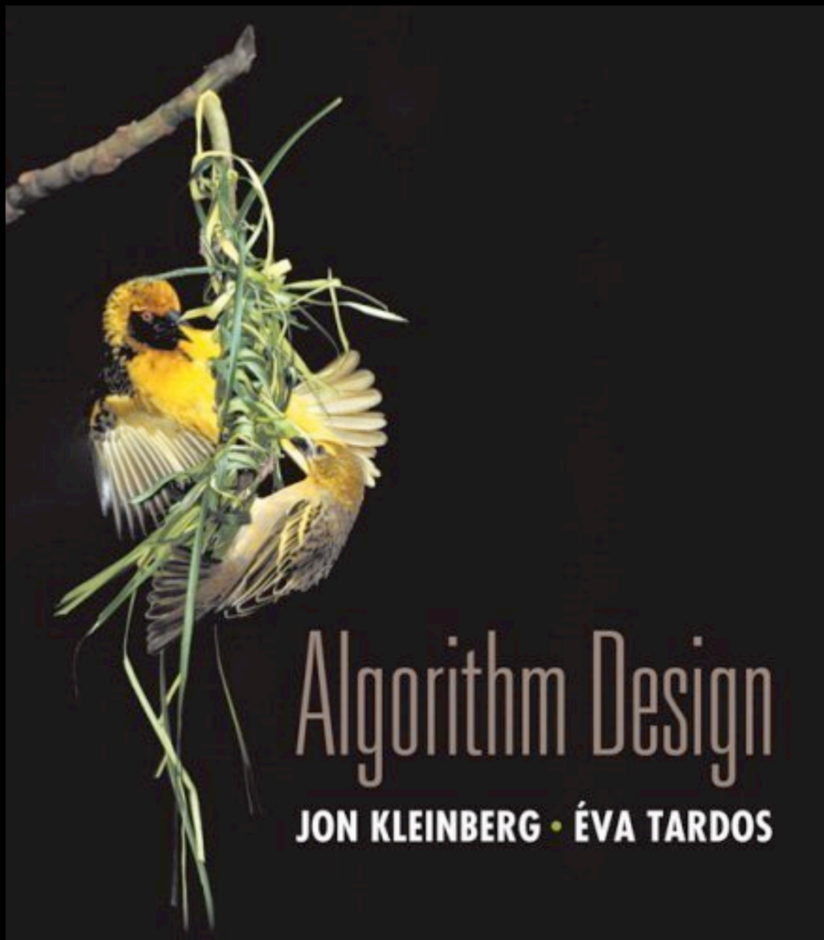



# How to Multiply

integers, matrices, and polynomials

Slides by Kevin Wayne.

# Complex Multiplication

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

**Grade-school.** $x = ac - bd,\ \ y = bc + ad.$

4 multiplications, 2 additions

**Q.** Is it possible to do with fewer multiplications?

# Complex Multiplication

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

Grade-school. $x = ac - bd,\ \ y = bc + ad.$

↖ 4 multiplications, 2 additions

Q. Is it possible to do with fewer multiplications?

A. Yes. [Gauss] $x = ac - bd,\ \ y = (a + b)\,(c + d) - ac - bd.$

↖ 3 multiplications, 5 additions

Remark. Improvement if no hardware multiply.

# 5.5 Integer Multiplication

# Integer Addition

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

Remark. Grade-school addition algorithm is optimal.

# Integer Multiplication

Multiplication. Given two $n$-bit integers $a$ and $b$, compute $a \times b$.

Grade-school. $\Theta(n^2)$ bit operations.

```
                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
```

Q. Is grade-school multiplication algorithm optimal?

# Divide-and-Conquer Multiplication: Warmup

To multiply two $n$-bit integers $a$ and $b$:

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

$$
\begin{aligned}
a &= 2^{n/2} \cdot a_1 + a_0 \\
b &= 2^{n/2} \cdot b_1 + b_0 \\
ab &= \left(2^{n/2} \cdot a_1 + a_0\right)\left(2^{n/2} \cdot b_1 + b_0\right) = 2^n \cdot a_1 b_1 + 2^{n/2} \cdot \left(a_1 b_0 + a_0 b_1\right) + a_0 b_0
\end{aligned}
$$

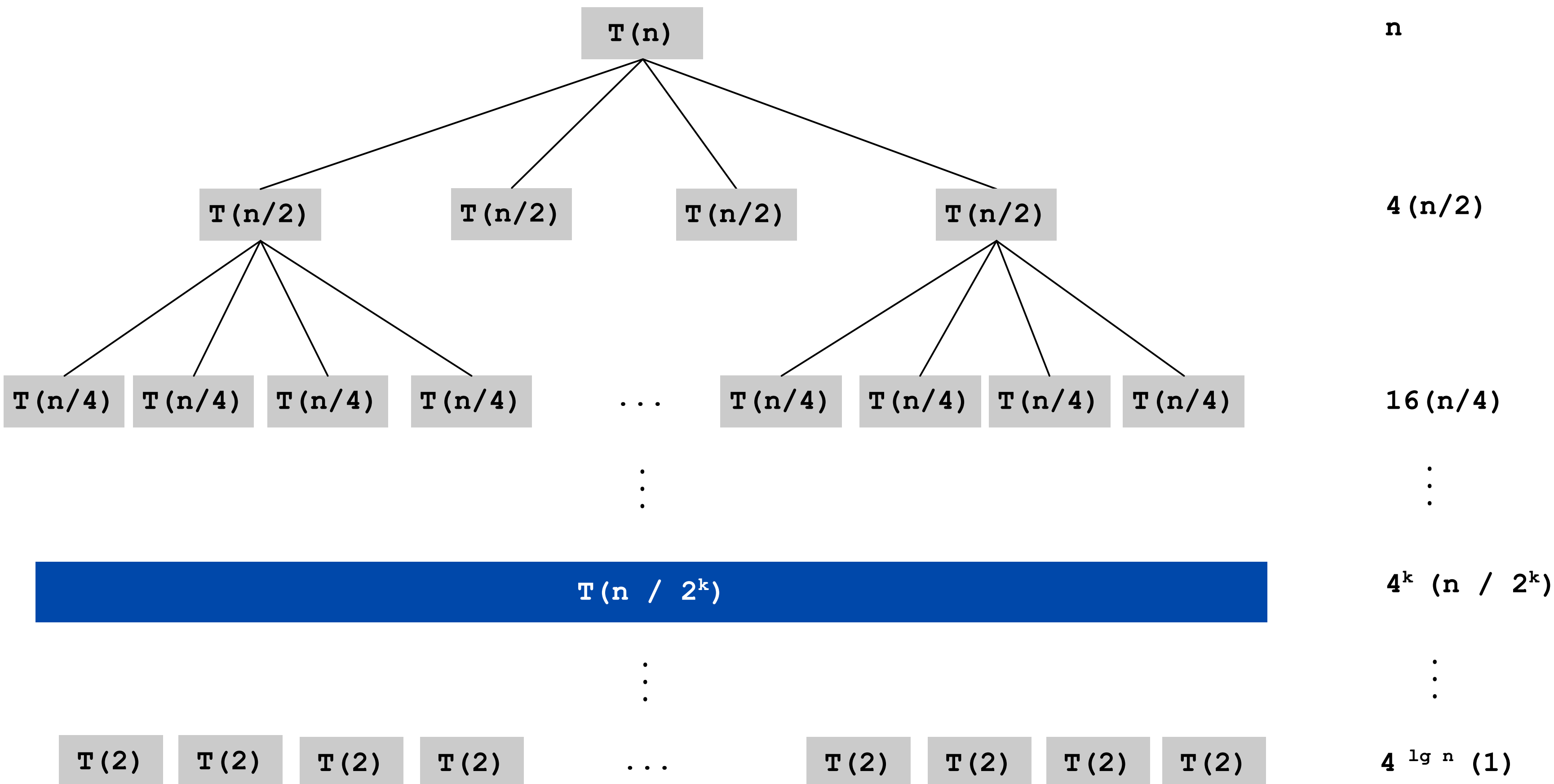
Ex. $\mathbf{a} = \underbrace{\texttt{1000}}_{a_1}\underbrace{\texttt{1101}}_{a_0}$ $\quad \mathbf{b} = \underbrace{\texttt{1110}}_{b_1}\underbrace{\texttt{0001}}_{b_0}$

