算法设计与分析

分治法 (II)

目录

快速幂问题

输入： $a \in \mathbb{R},\ n \in \mathbb{N}$
输出： a^n

朴素算法：顺序乘法

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

乘法次数 $= n - 1$

分治法：分解

n 为偶数时：

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

n 为奇数时：

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

$$a^n = \begin{cases} a^{n/2} \times a^{n/2} & n \text{ 为偶数} \\ a^{(n-1)/2} \times a^{(n-1)/2} \times a & n \text{ 为奇数} \end{cases}$$

复杂度分析

基本操作： 乘法

- 子问题规模：小于 $n/2$
- 两个子问题（规模约为 $n/2$）是相同的，只需计算一次

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

主定理（情形 1）$\Rightarrow W(n) = \Theta(\log n)$

如何实现这个算法？递归 *vs.* 迭代

递归方法

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

1: **if** $n < 0$ **then**
2: **return** Power$(1/a, -n)$ // 处理负整数指数
3: **end if**
4: **if** $n = 0$ **then**
5: **return** 1
6: **end if**
7: **if** $n = 1$ **then**
8: **return** a
9: **end if**
10: **if** n 为偶数 **then**
11: **return** Power$(a^2, n/2)$
12: **end if**
13: **if** n 为奇数 **then**
14: **return** $a\times$ Power$(a^2, (n-1)/2)$
15: **end if**

迭代方法

$$y = a^n = a^{\sum_{i=0}^{k} b_i 2^i} = \prod_{i=0}^{k} (a^{2^i})^{b_i}$$

Algorithm 2 平方-乘法 (Square-and-Multiply)(a, n)

```
(b_k, b_{k-1}, ..., b_1, b_0) ← BinaryDecomposition(n)
y ← 1
power ← a
for i = 0 to k do
    if b_i = 1 then
        y ← y × power                                  // 加
    end if
    power ← power × power                              // 倍
end for
return y
```

快速幂算法的应用

斐波那契数列：$1, 1, 2, 3, 5, 8, 13, 21, \ldots$
添加 $F_0 = 0$，得到：

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

问题：给定初值 $F_0 = 0$, $F_1 = 1$，计算 F_n
朴素算法：从 $F_0, F_1, \ldots$ 开始，反复计算

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

复杂度：顺序加法：$\Theta(n)$

斐波那契数列的性质

更好的算法？如何推导通项公式?

斐波那契数列的性质

更好的算法？如何推导通项公式?

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

观察： F_n 是 F_{n-1} 和 F_{n-2} 的线性组合。这启发我们用线性代数来表示递推关系。

斐波那契数列的性质

更好的算法？如何推导通项公式?

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

观察： F_n 是 F_{n-1} 和 F_{n-2} 的线性组合。这启发我们用线性代数来表示递推关系。
命题： 设 $\{F_n\}$ 是斐波那契数列，则

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

数学归纳法证明
基础： $n = 1$:

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

证明（归纳步骤）

假设对任意 n，公式成立，即：

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

则对于 $n+1$，根据斐波那契数列的定义：

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

由归纳假设 $\Rightarrow$

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

通过快速幂改进算法

令

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

使用推广的快速幂算法计算 M^n

时间复杂度

- 矩阵乘法次数 $T(n) = \Theta(\log n)$
- 每次矩阵乘法需要 8 次数值乘法
- 总体复杂度为 $\Theta(\log n)$

通过快速幂改进算法

令

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

使用推广的快速幂算法计算 M^n

时间复杂度

- 矩阵乘法次数 $T(n) = \Theta(\log n)$
- 每次矩阵乘法需要 8 次数值乘法
- 总体复杂度为 $\Theta(\log n)$

进一步改进

- M 可以对角化 $(M = PM'P^{-1}) \Rightarrow$ 我们可以直接使用快速幂算法以获得更好的基本计算步骤（矩阵乘法）效率。

目录

整数加法

加法： 给定两个 n 位整数 a 和 b，计算 $a+b$。
减法： 给定两个 n 位整数 a 和 b，计算 $a-b$。
小学算法： $\Theta(n)$ 位操作。

