南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Algorithm Design and Analysis (H)

CS216

Instructor: Shan CHEN (陈杉)

chens3@sustech.edu.cn

(slides edited from Prof. Shiqi Yu)

Divide and Conquer

4. Integer and Matrix Multiplication

Integer Addition and Subtraction

- **Addition.** Given two n-bit integers a and b, compute $a + b$.
- **Subtraction.** Given two n-bit integers a and b, compute $a - b$.
- **Grade-school algorithm.** $\Theta(n)$ 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 and subtraction algorithms are optimal.

Integer Multiplication

- **Multiplication.** Given two n-bit integers a and b, compute $a \times b$.
- **Grade-school algorithm.** $\Theta(n^2)$ 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
```

- **Conjecture.** [Kolmogorov 1956]
  Grade-school algorithm is optimal.
- **Theorem.** [Karatsuba 1960] Conjecture is false.

Integer Multiplication: Divide and Conquer

- **To multiply two *n*-bit integers *x* and *y*:**
  - Divide x and y into low- and high-order bits.
  - Multiply four $n/2$-bit integers, recursively.
  - Add and shift to obtain result.
- **Ex.** $n = 8$, $m = \lceil n/2 \rceil = 4$.

$$x = \underbrace{1000}_{a}\underbrace{1101}_{b} \qquad y = \underbrace{1110}_{c}\underbrace{0001}_{d}$$

$$xy = (2^m a + b)(2^m c + d) = 2^{2m} ac + 2^m (bc + ad) + bd$$

1 2 3 4

- **Time complexity.** $\Theta(n^2)$ ← $T(n) = 4T(n/2) + O(n)$

Integer Multiplication: Karatsuba's Trick

- **To multiply two n-bit integers x and y:**
  - Divide x and y into low- and high-order bits.
  - Multiply three $n/2$-bit integers, recursively. $bc + ad = ac + bd - (a - b)(c - d)$
  - Add and shift to obtain result.
- **Ex.** $n = 8$, $m = \lceil n/2 \rceil = 4$.

$$x = \underbrace{1000}_{a}\underbrace{1101}_{b} \qquad y = \underbrace{1110}_{c}\underbrace{0001}_{d}$$

$$xy = (2^m a + b)(2^m c + d) = 2^{2m} \underset{1}{ac} + 2^m(\underset{1}{ac} + \underset{3}{bd} - \underset{2}{(a - b)(c - d)}) + \underset{3}{bd}$$

- **Time complexity.** $\Theta(n^{\log_2 3}) = \Theta(n^{1.585})$ $\longleftarrow$ $T(n) = 3T(n/2) + O(n)$

Karatsuba Multiplication

```
Karatsuba-Multiply(x, y, n) {
   if n == 1
      return x * y
   else
      m = ⌈ n / 2 ⌉
      a = ⌊ x / 2^m ⌋; b = x mod 2^m
      c = ⌊ y / 2^m ⌋; d = y mod 2^m

      e = Karatsuba-Multiply(a, c, m)
      f = Karatsuba-Multiply(b, d, m)
      g = Karatsuba-Multiply(|a - b|, |c - d|, m)
      Flip sign of g if needed