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

# Recursion Tree

$$T(n) = \begin{cases} 0 & \text{if } n = 0 \\ 4T(n/2) + n & \text{otherwise} \end{cases}$$

$$T(n) = \sum_{k=0}^{\lg n} n\,2^k = n\left(\frac{2^{1+\lg n} - 1}{2 - 1}\right) = 2n^2 - n$$



# Karatsuba Multiplication

To multiply two $n$-bit integers $a$ and $b$:

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

$$
\begin{aligned}
a &= 2^{n/2} \cdot a_1 + a_0 \\
b &= 2^{n/2} \cdot b_1 + b_0 \\
ab &= 2^{n} \cdot a_1 b_1 + 2^{n/2} \cdot \left(a_1 b_0 + a_0 b_1\right) + a_0 b_0 \\
&= 2^{n} \cdot \underbrace{a_1 b_1}_{1} + 2^{n/2} \cdot \left( \underbrace{(a_1 + a_0)(b_1 + b_0)}_{2} - \underbrace{a_1 b_1}_{1} - \underbrace{a_0 b_0}_{3} \right) + \underbrace{a_0 b_0}_{3}
\end{aligned}
$$

# Karatsuba Multiplication

To multiply two $n$-bit integers $a$ and $b$:

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

$$
\begin{aligned}
a &= 2^{n/2} \cdot a_1 + a_0 \\
b &= 2^{n/2} \cdot b_1 + b_0 \\
ab &= 2^{n} \cdot a_1 b_1 + 2^{n/2} \cdot \left(a_1 b_0 + a_0 b_1\right) + a_0 b_0 \\
&= 2^{n} \cdot \underset{①}{a_1 b_1} + 2^{n/2} \cdot \left( \underset{②}{(a_1 + a_0)(b_1 + b_0)} - \underset{①}{a_1 b_1} - \underset{③}{a_0 b_0} \right) + \underset{③}{a_0 b_0}
\end{aligned}
$$

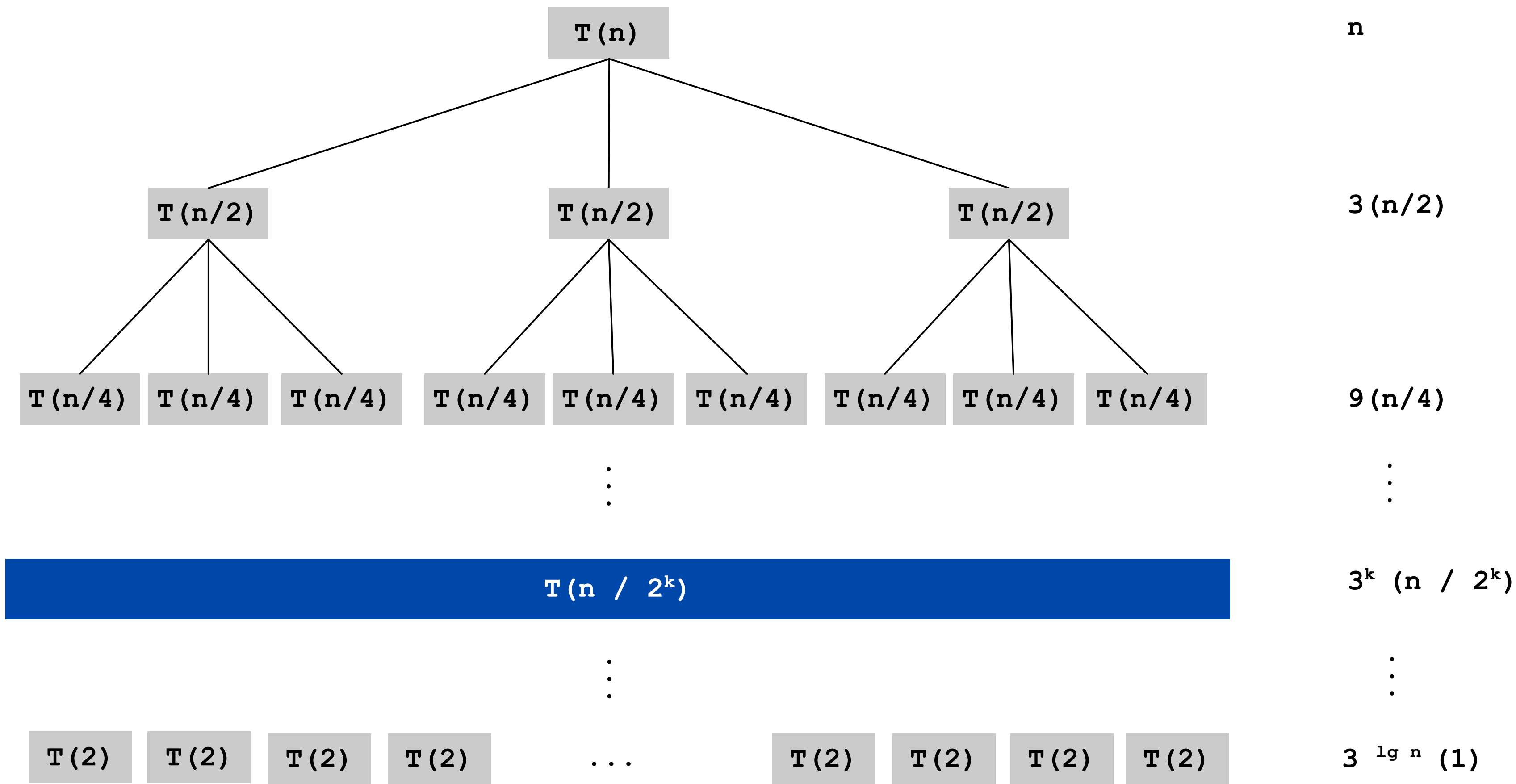
Theorem. [Karatsuba-Ofman 1962] Can multiply two $n$-bit integers in $O(n^{1.585})$ bit operations.

