Algorithm Design and Implementation

Principle of Algorithms VII

Divide and Conquer II

Guoqiang Li

School of Software, Shanghai Jiao Tong University

Master Theorem

Divide-and-conquer recurrences

Goal. Recipe for solving common divide-and-conquer recurrences:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

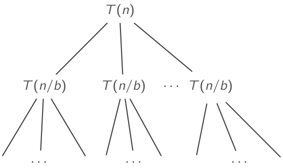
with $T(0) = 0$ and $T(1) = \Theta(1)$.

Terms.

- $a \geq 1$ is the number of subproblems.
- $b \geq 2$ is the factor by which the subproblem size decreases.
- $f(n) \geq 0$ is the work to divide and combine subproblems.

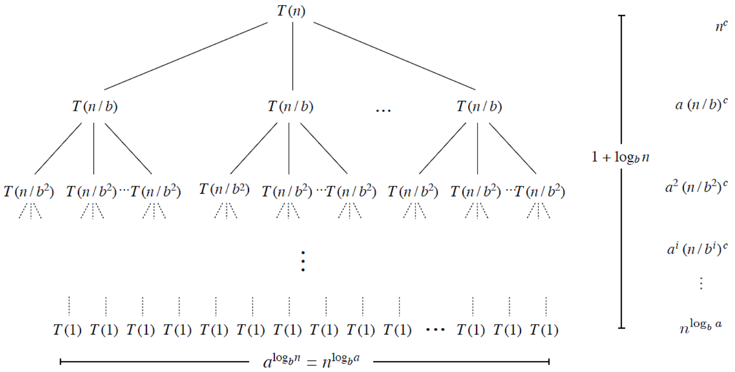
Recursion tree. [assuming n is a power of b]

- a= branching factor.
- a^i=number of subproblems at level i.
- $1 + \log_b n$ levels.
- n/b^i = size of subproblem at level i.



Divide-and-conquer recurrences: recursion tree

Suppose $T(n)$ satisfies $T(n) = aT(\frac{n}{b}) + n^c$ with $T(1) = 1$, n a power of b.



$$r = a/b^c \qquad T(n) = n^c \sum_{i=0}^{\log_b n} r^i$$

Divide-and-conquer recurrences: recursion tree analysis

Suppose $T(n)$ satisfies $T(n) = aT(\frac{n}{b}) + n^c$ with $T(1) = 1$, n a power of b.

Let $r = a/b^c$. Note that $r < 1$ iff $c > \log_b a$.

$$T(n) = n^c \sum_{i=0}^{\log_b n} r^i = \begin{cases} \Theta\left(n^c\right) & \text{if } r < 1 \quad c > \log_b a \\ \Theta\left(n^c \log n\right) & \text{if } r = 1 \quad c = \log_b a \\ \Theta\left(n^{\log_b a}\right) & \text{if } r > 1 \quad c < \log_b a \end{cases}$$

Geometric series.

- If $0 < r < 1$, then $1 + r + r^2 + r^3 + \ldots + r^k \leq 1/(1-r)$.
- If $r = 1$, then $1 + r + r^2 + r^3 + \ldots + r^k = k + 1$.
- If $r > 1$, then $1 + r + r^2 + r^3 + \ldots + r^k = \left(r^{k+1} - 1\right)/(r-1)$.

Divide-and-conquer recurrences: master theorem

Theorem

Let $a \geq 1$, $b \geq 2$ and $c > 0$ and suppose that $T(n)$ is a function on the non-negative integers that satisfies the recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta\left(n^c\right)$$

with $T(0) = 0$ and $T(1) = \Theta(1)$, where n/b means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then,

- *If $c < \log_b a$, then $T(n) = \Theta\left(n^{\log_b a}\right)$.*
- *If $c = \log_b a$, then $T(n) = \Theta\left(n^c \log n\right)$.*
- *If $c > \log_b a$, then $T(n) = \Theta\left(n^c\right)$.*