      return 2^(2m) e + 2^m(e + f - g) + f
}
```

$O(n)$

$3T(n/2)$

$O(n)$

- **Practice.** Use base 32/64 and faster than grade school for 320~640 bits.

Integer Multiplication: Asymptotic Complexity

year	algorithm	bit operations
12xx	grade school	$O(n^2)$
1962	Karatsuba–Ofman	$O(n^{1.585})$
1963	Toom-3, Toom-4	$O(n^{1.465}),\ O(n^{1.404})$
1966	Toom–Cook	$O(n^{1+\varepsilon})$
1971	Schönhage–Strassen	$O(n \log n \cdot \log \log n)$
2007	Fürer	$n \log n \, 2^{O(\log^* n)}$
2019	Harvey–van der Hoeven	$O(n \log n)$
	???	$O(n)$

Matrix Multiplication

- **Matrix multiplication.** Given n-by-n matrices A and B, compute $C = AB$.
- **Grade-school.** $\Theta(n^3)$ arithmetic operations.
- **Block matrix multiplication:**

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

C_{11} A_{11} A_{12} B_{11} B_{21}

$$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21}$$

Matrix Multiplication: Divide and Conquer

- **To multiply two *n*-by-*n* matrices *A* and *B*:**
  - **Divide:** partition *A* and *B* into *n*/2-by-*n*/2 blocks.
  - **Conquer:** multiply 8 pairs of *n*/2-by-*n*/2 matrices, recursively.
  - **Combine:** add appropriate products using 4 matrix additions.

n-by-n matrices

$$C = A \times B$$

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

½n-by-½n matrices

8 matrix multiplications (of ½n-by-½n matrices)

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

4 matrix additions (of ½n-by-½n matrices)

$$T(n) = 8T(n/2) + O(n^2)$$

$$T(n) = O(n^3)$$

Matrix Multiplication: Strassen's Trick

- **To multiply two *n*-by-*n* matrices *A* and *B*:**
  - **Divide:** partition *A* and *B* into *n/2*-by-*n/2* blocks.
  - **Conquer:** multiply 7 pairs of *n/2*-by-*n/2* matrices, recursively.
  - **Combine:** 11 matrix additions and 7 matrix subtractions.

scalars

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

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

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

$T(n) = 7T(n/2) + O(n^2)$

$T(n) = O(n^{\log_2 7}) = O(n^{2.81})$

Strassen's Algorithm in Practice

- **Implementation issues:**
  - Sparsity.
  - Caching.
  - *n* not a power of 2.
  - Numerical stability.
  - Non-square matrices.
  - Storage for intermediate submatrices.