	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

注： 小学加法和减法算法是渐近最优的。

整数乘法

乘法： 给定两个 n 位整数 a 和 b，计算 $a \times b$。
小学方法： $\Theta(n^2)$ 位操作
$\Theta(n^2)$ 原子位乘法 $+$ $\Theta(n^2)$ 原子位加法

[见原文第 14 页]

分治法：第一次尝试 (1/2)

分解： 将两个 n 位整数 x 和 y 分成左右两半（低位和高位）。令 $m = n/2$。

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

使用位移操作计算

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

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

分治法：第一次尝试 (2/2)

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

求解：递归地计算四个 $n/2$ 位整数的乘积。（重要操作）
合并：通过加法和移位得到结果。

$$T(n) = \underbrace{4\,T(n/2)}_{\text{递归调用}} + \underbrace{\Theta(n)}_{\text{加法、移位}}$$

主定理（情形 1）$\Rightarrow T(n) = \Theta(n^2)$
子问题数量太多 $\rightarrow$ 与传统小学方法运行时间相同，效率没有提升。
如何加速这个方法?

高斯技巧

高斯曾经注意到，尽管两个复数的乘积

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

看起来需要 4 次乘法，但实际上可以用 3 次完成：

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

[见原文第 17 页高斯图像]

Karatsuba 算法

1960 年，Kolmogorov 在一次研讨会上猜想小学乘法算法是最优的。一周之内，当时还是 23 岁学生的 Karatsuba 就找到了一个更好的算法，从而推翻了这个猜想。Kolmogorov 对这一发现非常兴奋，并于 1962 年发表了一篇论文。

[见原文第 18 页 Karatsuba 图像]

Karatsuba 算法： 第一个渐近快于二次方" 小学" 算法的算法。

减少子问题数量

思想：通过高斯技巧利用子问题之间的依赖关系

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

Algorithm 3 Karatsuba(x, y, n)

1: **if** $n = 1$ **then**
2: **return** $x \times y$
3: **else**
4: $m \leftarrow \lceil n/2 \rceil$
5: **end if**
6: $x_L \leftarrow \lfloor x/2^m \rfloor$; $x_R \leftarrow x \mod 2^m$
7: $y_L \leftarrow \lfloor y/2^m \rfloor$; $y_R \leftarrow y \mod 2^m$
8: $e \leftarrow$ Karatsuba(x_L, y_L, m)
9: $f \leftarrow$ Karatsuba(x_R, y_R, m)
10: $g \leftarrow$ Karatsuba$(x_L + x_R, y_L + y_R, m)$

理论分析

复杂度分析： 现在，递推关系为

$$\begin{cases} T(n) = 3\,T(n/2) + \Theta(n) \\ T(1) = 1 \end{cases} \quad \Rightarrow\ T(n) = \Theta(n^{\log_2 3}) = \Theta(n^{1.585})$$

合并和分解代价 $f(n)$ 很小 $\rightarrow$ $h(n)$ 主导总体复杂度。从 4 到 3 的常数因子改进发生在递归的每一层，复合效应导致了显著更低的界。

[Toom-Cook (1963)] Karatsuba 方法的更快推广
[Schönhage-Strassen (1971)] 对于足够大的 n，更快

一些说明

实践说明：

- 通常不需要一直递归到 1 位。对于大多数处理器，16 位或 32 位乘法是单条指令。
- GNU 多精度算术库根据操作数大小使用不同的算法。（用于 Maple、Mathematica、gcc、密码学等）

理论回顾（非正式）：

- 小学加/减法算法是最优的，因为它们已经是本地的
- 小学乘法算法不是最优的，因为它不是很本地的（全局相关）：分解有助于缩小局部性

目录

内积

内积： 给定两个长度为 n 的向量 $\mathbf{a} = (a_1, \ldots, a_n)$ 和 $\mathbf{b} = (b_1, \ldots, b_n)$，计算

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

小学方法： $\Theta(n)$ 次算术运算。

注： 小学点积算法是渐近最优的。

矩阵乘法

矩阵乘法： 给定两个 $n \times n$ 矩阵 X 和 Y，计算

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

[见原文第 24 页矩阵乘法示意图]

