Design and Analysis of Algorithm

Divide-and-Conquer (II)

Outline

Fast Power/Exponentiation Problem

Input. $a \in \mathbb{R}$, $n \in \mathbb{N}$

Output. a^n

Naive algorithm. Sequential multiplication

$$a^n = \underbrace{a \cdot a \ldots \ldots a \cdot a}_{n}$$

$\#$(multiplication)$= n - 1$

Divide-and-Conquer: Divide

n is even

$$\underbrace{a \ldots\ldots\ldots a}_{n/2} \mid \underbrace{a \ldots\ldots\ldots a}_{n/2}$$

n is odd

$$\underbrace{a \ldots\ldots\ldots a}_{(n-1)/2} \mid \underbrace{a \ldots\ldots\ldots a}_{(n-1)/2} \mid a$$

$$a^n = \begin{cases} a^{n/2} \times a^{n/2} & n \text{ is even} \\ a^{(n-1)/2} \times a^{(n-1)/2} \times a & n \text{ is odd} \end{cases}$$

Complexity Analysis

Basic operation. multiplication

- size of subproblem: smaller than $n/2$
- two subproblems (with size roughly $n/2$) are identical, only need computing once

$$W(n) = W(n/2) + \Theta(1)$$

master theorem (case 1) $\Rightarrow W(n) = \Theta(\log n)$

How to realize this algorithm? recursion vs. iteration

A Recursive Approach

Algorithm 1: Power(a, n): $a^n = (a^{-1})^{-n}$

---

if $n < 0$ **then return** Power$(1/a, -n)$; //handle negative integer exponent

if $n = 0$ **then return** 1;

if $n = 1$ **then return** a;

if n *is even* **then return** Power$(a^2, n/2)$; //smart trick

if n *is odd* **then return** $a \times$ Power$(a^2, (n-1)/2)$;

- Naive implementation: $x \leftarrow$ Power$(a, n/2)$, return $x \times x$.

An Iterative Approach

Algorithm 2: Square-and-Multiply(a, n)

$(b_k, b_{k-1}, \ldots, b_1, b_0) \leftarrow$ BinaryDecomposition(n);
$y \leftarrow 0$;
$power \leftarrow 1$;
for $i = 0$ *to* k **do**
    if $b_i = 1$ **then** $y \leftarrow y + power$;
    $power \leftarrow power \times 2$;
end
return y

Also known as binary exponentiation

Naturally extend to additive semigroups: double-and-add

- 2 is the window size

How to extend to other window size? Can the efficiency be further improved?

Application of Power Algorithm

Fibonacci sequence: $1, 1, 2, 3, 5, 8, 13, 21, \ldots$

add $F_0 = 0$, we obtain:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

Problem. Given initial values $F_0 = 0$, $F_1 = 1$, compute F_n

Naive algorithm. From $F_0, F_1, \ldots$, repeatedly compute

$$F_n = F_{n-1} + F_{n-2}$$

Complexity. sequential addition: $\Theta(n)$

Properties of Fibonacci Sequence

Proposition. Let $\{F_n\}$ be a Fibonacci sequence, then

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

Proof by mathematical induction

Basis. $n = 1$:

$$\begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

Proof (Induction Step)

Suppose for any n, the formula is correct, i.e.:

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

Then for $n+1$, according to the definition of Fibonacci sequence:

$$\begin{pmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{pmatrix} = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

$$\text{Induction premise} \Rightarrow \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n+1}$$

Improved Algorithm via Fast Power

Let

$$M = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

Compute M^n using generalized fast power algorithm

- M can be diagonalized $(M = PM'P^{-1}) \Rightarrow$ we could directly use fast power algorithm for better efficiency on basic computer step (matrix mul).

Time complexity:

- The number of matrix multiplication $T(n) = \Theta(\log n)$
- Each matrix multiplication requires 8 number multiplication
- The overall complexity is $\Theta(\log n)$

Outline

Integer Addition

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 asymptotically optimal.

Integer Multiplication

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

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

$\Theta(n^2)$ atomic bit multiplications $+$ $\Theta(n^2)$ atomic bit additions

								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	
						1	1	0	1	0	1	0	1		
					1	1	0	1	0	1	0	1			
				1	1	0	1	0	1	0	1				
			1	1	0	1	0	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	0	0	0	0	0	0	0	0	0	1

Divide-and-Conquer: First Attempt (1/2)

Divide Split two n-bit integer x and y into their left and right halves (low- and high-order bits). Let $m = n/2$.

x	x_L	x_R	$2^{n/2}x_L + x_R$
y	y_L	y_R	$2^{n/2}y_L + y_R$

Use bit shifting to compute

$$x_L = \lfloor x/2^m \rfloor, x_R = x \bmod 2^m$$
$$y_L = \lfloor y/2^m \rfloor, y_R = y \bmod 2^m$$

Example. $x = \underbrace{1011}_{x_L}\underbrace{0110}_{x_R} = 1011 \times 2^4 + 0110$

Divide-and-Conquer: First Attempt (2/2)

$$\begin{aligned} xy &= (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) \\ &= 2^n \underline{x_L y_L} + 2^{n/2}(\underline{x_L y_R} + \underline{x_R y_L}) + \underline{x_R y_R} \end{aligned}$$

Conquer. Multiply four $n/2$-bit integers, recursively. (significant operations)

Combine. Add and shift to obtain result.

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

master theorem (case 1) $\Rightarrow T(n) = \Theta(n^2)$

$\#$ (Subproblems) too many $\rightsquigarrow$ Same running time as traditional grade school method, no progress in efficiency.

How can this method be sped up?