Divide-and-conquer recurrences: master theorem

Theorem

Let $a \geq 1$, $b \geq 2$ *and* $c > 0$ *and suppose that* $T(n)$ *is a function on the non-negative integers that satisfies the recurrence*

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta\left(n^c\right)$$

with $T(0) = 0$ *and* $T(1) = \Theta(1)$, *where* n/b *means either* $\lfloor n/b \rfloor$ *or* $\lceil n/b \rceil$. *Then,*

- *If* $c < \log_b a$, *then* $T(n) = \Theta\left(n^{\log_b a}\right)$.
- *If* $c = \log_b a$, *then* $T(n) = \Theta\left(n^c \log n\right)$.
- *If* $c > \log_b a$, *then* $T(n) = \Theta\left(n^c\right)$.

Extensions.

- Can replace Θ with O everywhere.
- Can replace Θ with Ω everywhere.
- Can replace initial conditions with $T(n) = \Theta(1)$ for all $n \leq n_0$ and require recurrence to hold only for all $n > n_0$.

Divide-and-conquer recurrences: master theorem

Theorem

Let $a \geq 1$, $b \geq 2$ *and* $c > 0$ *and suppose that* $T(n)$ *is a function on the non-negative integers that satisfies the recurrence*

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta\left(n^c\right)$$

with $T(0) = 0$ *and* $T(1) = \Theta(1)$, *where* n/b *means either* $\lfloor n/b \rfloor$ *or* $\lceil n/b \rceil$. *Then,*

- *If* $c < \log_b a$, *then* $T(n) = \Theta\left(n^{\log_b a}\right)$.
- *If* $c = \log_b a$, *then* $T(n) = \Theta\left(n^c \log n\right)$.
- *If* $c > \log_b a$, *then* $T(n) = \Theta\left(n^c\right)$.

Exercise 1. $T(n) = 3T(\lfloor n/2 \rfloor) + 5n$.

- $a = 3, b = 2, c = 1, \log_b a < 1.58$.
- $T(n) = \Theta\left(n^{\log_2 3}\right) = \mathrm{O}\left(n^{1.58}\right)$.

Divide-and-conquer recurrences: master theorem

Theorem

Let $a \geq 1$, $b \geq 2$ and $c > 0$ and suppose that $T(n)$ is a function on the non-negative integers that satisfies the recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta\left(n^c\right)$$

with $T(0) = 0$ and $T(1) = \Theta(1)$, where n/b means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then,

- *If $c < \log_b a$, then $T(n) = \Theta\left(n^{\log_b a}\right)$.*
- *If $c = \log_b a$, then $T(n) = \Theta\left(n^c \log n\right)$.*
- *If $c > \log_b a$, then $T(n) = \Theta\left(n^c\right)$.*

Exercise 2. $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 17n$.

- $a = 2, b = 2, c = 1, \log_b a = 1$.
- $T(n) = \Theta(n \log n)$.

Divide-and-conquer recurrences: master theorem

Theorem

Let $a \geq 1$, $b \geq 2$ and $c > 0$ and suppose that $T(n)$ is a function on the non-negative integers that satisfies the recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta\left(n^c\right)$$

with $T(0) = 0$ and $T(1) = \Theta(1)$, where n/b means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then,

- *If $c < \log_b a$, then $T(n) = \Theta\left(n^{\log_b a}\right)$.*
- *If $c = \log_b a$, then $T(n) = \Theta\left(n^c \log n\right)$.*
- *If $c > \log_b a$, then $T(n) = \Theta\left(n^c\right)$.*

Exercise 3. $T(n) = 48T(\lfloor n/4 \rfloor) + n^3$.

- $a = 48, b = 4, c = 3, \log_b a > 2.79$.
- $T(n) = \Theta\left(n^3\right)$.

Quiz 1

Consider the following recurrence. Which case of the master theorem?

