

> The FFT is one of the truly great computational developments of [the 20th] century. It has changed the face of science and engineering so much that it is not an exaggeration to say that life as we know it would be very different without the FFT. -Charles van Loan

Fast Fourier Transform: Brief History

Gauss (1805, 1866). Analyzed periodic motion of asteroid Ceres.

Runge-König (1924). Laid theoretical groundwork.

Danielson-Lanczos (1942). Efficient algorithm, x-ray crystallography.

Cooley-Tukey (1965). Monitoring nuclear tests in Soviet Union and tracking submarines. Rediscovered and popularized FFT.

Importance not fully realized until advent of digital computers.

Polynomials: Coefficient Representation

Polynomial. [coefficient representation]

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

$$B(x) = b_0 + b_1 x + b_2 x^2 + \cdots + b_{n-1} x^{n-1}$$

Add. O(n) arithmetic operations.

$$A(x) + B(x) = (a_0 + b_0) + (a_1 + b_1) x + \cdots + (a_{n-1} + b_{n-1}) x^{n-1}$$

Evaluate. O(n) using Horner's method.

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

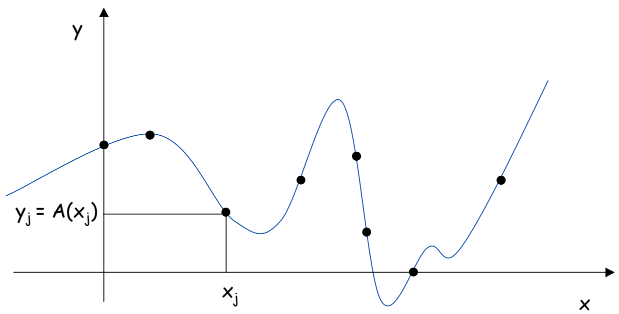
Multiply (convolve). O(n^2) using brute force.

$$A(x) \times B(x) = \sum_{i=0}^{2n-2} c_i x^i, \text{ where } c_i = \sum_{j=0}^{i} a_j b_{i-j}$$

Polynomials: Point-Value Representation

Fundamental theorem of algebra. [Gauss, PhD thesis] A degree n polynomial with complex coefficients has exactly n complex roots.

Corollary. A degree n-1 polynomial A(x) is uniquely specified by its evaluation at n distinct values of x.



Polynomials: Point-Value Representation

Polynomial. [point-value representation]

$$A(x): \ (x_0, y_0), \ \ldots, (x_{n-1}, y_{n-1})$$
$$B(x): \ (x_0, z_0), \ \ldots, (x_{n-1}, z_{n-1})$$

Add. O(n) arithmetic operations.

$$A(x)+B(x): \ (x_0, y_0+z_0), \ldots, (x_{n-1}, y_{n-1}+z_{n-1})$$

Multiply (convolve). O(n), but need 2n-1 points.

$$A(x) \times B(x): \ (x_0, y_0 \times z_0), \ldots, (x_{2n-1}, y_{2n-1} \times z_{2n-1})$$

Evaluate. O(n^2) using Lagrange's formula.

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k} (x - x_j)}{\prod_{j \neq k} (x_k - x_j)}$$

Converting Between Two Polynomial Representations

Tradeoff. Fast evaluation or fast multiplication. We want both!

Representation	Multiply	Evaluate
Coefficient	O(n^2)	O(n)
Point-value	O(n)	O(n^2)

Goal. Make all ops fast by efficiently converting between two representations.

$$a_0, a_1, \ldots, a_{n-1} \rightleftarrows (x_0, y_0), \ \ldots, (x_{n-1}, y_{n-1})$$

coefficient representation

point-value representation

Converting Between Two Representations: Brute Force

Coefficient ⇒ point-value. Given a polynomial $a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$, evaluate it at n distinct points $x_0, \ldots, x_{n-1}$.

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

O(n^2) via matrix-vector multiply

O(n^3) via Gaussian elimination

Vandermonde matrix is invertible iff x_i distinct

Point-value ⇒ coefficient. Given n distinct points $x_0, \ldots, x_{n-1}$ and values $y_0, \ldots, y_{n-1}$, find unique polynomial $a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$ that has given values at given points.

Coefficient to Point-Value Representation: Intuition

Coefficient ⇒ point-value. Given a polynomial $a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$, evaluate it at n distinct points $x_0, \ldots, x_{n-1}$.

we get to choose which ones!

Divide. Break polynomial up into even and odd powers.