  - Crossover to classical algorithm when *n* is "small".
  - Parallelism for multi-core and many-core architectures.
- **However, it is still useful in practice.**
  - Apple reports 8x speedup when n ≈ 2048.

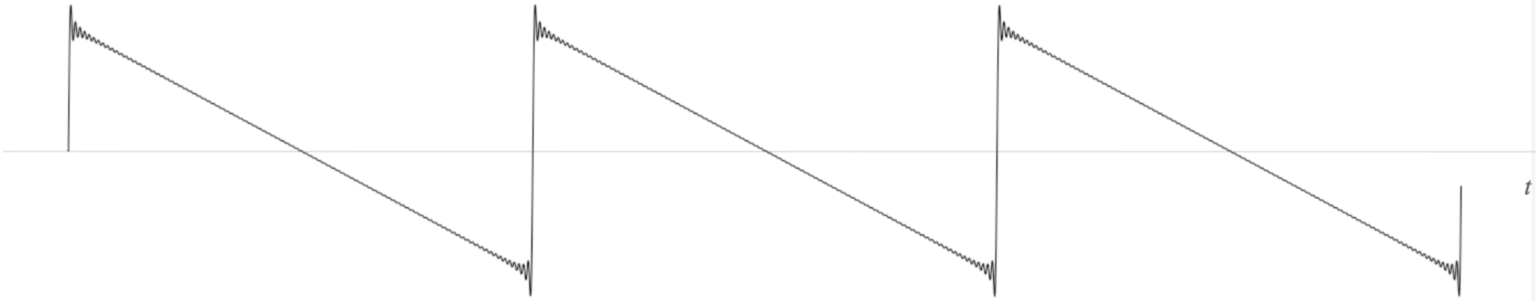
Integer Multiplication: Asymptotic Complexity

year	algorithm	arithmetic operations
1858	**"grade school"**	$O(n^3)$
1969	**Strassen**	$O(n^{2.808})$
1978	**Pan**	$O(n^{2.796})$
1979	**Bini**	$O(n^{2.780})$
1981	**Schönhage**	$O(n^{2.522})$
1982	**Romani**	$O(n^{2.517})$
1982	**Coppersmith–Winograd**	$O(n^{2.496})$
1986	**Strassen**	$O(n^{2.479})$
1989	**Coppersmith–Winograd**	$O(n^{2.3755})$
2010	**Strother**	$O(n^{2.3737})$
2011	**Williams**	$O(n^{2.372873})$
2014	**Le Gall**	$O(n^{2.372864})$
	???	$O(n^{2+\varepsilon})$

5. Convolution and FFT

Fourier Analysis and Euler's Identity

- **Fourier theorem.** [Fourier, Dirichlet, Riemann] Any (sufficiently smooth) periodic function can be expressed as the sum of a series of sinusoids.



$$y(t) = \frac{2}{\pi} \sum_{k=1}^{n} \frac{\sin kt}{k} \qquad n = 100$$

- **Euler's identity.** $e^{ix} = \cos x + i \sin x$.
  - Sum of sines and cosines = sum of complex exponentials

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). Detect nuclear tests in Soviet Union and tracking submarines. Rediscovered and popularized FFT.

- Importance not fully realized until emergence of digital computers.

Fast Fourier Transform: Applications

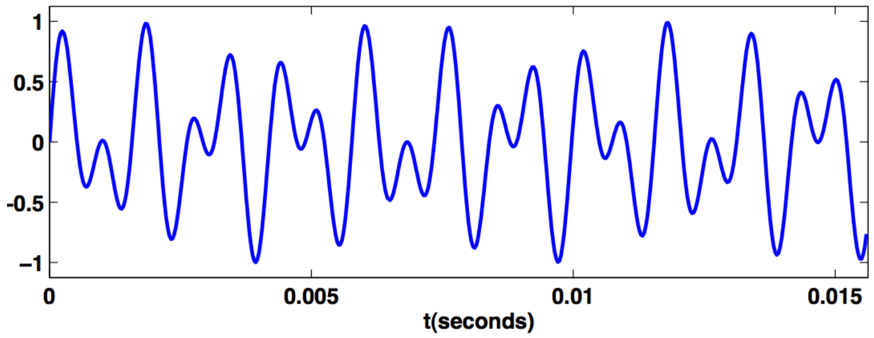
- **Applications.**
  - Optics, acoustics, quantum physics, telecommunications, radar, control systems, signal processing, speech recognition, data compression, image processing, seismology, mass spectrometry, ...
  - Digital media. [DVD, JPEG, MP3, H.264]
  - Medical diagnostics. [MRI, CT, PET scans,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

> The FFT is one of the truly great computational developments of this [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

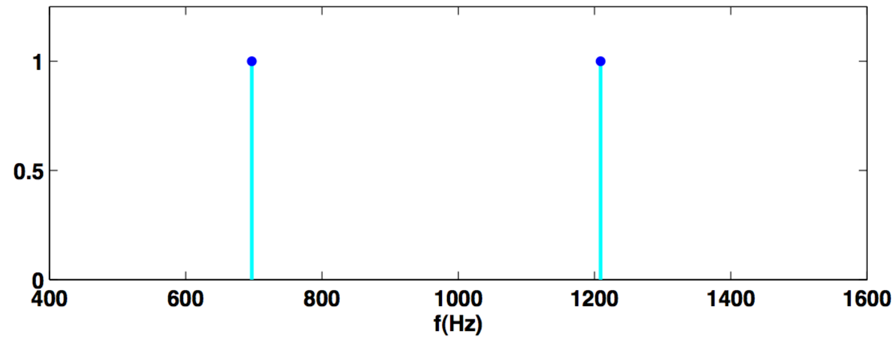
Example: Touch Tone

- Signal for 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

Fast Fourier Transform (FFT)

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

- **Alternative viewpoint.** Fast way to multiply and evaluate polynomials.