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

# Karatsuba: Recursion Tree

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

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



# Fast Integer Division Too (!)

Integer division. Given two $n$-bit (or less) integers $s$ and $t$, compute quotient $q = s / t$ and remainder $r = s \bmod t$.

Fact. Complexity of integer division is same as integer multiplication.
To compute quotient $q$:

- Approximate $x = 1 / t$ using Newton's method: $x_{i+1} = 2x_i - t\, x_i^2$
- After $\log n$ iterations, either $q = \lfloor s\, x \rfloor$ or $q = \lceil s\, x \rceil$.

↖ using fast multiplication

# Matrix Multiplication

# Dot Product

**Dot product.** Given two length $n$ vectors $a$ and $b$, compute $c = a \cdot b$.

**Grade-school.** $\Theta(n)$ arithmetic operations.

$$a \cdot b = \sum_{i=1}^{n} a_i b_i$$

$$a = [\,.70 \quad .20 \quad .10\,]$$
$$b = [\,.30 \quad .40 \quad .30\,]$$
$$a \cdot b = (.70 \times .30) + (.20 \times .40) + (.10 \times .30) = .32$$

**Remark.** Grade-school dot product algorithm is optimal.

# Matrix Multiplication

Matrix multiplication. Given two $n$-by-$n$ matrices $A$ and $B$, compute $C = AB$.

Grade-school. $\Theta(n^3)$ arithmetic operations.

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

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

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

Q. Is grade-school matrix multiplication algorithm optimal?

# Block Matrix Multiplication

$C_{11}$, $A_{11}$, $A_{12}$, $B_{11}$

$$
\begin{bmatrix} 152 & 158 & 164 & 170 \\ 504 & 526 & 548 & 570 \\ 856 & 894 & 932 & 970 \\ 1208 & 1262 & 1316 & 1370 \end{bmatrix}
=
\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}
\times
\begin{bmatrix} 16 & 17 & 18 & 19 \\ 20 & 21 & 22 & 23 \\ 24 & 25 & 26 & 27 \\ 28 & 29 & 30 & 31 \end{bmatrix}
$$

$B_{11}$

$$
C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21} = \begin{bmatrix} 0 & 1 \\ 4 & 5 \end{bmatrix} \times \begin{bmatrix} 16 & 17 \\ 20 & 21 \end{bmatrix} + \begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix} \times \begin{bmatrix} 24 & 25 \\ 28 & 29 \end{bmatrix} = \begin{bmatrix} 152 & 158 \\ 504 & 526 \end{bmatrix}
$$

# Matrix Multiplication: Warmup

To multiply two $n$-by-$n$ matrices $A$ and $B$:

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

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

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

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

# Fast Matrix Multiplication

Key idea. multiply 2-by-2 blocks with only 7 multiplications.

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

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

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

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

# Fast Matrix Multiplication

To multiply two $n$-by-$n$ matrices $A$ and $B$: [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: 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."*

- Apple reports $8$x speedup on G4 Velocity Engine when $n \approx 2{,}500$.
- Range of instances where it's useful is a subject of controversy.

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

# Fast Matrix Multiplication: 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: Theory

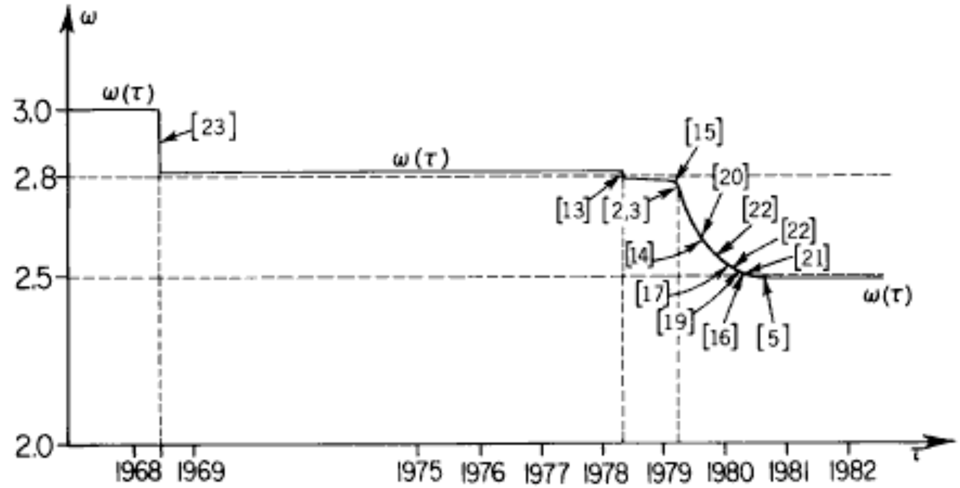



FIG. 1. *ω(t) is the best exponent announced by time τ.*

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

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

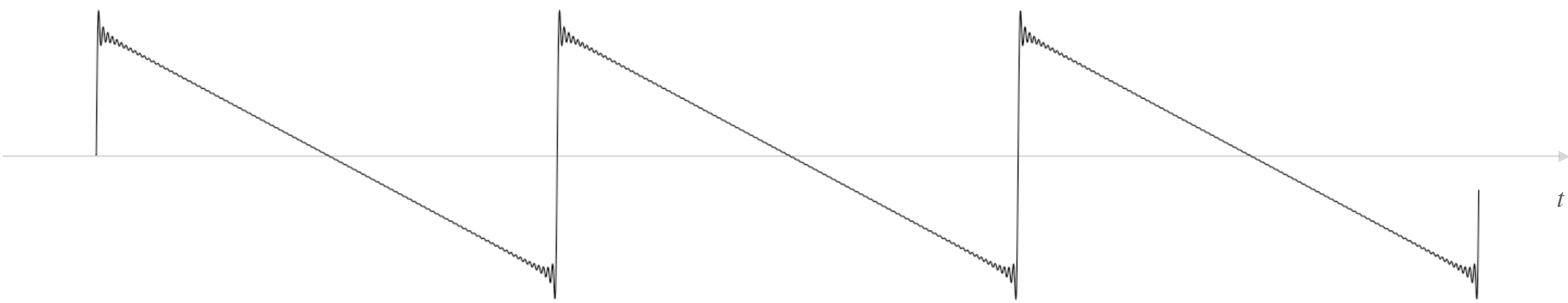
Caveat. Theoretical improvements to Strassen are progressively less practical.