Gauss's Trick

Gauss once noticed that although the product of two complex numbers

$$(a+bi)(c+di) = ac - bd + (bc + ad)i$$

seems involving 4 multiplications, it can in fact be done with 3:

$$bc + ad = (a+b)(c+d) - ac - bd$$

Figure: Carl Friedrich Gauss

Karatsuba's Algorithm

In 1960, Kolmogorov conjectured grade-school multiplication algorithm is optimal in a seminar. Within a week, Karatsuba, then a 23-year-old student, found a much better algorithm thus disproving the conjecture. Kolmogorov was very excited about the discovery and published a paper in 1962.

Figure: Anatolii Karatsuba

Karatsuba algorithm: the first algorithm asymptotically faster than the quadratic "grade school" algorithm.

Reduce the Number of Subproblem

Idea. Exploit the dependence among subproblems via Gauss's trick

$$\underbrace{x_L y_R + x_R y_L}_{\text{middle term}} = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$$

Algorithm 3: KARATSUBA(x, y, n)

if $n = 1$ **then return** $x \times y$;
else $m \leftarrow \lceil n/2 \rceil$;
$x_L \leftarrow \lfloor x/2^m \rfloor$; $x_R \leftarrow x \bmod 2^m$;
$y_L \leftarrow \lfloor y/2^m \rfloor$; $y_R \leftarrow y \bmod 2^m$;
$e \leftarrow$ KARATSUBA(x_L, y_L, m);
$f \leftarrow$ KARATSUBA(x_R, y_R, m);
$g \leftarrow$ KARATSUBA$(x_L + x_R, y_L + y_R, m)$;
return $2^{2m}e + 2^m(g - e - f) + f$;

Theory

Complexity Analysis. Now, the recurrence relation is

$$\left.\begin{array}{r} T(n) = 3T(n/2) + O(n) \\ T(1) = 1 \end{array}\right\} \Rightarrow T(n) = O(n^{\log_2 3}) = O(n^{1.585})$$

$\boxed{\text{Combining cost } f(n) \text{ is cheap} \rightsquigarrow h(n) \text{ dominates the overall complexity.}}$

Combining cost $f(n)$ is cheap $\rightsquigarrow$ $h(n)$ dominates the overall complexity. The constant factor improvement from 4 to 3 occurs at the *every level of the recursion*, the compounding effect leads to a dramatically lower bound.

[Toom-Cook (1963)] faster generalization of Karatsuba's method

[Schönhage-Strassen (1971)] even faster, for sufficiently large n.

Practice

A practical note: it generally does not make sense to recurse all the way down to 1 bit. For most processors, 16- or 32-bit multiplication is single operation.

GNU Multiple Precision Library uses different algorithms depending on size of operands. (used in Maple, Mathematica, gcc, cryptography, . . .)

Outline

Inner Product

Inner product. Given two length n vectors $\mathbf{a} = (a_1, \ldots, a_n)$ and $\mathbf{b} = (b_1, \ldots, b_n)$, compute

$$c = \langle \mathbf{a}, \mathbf{b} \rangle = \sum_{i=1}^{n} a_i b_i$$

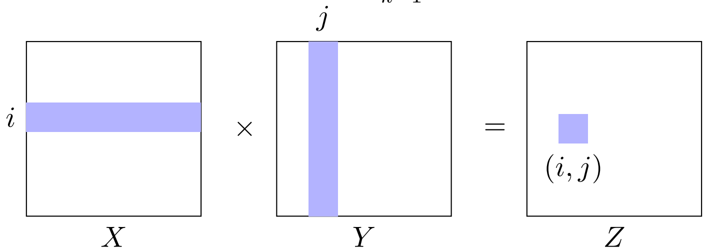
Grade school. $\Theta(n)$ arithmetic operations.

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

Matrix Multiplication

Matrix multiplication. Given two $n \times n$ matrix X and Y, compute

$$Z = XY, Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}$$



College-school method: $\Theta(n^3)$ arithmetic operations

- there are n^2 elements in Z
- computing each element requires n arithmetic multiplications

Is college-school matrix multiplication algorithm asymptotically optimal? Can divide-and-conquer strategy do better?

Naive Divide-and-Conquer

Strategy. Split matrix into blocks:

$$\begin{pmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{pmatrix} \begin{pmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{pmatrix} = \begin{pmatrix} Z_{11} & Z_{12} \\ Z_{21} & Z_{22} \end{pmatrix}$$

in which:

$$\begin{array}{ll} Z_{11} = X_{11}Y_{11} + X_{12}Y_{21} & Z_{12} = X_{11}Y_{12} + X_{12}Y_{22} \\ Z_{21} = X_{21}Y_{11} + X_{22}Y_{21} & Z_{22} = X_{21}Y_{12} + X_{22}Y_{22} \end{array}$$

Recurrence relation: master theorem (case 1)

$$\left.\begin{array}{c} T(n) = \overbrace{8T(n/2)}^{\text{recursive calls}} + \overbrace{\Theta(n^2)}^{\text{add/form submatrices}} \\ T(1) = 1 \end{array}\right\} \Rightarrow T(n) = \Theta(n^3)$$

Breakthrough

College algorithm: $\Theta(n^3)$

Naive divide-and-conquer strategy: $\Theta(n^3)$ (unimpressive)

For a quite while, this was widely believed to the the best running time possible, it was was even proved that *in certain models* no algorithms can do better.

Great excitement: This effciency can be further improved by some clever algebra.

Strassen Algorithm (1/3)

Volker Strassen first published this algorithm in 1969

- proved that the $\Theta(n^3)$ general matrix multiplication algorithm wasn't optimal
- faster than the standard matrix multiplication algorithm and is useful in practice for large matrices,
- inspire more research about matrix multiplication that led to faster approaches, e.g. the Coppersmith-Winograd algorithm.