we take this viewpoint

> *" 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*

Polynomials: Coefficient Representation

- **Polynomial.** [coefficient representation]

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

- **Addition.** O(n) arithmetic operations.

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

- **Evaluation.** O(n) using Horner's method.

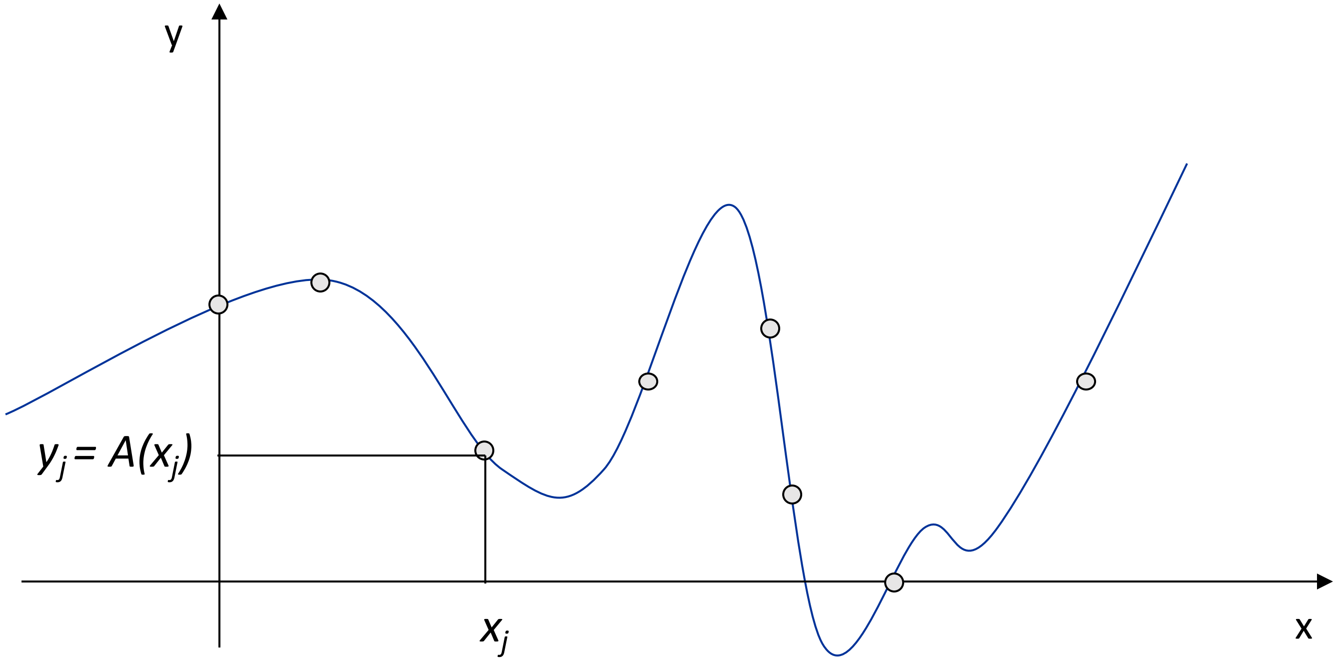
$$A(x) = a_0 + (x(a_1 + x(a_2 + \cdots + x(a_{n-2} + x(a_{n-1}))\cdots))$$

- **Multiplication** (linear convolution): $O(n^2)$ using brute force.

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

Polynomials: Point-Value Representation

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

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

- **Addition.** O(n) arithmetic operations.

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

- **Multiplication.** O(n), but represent $A(x)$ and $B(x)$ using $2n$ points.

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

- **Evaluation.** O(n^2) using Lagrange's method.

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

← not used in FFT

Converting between Two 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.** Efficiently convert between two representations, i.e., all ops fast.

$$\begin{tikzcd} a_0, a_1, \ldots, a_{n-1} \arrow[r, shift left] & (x_0, y_0), \ldots, (x_{n-1}, y_{n-1}) \arrow[l, shift left] \end{tikzcd}$$

coefficient representation — point-value representation

Converting between Two Representations

- **Application.** Polynomial multiplication (coefficient representation).

coefficient representation

$a_0, a_1, \ldots, a_{n-1}$
$b_0, b_1, \ldots, b_{n-1}$

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

point-value representation

$\xrightarrow[\text{point-value multiplication}]{O(n)}$

$C(x_0), C(x_1), \ldots, C(x_{2n-1})$

point-value representation

coefficient representation

$c_0, c_1, \ldots, c_{2n-2}$

Converting between Two Representations

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

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

$O(n^3)$ via Gaussian elimination
or $O(n^{2.37})$ via fast matrix multiplication

Vandermonde matrix is invertible iff x_i s are distinct