# 5.6 Convolution and FFT

# Fourier Analysis

Fourier theorem. [Fourier, Dirichlet, Riemann] Any periodic function can be expressed as the sum of a series of sinusoids.

sufficiently smooth




$$y(t) = \frac{2}{\pi} \sum_{k=1}^{N} \frac{\sin kt}{k}$$

$N = 100$

# Euler's Identity

Sinusoids. Sum of sine an cosines.

$$e^{ix} = \cos x + i \sin x$$

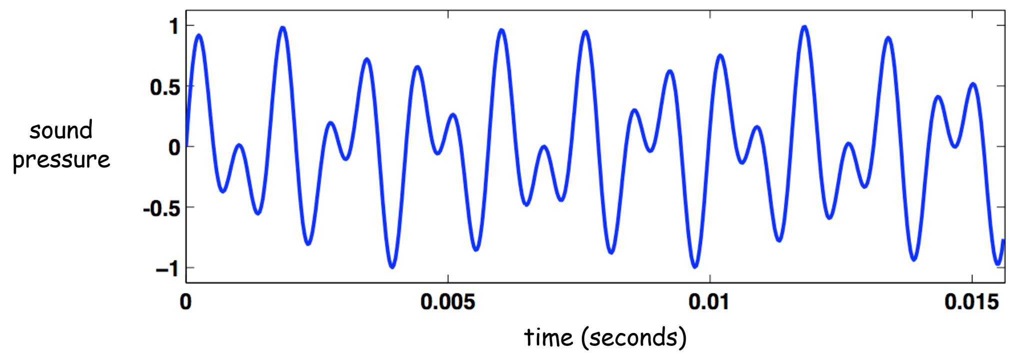
*Euler's identity*

Sinusoids. Sum of complex exponentials.