大学方法： $\Theta(n^3)$ 次算术运算

- Z 中有 n^2 个元素
- 计算每个元素需要 n 次算术乘法

大学矩阵乘法算法是渐近最优的吗？分治策略能做得更好吗？

朴素分治法

策略：将矩阵分成块：

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

其中：

$$Z_{11} = X_{11}\,Y_{11} + X_{12}\,Y_{21} \qquad Z_{12} = X_{11}\,Y_{12} + X_{12}\,Y_{22}$$
$$Z_{21} = X_{21}\,Y_{11} + X_{22}\,Y_{21} \qquad Z_{22} = X_{21}\,Y_{12} + X_{22}\,Y_{22}$$

递推关系：主定理（情形 1）

$$\begin{cases} T(n) = \overbrace{8\,T(n/2)}^{\text{递归调用}} + \overbrace{\Theta(n^2)}^{\text{加法/形成子矩阵}} \\ T(1) = 1 \end{cases} \quad \Rightarrow\ T(n) = \Theta(n^3)$$

突破

- 大学算法：$\Theta(n^3)$
- 朴素分治策略：$\Theta(n^3)$（令人失望）

在相当长的一段时间里，人们普遍认为这是可能达到的最佳运行时间，甚至在某些模型中证明了没有算法可以做得更好。

重大突破：这个效率可以通过一些巧妙的代数进一步提高。

Strassen 算法 (1/3)

Volker Strassen 于 1969 年首次发表了这个算法

- 证明了 $\Theta(n^3)$ 的一般矩阵乘法算法不是最优的
- 比标准矩阵乘法算法更快，对于大矩阵在实践中很有用
- 启发了更多关于矩阵乘法的研究，产生了更快的方法，例如 Coppersmith-Winograd 算法

[见原文第 27 页 Strassen 图像]

Strassen 算法 (2/3)

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

定义 **7** 个中间矩阵:

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

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

通过中间矩阵表示 Z_{ij}:

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

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

Strassen 算法 (3/3)

将子问题数量从 8 减少到 7
时间复杂度的递推关系（18 是算法每次应用时执行的加法/减法次数）

$$\begin{cases} T(n) = 7T(n/2) + 18n^2 \\ T(1) = 1 \end{cases} \Rightarrow T(n) = \Theta(n^{\log_2 7}) = \Theta(n^{2.8075})$$

Strassen 算法 (3/3)

将子问题数量从 8 减少到 7
时间复杂度的递推关系（18 是算法每次应用时执行的加法/减法次数）

$$\begin{cases} T(n) = 7T(n/2) + 18n^2 \\ T(1) = 1 \end{cases} \Rightarrow T(n) = \Theta(n^{\log_2 7}) = \Theta(n^{2.8075})$$

问：如果 n 不是 2 的幂怎么办?
答：可以用零填充矩阵。

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

更多关于矩阵乘法

这个分解如此巧妙和精妙，以至于人们不禁好奇 Strassen 是如何发现它的！

[见原文第 30 页图像]

更多关于矩阵乘法

这个分解如此巧妙和精妙，以至于人们不禁好奇 Strassen 是如何发现它的！

[见原文第 30 页图像]

矩阵乘法的复杂度

- 最佳上界：$O(n^{2.376})$ —Coppersmith-Winograd 算法
- 已知下界：$\Omega(n^2)$

更多关于矩阵乘法

这个分解如此巧妙和精妙，以至于人们不禁好奇 Strassen 是如何发现它的！

[见原文第 30 页图像]

矩阵乘法的复杂度

- 最佳上界：$O(n^{2.376})$ —Coppersmith-Winograd 算法
- 已知下界：$\Omega(n^2)$

应用

- 科学计算、图像处理、数据挖掘（回归、聚合、决策树）

矩阵乘法：最新进展

[见原文第 31 页时间线图]

矩阵乘法：最新进展

[见原文第 31 页周任飞介绍及论文]

矩阵乘法：最新进展

矩阵乘法可能看起来是一个晦涩的问题，但它是一种基本的计算操作。它被融入了人们每天使用的大部分算法中，用于各种任务，从显示更清晰的计算机图形到解决网络理论中的物流问题。就像在计算的其他领域一样，速度至关重要。即使是微小的改进最终也可能大大减少所需要的时间、计算能力和金钱。