Figure: Volker Strassen

Strassen Algorithm (2/3)

$$\begin{pmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{pmatrix} \begin{pmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{pmatrix} = \begin{pmatrix} Z_{11} & Z_{12} \\ Z_{21} & Z_{22} \end{pmatrix}$$

Define 7 instead matrix:

$$\begin{aligned}
M_1 &= X_{11}(Y_{12} - Y_{22}) \\
M_2 &= (X_{11} + X_{12})Y_{22} \\
M_3 &= (X_{21} + X_{22})Y_{11} \\
M_4 &= X_{22}(Y_{21} - Y_{11}) \\
M_5 &= (X_{11} + X_{22})(Y_{11} + Y_{22}) \\
M_6 &= (X_{12} - X_{22})(Y_{21} + Y_{22}) \\
M_7 &= (X_{11} - X_{21})(Y_{11} + Y_{12})
\end{aligned}$$

Express Z_{ij} via instead matrices

$$\begin{aligned}
Z_{11} &= M_5 + M_4 - M_2 + M_6 \\
Z_{12} &= M_1 + M_2 \\
Z_{21} &= M_3 + M_4 \\
Z_{22} &= M_5 + M_1 - M_3 - M_7
\end{aligned}$$

$$\begin{aligned}
Z_{12} = M_1 + M_2 &= X_{11} \times (Y_{12} - Y_{22}) + (X_{11} + X_{12}) \times Y_{22} \\
&= X_{11} \times Y_{12} + X_{12} \times Y_{22}
\end{aligned}$$

Strassen Algorithm (3/3)

Reduce the number of subproblems from 8 to 7

Recurrence relation for time complexity (18 is number of additions/substraction performed at each application of the algorithm)

$$\left.\begin{array}{c} T(n) = 7T(n/2) + 18n^2 \\ T(1) = 1 \end{array}\right\} \Rightarrow T(n) = \Theta(n^{\log_2^7}) = \Theta(n^{2.8075})$$

Strassen Algorithm (3/3)

Reduce the number of subproblems from 8 to 7

Recurrence relation for time complexity (18 is number of additions/substraction performed at each application of the algorithm)

$$\left.\begin{array}{c} T(n) = 7T(n/2) + 18n^2 \\ T(1) = 1 \end{array}\right\} \Rightarrow T(n) = \Theta(n^{\log_2^7}) = \Theta(n^{2.8075})$$

Q. What if n is not the power of 2?

A. Could pad matrices with zeros.

$$\begin{pmatrix} 1 & 2 & 3 & 0 \\ 4 & 5 & 7 & 0 \\ 7 & 8 & 9 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 10 & 11 & 12 & 0 \\ 13 & 14 & 15 & 0 \\ 16 & 17 & 18 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 84 & 90 & 96 & 0 \\ 201 & 216 & 231 & 0 \\ 318 & 342 & 366 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

More about Matrix Multiplication

The decompsition is *so tricky and intricate* that one wonders how Strassen was ever able to discover it!

More about Matrix Multiplication

The decompsition is *so tricky and intricate* that one wonders how Strassen was ever able to discover it!

Complexity of Matrix Multiplication

- Best upper-bound: $O(n^{2.376})$ — Coppersmith-Winograd algorithm
- Known lower-bound: $\Omega(n^2)$

More about Matrix Multiplication

The decompsition is *so tricky and intricate* that one wonders how Strassen was ever able to discover it!

Complexity of Matrix Multiplication

- Best upper-bound: $O(n^{2.376})$ — Coppersmith-Winograd algorithm
- Known lower-bound: $\Omega(n^2)$

Applications

- scientific computation, image processing, data mining (regression, aggregation, decision tree)

Outline

Motivation

We have studied how to multiply

- Integers: Gauss's trick
- Matrix: Strassen algorithm

Motivation

We have studied how to multiply

- Integers: Gauss's trick
- Matrix: Strassen algorithm

How to multiply polynomials?

Motivation

We have studied how to multiply

- Integers: Gauss's trick
- Matrix: Strassen algorithm

How to multiply polynomials?

Applications of polynomial multiplication

- Fastest polynomial multiplication implies fastest integers multiplication
  - polynomials and binary integers are quite similar — just replace the variable x by the base 2 and watch out for carries
- Multiplying polynomials is crucial for *signal processing*

Polynomials: Coefficient Representation

Polynomial. coefficient representation

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

Polynomial Operation

Add. $\Theta(n)$

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

Polynomial Operation

Add. $\Theta(n)$

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

Evaluate. three choices:

- Naive algorithm. compute each term one by one: $\Theta(n^2)$
- Caching algorithm. cache x^i: $\Theta(n)$
- Horner algorithm.
  $a_0 + (x(a_1 + x(a_2 + \cdots + x(a_{n-2} + x(a_{n-1}))))):$ $\Theta(n)$
  秦九韶 discovered this algorithm hundreds of years earlier
  参考链接: https://zhuanlan.zhihu.com/p/22166332

Polynomial Operation

Add. $\Theta(n)$

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

Evaluate. three choices:

- Naive algorithm. compute each term one by one: $\Theta(n^2)$
- Caching algorithm. cache x^i: $\Theta(n)$
- Horner algorithm.
  $a_0 + (x(a_1 + x(a_2 + \cdots + x(a_{n-2} + x(a_{n-1})))))$: $\Theta(n)$
  秦九韶 discovered this algorithm hundreds of years earlier
  参考链接: https://zhuanlan.zhihu.com/p/22166332

