5.5 Integer Multiplication

Complex Multiplication

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

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

**Gauss.** \(x = ac - bd, \ y = (a + b) (c + d) - ac - bd.\)

**Remark.** Improvement if no hardware multiply.

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 \(a \times b.\)

**Grade-school.** \(\Theta(n^2)\) bit operations.
Karatsuba: Recursion Tree

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.

\[
\begin{align*}
x &= 2^x x_1 + x_0 \\
y &= 2^y y_1 + y_0 \\
x y &= (2^x x_1 + x_0)(2^y y_1 + y_0) = 2^n x_1 y_1 + 2^{n/2} (x_0 y_1 + x_1 y_0) + x_0 y_0
\end{align*}
\]

Ex. \( x = 10001101 \) \( y = 11100001 \)

\[
\begin{align*}
t(n) &= 4t(n/2) + \Theta(n) \Rightarrow t(n) = \Theta(n^2)
\end{align*}
\]

Karatsuba: Recursion Tree

Fast Integer Division Too (!)

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.

\[
\begin{align*}
x &= 2^x x_1 + x_0 \\
y &= 2^y y_1 + y_0 \\
x y &= 2^x x_1 y_1 + 2^{x/2} (x_0 y_1 + x_1 y_0) + x_0 y_0 \\
&= 2^x x_1 y_1 + 2^y y_1 + (x_0 y_1 + y_0) (x_1 y_0 + x_0) + x_0 y_0
\end{align*}
\]

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

\[
\begin{align*}
t(n) &= T\left( \left\lfloor \frac{n}{2} \right\rfloor \right) + T\left( \left\lfloor \frac{n}{2} \right\rfloor \right) + T\left( \left\lfloor \frac{n}{2} \right\rfloor \right) + \Theta(n) \\
&= O(2 \log^2 n) = O(n^{1.585})
\end{align*}
\]

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: \( x_{i+1} = 2x_i - b x_i^2 \)
- After \( \log n \) iterations, either \( q = \lfloor a x \rfloor \) or \( q = \lceil a x \rceil \),

using fast multiplication.
Matrix Multiplication

Matrix Multiplication: Warmup

Divide-and-conquer.
- Divide: partition A and B into \( \frac{1}{2} \times \frac{1}{2} \) block matrices.
- Conquer: multiply \( \frac{1}{2} \times \frac{1}{2} \) block matrices recursively.
- Combine: add appropriate products using 4 matrix additions.

\[
\begin{pmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{pmatrix} =
\begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix} \times
\begin{pmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{pmatrix}
\]

\[
C_{11} = (A_{11} \times B_{11}) + (A_{12} \times B_{21})
\]

\[
C_{12} = (A_{11} \times B_{12}) + (A_{12} \times B_{22})
\]

\[
C_{21} = (A_{21} \times B_{11}) + (A_{22} \times B_{21})
\]

\[
C_{22} = (A_{21} \times B_{12}) + (A_{22} \times B_{22})
\]

\[
T(n) = 8T(n/2) + \Theta(n^2) \quad \Rightarrow \quad T(n) = \Theta(n^3)
\]

Matrix Multiplication

Matrix multiplication. Given two \( n \times n \) matrices A and B, compute \( C = AB \).

\[
\begin{pmatrix}
c_1 & c_2 & \cdots & c_n \\
c_n & c_{n-1} & \cdots & c_1
\end{pmatrix}
=\begin{pmatrix}
a_1 & a_2 & \cdots & a_n \\
a_n & a_{n-1} & \cdots & a_1
\end{pmatrix}
\times
\begin{pmatrix}
b_1 & b_2 & \cdots & b_n \\
b_n & b_{n-1} & \cdots & b_1
\end{pmatrix}
\]

Ex.
\[
\begin{pmatrix}
59 & 32 & .41 \\
.31 & .36 & .25
\end{pmatrix}
\times
\begin{pmatrix}
.80 & .20 & .10 \\
.10 & .40 & .10
\end{pmatrix}
\]

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

Fundamental question. Can we improve upon brute force?

Matrix Multiplication: Key Idea

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

\[
\begin{pmatrix}
P_1 \\
P_2 \\
P_3 \\
P_4 \\
P_5 \\
P_6 \\
P_7
\end{pmatrix} =
\begin{pmatrix}
P_3 + P_2 - P_5 + P_6 \\
P_4 \\
P_5 \\
P_1 + P_2 + P_3 \\
P_6 \\
P_5 - P_1 - P_2 + P_3
\end{pmatrix}
\times
\begin{pmatrix}
P_1 & P_2 & P_3 & P_4 & P_5 & P_6 & P_7
\end{pmatrix}
\]

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

Fast matrix multiplication. [Strassen, 1969]
- Divide: partition A and B into 1/2n-by-1/2n blocks.
- Compute: 14 1/2n-by-1/2n matrices via 10 matrix additions.
- Conquer: multiply 7 pairs of 1/2n-by-1/2n matrices recursively.
- Combine: 7 products into 4 terms using 8 matrix additions.