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 3T([n/2]) + \Theta(n) & \text{if } n > 1 \end{cases}$$

A. Case 1: $T(n) = \Theta\left(n^{\log_2 3}\right) = O\left(n^{1.585}\right)$.

B. Case 2: $T(n) = \Theta(n \log n)$.

C. Case 3: $T(n) = \Theta(n)$.

D. Master theorem not applicable.

Quiz 2

Consider the following recurrence. Which case of the master theorem?

$$T(n) = \begin{cases} 0 & \text{if } n \leq 1 \\ T(\lfloor n/5 \rfloor) + T(n - 3\lfloor n/10 \rfloor) + \frac{11}{5}n & \text{if } n > 1 \end{cases}$$

A. Case 1: $T(n) = \Theta(n)$.

B. Case 2: $T(n) = \Theta(n \log n)$.

C. Case 3: $T(n) = \Theta(n)$.

D. Master theorem not applicable.

Master theorem need not apply

Gaps in master theorem.

- Number of subproblems is not a constant.

$$(n) = n\,T(n/2) + n^2$$

- Number of subproblems is less than 1.

$$T(n) = \tfrac{1}{2}\,T(n/2) + n^2$$

- Work to divide and combine subproblems is not $\Theta\left(n^c\right)$.

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

Akra–Bazzi theorem

Theorem (Akra–Bazzi 1998)

Given constants $a_i > 0$ *and* $0 < b_i < 1$, *functions* $|h_i(n)| = O\left(n/\log^2 n\right)$ *and* $g(n) = O\left(n^c\right)$. *If* $T(n)$ *satisfies the recurrence:*

$$T(n) = \sum_{i=1}^{k} a_i T\left(b_i n + h_i(n)\right) + g(n)$$

then, $T(n) = \Theta\left(n^p\left(1 + \int_1^n \frac{g(u)}{u^{p+1}}\,du\right)\right)$, *where* p *satisfies* $\sum_{i=1}^{k} a_i b_i^p = 1$.

Example. $T(n) = T(\lfloor n/5 \rfloor) + T(n - 3\lfloor n/10 \rfloor) + 11/5n$, with $T(0) = 0$ and $T(1) = 0$.

- $a_1 = 1, b_1 = 1/5, a_2 = 1, b_2 = 7/10 \Rightarrow p = 0.83978\ldots < 1$.
- $h_1(n) = \lfloor n/5 \rfloor - n/5, h_2(n) = 3/10n - 3\lfloor n/10 \rfloor$.
- $g(n) = 11/5n \Rightarrow T(n) = \Theta(n)$.

Integer 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	1	1	1	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. $\Theta\left(n^2\right)$ bit operations.

								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	
						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	1							
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.

Divide-and-conquer multiplication

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.

$$\begin{aligned} m &= \lceil n/2 \rceil \\ a &= \lfloor x/2^m \rfloor \qquad b = x \bmod 2^m \\ c &= \lfloor y/2^m \rfloor \qquad d = y \bmod 2^m \end{aligned}$$

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

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

Divide-and-conquer multiplication

Multiply(x, y, n)

if $n = 1$ **then** Return $x \times y$;
$m \leftarrow \lceil n/2 \rceil$;
$a \leftarrow \lfloor x/2^m \rfloor$; $b \leftarrow x \bmod 2^m$;
$c \leftarrow \lfloor y/2^m \rfloor$; $d \leftarrow y \bmod 2^m$;
$e \leftarrow$ Multiply(a, c, m);
$f \leftarrow$ Multiply(b, d, m);
$g \leftarrow$ Multiply(b, c, m);
$h \leftarrow$ Multiply(a, d, m);
Return $2^n e + 2^m(g + h) + f$;

Quiz 3

How many bit operations to multiply two n-bit integers using the divide-and-conquer multiplication algorithm?

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 4T(\lceil n/2 \rceil) + \Theta(n) & \text{if } n > 1 \end{cases}$$

A. $T(n) = \Theta\left(n^{1/2}\right)$

B. $T(n) = \Theta(n \log n)$.

C. $T(n) = \Theta\left(n^{\log_2 3}\right) = O\left(n^{1.585}\right)$.

D. $T(n) = \Theta\left(n^2\right)$

Gauss's trick

To multiply two n-bit integers x and y:

- Divide x and y into low- and high-order bits.
- To compute middle term $bc + ad$, use identity:

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

- Multiply only three $n/2$-bit integers, recursively.