Multiply (convolve). $\Theta(n^2)$ using brute force algorithm

$$A(x) \times B(x) = a_0b_0 + (a_0b_1 + a_1b_0)x + \cdots + a_{n-1}b_{n-1}x^{2n-2}$$

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

Pictorial View of Convolution

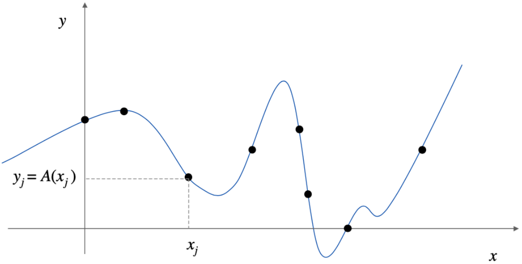
$$(c_0, \ldots, c_{2n-2}) = (a_0, \ldots, a_{n-1}) \otimes (b_0, \ldots, b_{n-1})$$

$$\begin{array}{ccccc}
a_0 b_0 & a_0 b_1 & \cdots & a_0 b_{n-2} & a_0 b_{n-1} \\
a_1 b_0 & a_1 b_1 & \cdots & a_1 b_{n-2} & a_1 b_{n-1} \\
\vdots & \vdots & \vdots & & \\
a_{n-2} b_0 & a_{n-2} b_1 & \cdots & a_{n-2} b_{n-2} & a_{n-2} b_{n-1} \\
a_{n-1} b_0 & a_{n-1} b_1 & \cdots & a_{n-1} b_{n-2} & a_{n-1} b_{n-1}
\end{array}$$

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]

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

Polynomial Operation

Add. $\Theta(n)$ add operations.

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

Multiply (convolve). $\Theta(n)$, 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. $\Theta(n^2)$ using Lagrange's formula

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

Converting Between Two Representations

Trade-off. Fast evaluation or fast multiplication. We want both!

representation	multiply	evaluate
coefficient	$\Theta(n^2)$	$\Theta(n)$
point-value	$\Theta(n)$	$\Theta(n^2)$

Goal. Efficient conversion between two representations $\Rightarrow$ have the good of both: all operations fast

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

coefficient representation | point-value representation

Converting Between Two Representations: Evaluation

Coefficient $\Rightarrow$ Point-value

Given $A(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$, evaluate it at n distinct points $x_0, \ldots, x_{n-1}$.

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

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

Converting Between Two Representations: Interpolation

Point-value $\Rightarrow$ Coefficient

Given n distinct points $x_0, \ldots, x_{n-1}$ and values $(y_0, \ldots, y_{n-1})$, find unique polynomial $A(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$, that has given values at given points.

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

Vandermonde matrix is invertible iff x_i's are distinct.

Running time. $\Theta(n^3)$ for Gaussian elimination

Restate Our Goal

Both known conversions are inefficient

- coefficients $\Rightarrow$ point-value: $\Theta(n^2)$
- point-value $\Rightarrow$ coefficients: $\Theta(n^3)$

More efficient conversions are needed.

Next, we begin with the first direction. We restate our goal:

> Given n coefficients, computing n point-value tuples quickly.

Divide-and-Conquer for Evaluation

$$A(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5 + a_6 x^6 + a_7 x^7$$

two choices for dividing: frequency vs. time

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

$$A_{\mathsf{low}}(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3$$
$$A_{\mathsf{high}}(x) = a_4 + a_5 x + a_6 x^2 + a_7 x^3$$
$$A(x) = A_{\mathsf{low}}(x) + x^4 A_{\mathsf{high}}(x).$$

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

$$A_{\mathsf{even}}(x) = a_0 + a_2 x + a_4 x^2 + a_6 x^3$$
$$A_{\mathsf{odd}}(x) = a_1 + a_3 x + a_5 x^2 + a_7 x^3$$
$$A(x) = A_{\mathsf{even}}(x^2) + x A_{\mathsf{odd}}(x^2)$$

radix-2 decimation-in-time (DIT)

Naive Idea

Naive idea. Randomly pick n distinct points $x_0, \ldots, x_{n-1}$, then compute $A(x)$ via $A_{\mathsf{even}}(x^2) + xA_{\mathsf{odd}}(x^2)$.

- $T(n)$: evaluate a degree $n-1$ polynomial at n points
- $E(n)$: evaluate a degree $n-1$ polynomial at one point

Naive Idea

Naive idea. Randomly pick n distinct points $x_0, \ldots, x_{n-1}$, then compute $A(x)$ via $A_{\text{even}}(x^2) + xA_{\text{odd}}(x^2)$.

- $T(n)$: evaluate a degree $n-1$ polynomial at n points
- $E(n)$: evaluate a degree $n-1$ polynomial at one point

Issue. Efficiency does not improve

- Evaluating $A(x)$ of degree $n-1$ at n points: $T(n) = n \cdot E(n)$
- Evaluating $A_{\text{even}}(x)$ and $A_{\text{odd}}(x)$ both of degree $n/2 - 1$ at n points: $2 \times n \cdot E(n/2) = 2n \cdot E(n/2)$

$E(n)$ is a linear function $\rightsquigarrow$ no efficiency improvement

Goal. Reduce the number of evaluated points

Basic Idea (1/2)

Basic idea. Introduce simple structure by choosing the n points to be positive-negative pairs, that is,

$$\pm x_0, \pm x_1, \ldots, \pm x_{n/2-1}$$

Note that the even powers of x_i coincide with those of $-x_i \Rightarrow$ the computations required for each $A(x_i)$ and $A(-x_i)$ overlap a lot.