- $A(x) \ = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7$.
- $A_{even}(x) = a_0 + a_2x + a_4x^2 + a_6x^3$.
- $A_{odd}(x) = a_1 + a_3x + a_5x^2 + a_7x^3$.
- $A(x) = A_{even}(x^2) + x \, A_{odd}(x^2)$.
- $A(-x) = A_{even}(x^2) - x \, A_{odd}(x^2)$.

Intuition. Choose two points to be ±1.

- $A(1) = A_{even}(1) + 1 \, A_{odd}(1)$.
- $A(-1) = A_{even}(1) - 1 \, A_{odd}(1)$.

Can evaluate polynomial of degree ≤ n at 2 points by evaluating two polynomials of degree ≤ ½n at 1 point.

Coefficient to Point-Value Representation: Intuition

Coefficient $\Rightarrow$ point-value. Given a polynomial $a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$, evaluate it at n distinct points $x_0, \ldots, x_{n-1}$.

we get to choose which ones!

Divide. Break polynomial up into even and odd powers.

- $A(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7$.
- $A_{even}(x) = a_0 + a_2x + a_4x^2 + a_6x^3$.
- $A_{odd}(x) = a_1 + a_3x + a_5x^2 + a_7x^3$.
- $A(x) = A_{even}(x^2) + x\, A_{odd}(x^2)$.
- $A(-x) = A_{even}(x^2) - x\, A_{odd}(x^2)$.

Intuition. Choose four points to be $\pm 1, \pm i$.

- $A(1) = A_{even}(1) + 1\, A_{odd}(1)$.
- $A(-1) = A_{even}(1) - 1\, A_{odd}(1)$.
- $A(i) = A_{even}(-1) + i\, A_{odd}(-1)$.
- $A(-i) = A_{even}(-1) - i\, A_{odd}(-1)$.

Can evaluate polynomial of degree $\leq n$ at 4 points by evaluating two polynomials of degree $\leq \frac{1}{2}n$ at 2 points.

Discrete Fourier Transform

Coefficient $\Rightarrow$ point-value. Given a polynomial $a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$, evaluate it at n distinct points $x_0, \ldots, x_{n-1}$.

Key idea. Choose $x_k = \omega^k$ where ω is principal n^{th} root of unity.

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(n-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \omega^{3(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

Discrete Fourier transform

Fourier matrix F_n

Roots of Unity

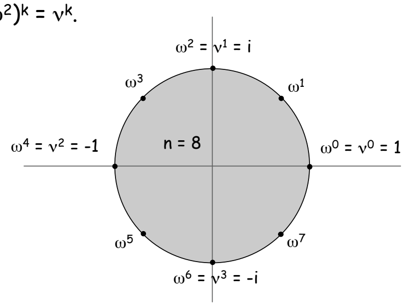
Def. An n^{th} root of unity is a complex number x such that $x^n = 1$.

Fact. The n^{th} roots of unity are: $\omega^0, \omega^1, \ldots, \omega^{n-1}$ where $\omega = e^{2\pi i / n}$.

Pf. $(\omega^k)^n = (e^{2\pi i k / n})^n = (e^{\pi i})^{2k} = (-1)^{2k} = 1$.

Fact. The $\frac{1}{2}n^{th}$ roots of unity are: $\nu^0, \nu^1, \ldots, \nu^{n/2-1}$ where $\nu = e^{4\pi i / n}$.

Note. $\omega^2 = \nu$ and $(\omega^2)^k = \nu^k$.



Fast Fourier Transform

Goal. Evaluate a degree n-1 polynomial $A(x) = a_0 + \ldots + a_{n-1} x^{n-1}$ at its n^{th} roots of unity: $\omega^0, \omega^1, \ldots, \omega^{n-1}$.

Divide. Break up polynomial into even and odd powers.

- $A_{even}(x) = a_0 + a_2x + a_4x^2 + \ldots + a_{n/2-2}\, x^{(n-1)/2}$.
- $A_{odd}(x) = a_1 + a_3x + a_5x^2 + \ldots + a_{n/2-1}\, x^{(n-1)/2}$.
- $A(x) = A_{even}(x^2) + x\, A_{odd}(x^2)$.

Conquer. Evaluate $A_{even}(x)$ and $A_{odd}(x)$ at the $\frac{1}{2}n^{th}$ roots of unity: $\nu^0, \nu^1, \ldots, \nu^{n/2-1}$.

Combine.