Carl Friedrich Gauss(1777-1855) noticed that although the product of two complex numbers

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

seems to involve four real-number multiplications, it can in fact be done with just three: ac, bd, and $(a + b)(c + d)$.

Karatsuba's trick

To multiply two n-bit integers x and y:

- Divide x and y into low- and high-order bits.
- To compute middle term $bc + ad$, use identity:

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

- Multiply only three $n/2$-bit integers, recursively.

Recursive Multiplication

Suppose x and y are two n-integers, and assume for convenience that n is a power of 2.

[Hints: For every n there is an n' with $n \leq n' \leq 2n$, where n' a power of 2.]

$$m = \lceil n/2 \rceil$$
$$a = \lfloor x/2^m \rfloor \qquad b = x \textbf{ mod } 2^m$$
$$c = \lfloor y/2^m \rfloor \qquad d = y \textbf{ mod } 2^m$$

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

Gauss's trick

$$\begin{aligned} xy = (2^m a + b)(2^m c + d) &= 2^n ac + 2^m(bc + ad) + bd \\ &= 2^{2m} ac + 2^m((a+b)(c+d) - ac - bd) + bd \end{aligned}$$

Karatsuba's trick

$$\begin{aligned} xy = (2^m a + b)(2^m c + d) &= 2^n ac + 2^m(bc + ad) + bd \\ &= 2^{2m} ac + 2^m(ac + bd - (a-b)(c-d)) + bd \end{aligned}$$

Gauss multiplication

Gauss-Multiply(x, y, n)

if $n = 1$ **then** Return $x \times y$;
$m \leftarrow \lceil n/2 \rceil$;
$a \leftarrow \lfloor x/2^m \rfloor$; $b \leftarrow x \bmod 2^m$;
$c \leftarrow \lfloor y/2^m \rfloor$; $d \leftarrow y \bmod 2^m$;
$e \leftarrow$ Gauss-Multiply(a, c, m);
$f \leftarrow$ Gauss-Multiply(b, d, m);
$g \leftarrow$ Gauss-Multiply($a + b$, $c + d$, m);
Return $2^n e + 2^m(g - e - f) + f$;

Karatsuba analysis

Proposition

Gauss(Karatsuba)'s algorithm requires $O(n^{1.585})$ bit operations to multiply two n-bit integers.

Proof. Apply Case 1 of the master theorem to the recurrence:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 3T(\lceil n/2 \rceil) + \Theta(n) & \text{if } n > 1 \end{cases}$$

$$\Longrightarrow \quad T(n) = \Theta\left(n^{\log_2 3}\right) = O\left(n^{1.585}\right)$$

Practice.

- Use base 32 or 64 (instead of base 2).
- Faster than grade-school algorithm for about 320–640 bits.

Integer arithmetic reductions

Integer multiplication. Given two n-bit integers, compute their product. bigskip

arithmetic problem	formula	bit complexity
integer multiplication	$a \times b$	$M(n)$
integer square	a^2	$\Theta(M(n))$
integer division	$\lfloor a/b \rfloor$, $a \bmod b$	$\Theta(M(n))$
integer square root	$\lfloor \sqrt{a} \rfloor$	$\Theta(M(n))$

integer arithmetic problems with the same bit complexity $M(n)$ as integer multiplication

History of asymptotic complexity of integer multiplication

year	algorithm	bit operations
12xx	grade school	$O\left(n^2\right)$
1962	Karatsuba–Ofman	$O\left(n^{1.585}\right)$
1963	Toom-3, Toom-4	$O\left(n^{1.465}\right), \quad O\left(n^{1.404}\right)$
1966	Toom–Cook	$O\left(n^{1+\varepsilon}\right)$
1971	Schönhage–Strassen	$O(n \log n \cdot \log \log n)$
2007	Fürer	$n \log n 2^{O(\log^* n)}$
2018	Harvey–van der Hoeven	$O\left(n \log n \cdot 2^{2 \lg^* n}\right)$
20XX	???	$O(n)$

number of bit operations to multiply two n-bit integers

Remark. GNU Multiple Precision library uses one of first five algorithms depending on n.