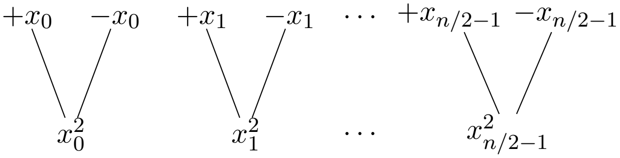
$$A(x_i) = A_{\text{even}}(x_i^2) + x_i A_{\text{odd}}(x_i^2)$$
$$A(-x_i) = A_{\text{even}}(x_i^2) - x_i A_{\text{odd}}(x_i^2)$$

Now, evaluating degree $(n-1)$ $A(x)$ at n paired points $\pm x_0, \ldots, \pm x_{n/2-1} \Rightarrow$ evaluating degree $(n/2-1)$ $A_{\text{even}}(x)$ and $A_{\text{odd}}(x)$ at just $n/2$ points: $x_0^2, \ldots, x_{n/2-1}^2$.

Basic Idea (2/2)

$A(x)$
degree $= n - 1$

$A_{\text{even}}(x),\ A_{\text{odd}}(x)$
degree $= n/2 - 1$



Now, the original problem of size n is in this way recast as two subproblems of size $n/2$ followed by some linear-time arithmetic.

$T(n)$: evaluate a degree $(n-1)$ polynomial at n points

- If we could recurse, we would get a divide-and-conquer procedure with running time:

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

which is $\Theta(n \log n)$, exactly what we want.

Technical Hurdle

Technical hurdle. The plus-minus trick only works at the top level of the recursion.

- To recurse at the next level, we need the $n/2$ evaluation points $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$ *themselves* to be plus-minus pairs.

Technical Hurdle

Technical hurdle. The plus-minus trick only works at the top level of the recursion.

- To recurse at the next level, we need the $n/2$ evaluation points $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$ *themselves* to be plus-minus pairs.

But how can a square to be negative?

Technical Hurdle

Technical hurdle. The plus-minus trick only works at the top level of the recursion.

- To recurse at the next level, we need the $n/2$ evaluation points $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$ *themselves* to be plus-minus pairs.

But how can a square to be negative?

Technical Hurdle

Technical hurdle. The plus-minus trick only works at the top level of the recursion.

- To recurse at the next level, we need the $n/2$ evaluation points $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$ *themselves* to be plus-minus pairs.

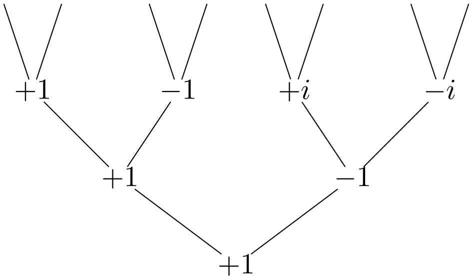
But how can a square to be negative?

Unless, of course, we use complex numbers.

Which Complex Numbers?

Fine, but which complex numbers? Let us figure out by "reverse engineer" the process.

- At the bottom of the recursion, we have a single point, say, 1.
- In the level above it must consists of its square roots, ± 1.
- The next level up is $(+1, -1)$ and $(+i, -i)$, until we reach $n = 2^k$ leaf nodes.



The Choice of n Complex Numbers

An nth root of unity is a complex number such that $x^n = 1$.

The Choice of n Complex Numbers

An nth root of unity is a complex number such that $x^n = 1$.

Fact. The nth roots of unity are $\omega^0 = 1, \omega^1, \omega^2, \ldots, \omega^{n-1}$, where $\omega = e^{2\pi i/n} = \cos\frac{2\pi}{n} + i\sin\frac{2\pi}{n}$

The Choice of n Complex Numbers

An nth root of unity is a complex number such that $x^n = 1$.

Fact. The nth roots of unity are $\omega^0 = 1, \omega^1, \omega^2, \ldots, \omega^{n-1}$, where $\omega = e^{2\pi i/n} = \cos\frac{2\pi}{n} + i\sin\frac{2\pi}{n}$

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

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

If n is even, the nth roots are plus-minus paired, $\omega^{n/2+j} = -\omega^j$

- Squaring them produces the $(n/2)$-th roots of unity: $v_0, v_1, \ldots, v^{n/2-1}$, where $v = \omega^2 = e^{4\pi i/n}$.

The Choice of n Complex Numbers

An nth root of unity is a complex number such that $x^n = 1$.

Fact. The nth roots of unity are $\omega^0 = 1, \omega^1, \omega^2, \ldots, \omega^{n-1}$, where $\omega = e^{2\pi i/n} = \cos\frac{2\pi}{n} + i\sin\frac{2\pi}{n}$

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

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

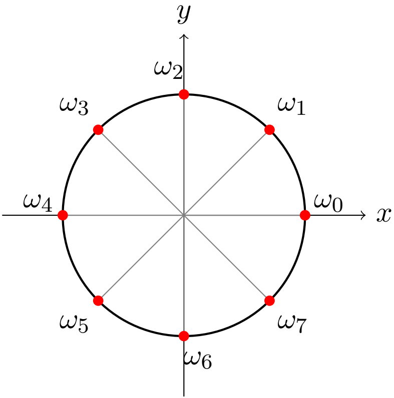
If n is even, the nth roots are plus-minus paired, $\omega^{n/2+j} = -\omega^j$

- Squaring them produces the $(n/2)$-th roots of unity: $v_0, v_1, \ldots, v^{n/2-1}$, where $v = \omega^2 = e^{4\pi i/n}$.

If we start with $\omega^0, \omega^1, \omega^2, \ldots, \omega^{n-1}$ for some $n = 2^k$, then at k-level of recursion we will have the $(n/2^k)$-th roots of unity.