- **Point-value to 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: Intuition

- **Coefficient to 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 these!
- **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/-1.
  - $A(1) = A_{even}(1) + 1 A_{odd}(1)$.
  - $A(-1) = A_{even}(1) - 1 A_{odd}(1)$.

can evaluate polynomial of degree $n - 1$ at 2 points by evaluating two polynomials of degree ½ n - 1 at 1 point.

Coefficient to Point-Value: Intuition

- **Coefficient to 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 these!
- **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 complex points to be +1/-1, +i/-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 $n - 1$ at 4 points by evaluating two polynomials of degree ½ n - 1 at 2 points.

Discrete Fourier Transform (DFT)

- **Coefficient to 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.

$y_k = A(\omega^k) \longrightarrow$

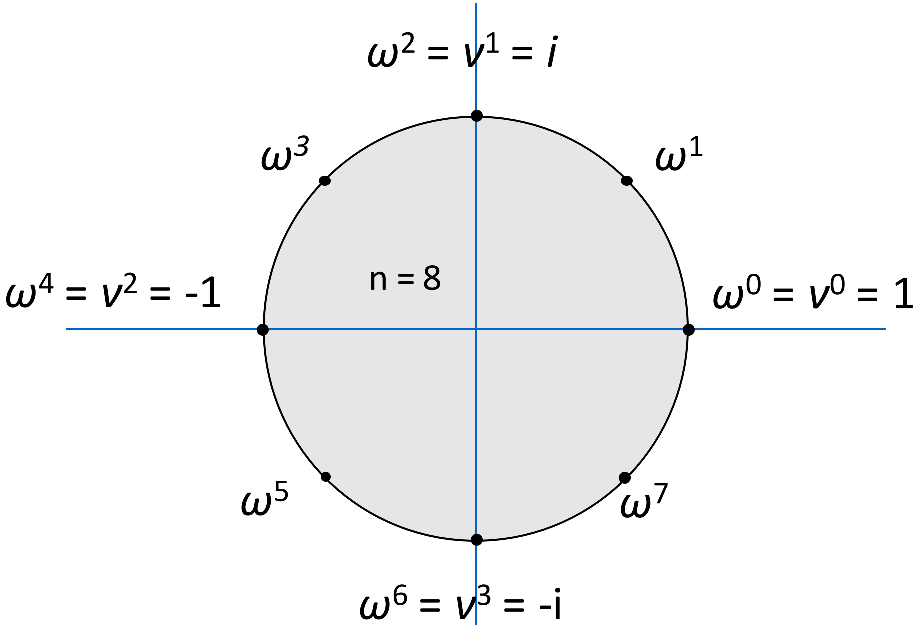
$$
\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{bmatrix}
=
\begin{bmatrix}
1 & 1 & 1 & 1 & \cdots & 1 \\
1 & \omega^1 & \omega^2 & \omega^3 & \cdots & \omega^{n-1} \\
1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(n-1)} \\
1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3(n-1)}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

DFT

Fourier matrix F_n

Roots of Unity

- **Def.** An nth root of unity is a complex number x such that $x^n = 1$.
- **Fact.** The nth 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 ½nth 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 (FFT)

- **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.
  - $A_{even}(x) = a_0 + a_2 x + a_4 x^2 + \ldots + a_{n/2-2}\, x^{(n-1)/2}$.
  - $A_{odd}(x) = a_1 + a_3 x + a_5 x^2 + \ldots + a_{n/2-1}\, x^{(n-1)/2}$.
  - $A(x) = A_{even}(x^2) + x\, A_{odd}(x^2)$.
  - $A(-x) = A_{even}(x^2) - x\, A_{odd}(x^2)$.
- **Conquer.** Evaluate $A_{even}(x)$, $A_{odd}(x)$ at ½n^{th} roots of unity: $\nu^0, \nu^1, \ldots, \nu^{n/2-1}$.
- **Combine.** (Note that $\omega^2 = \nu$ and $\omega^{n/2} = -1$.)
  - $A(\omega^k) = A_{even}(\nu^k) + \omega^k A_{odd}(\nu^k), \quad 0 \le k < n/2$
  - $A(\omega^{k+n/2}) = A_{even}(\nu^k) - \omega^k A_{odd}(\nu^k), \quad 0 \le k < n/2$

Fast Fourier Transform (FFT): Algorithm