- $A(\omega^k) = A_{even}(\nu^k) + \omega^k A_{odd}(\nu^k)$, $0 \leq k < \frac{1}{2}n$
- $A(\omega^{k+\frac{1}{2}n}) = A_{even}(\nu^k) - \omega^k A_{odd}(\nu^k)$, $0 \leq k < \frac{1}{2}n$

$\omega^{k+\frac{1}{2}n} = -\omega^k$

$\nu^k = (\omega^k)^2 = (\omega^{k+\frac{1}{2}n})^2$

FFT Algorithm

```
fft(n, a_0,a_1,…,a_{n-1}) {
   if (n == 1) return a_0

   (e_0,e_1,…,e_{n/2-1}) ← FFT(n/2, a_0,a_2,a_4,…,a_{n-2})
   (d_0,d_1,…,d_{n/2-1}) ← FFT(n/2, a_1,a_3,a_5,…,a_{n-1})

   for k = 0 to n/2 - 1 {
      ω^k ← e^{2πik/n}
      y_k     ← e_k + ω^k d_k
      y_{k+n/2} ← e_k - ω^k d_k
   }

   return (y_0,y_1,…,y_{n-1})
}
```

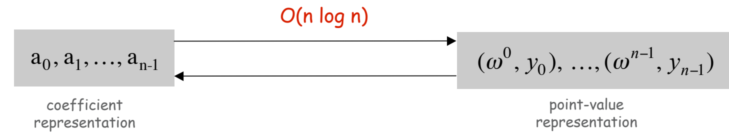

FFT Summary

Theorem. FFT algorithm evaluates a degree n-1 polynomial at each of the n^{th} roots of unity in O(n log n) steps.

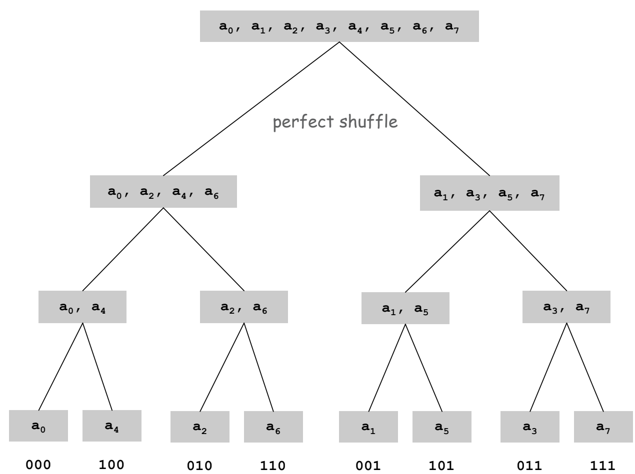
assumes n is a power of 2

Running time.

$$T(n) = 2T(n/2) + \Theta(n) \Rightarrow T(n) = \Theta(n \log n)$$



Recursion Tree



"bit-reversed" order

Point-Value to Coefficient Representation: Inverse DFT

Goal. Given the values $y_0, \ldots, y_{n-1}$ of a degree n-1 polynomial at the n points $\omega^0, \omega^1, \ldots, \omega^{n-1}$, find (unique) polynomial $a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$ that has given values at given points.

$$\underbrace{\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{bmatrix}}_{\text{Inverse DFT}} = \underbrace{\begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(n-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \omega^{3(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix}^{-1}}_{\text{Fourier matrix inverse } (F_n)^{-1}} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{bmatrix}$$

Inverse FFT

Claim. Inverse of Fourier matrix is given by following formula.

$$G_n = \frac{1}{n}\begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \omega^{-3} & \cdots & \omega^{-(n-1)} \\ 1 & \omega^{-2} & \omega^{-4} & \omega^{-6} & \cdots & \omega^{-2(n-1)} \\ 1 & \omega^{-3} & \omega^{-6} & \omega^{-9} & \cdots & \omega^{-3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \omega^{-3(n-1)} & \cdots & \omega^{-(n-1)(n-1)} \end{bmatrix}$$

Consequence. To compute inverse FFT, apply same algorithm but use $\omega^{-1} = e^{-2\pi i / n}$ as principal n^{th} root of unity (and divide by n).

Inverse FFT: Proof of Correctness

Claim. F_n and G_n are inverses.

Pf.

$$\left(F_n\, G_n\right)_{k\,k'} = \frac{1}{n}\sum_{j=0}^{n-1} \omega^{kj}\,\omega^{-jk'} = \frac{1}{n}\sum_{j=0}^{n-1} \omega^{(k-k')j} = \begin{cases} 1 & \text{if } k = k' \\ 0 & \text{otherwise} \end{cases}$$

summation lemma

Summation lemma. Let ω be a principal n^{th} root of unity. Then

$$\sum_{j=0}^{n-1} \omega^{kj} = \begin{cases} n & \text{if } k \equiv 0 \bmod n \\ 0 & \text{otherwise} \end{cases}$$

Pf.