- All these sets of roots are plus-minus paired $\Rightarrow$ Divide-and-conquer algorithm will work perfectly

Demo of $n = 8$



$$\omega_0 = 1$$

$$\omega_1 = e^{\frac{\pi}{4} i} = \frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2} \cdot i$$

$$\omega_2 = e^{\frac{\pi}{2} i} = i$$

$$\omega_3 = e^{\frac{3\pi}{4} i} = -\frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2} \cdot i$$

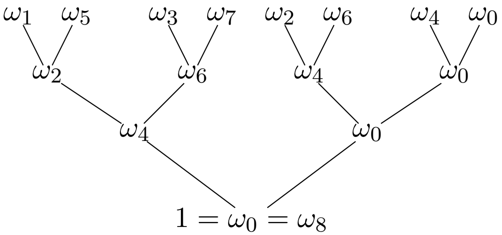
$$\omega_4 = e^{\pi i} = -1$$

$$\omega_5 = e^{\frac{5\pi}{4} i} = -\frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2} \cdot i$$

$$\omega_6 = e^{\frac{3\pi}{2} i} = -i$$

$$\omega_7 = e^{\frac{7\pi}{4} i} = \frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2} \cdot i$$

Recursion Structure and FFT



DFT: Fourier Matrix $M_n(\omega)$

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & \omega_0 & \omega_0^1 & \dots & \omega_0^{n-1} \\ 1 & \omega_1 & \omega_1^2 & \dots & \omega_1^{n-1} \\ 1 & \omega_2 & \omega_2^2 & \dots & \omega_2^{n-1} \\ 1 & \omega_3 & \omega_3^2 & \dots & \omega_3^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_{n-1} & \omega_{n-1}^2 & \dots & \omega_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

Fast Fourier Transform (FFT)

Refined Goal. Evaluate $A(x) = a_0 + \cdots + a_{n-1}x^{n-1}$ at its nth root of unity: $\omega^0, \omega^1, \ldots, \omega^{n-1}$

Divide. Break up polynomial into even and odd powers:

$$A_{\text{even}}(x) = a_0 + a_2x + a_4x^2 + \cdots + a_{n-2}x^{n/2-1}$$

$$A_{\text{odd}}(x) = a_1 + a_3x + a_5x^2 + \cdots + a_{n-1}x^{n/2-1}$$

$$A(x) = A_{\text{even}}(x^2) + xA_{\text{odd}}(x^2)$$

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

Combine.

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

$$A(\omega^k) = A_{\text{even}}(v^k) + \omega^k A_{\text{odd}}(v^k), 1 \le k < n/2$$

$$A(\omega^{k+n/2}) = A_{\text{even}}(v^k) - \omega^k A_{\text{odd}}(v^k), 1 \le k < n/2$$

$v^k = (\omega^{k+n/2})^2$ $\quad$ $(\omega^{k+n/2}) = -\omega^k$

Pseudocode of FFT Algorithm

Algorithm 4: FFT(A, n, ω)

Input: coefficient representation of degree $n-1$ polynomial A, principal n-th root of unity $\omega = e^{2\pi i/n}$

Output: value representation $A(\omega^0), \ldots, A(\omega^{n-1})$

if $n = 1$ **then return** a_0;

express $A(x) = A_{\mathsf{even}}(x^2) + xA_{\mathsf{odd}}(x^2)$;

FFT$(A_{\mathsf{even}}, \frac{n}{2}, \omega^2) \rightarrow (A_{\mathsf{even}}((\omega^2)^0), \ldots, A_{\mathsf{even}}((\omega^2)^{n/2-1}))$;

FFT$(A_{\mathsf{odd}}, \frac{n}{2}, \omega^2) \rightarrow (A_{\mathsf{odd}}((\omega^2)^0), \ldots, A_{\mathsf{odd}}((\omega^2)^{n/2-1}))$;

for $j = 0$ *to* $n - 1$ **do**

  $A(w^j) = A_{\mathsf{even}}(\omega^{2j}) + \omega^j A_{\mathsf{odd}}(\omega^{2j})$ // $\Theta(n)$;

end

return $A(\omega^0), \ldots, A(\omega^{n-1})$;

FFT Summary

Theorem. Assume $n = 2^k$. FFT algorithm evaluates a degree $n-1$ polynomial at each of the n-th roots of unity in $\Theta(n \log n)$ steps.

Running time

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

Essence: choose n points with special structure to accelerate DFT computation.

$$\begin{tikzcd} a_0, a_1, \ldots, a_{n-1} \arrow[r, shift left, "\Theta(n \log n)"] & (\omega_0, y_0), \ldots, (\omega_{n-1}, y_{n-1}) \arrow[l, shift left, "???"] \end{tikzcd}$$

coefficient representation | point-value representation

Recap

We first developed a high-level scheme for multiplying polynomials

coefficient representation $\Rightarrow$ point-value representation

Point-value representation makes it trivial to multiply polynomials, but the input-output form of algorithm is specified as coefficient representation.

- So we designed FFT: coefficient $\Rightarrow$ point-value in time just $\Theta(n \log n)$, where the points $\{x_i\}_n$ are complex n-th roots of unity $(1, \omega, \omega^2, \ldots, \omega^{n-1})$.

$$\langle \text{values} \rangle = \text{FFT}(\langle \text{coefficients} \rangle, \omega)$$

$$\underbrace{a_0, a_1, \ldots, a_{n-1}}_{\text{coefficient representation}} \;\underset{\text{Interpolation}}{\overset{\text{Evaluation}}{\rightleftarrows}}\; \underbrace{(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})}_{\text{point-value representation}}$$

Interpolation

The last remaining piece of the puzzle is the inverse operation — interpolation. It turns out amazingly that:

$$\langle \text{coefficients} \rangle = \frac{1}{n} \mathsf{FFT}(\langle \text{values} \rangle, \omega^{-1})$$

Interpolation is thus solved in the most simple and elegant way, using the same FFT algorithm, but called with ω^{-1} in place of ω!