段然团队修改了激光法标记块的方式，将 ω 的新上限设定在了 2.371866 左右，这要比 2020 年设定的上限 2.3728596 有所改进。这看起来是一个不大的变化，将上限降低了大约 0.001，但这是自 2010 年以来科学家们看到的最大进步。

Le Gall 说：" 要想进一步改进，就必须在 Coppersmith and Winograd 的原始方法基础上加以改进，而这种方法自 1987 年以来就没有真正改变过。" 但到目前为止，还没有人提出更好的方法。也许根本就没有。

周任飞说：" 改进 ω 实际上是理解这个问题的一部分。如果我们能很好地理解这个问题，就能设计出更好的算法。不过，人们对这个古老问题的理解还处于非常初级的阶段。"

1. 快速幂
2. 整数乘法
3. 矩阵乘法
4. 多项式乘法

动机

我们已经学习了如何进行乘法运算

- 整数：Gauss 技巧
- 矩阵：Strassen 算法

动机

我们已经学习了如何进行乘法运算

- 整数：Gauss 技巧
- 矩阵：Strassen 算法

如何进行多项式乘法?

动机

我们已经学习了如何进行乘法运算

- 整数：Gauss 技巧
- 矩阵：Strassen 算法

如何进行多项式乘法?

多项式乘法的应用

- 最快的多项式乘法意味着最快的整数乘法
  - 多项式和二进制整数非常相似——只需将变量 x 替换为基数 2 并注意进位
- 多项式乘法对于信号处理至关重要

多项式：系数表示

多项式 [系数表示]

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

多项式运算

加法。$\Theta(n)$

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

多项式运算

加法。$\Theta(n)$

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

求值。三种选择：

- 朴素算法。逐项计算：$\Theta(n^2)$
- 缓存算法。缓存 x^i：$\Theta(n)$
- Horner 算法。$a_0 + (x(a_1 + x(a_2 + \cdots + x(a_{n-2} + x(a_{n-1})))))$：$\Theta(n)$
  秦九韶早在数百年前就发现了这个算法
  参考链接：`https://zhuanlan.zhihu.com/p/22166332`

多项式运算

加法。$\Theta(n)$

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

求值。三种选择：

- 朴素算法。逐项计算：$\Theta(n^2)$
- 缓存算法。缓存 x^i：$\Theta(n)$
- Horner 算法。$a_0+(x(a_1+x(a_2+\cdots+x(a_{n-2}+x(a_{n-1})))))$：$\Theta(n)$
  秦九韶早在数百年前就发现了这个算法
  参考链接：`https://zhuanlan.zhihu.com/p/22166332`
  乘法（卷积）。使用暴力算法 $\Theta(n^2)$

$$\begin{aligned}A(x)\times B(x)&=a_0b_0+(a_0b_1+a_1b_0)x+\cdots+a_{n-1}b_{n-1}x^{2n-2}\\&=\sum_{i=0}^{2n-2}c_ix^i,\text{其中}c_i=\sum_{j=0}^{i}a_jb_{i-j}\end{aligned}$$

卷积的图示

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

[见原文第 36 页 - 卷积矩阵图示]

多项式：点值表示

代数基本定理 [Gauss，博士论文]

每个具有复系数的非零单变量 n 次多项式恰好有 n 个复根。

推论

一个 $n-1$ 次多项式 $A(x)$ 由其在 n 个不同点处的求值唯一确定。

[见原文第 37 页 - 多项式曲线图]

证明思路

假设另一个 $n-1$ 次多项式 $A'(x)$ 在 n 个不同点处与 $A(x)$ 有相同的值
$\Rightarrow A(x)-A'(x)$ 至多是 $n-1$ 次但有 n 个根 $\rightsquigarrow$ 与代数基本定理矛盾

多项式：点值表示

多项式 [点值表示]