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

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

\[ T(n) = 7T(n/2) + \Theta(n^2) \Rightarrow T(n) = \Theta(n^{\log_2 7}) = O(n^{2.7799}) \]

Remark.

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

A year later.


Fast Matrix Multiplication in Theory

Q. Multiply two 2-by-2 matrices with 7 scalar multiplications?
A. Yes! [Strassen, 1969]

Q. Multiply two 2-by-2 matrices with 6 scalar multiplications?
A. Impossible. [Hopcroft and Kerr, 1971]

Q. Two 3-by-3 matrices with 21 scalar multiplications?
A. Also impossible.

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

- Two 20-by-20 matrices with 4,460 scalar multiplications.
- Two 48-by-48 matrices with 47,217 scalar multiplications.
- A year later.

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, ...
5.6 Convolution and FFT

Time Domain vs. Frequency Domain

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

**Time domain.**

**Frequency domain.**

- 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.

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

---

Reference: Cleve Moler, *Numerical Computing with MATLAB*
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.

Importance not fully realized until advent of digital computers.

Polynomials: Coefficient Representation

Polynomial. [coefficient representation]
\[ A(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 \]
\[ B(x) = b_n x^n + b_{n-1} x^{n-1} + \ldots + b_1 x + b_0 \]

Add. O(n) arithmetic operations.
\[ A(x) + B(x) = (a_n + b_n) + (a_{n-1} + b_{n-1}) x + \ldots + (a_1 + b_1) x + (a_0 + b_0) \]

Evaluate. O(n) using Horner’s method.
\[ A(x) = a_n + x (a_{n-1} + x (a_{n-2} + x \ldots (a_1 + x a_0) \ldots)) \]

Multiply (convolve). O(n^2) using brute force.
\[ A(x) \times B(x) = \sum_{i,j=0}^{n} c_{ij} x^i, \text{ where } c_{ij} = \sum_{j=0}^{i} a_j b_{i-j} \]


Corollary. A degree n-1 polynomial \( A(x) \) is uniquely specified by its evaluation at n distinct values of x.
Polynomials: Point-Value Representation

[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.

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

Converting Between Two Representations: Brute Force

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

- O(n^2) via matrix-vector multiply
- O(n^2) via Gaussian elimination

Point-value \( \Rightarrow \) coefficients. 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.

Converting Between Two Polynomial Representations

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

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

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

Coefficient to Point-Value Representation: Intuition

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

Divide. Break polynomial up into even and odd powers.

- \( A(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5 + \ldots \)
- \( A_{\text{even}}(x) = a_0 + a_2 x^2 + a_4 x^4 + a_6 x^6 + a_8 x^8 + \ldots \)
- \( A_{\text{odd}}(x) = a_1 + a_3 x^2 + a_5 x^2 + a_7 x^8 + \ldots \)
- \( A(-x) = A_{\text{even}}(x^2) - A_{\text{odd}}(x^2) \)
- \( A(-1) = A_{\text{even}}(1) - A_{\text{odd}}(1) \)

Intuition. Choose two points to be \( \pm 1 \).

- \( A(1) = A_{\text{even}}(1) + A_{\text{odd}}(1) \)
- \( A(-1) = A_{\text{even}}(-1) - A_{\text{odd}}(1) \)

Can evaluate polynomial of degree \( \leq n \) at 2 points by evaluating two polynomials of degree \( \leq \frac{n}{2} \) at 1 point.
Coefficient to Point-Value Representation: Intuition

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

**Divide.** Break polynomial up into even and odd powers.
- \(A(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5 + a_6 x^6 + a_7 x^7\).
- \(A_{\text{even}}(x) = a_0 + a_2 x^2 + a_4 x^4 + a_6 x^6\).
- \(A_{\text{odd}}(x) = a_1 + a_3 x^3 + a_5 x^5 + a_7 x^7\).
- \(A(x) = A_{\text{even}}(x^2) + x A_{\text{odd}}(x^3)\).
- \(A(-x) = A_{\text{even}}(x^2) - x A_{\text{odd}}(x^3)\).

**Intuition.** Choose four points to be \(\pm 1, \pm i\).
- \(A(1) = A_{\text{even}}(1) + 1 A_{\text{odd}}(1)\).
- \(A(-1) = A_{\text{even}}(1) - 1 A_{\text{odd}}(1)\).
- \(A(i) = A_{\text{even}}(-1) + i A_{\text{odd}}(-1)\).
- \(A(-i) = A_{\text{even}}(-1) - i A_{\text{odd}}(-1)\).

\[
\begin{array}{c|cccccc}
\text{polynomial} & 1 & i & -1 & -i & \omega^0 & \omega^1 & \omega^{n-1} \\
\hline
A(1) & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
A(-1) & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
A(i) & 1 & i & -1 & -i & \omega & \omega^2 & \omega^{n-1} \\
A(-i) & 1 & -i & -1 & i & \omega & \omega^2 & \omega^{n-1} \\
\end{array}
\]

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

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

**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 \((\omega^2)^k = \nu^k\).

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 \(\omega\) is principal \(n\)th root of unity.