This might seem like a miraculous coincidence, but it will make a lot more sense when recasting polynomial operations in the language of linear algebra.

Inverse Discrete Fourier Transform

Point-value $\Rightarrow$ Coefficient

Given n distinct points $x_0, \ldots, x_{n-1}$ and values $y_0, \ldots, y_{n-1}$, find unique polynomial $A(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$, that has given values at given points.

Inverse DFT: Fourier Matrix inverse $F_n(\omega)^{-1}$

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & \omega_0 & \omega_0^1 & \ldots & \omega_0^{n-1} \\ 1 & \omega_1 & \omega_1^2 & \ldots & \omega_1^{n-1} \\ 1 & \omega_2 & \omega_2^2 & \ldots & \omega_2^{n-1} \\ 1 & \omega_3 & \omega_3^2 & \ldots & \omega_3^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_{n-1} & \omega_{n-1}^2 & \ldots & \omega_{n-1}^{n-1} \end{bmatrix}^{-1} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{bmatrix}$$

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

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

Key Fact

$$G_n = \frac{1}{n} F_n(\omega^{-1}) = F_n(\omega)^{-1}$$

Claim. F_n and G_n are inverses

Proof. Examine $F_n G_n$

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

Summation lemma. Let ω be the principal n-th root of unity. Then

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

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

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 the result by n).
- switch the role of $\langle a_0, \ldots, a_{n-1} \rangle$ and $\langle y_0, \ldots, y_{n-1} \rangle$

Interpolation Resolved

$$\langle \text{values} \rangle = \text{FFT}(\langle \text{coefficients} \rangle, \omega)$$

$$\begin{bmatrix} A(\omega_0) \\ A(\omega_1) \\ A(\omega_2) \\ \vdots \\ A(\omega_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & \omega_0 & \omega_0^2 & \dots & \omega_0^{n-1} \\ 1 & \omega_1 & \omega_1^2 & \dots & \omega_1^{n-1} \\ 1 & \omega_2 & \omega_2^2 & \dots & \omega_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_{n-1} & \omega_{n-1}^2 & \dots & \omega_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

$$\langle \text{coefficients} \rangle = \frac{1}{n}\text{FFT}(\langle \text{values} \rangle, \omega^{-1})$$

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & \omega_0 & \omega_0^2 & \dots & \omega_0^{n-1} \\ 1 & \omega_1 & \omega_1^2 & \dots & \omega_1^{n-1} \\ 1 & \omega_2 & \omega_2^2 & \dots & \omega_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_{n-1} & \omega_{n-1}^2 & \dots & \omega_{n-1}^{n-1} \end{bmatrix}^{-1} \begin{bmatrix} A(\omega_0) \\ A(\omega_1) \\ A(\omega_2) \\ \vdots \\ A(\omega_{n-1}) \end{bmatrix}$$

Inverse FFT Summary

Theorem. Assume $n = 2^k$. Inverse FFT algorithm interpolated a degree $n-1$ polynomial given values at each of the n-th roots of unity in $\Theta(n \log n)$ steps.

Running time. Almost the same algorithm as FFT.

$$\begin{tikzcd} \underset{\text{coefficient representation}}{a_0, a_1, \ldots, a_{n-1}} \arrow[r, shift left, "\Theta(n \log n)"] & \underset{\text{point-value representation}}{(\omega_0, y_0), \ldots, (\omega_{n-1}, y_{n-1})} \arrow[l, shift left, "\Theta(n \log n)"] \end{tikzcd}$$

Polynomial Multiplication

Theorem. Two degree $(n-1)$-polynomials can be multiplied in $\Theta(n \log n)$ steps. (pad with 0s to make n a power of 2)

coefficient representation

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

2 FFTs $\Theta(n \log n)$

$A(\omega_0), \ldots, A(\omega_{2n-1})$
$B(\omega_0), \ldots, B(\omega_{2n-1})$

point-value multiplication $\Theta(n)$

$C(\omega_0), \ldots, C(\omega_{2n-1})$

1 inverse FFT $\Theta(n \log n)$

coefficient representation

$c_0, c_1, \ldots, c_{2n-2}, c_{2n-1} = 0$

Actually, $2n-1$ point-value tuples are sufficient.
But, FFT requires the input size to be 2^k, so is the output size

Remarks

Standard FFT. Evaluating degree $(n-1)$-$A(x)$ at its n-th root of unity $\omega^0, \omega^1, \ldots, \omega^{n-1}$ by evaluating degree $n/2-1$ polynimials $A_{\text{even}}(x)$ and $A_{\text{odd}}(x)$ at their $n/2$-th root of unity.

We choose the degree of polynomial as input size, since it determines the depth of recursion call.

Standard FFT can be easily extended to evaluating degree $(n-1)$ polynomial $A(x)$ at its $2n$-th root of unity $\omega^0, \omega^1, \ldots, \omega^{2n-1}$ by evaluating degree $(n/2-1)$ polynomials $A_{\text{even}}(x)$ and $A_{\text{odd}}(x)$ at their n-th root of unity.

We still choose the degree of polynomial as input size, the recurrence relation is similar,

$$f(n) : \Theta(n) \to \Theta(2n)$$

The overall complexity does not change in asymptotic sense.