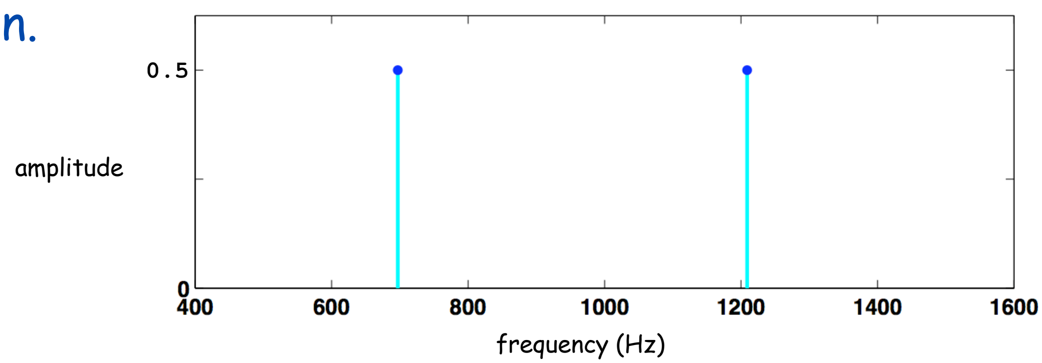
# Time Domain vs. Frequency Domain

**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.**



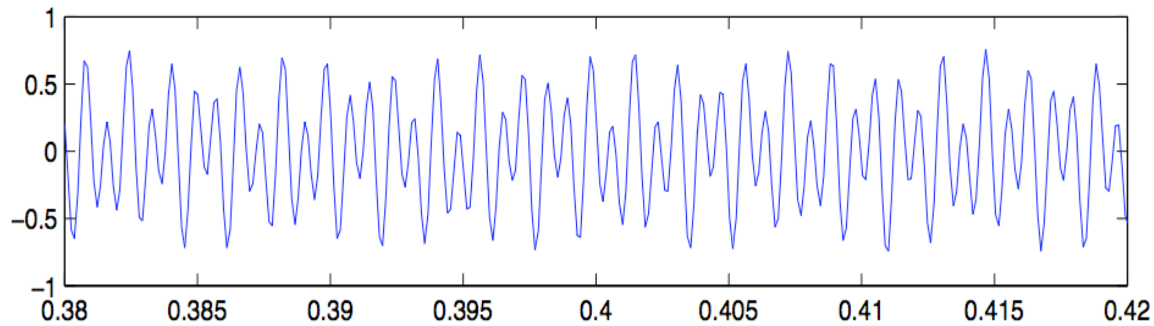

**Frequency domain.**



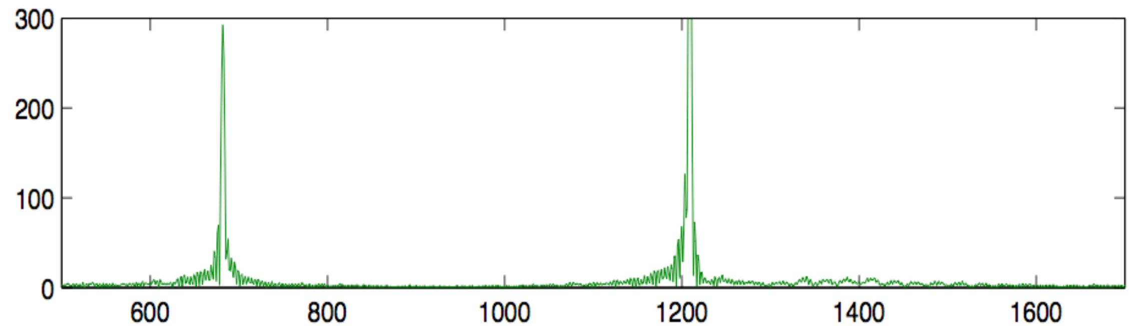

Reference: Cleve Moler, Numerical Computing with MATLAB

# 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 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.

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

# A Modest PhD Dissertation Title

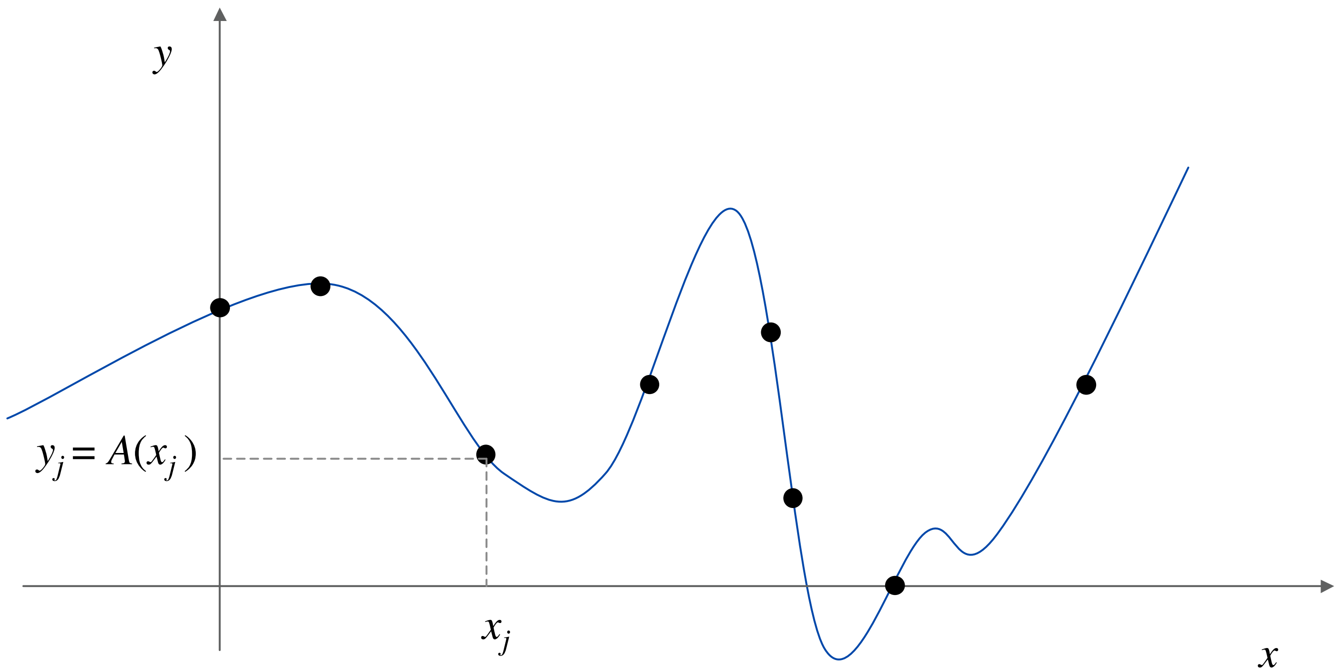
"New Proof of the Theorem That Every Algebraic Rational Integral Function In One Variable can be Resolved into Real Factors of the First or the Second Degree."

- PhD dissertation, 1799 the University of Helmstedt

# 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. Efficient conversion between two representations $\Rightarrow$ all ops fast.

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

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 $x_0, \ldots, x_{n-1}$.

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

Running time. $O(n^2)$ for matrix-vector multiply (or $n$ Horner's).

# Converting Between Two Representations: Brute Force

**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.

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

↑ Vandermonde matrix is invertible iff $x_i$ distinct

**Running time.** $O(n^3)$ for Gaussian elimination.

↑ or $O(n^{2.376})$ via fast matrix multiplication

# Divide-and-Conquer

Decimation in frequency. Break up polynomial into low and high powers.

- $A(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7$.
- $A_{low}(x) = a_0 + a_1x + a_2x^2 + a_3x^3$.
- $A_{high}(x) = a_4 + a_5x + a_6x^2 + a_7x^3$.
- $A(x) = A_{low}(x) + x^4 A_{high}(x)$.

Decimation in time. Break polynomial up into even and odd powers.

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

# Coefficient to Point-Value Representation: Intuition

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

we get to choose which ones!

Divide. Break polynomial up into even and odd powers.

- $A(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5 + a_6 x^6 + a_7 x^7$.
- $A_{even}(x) = a_0 + a_2 x + a_4 x^2 + a_6 x^3$.
- $A_{odd}(x) = a_1 + a_3 x + a_5 x^2 + a_7 x^3$.
- $A(x) = A_{even}(x^2) + x\, A_{odd}(x^2)$.
- $A(-x) = A_{even}(x^2) - x\, A_{odd}(x^2)$.

Intuition. Choose two points to be $\pm 1$.

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

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

# Coefficient to Point-Value Representation: Intuition

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

we get to choose which ones!

Divide. Break polynomial up into even and odd powers.

- $A(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5 + a_6 x^6 + a_7 x^7$.
- $A_{even}(x) = a_0 + a_2 x + a_4 x^2 + a_6 x^3$.
- $A_{odd}(x) = a_1 + a_3 x + a_5 x^2 + a_7 x^3$.
- $A(x) = A_{even}(x^2) + x\, A_{odd}(x^2)$.
- $A(-x) = A_{even}(x^2) - x\, A_{odd}(x^2)$.

Intuition. Choose four complex 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 ≤ $n$ at $4$ points by evaluating two polynomials of degree ≤ $\frac{1}{2}n$ at $2$ points.

# Discrete Fourier Transform

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

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

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

↑ DFT

↑ Fourier matrix $F_n$

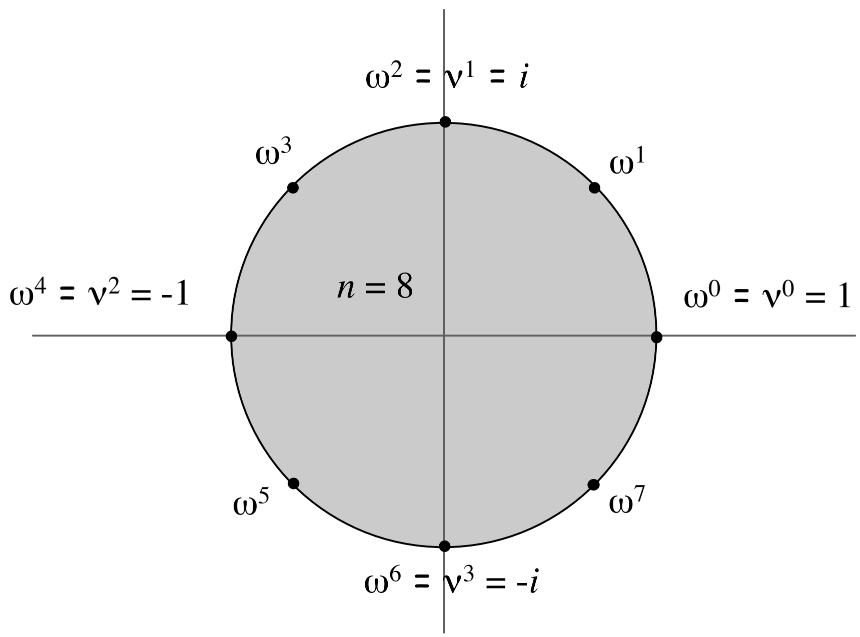
# Roots of Unity

**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 $\nu = \omega^2 = e^{4\pi i / n}$.



# Fast Fourier Transform

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

Divide. Break up polynomial into even and odd powers.