\[
\begin{bmatrix}
y_0 \\
y_1 \\
y_2 \\
y_3 \\
\vdots \\
y_{n-1}
\end{bmatrix}
= 
\begin{bmatrix}
1 & 1 & 1 & 1 & \cdots & 1 \\
\omega & \omega^2 & \omega^3 & \cdots & \omega^{n-1} \\
\omega^2 & \omega^4 & \omega^5 & \cdots & \omega^{n-2} \\
\omega^3 & \omega^5 & \omega^6 & \cdots & \omega^{n-3} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
\omega^{n-1} & \omega^{2(n-1)} & \omega^{3(n-1)} & \cdots & \omega^{(n-1)(n-1)}
\end{bmatrix}
\begin{bmatrix}
a_0 \\
a_1 \\
a_2 \\
a_3 \\
\vdots \\
a_{n-1}
\end{bmatrix}
\]

Discrete Fourier transform Fourier matrix \(F_n\)

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.
- \(A_{\text{even}}(x) = a_0 + a_2 x^2 + a_4 x^4 + \ldots + a_{n-2} x^{n-2}\).
- \(A_{\text{odd}}(x) = a_1 + a_3 x^3 + a_5 x^5 + \ldots + a_{n-1} x^{n-1}\).
- \(A(x) = A_{\text{even}}(x^2) + x A_{\text{odd}}(x^3)\).

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

**Combine.**
- \(A(\omega^k) = A_{\text{even}}(\nu^k) + \omega^k A_{\text{odd}}(\nu^k), \quad 0 \leq k \leq \frac{1}{2}n\)
- \(A(\omega^{k+n}) = A_{\text{even}}(\nu^k) - \omega^k A_{\text{odd}}(\nu^k), \quad 0 \leq k \leq \frac{1}{2}n\)

\[
\omega^{k+n} = \omega^k \nu^n = -\omega^k
\]

\[

\nu^n = (\omega^k)^2 = (\omega^{kn})^2
\]
**FFT Algorithm**

```plaintext
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 {
        \omega^k ← e^{2\pi ik/n}
        y_k ← e_k + \omega^k d_k
        y_{k+n/2} ← e_k - \omega^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. 

Running time. \[ T(n) = 2T(n/2) + \Theta(n) \Rightarrow T(n) = \Theta(n \log n) \]

Point-Value to Coefficient Representation: Inverse DFT

**Goal.** Given the values \(y_0, y_1, ..., y_{n-1}\) of a degree n-1 polynomial at the n points \(\omega^0, \omega^1, ..., \omega^{n-1}\), find (unique) polynomial \(a_0 + a_1 x + ... + a_{n-1} x^{n-1}\) that has given values at given points.

\[
\begin{bmatrix}
    a_0 \\
    a_1 \\
    a_2 \\
    \vdots \\
    a_{n-1}
\end{bmatrix} =
\begin{bmatrix}
    1 & 1 & 1 & \cdots & 1 \\
    1 & \omega^1 & \omega^2 & \cdots & \omega^{n-1} \\
    1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\
    \vdots & \vdots & \vdots & \ddots & \vdots \\
    1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)}
\end{bmatrix}^{-1}
\begin{bmatrix}
    y_0 \\
    y_1 \\
    y_2 \\
    \vdots \\
    y_{n-1}
\end{bmatrix}
\]

Inverse DFT

Fourier matrix inverse \((F_2)^{-1}\)
Inverse FFT: Proof of Correctness

Claim. \( F_n \) and \( G_n \) are inverses.

Pf.

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

Summation lemma. Let \( \omega \) be a principal \( n \)th root of unity. Then

\[
\frac{1}{n} \sum_{j=0}^{n-1} \omega^j = \begin{cases} 
n & \text{if } k \equiv 0 \mod n \\
0 & \text{otherwise}
\end{cases}
\]

Pf.

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

Inverse FFT

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

\[
G_n = \frac{1}{n} \begin{cases} 
1 & \\
\omega^j & \\
\omega^{2j} & \\
\vdots & \\
\omega^{(n-1)j} & 
\end{cases}
\]

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: Algorithm

```
ifft(n, a0, a1, ..., an-1) {
    if (n == 1) return a0
    (e0, e1, ..., en/2-1) ← FFT(n/2, a0, a1, ..., an/2-1)
    (d0, d1, ..., dn/2-2) ← FFT(n/2, a1, a2, ..., an-1)
    for k = 0 to n/2 - 1 {
        \( \omega^k \) ← \( \omega^{2nk/n} \)
        \( y_k \) ← (e0 + \( \omega^k \) d0) / n
        \( y_{k+n/2} \) ← (e0 - \( \omega^k \) d0) / n
    }
    return (y0, y1, ..., yn-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

Coeficient representation

\( a_0, a_1, \ldots, a_{n-1} \)

\( O(n \log n) \)

Point-value representation

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

\( O(n \log n) \)
**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](http://www.fftw.org)

---

**Integer Multiplication, Redux**

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

**Convolution algorithm.**
- Form two polynomials. $A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$
- 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.

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

**Practice.** [GNU Multiple Precision Arithmetic Library]

It uses brute force, Karatsuba, and FFT, depending on the size of n.