```
FFT(n, a0,a1,…,an-1) {
   if (n == 1) return a0

   (e0,e1,…,en/2-1) = FFT(n/2, a0,a2,a4,…,an-2)
   (d0,d1,…,dn/2-1) = FFT(n/2, a1,a3,a5,…,an-1)

   for k = 0 to n/2 - 1 {
      ωk = e2πik/n
      yk     = ek + ωk dk
      yk+n/2 = ek - ωk dk
   }

   return (y0,y1,…,yn-1)
}
```

$2T(n/2)$

$O(n)$

FFT Summary

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

assume n is a power of 2

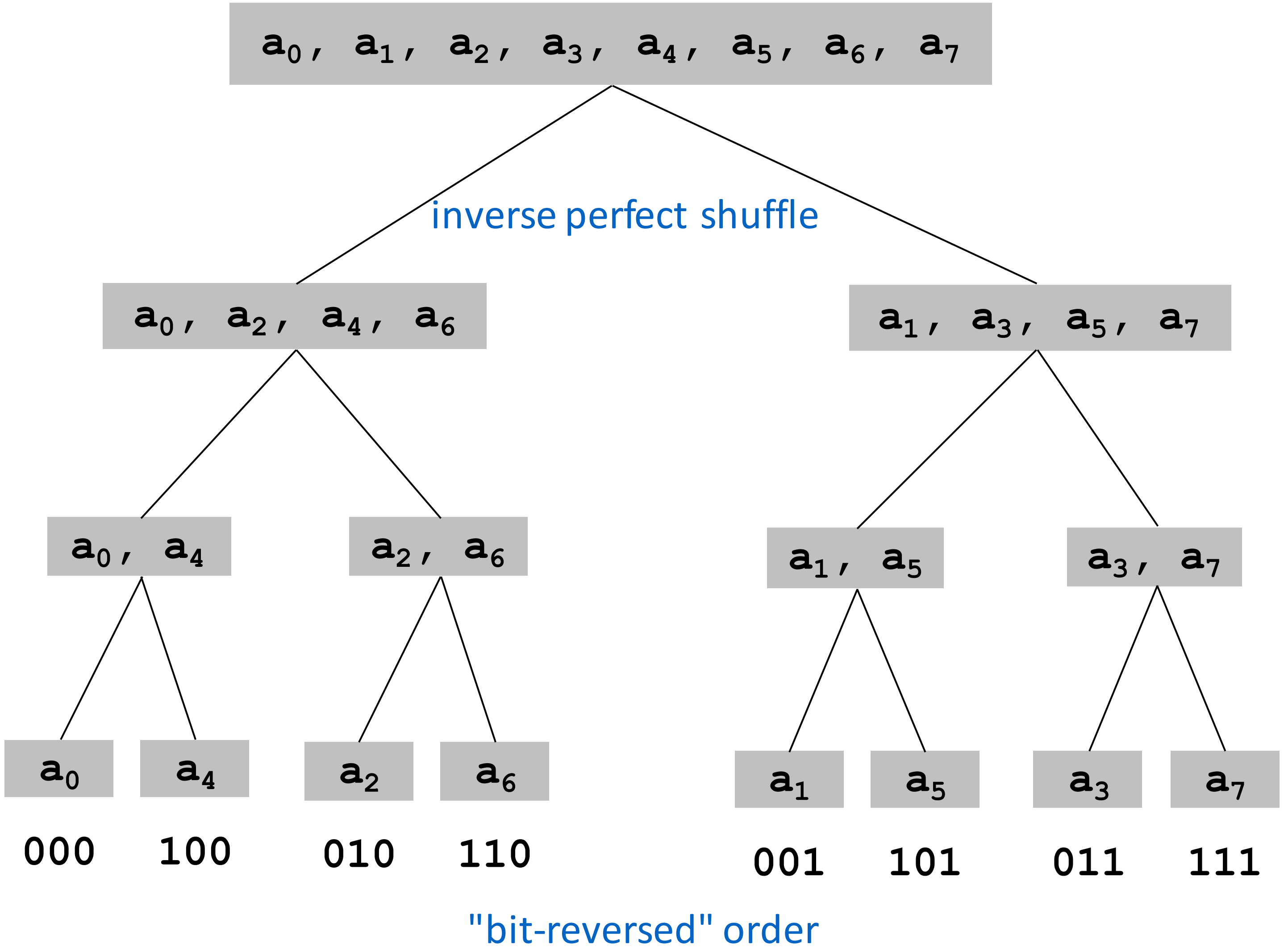
- **Time complexity.** $T(n) = 2T(n/2) + O(n) \Rightarrow T(n) = O(n \log n)$.

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

coefficient representation

point-value representation

FFT Recursion Tree



Point-Value to Coefficient: 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.

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

inverse DFT

Fourier matrix inverse F_n^{-1}

Inverse DFT

- **Claim.** Inverse of Fourier matrix 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 the inverse DFT, apply same algorithm but use ω^{-1} 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 (below)

- **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 $\omega^k = 1$, so the series sums to n.
  - Each n^{th} root of unity ω^k is a root of $x^n - 1 = (x - 1)(1 + x + x^2 + \ldots + x^{n-1})$.
  - if $\omega^k \neq 1$, then $1 + \omega^k + \omega^{k(2)} + \ldots + \omega^{k(n-1)} = 0$, so the series also sums to 0. ▪

Inverse FFT: Algorithm