- $A_{even}(x) = a_0 + a_2 x + a_4 x^2 + \ldots + a_{n-2} x^{n/2 - 1}$.
- $A_{odd}(x) = a_1 + a_3 x + a_5 x^2 + \ldots + a_{n-1} x^{n/2 - 1}$.
- $A(x) = A_{even}(x^2) + x A_{odd}(x^2)$.

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.

$\nu^k = (\omega^k)^2$

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

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

# 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.**

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

$a_0, a_1, ..., a_{n-1}$

$O(n \log n)$ →

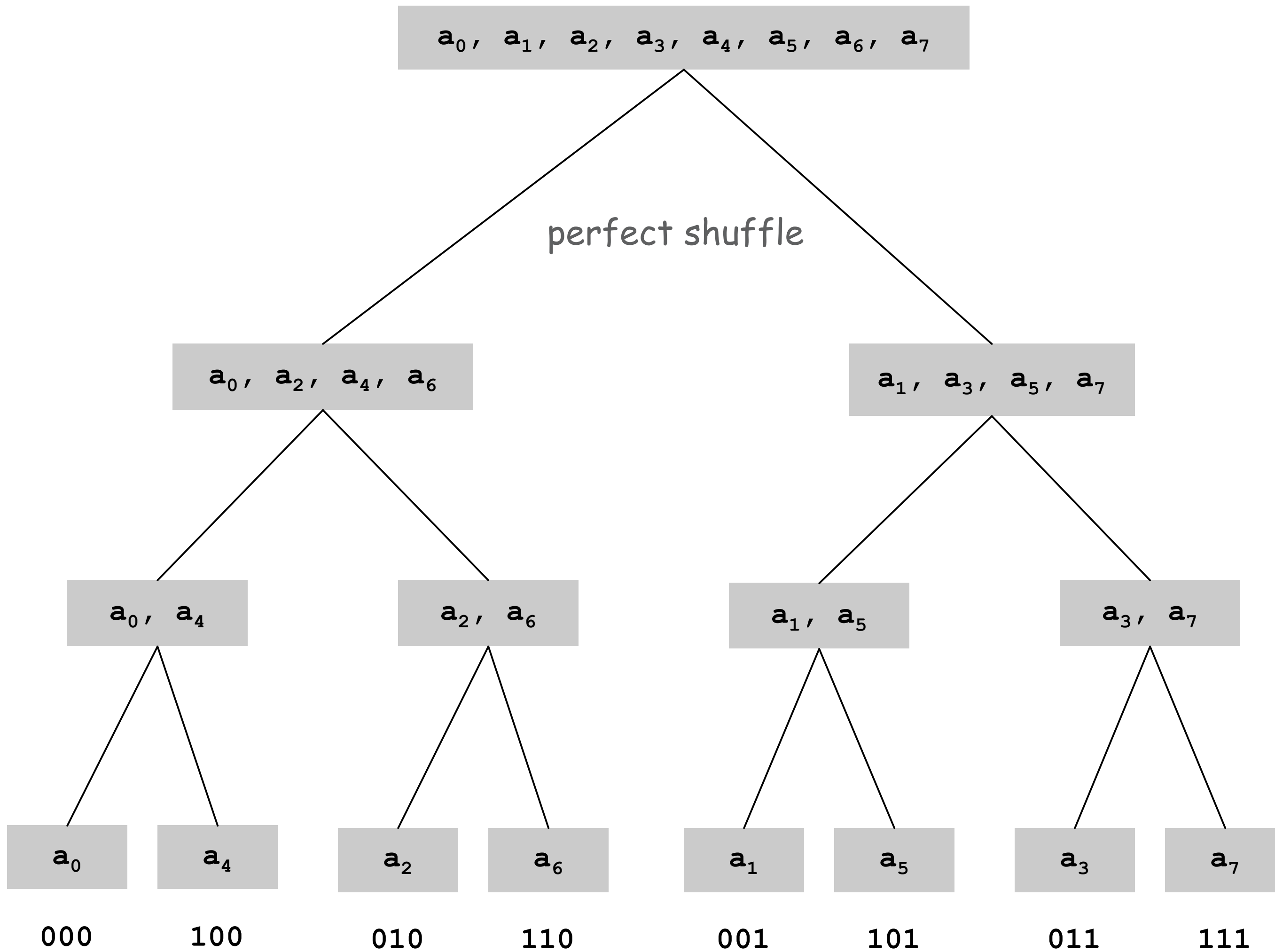
← ???

$(\omega^0, y_0), ..., (\omega^{n-1}, y_{n-1})$

coefficient representation

point-value representation

# Recursion Tree




"bit-reversed" order

# Inverse Discrete Fourier Transform

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.