Extension of FFT

FFT works in the field of complex numbers $\mathbb{C}$, the roots might be complex numbers $\rightsquigarrow$ precision lose is inevitable

Sometimes we only need to work in a finite field, e.g. $\mathbb{F} = \mathbb{Z}/p$, the integers modulo a prime p.

- Primitive n-th roots of unity exist whenever n divides $p - 1$, so we have $p = \xi n + 1$ for a positive integer ξ.
- Specially, let ω be a primitive $(p - 1)$-th root of unity, then an n-th root of unity — α can be found by letting $\alpha = \omega^{\xi}$

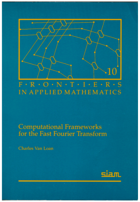
This is number-theoretic transform (NTT): obtained by specializing the discrete Fourier transform to $\mathbb{F}$.

- no precision loss
- much faster

Applications of FFT

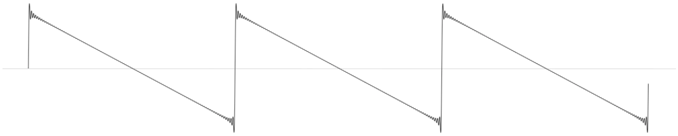
- Optics, acoustics, quantum physics, telecommunications, radar, control systems, signal processing, speech recognition, data compression, image processing, seismology, mass spectrometry...
- Digital media. [DVD, JPEG, MP3, H.264]
- Medical diagnostics. [MRI, CT, PET scans, ultrasound]
- Numerical solutions 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*



Fourier Analysis

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.

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

Sinusoids. Sum of sine and cosines = sum of complex exponentials.

Fourier Transform

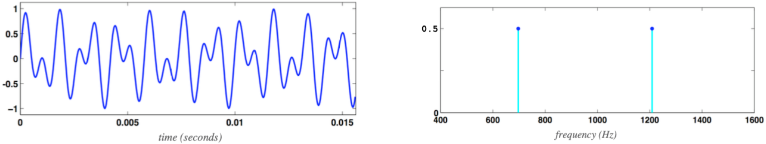


Figure: time domain vs. frequency domain

FFT.

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

Alternate viewpoint.

Fast way to multiply and evaluate polynomials

FFT: Brief History

Gauss. Analyzed periodic motion of asteroid Ceres (in Latin)

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.

An Algorithm for the Machine Calculation of Complex Fourier Series

By James W. Cooley and John W. Tukey

An efficient method for the calculation of the interactions of a 2^m factorial experiment was introduced by Yates and is widely known by his name. The generalization to 3^m was given by Box et al. [1]. Good [2] generalized these methods and gave elegant algorithms for which one class of applications is the calculation of Fourier series. In their full generality, Good's methods are applicable to certain problems in which one must multiply an N-vector by an $N \times N$ matrix which can be factored into m sparse matrices, where m is proportional to $\log N$. This results in a procedure requiring a number of operations proportional to $N \log N$ rather than N^2.

paper published only after IBM lawyers decided not to set precedent of patenting numerical algorithms (conventional wisdom: nobody could make money selling software!)

Importance not fully realized until advent of digital computers.

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 Wilkinson Prize '99.

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.
- $\Theta(n \log n)$, even for prime sizes.

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 ab.

1. Form two polynomials (base-2 representation $\Rightarrow a = A(2)$, $b = B(2)$)

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

2. Compute $C(x) = A(x)B(x)$ via FFT, evaluate $C(2) = ab$

Running time: $\Theta(n \log n)$ complex arithmetic operations.

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

Figure: the fastest bignum library on the planet

Summary of This Lecture (1/3)

Concept of Divide-and-Conquer

Main Idea. Reduce problems to subproblems

Principle. Subproblems should be of the same type of the original problem, and can be solved individually.

- Direct dividing: splitting original problem into subproblems with roughly same size
  - FindMinMax, Merge Sort
- Sophisticated dividing
  - General selection: using median as pivot (finding the pivot itself requires effort)
  - Cloest pair of points: analysis of the strip around the midline
  - Convex hull: sometime it is hard to split in a balance manner (convex hull)

Summary of This Lecture (2/3)

Implementation. Recursion or iteration (be careful of the smallest subproblem which can be solved outright)

Time complexity

- Finding the recurrence relation and initial values, solving the recurrence relation

Recurrence relation of divide-and-conquer algorithm

$$T(n) = aT(n/b) + f(n)$$

- a: #(subproblems), n/b: size of subproblems
- $f(n)$: cost of dividing and combining

Summary of This Lecture (3/3)

Optimization trick 1. Reduce the number of subproblems: when $f(n)$ is not very large, $h(n) = n^{\log_b^a}$ dominates the overall complexity $\Rightarrow T(n) = \Theta(h(n))$

- Reduce a can immediately lowering the order of $T(n)$
- When subproblems are related $\rightsquigarrow$ exploit relations to solve some subproblems by combining the solutions to other subproblems

Examples

- power algorithm: subproblems are same
- simple algebraic trick: integer multiplication ($f(n)$ is still low)
- exploit dependence: matrix multiplication ($f(n)$ may increase but does not affect the order)

Optimization trick 2. Reduce the cost of dividing and combining $f(n)$: add global preprocessing

- closest pair of points

Important Divide-and-Conquer Algorithms

Searching algorithm: binary search

Soring algorithm: quick sort, merge sort

Selection algorithm: find min/max, general selection algorithm

Cloest pair of points, Convex hull

Fast Power algorithm

Multiplying matrices: Strassen algorithm

Multiplying integers, polynomials: FFT