The Fast Fourier Transform

Polynomial multiplication

If $A(x) = a_0 + a_1 x + \ldots + a_d x^d$ and $B(x) = b_0 + b_1 x + \ldots + b_d x^d$, their product

$$C(x) = c_0 + c_1 x + \ldots + c_{2d} x^{2d}$$

has coefficients

$$c_k = a_0 b_k + a_1 b_{k-1} + \ldots + a_k b_0 = \sum_{i=0}^{k} a_i b_{k-i}$$

where for $i > d$, take a_i and b_i to be zero.

Computing c_k from this formula take $O(k)$ step, and finding all $2d+1$ coefficients would therefore seem to require $\Theta(d^2)$ time.

Q: Can we do better?

Fact: A degree-d polynomial is uniquely characterized by its values at any $d+1$ distinct points.

We can specify a degree-d polynomial $A(x) = a_0 + a_1 x + \ldots + a_d x^d$ by either of the following:

- Its coefficients $a_0, a_1, \ldots, a_d$. (coefficient representation).
- The values $A(x_0), A(x_1), \ldots A(x_d)$ (value representation).

$$\text{coefficient representation} \; \underset{\text{interpolation}}{\overset{\text{evaluation}}{\rightleftarrows}} \; \text{value representation}$$

An alternative representation

$$\text{coefficient representation} \underset{\text{interpolation}}{\overset{\text{evaluation}}{\rightleftarrows}} \text{value representation}$$

The product $C(x)$ has degree $2d$, it is determined by its value at any $2d+1$ points.

Its value at any given point z is just $A(z)$ times $B(z)$.

Therefore, polynomial multiplication takes linear time in the value representation.

The algorithm

Input: Coefficients of two polynomials, $A(x)$ and $B(x)$, of degree d
Output: Their product $C = A \cdot B$

Selection
Pick some points $x_0, x_1, \ldots, x_{n-1}$, where $n \geq 2d + 1$.

Evaluation
Compute $A(x_0), A(x_1), \ldots, A(x_{n-1})$ and $B(x_0), B(x_1), \ldots, B(x_{n-1})$.

Multiplication
Compute $C(x_k) = A(x_k)B(x_k)$ for all $k = 0, \ldots, n-1$.

Interpolation
Recover $C(x) = c_0 + c_1 x + \ldots + c_{2d} x^{2d}$

Fast Fourier Transform

The selection step and the multiplications are just linear time:

- In a typical setting for polynomial multiplication, the coefficients of the polynomials are real number.
- Moreover, are small enough that basic arithmetic operations take unit time.

Evaluating a polynomial of degree $d \leq n$ at a single point takes $O(n)$, and so the baseline for n points is $\Theta(n^2)$.

The Fast Fourier Transform (FFT) does it in just $O(n \log n)$ time, for a particularly clever choice of $x_0, \ldots, x_{n-1}$.

Evaluation by divide-and-conquer

Q: How to make it efficient?

First idea, we pick the n points,

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

then the computations required for each $A(x_i)$ and $A(-x_i)$ overlap a lot, because the even power of x_i coincide with those of $-x_i$.

We need to split $A(x)$ into its odd and even powers, for instance

$$3 + 4x + 6x^2 + 2x^3 + x^4 + 10x^5 = (3 + 6x^2 + x^4) + x(4 + 2x^2 + 10x^4)$$

More generally

$$A(x) = A_e(x^2) + xA_o(x^2)$$

where $A_e(\cdot)$, with the even-numbered coefficients, and $A_o(\cdot)$, with the odd-numbered coefficients, are polynomials of degree $\leq n/2 - 1$.