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

↑ Inverse DFT

↑ Fourier matrix inverse $(F_n)^{-1}$

# Inverse DFT

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

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

$\frac{1}{\sqrt{n}} F_n$ is unitary

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.**

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

summation lemma

**Summation lemma.** Let $\omega$ be a principal $n^{th}$ root of unity. Then

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

**Pf.**

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

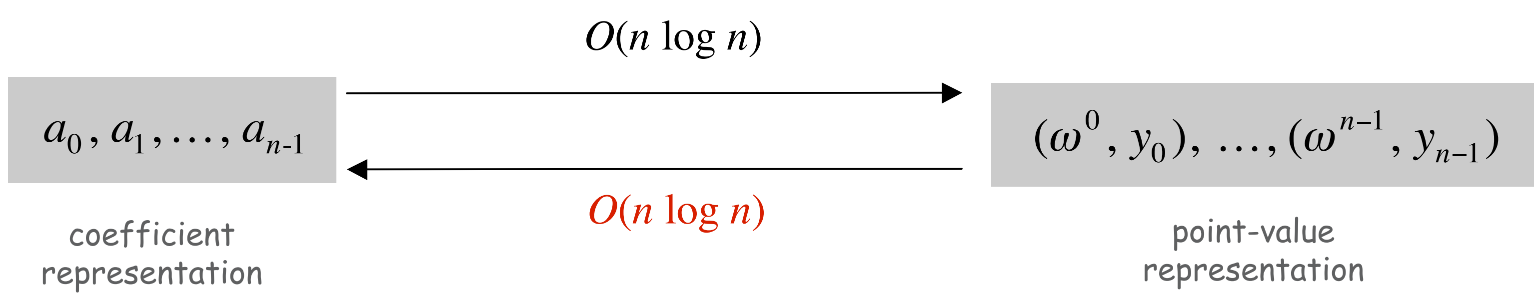
# Inverse FFT: Algorithm

```
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+n/2} ← (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.

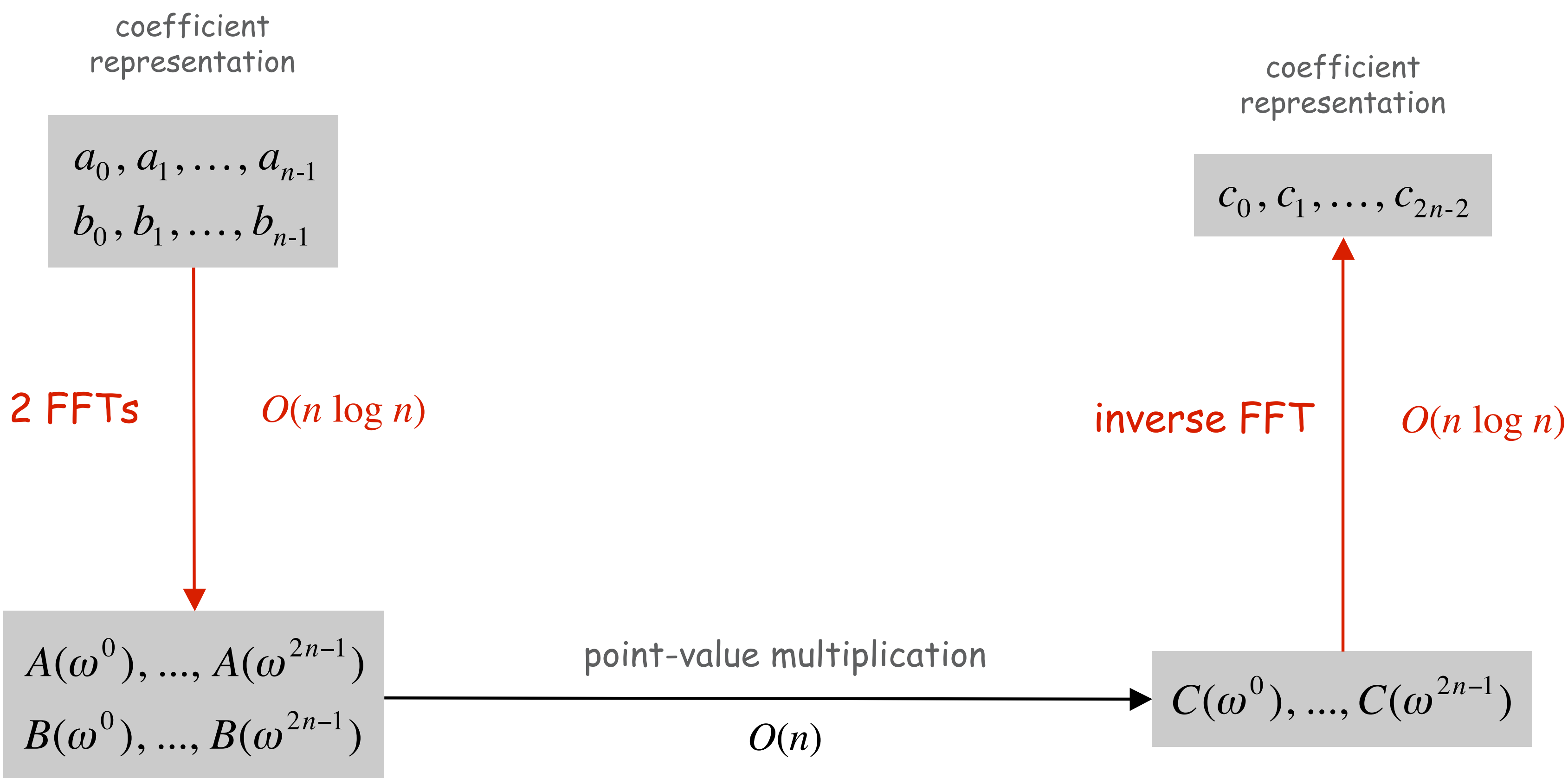
assumes $n$ is a power of 2



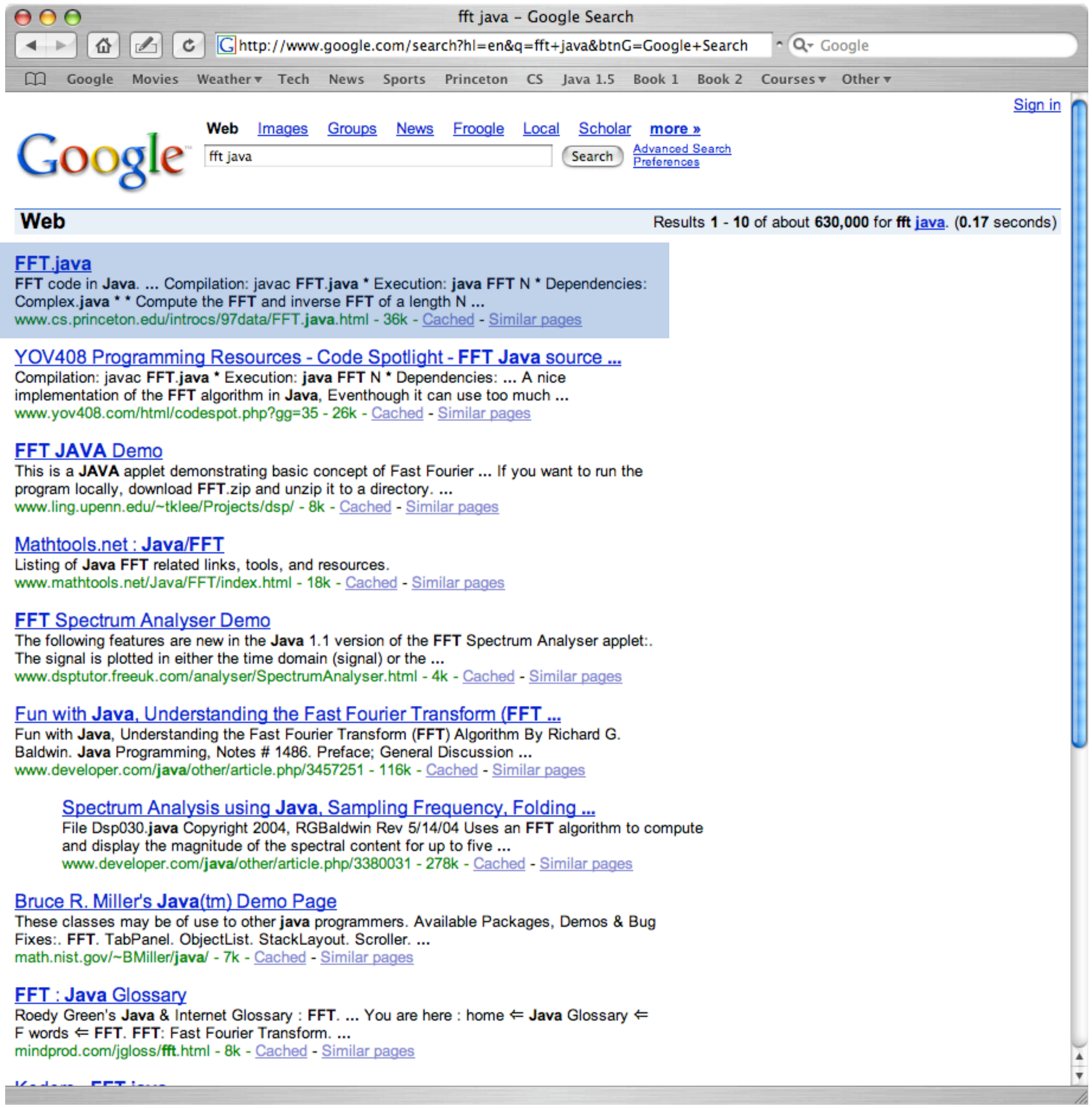
# Polynomial Multiplication

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

pad with 0s to make $n$ a power of 2



# FFT in Practice ?



# 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.
- $O(n \log n)$, 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.

Theory. [Fürer 2007] $O(n \log n \, 2^{O(\log^* n)})$ bit operations.

# 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 $a \cdot b$.

"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 $n$.

# Integer Arithmetic

Fundamental open question. What is complexity of arithmetic?

| Operation | Upper Bound | Lower Bound |
| --- | --- | --- |
| addition | $O(n)$ | $\Omega(n)$ |
| multiplication | $O(n \log n\ 2^{O(\log^* n)})$ | $\Omega(n)$ |
| division | $O(n \log n\ 2^{O(\log^* n)})$ | $\Omega(n)$ |

# Factoring

**Factoring.** Given an $n$-bit integer, find its prime factorization.

$$2773 = 47 \times 59$$

$$2^{67}-1 = 147573952589676412927 = 193707721 \times 761838257287$$

a disproof of Mersenne's conjecture that $2^{67} - 1$ is prime