- If k is a multiple of n then $\omega^k = 1 \Rightarrow$ sums to n.
- Each n^{th} root of unity ω^k is a root of $x^n - 1 = (x - 1)(1 + x + x^2 + ... + x^{n-1})$.
- if $\omega^k \neq 1$ we have: $1 + \omega^k + \omega^{k(2)} + \ldots + \omega^{k(n-1)} = 0 \Rightarrow$ sums to 0. ▪

Inverse FFT: Algorithm

```
ifft(n, a_0,a_1,…,a_{n-1}) {
   if (n == 1) return a_0

   (e_0,e_1,…,e_{n/2-1}) ← FFT(n/2, a_0,a_2,a_4,…,a_{n-2})
   (d_0,d_1,…,d_{n/2-1}) ← FFT(n/2, a_1,a_3,a_5,…,a_{n-1})

   for k = 0 to n/2 - 1 {
      ω^k ← e^{-2πik/n}
      y_k      ← (e_k + ω^k d_k) / n
      y_{k+n/2} ← (e_k - ω^k d_k) / n
   }

   return (y_0,y_1,…,y_{n-1})
}
```


Inverse FFT Summary

Theorem. Inverse FFT algorithm interpolates a degree n-1 polynomial given values at each of the n^{th} roots of unity in O(n log n) steps.

assumes n is a power of 2

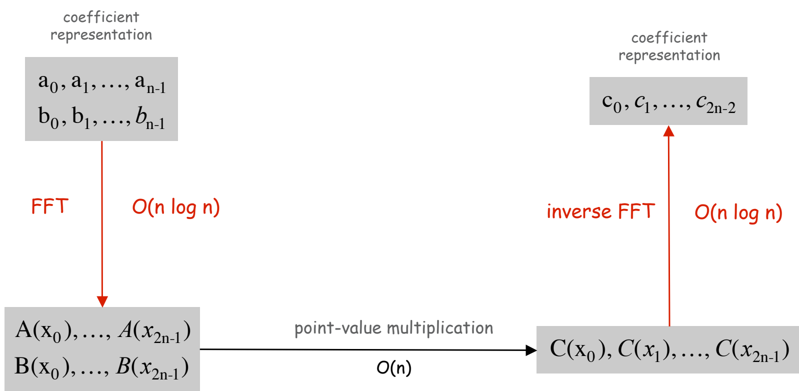
$$a_0, a_1, \ldots, a_{n-1} \underset{O(n \log n)}{\overset{O(n \log n)}{\rightleftarrows}} (\omega^0, y_0), \ldots, (\omega^{n-1}, y_{n-1})$$

coefficient representation

point-value representation

Polynomial Multiplication

Theorem. Can multiply two degree n-1 polynomials in O(n log n) steps.



FFT in Practice

Fastest Fourier transform in the West. [Frigo and Johnson]

- Optimized C library.
- Features: DFT, DCT, real, complex, any size, any dimension.
- Won 1999 Wilkinson Prize for Numerical Software.
- Portable, competitive with vendor-tuned code.

Implementation details.

- Instead of executing predetermined algorithm, it evaluates your hardware and uses a special-purpose compiler to generate an optimized algorithm catered to "shape" of the problem.
- Core algorithm is nonrecursive version of Cooley-Tukey radix 2 FFT.
- O(n log n), even for prime sizes.

Reference: http://www.fftw.org

Integer Multiplication, Redux

Integer multiplication. Given two n bit integers $a = a_{n-1} \ldots a_1 a_0$ and $b = b_{n-1} \ldots b_1 b_0$, compute their product $a \cdot b$.

Convolution algorithm.

- Form two polynomials.
- Note: $a = A(2)$, $b = B(2)$.
- Compute $C(x) = A(x) \cdot B(x)$.
- Evaluate $C(2) = a \cdot b$.
- Running time: O(n log n) complex arithmetic operations.

$$A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$
$$B(x) = b_0 + b_1 x + b_2 x^2 + \cdots + b_{n-1} x^{n-1}$$

Theory. [Schönhage-Strassen 1971] O(n log n log log n) bit operations.

"the fastest bignum library on the planet"

Practice. [GNU Multiple Precision Arithmetic Library]
It uses brute force, Karatsuba, and FFT, depending on the size of n.