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

多项式运算

加法。$\Theta(n)$ 次加法运算。

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

乘法（卷积）。$\Theta(n)$，但需要 $2n-1$ 个点。

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

求值。使用 Lagrange 公式 $\Theta(n^2)$

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

两种表示之间的转换

权衡。快速求值或快速乘法

表示方式	乘法	求值
系数表示	$\Theta(n^2)$	$\Theta(n)$
点值表示	$\Theta(n)$	$\Theta(n^2)$

[见原文第 40 页 - " 我全都要" 图片]

目标。两种表示之间的高效转换 $\Rightarrow$ 享受两个世界的最佳优势：所有操作都很快！

$$\underbrace{\boxed{a_0, a_1, \ldots, a_{n-1}}}_{\text{系数表示}} \longleftrightarrow \underbrace{\boxed{(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})}}_{\text{点值表示}}$$

两种表示之间的转换：求值

系数 $\Rightarrow$ 点值

给定 $A(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$，在 n 个不同点 $x_0, \ldots, x_{n-1}$ 处求值。

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

运行时间。矩阵-向量乘法 $\Theta(n^2)$（或 n 次 Horner 算法）。

两种表示之间的转换：插值

点值 ⇒ 系数

给定 n 个不同点 $x_0, \ldots, x_{n-1}$ 和值 $(y_0, \ldots, y_{n-1})$，找到唯一的多项式 $A(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$，使其在给定点处具有给定值。

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

Vandermonde 矩阵当且仅当 x_i 互不相同时可逆。

运行时间。高斯消元 $\Theta(n^3)$

重述我们的目标

已知的两种转换都不高效

- 系数 $\Rightarrow$ 点值：$\Theta(n^2)$
- 点值 $\Rightarrow$ 系数：$\Theta(n^3)$

需要更高效的转换方法。

重述我们的目标

已知的两种转换都不高效

- 系数 $\Rightarrow$ 点值：$\Theta(n^2)$
- 点值 $\Rightarrow$ 系数：$\Theta(n^3)$

需要更高效的转换方法。接下来，我们从第一个方向开始。重述我们的目标：

给定 n 个系数，快速计算 n 个点值元组。

重述我们的目标

已知的两种转换都不高效

- 系数 $\Rightarrow$ 点值：$\Theta(n^2)$
- 点值 $\Rightarrow$ 系数：$\Theta(n^3)$

需要更高效的转换方法。接下来，我们从第一个方向开始。重述我们的目标：

给定 n 个系数，快速计算 n 个点值元组。

多项式求值算法的最优复杂度是 $\Theta(n)$。因此，上述目标的 $\Theta(n^2)$ 复杂度似乎是不可避免的。

重述我们的目标

已知的两种转换都不高效

- 系数 $\Rightarrow$ 点值：$\Theta(n^2)$
- 点值 $\Rightarrow$ 系数：$\Theta(n^3)$

需要更高效的转换方法。接下来，我们从第一个方向开始。重述我们的目标：

给定 n 个系数，快速计算 n 个点值元组。

多项式求值算法的最优复杂度是 $\Theta(n)$。因此，上述目标的 $\Theta(n^2)$ 复杂度似乎是不可避免的。

高层次核心思想：施加结构以增强局部性 $\rightsquigarrow$ 降低复杂度

求值的分治法

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

划分有两种选择：按频率 vs. 按时间

频率抽取。将多项式分为低次项和高次项。

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

时间抽取。将多项式分为偶次项和奇次项。

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

基 2 时间抽取 (DIT)

澄清

我们强调，划分的目标不是提高单点多项式求值的效率，因为它已经是最优的。

最终目标是提高将 n 个点作为整体任务求值的效率。

初次尝试

朴素想法。随机选取 n 个不同点 $x_0, \ldots, x_{n-1}$，然后通过 $A_{\mathsf{even}}(x^2) + xA_{\mathsf{odd}}(x^2)$ 计算 $A(x)$。

- $T(n)$：在 n 个点处求值 $n-1$ 次多项式
- $E(n)$：在 1 个点处求值 $n-1$ 次多项式

初次尝试

朴素想法。随机选取 n 个不同点 $x_0, \ldots, x_{n-1}$，然后通过 $A_{\mathsf{even}}(x^2) + xA_{\mathsf{odd}}(x^2)$ 计算 $A(x)$。