```
74037563479561712828046796097429573142593188889231289084936232638972765034028266276891996419625117843995894330502127585370118968098286733173273108930900552505116877063299072396380786710086096962537934650563796359
```

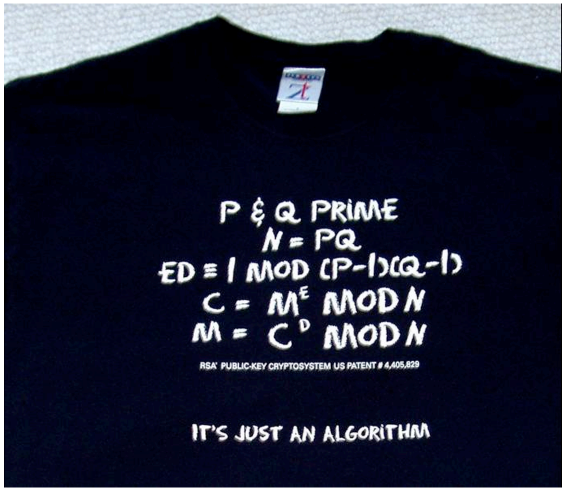
RSA-704
($30,000 prize if you can factor)

# Factoring and RSA

Primality. Given an $n$-bit integer, is it prime?
Factoring. Given an $n$-bit integer, find its prime factorization.

Significance. Efficient primality testing $\Rightarrow$ can implement RSA.
Significance. Efficient factoring $\Rightarrow$ can break RSA.

Theorem. [AKS 2002] Poly-time algorithm for primality testing.



# Shor's Algorithm

Shor's algorithm. Can factor an $n$-bit integer in $O(n^3)$ time on a quantum computer.

algorithm uses quantum QFT !

Ramification. At least one of the following is wrong:

- RSA is secure.
- Textbook quantum mechanics.
- Extending Church-Turing thesis.

# Shor's Factoring Algorithm

## Period finding.

| $2^i$ | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | … |
|---|---|---|---|---|---|---|---|---|---|
| $2^i \bmod 15$ | 1 | 2 | 4 | 8 | 1 | 2 | 4 | 8 | … |
| $2^i \bmod 21$ | 1 | 2 | 4 | 8 | 16 | 11 | 1 | 2 | … |

period = 4 (for $2^i \bmod 15$)

period = 6 (for $2^i \bmod 21$)

**Theorem.** [Euler] Let $p$ and $q$ be prime, and let $N = p\, q$. Then, the following sequence repeats with a period divisible by $(p\text{-}1)\ (q\text{-}1)$:

$$x \bmod N,\ \ x^2 \bmod N,\ \ x^3 \bmod N,\ \ x^4 \bmod N,\ \ldots$$

**Consequence.** If we can learn something about the period of the sequence, we can learn something about the divisors of $(p\text{-}1)\ (q\text{-}1)$.

by using random values of $x$, we get the divisors of $(p\text{-}1)\ (q\text{-}1)$, and from this, can get the divisors of $N = p\, q$

# Extra Slides

# Fourier Matrix Decomposition

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

$$I_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$D_4 = \begin{bmatrix} \omega^0 & 0 & 0 & 0 \\ 0 & \omega^1 & 0 & 0 \\ 0 & 0 & \omega^2 & 0 \\ 0 & 0 & 0 & \omega^3 \end{bmatrix}$$

$$a = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

$$y = F_n a = \begin{bmatrix} I_{n/2} & D_{n/2} \\ I_{n/2} & -D_{n/2} \end{bmatrix} \begin{bmatrix} F_{n/2}\, a_{even} \\ F_{n/2}\, a_{odd} \end{bmatrix}$$