Evaluation by divide-and-conquer

Given paired points $\pm x_i$, the calculations needed for $A(x_i)$ can be recycled toward computing $A(-x_i)$:

$$
\begin{aligned}
A(x_i) &= A_e(x_i^2) + x_i A_o(x_i^2) \\
A(-x_i) &= A_e(x_i^2) - x_i A_o(x_i^2)
\end{aligned}
$$

Evaluating $A(x)$ at n paired points $\pm x_0, \ldots, \pm x_{n/2-1}$ reduces to evaluating $A_e(x)$ and $A_o(x)$ at just $n/2$ points, $x_0^2, \ldots, x_{n/2-1}^2$.

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

$$T(n) = 2T(n/2) + O(n) = O(n \log n)$$

How to choose n points?

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

Q: How can a square be negative?

- We use complex numbers.

At the very bottom of the recursion, we have a single point, 1, in which case the level above it must consist of its square roots, $\pm\sqrt{1} = \pm 1$.

The next level up then has $\pm\sqrt{+1} = \pm 1$, as well as the complex numbers $\pm\sqrt{-1} = \pm i$.

By continuing in this manner, we eventually reach the initial set of n points: the complex $n\,th$ roots of unity, that is the n complex solutions of the equation

$$z^n = 1$$

The n-th complex roots of unity

Solutions to the equation $z^n = 1$

- by the multiplication rules: solutions are $z = (1, \theta)$, for θ a multiple of $2\pi/n$.
- It can be represented as

$$1, \omega, \omega^2, \ldots, \omega^{n-1}$$

where

$$\omega = e^{2\pi i/n}$$

For n is even:

- These numbers are plus-minus paired.
- Their squares are the $(n/2)$-nd roots of unity.

The FFT algorithm

FFT(A, ω)
input : coefficient reprentation of a polynomial $A(x)$ of degree $\leq n-1$, where n is a power of 2; ω, an n-th root of unity
output: value representation $A(\omega^0), \ldots, A(\omega^{n-1})$

if $\omega = 1$ **then** return $A(1)$;
express $A(x)$ in the form $A_e(x^2) + xA_o(x^2)$;
call FFT(A_e,ω^2) to evaluate A_e at even powers of ω;
call FFT(A_o,ω^2) to evaluate A_o at even powers of ω;
for $j = 0$ *to* $n-1$ **do**
  compute $A(\omega^j) = A_e(\omega^{2j}) + \omega^j A_o(\omega^{2j})$;
end
return($A(\omega^0), \ldots, A(\omega^{n-1})$);

Interpolation

FFT moves from coefficients to values in time just $O(n \log n)$, when the points $\{x_i\}$ are complex n-th roots of unity $(1, \omega, \omega^2, \ldots, \omega^{n-1})$.

That is,

$$\langle value \rangle = \texttt{FFT}(\langle coefficients \rangle, \omega)$$

We will see that the interpolation can be computed by

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

A matrix reformation

Let's explicitly set down the relationship between our two representations for a polynomial $A(x)$ of degree $\leq n-1$.

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ & & \vdots & & \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

Let M be the matrix in the middle, which is a Vandermonde matrix.

- If $x_0, x_1, \dots, x_{n-1}$ are distinct numbers, then M is invertible.
- evaluation is multiplication by M, while interpolation is multiplication by M^{-1}.

A matrix reformation

This reformulation of our polynomial operations reveals their essential nature more clearly.

It justifies an assumption that $A(x)$ is uniquely characterized by its values at any n points.

Vandermonde matrices also have the distinction of being quicker to invert than more general matrices, in $O(n^2)$ time instead of $O(n^3)$.

However, using this for interpolation would still not be fast enough for us..

Interpolation resolved

In linear algebra terms, the FFT multiplies an arbitrary n-dimensional vector, which we have been calling the coefficient representation, by the $n \times n$ matrix.

$$M_n(\omega) = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{n-1} \\ & & \vdots & & \\ 1 & \omega^j & \omega^{2j} & \dots & \omega^{(n-1)j} \\ & & \vdots & & \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \dots & x^{(n-1)(n-1)} \end{bmatrix}$$

Its (j, k)-th entry (starting row- and column-count at zero) is ω^{jk}

Interpolation resolved

The columns of M are orthogonal to each other, which is often called the Fourier basis.