- $T(n)$：在 n 个点处求值 $n-1$ 次多项式
- $E(n)$：在 1 个点处求值 $n-1$ 次多项式

问题。效率没有提高

- 在 n 个点处求值 $n-1$ 次 $A(x)$：$T(n) = n \cdot E(n)$
- 在 n 个点处求值 $n/2-1$ 次的 $A_{\mathsf{even}}(x)$ 和 $A_{\mathsf{odd}}(x)$：$2 \times n \cdot E(n/2) = 2n \cdot E(n/2)$

$E(n)$ 是线性函数 $\rightsquigarrow$ 没有效率提升

- 根本原因是问题规模没有真正减半。

解决方案。减少求值点的数量

基本思想 (1/2)

基本思想。通过选择 n 个点为正负配对来引入简单结构，即

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

注意 x_i 的偶次幂与 $-x_i$ 的偶次幂相同 $\Rightarrow$ 每个 $A(x_i)$ 和 $A(-x_i)$ 所需的计算大量重叠。

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

现在，在 n 个配对点 $\pm x_0, \ldots, \pm x_{n/2-1}$ 处求值 $n-1$ 次多项式 $A(x)$
$\Rightarrow$ 仅在 $n/2$ 个点 $x_0^2, \ldots, x_{n/2-1}^2$ 处求值 $n/2-1$ 次多项式 $A_{\mathsf{even}}(x)$ 和 $A_{\mathsf{odd}}(x)$。

基本思想 (2/2)

[见原文第 48 页 - 递归树图示]

现在，原始规模为 n 的问题以这种方式被重构为两个规模为 $n/2$ 的子问题，再加上一些线性时间的算术运算。

$T(n)$：在 n 个点处求值 $n-1$ 次多项式

- 如果我们能递归，就能得到运行时间为

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

的分治过程

这是 $\Theta(n \log n)$，正是我们想要的。

技术障碍

技术障碍。正负配对技巧只在递归的最顶层有效。

- 要在下一层递归，我们需要 $n/2$ 个求值点 $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$ 本身也是正负配对的。

技术障碍

技术障碍。正负配对技巧只在递归的最顶层有效。

- 要在下一层递归，我们需要 $n/2$ 个求值点 $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$ 本身也是正负配对的。

但是平方怎么可能是负数呢?

技术障碍

技术障碍。正负配对技巧只在递归的最顶层有效。

- 要在下一层递归，我们需要 $n/2$ 个求值点 $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$ 本身也是正负配对的。

但是平方怎么可能是负数呢?

[见原文第 49 页 - Mission Impossible 图片]

技术障碍

技术障碍。正负配对技巧只在递归的最顶层有效。

- 要在下一层递归，我们需要 $n/2$ 个求值点 $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$ 本身也是正负配对的。

但是平方怎么可能是负数呢?

[见原文第 49 页 - Mission Impossible 图片]

当然，除非我们使用**复数**。

选择哪些复数?

好吧，但是选择哪些复数呢？让我们通过"逆向工程"这个过程来弄清楚。

- 在递归的底部，我们有一个点，比如说，1。
- 在它上面的层必须由它的平方根组成，± 1。
- 再上一层是 $(+1, -1)$ 和 $(+i, -i)$，直到我们达到 $n = 2^k$ 个叶节点。

[见原文第 50 页 - 递归树图示]

选择 n 个复数

n 次单位根是满足 $x^n = 1$ 的复数。

选择 n 个复数

n 次单位根是满足 $x^n = 1$ 的复数。事实。n 次单位根是 $\omega^0 = 1, \omega^1, \omega^2, \ldots, \omega^{n-1}$，其中 $\omega = e^{2\pi i/n} = \cos\frac{2\pi}{n} + i\sin\frac{2\pi}{n}$

选择 n 个复数

n 次单位根是满足 $x^n = 1$ 的复数。事实。n 次单位根是 $\omega^0 = 1, \omega^1, \omega^2, \ldots, \omega^{n-1}$，其中 $\omega = e^{2\pi i/n} = \cos\frac{2\pi}{n} + i\sin\frac{2\pi}{n}$ 证明。$(\omega^k)^n = (e^{2\pi ik/n})^n = (e^{\pi i})^{2k} = (-1)^{2k} = 1$