```
Inverse-FFT(n, y_0,y_1,…,y_{n-1}) {
   if (n == 1) return a_0

   (e_0,e_1,…,e_{n/2-1}) = Inverse-FFT(n/2, y_0,y_2,y_4,…,y_{n-2})
   (d_0,d_1,…,d_{n/2-1}) = Inverse-FFT(n/2, y_1,y_3,y_5,…,y_{n-1})

   for k = 0 to n/2 - 1 {
      ω^k = e^{-2πik/n}
      a_k     = e_k + ω^k d_k
      a_{k+n/2} = e_k - ω^k d_k
   }

   return (a_0,a_1,…,a_{n-1})
}

Output: Inverse-FFT(n, a_0,a_1,…,a_{n-1}) / n
```

$2T(n/2)$

$O(n)$

Inverse FFT Summary

- **Theorem.** The inverse FFT algorithm interpolates a degree $n - 1$ polynomial at each of the n^{th} roots of unity in $O(n \log n)$ steps.

assume n is a power of 2

- **FFT + Inverse FFT.** Can convert between coefficient and point-value representations in $O(n \log n)$ arithmetic operations.

$$
\begin{array}{ccc}
a_0, a_1, \ldots, a_{n-1} & \underset{O(n \log n)}{\overset{O(n \log n)}{\rightleftarrows}} & (\omega^0, y_0), \ldots, (\omega^{n-1}, y_{n-1}) \\
\text{coefficient representation} & & \text{point-value representation}
\end{array}
$$

Polynomial Multiplication

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

pad 0 items to make n a power of 2

coefficient representation

$a_0, a_1, \ldots, a_{n-1}$
$b_0, b_1, \ldots, b_{n-1}$

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

coefficient representation

$c_0, c_1, \ldots, c_{2n-2}$

O(n log n) 2FFTs

O(n log n) inverse FFT

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

O(n)

point-value multiplication

$C(x_0), C(x_1), \ldots, C(x_{2n-1})$

point-value representation

point-value representation

Integer Multiplication Revisited

- **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 $c = ab$.
- **Convolution algorithm.**
  - Form two polynomials, ($a = A(2)$, $b = B(2)$)
  - Compute $C(x) = A(x)\, B(x)$.
  - Evaluate $C(2) = ab$.
  - 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]
  - $O(n \log^2 n)$ bit operations over complex numbers (with $O(\log n)$ bit precision)
  - $O(n \log n \log \log n)$ bit operations over ring of integers (modulo Fermat number)
- **Practice.** [GNU Multiple Precision Arithmetic Library] It uses FFT-based algorithms when n is large (≥ 5~10K)

FFT in Practice

- **FFT in the West (FFTW)** [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.

FFTW

Reference: http://www.fftw.org

- **Implementation details.**
  - Core algorithm is an in-place, nonrecursive version of Cooley–Tukey.
  - Instead of executing a fixed algorithm, it evaluates the hardware and uses a special-purpose compiler to generate an optimized algorithm catered to "shape" of the problem.
  - Runs in $O(n \log n)$ time, even when n is prime.
  - Multidimensional FFTs.
  - Parallelism.