The FFT is thus a change of basis, a rigid rotation. The inverse of M is the opposite rotation, from the Fourier basis back into the standard basis.

Inversion formula

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

Interpolation resolved

Take ω to be $e^{2\pi i/n}$, and think of M as vectors in $\mathbb{C}^n$.

Recall that the angle between two vectors $u = (u_0, \ldots, u_{n-1})$ and $v(v_0, \ldots, v_{n-1})$ in $\mathbb{C}^n$ is just a scaling factor times their inner product

$$u \cdot v^* = u_0 v_0^* + u_1 v_1^* + \ldots + u_{n-1} v_{n-1}^*$$

where z^* denotes the complex conjugate of z.

The above quantity is maximized when the vectors lie in the same direction and is zero when the vectors are orthogonal to each other.

Interpolation resolved

Lemma

The columns of matrix M are orthogonal to each other.

Proof.

- Take the inner product of of any columns j and k of matrix M,

$$1 + \omega^{j-k} + \omega^{2(j-k)} + \ldots + \omega^{(n-1)(j-k)}$$

  This is a geometric series with first term 1, last term $\omega^{(n-1)(j-k)}$, and ratio ω^{j-k}.

- Therefore, if $j \neq k$, it evaluates to

$$\frac{1 - \omega^{n(j-k)}}{1 - \omega^{(j-k)}} = 0$$

- If $j = k$, then it evaluates to n.

Interpolation resolved

Corollary

$MM^* = nI$, *i.e.*,

$$M_n^{-1} = \frac{1}{n} M_n^*$$

The definitive FFT algorithm

The FFT takes as input a vector $a = (a_0, \ldots, a_{n-1})$ and a complex number ω whose powers $1, \omega, \omega^2, \ldots, \omega^{n-1}$ are the complex n-th roots of unity.

It multiplies vector a by the $n \times n$ matrix $M_n(\omega)$, which has (j, k)-th entry ω^{jk}.

The potential for using divide-and-conquer in this matrix-vector multiplication becomes apparent when M's columns are segregated into evens and odds.

The product of $M_n(\omega)$ with vector $a = (a_0, \ldots, a_{n-1})$, a size-$n$ problem, can be expressed in terms of two size-$n/2$ problems: the product of $M_{n/2}(\omega^2)$ with $(a_0, a_2, \ldots, a_{n-2})$ and with $(a_1, a_3, \ldots, a_{n-1})$.

This divide-and-conquer strategy leads to the definitive FFT algorithm, whose running time is $T(n) = 2T(n/2) + O(n) = O(nlogn)$.

The general FFT algorithm

FFT(a, ω)

input : An array $a = (a_0, a_1, \ldots, a_{n-1})$ for n is a power of 2; ω, an n-th root of unity

output: $M_n(\omega)a$

if $\omega = 1$ **then** return a;

$(s_0, s_1, \ldots, s_{n/2-1})$=FFT $((a_0, a_2, \ldots, a_{n-2}), \omega^2)$;

$(s'_0, s'_1, \ldots, s'_{n/2-1})$=FFT $((a_1, a_3, \ldots, a_{n-1}), \omega^2)$;

for $j = 0$ *to* $n/2 - 1$ **do**

  $r_j = s_j + \omega^j s'_j$;

  $r_{j+n/2} = s_j - \omega^j s'_j$;

end

return $(r_0, r_1, \ldots, r_{n-1})$;

Top 10 algorithms of the 20th century

1946: The Metropolis Algorithm

1947: Simplex Method

1950: Krylov Subspace Method

1951: The Decompositional Approach to Matrix Computations

1957: The Fortran Optimizing Compiler

1959: QR Algorithm

1962: Quicksort

1965: Fast Fourier Transform

1977: Integer Relation Detection

1987: Fast Multipole Method