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

选择 n 个复数

n 次单位根是满足 $x^n = 1$ 的复数。事实。n 次单位根是 $\omega^0 = 1, \omega^1, \omega^2, \ldots, \omega^{n-1}$，其中 $\omega = e^{2\pi i/n} = \cos\frac{2\pi}{n} + i\sin\frac{2\pi}{n}$ 证明。$(\omega^k)^n = (e^{2\pi ik/n})^n = (e^{\pi i})^{2k} = (-1)^{2k} = 1$

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

如果 n 是偶数，n 次单位根是正负配对的，$\omega^{n/2+j} = -\omega^j$

- 将它们平方得到 $(n/2)$ 次单位根：$v^0, v^1, \ldots, v^{n/2-1}$，其中 $v = \omega^2 = e^{4\pi i/n}$。

选择 n 个复数

n 次单位根是满足 $x^n = 1$ 的复数。事实。n 次单位根是 $\omega^0 = 1, \omega^1, \omega^2, \ldots, \omega^{n-1}$，其中 $\omega = e^{2\pi i/n} = \cos\frac{2\pi}{n} + i\sin\frac{2\pi}{n}$ 证明。$(\omega^k)^n = (e^{2\pi ik/n})^n = (e^{\pi i})^{2k} = (-1)^{2k} = 1$

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

如果 n 是偶数，n 次单位根是正负配对的，$\omega^{n/2+j} = -\omega^j$

- 将它们平方得到 $(n/2)$ 次单位根：$v^0, v^1, \ldots, v^{n/2-1}$，其中 $v = \omega^2 = e^{4\pi i/n}$。

如果我们从 $\omega^0, \omega^1, \omega^2, \ldots, \omega^{n-1}$ 开始，其中 $n = 2^k$，那么在第 k 层递归时我们将得到 $(n/2^k)$ 次单位根。

- 所有这些单位根集合都是正负配对的 $\Rightarrow$ 分治算法将完美运行

$n = 8$ 的演示

[见原文第 52 页 - 单位圆上 8 次单位根的图示]

$$\omega^0 = 1$$

$$\omega^1 = e^{\frac{\pi}{4}i} = \frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2} \cdot i$$

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

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

$$\omega^4 = e^{\pi i} = -1$$

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

$$\omega^6 = e^{\frac{3\pi}{2}i} = -i$$

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

递归结构与 FFT

[见原文第 53 页 - 递归树图示]

DFT：傅里叶矩阵 $M_n(\omega)$

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & \omega^0 & \omega^{2\cdot 0} & \cdots & \omega^{(n-1)\cdot 0} \\ 1 & \omega^1 & \omega^{2\cdot 1} & \cdots & \omega^{(n-1)\cdot 1} \\ 1 & \omega^2 & \omega^{2\cdot 2} & \cdots & \omega^{(n-1)\cdot 2} \\ 1 & \omega^3 & \omega^{2\cdot 3} & \cdots & \omega^{(n-1)\cdot 3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

快速傅里叶变换 (FFT)

精确目标。在 n 次单位根 $\omega^0, \omega^1, \ldots, \omega^{n-1}$ 处求值 $A(x) = a_0 + \cdots + a_{n-1}x^{n-1}$
分解。将多项式分为偶次幂和奇次幂:

$$\begin{aligned} A_{\mathsf{even}}(x) &= a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2}x^{n/2-1} \\ A_{\mathsf{odd}}(x) &= a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1}x^{n/2-1} \\ A(x) &= A_{\mathsf{even}}(x^2) + xA_{\mathsf{odd}}(x^2) \end{aligned}$$

解决。在 $(n/2)$ 次单位根 $v^0, v^1, \ldots, v^{n/2-1}$ 处求值 $A_{\mathsf{even}}(x)$ 和 $A_{\mathsf{odd}}(x)$
合并。$v^k = (\omega^k)^2$

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

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

FFT 算法伪代码

Algorithm 4 $\mathsf{FFT}(A, n, \omega)$

输入: $n-1$ 次多项式 A 的系数表示，主 n 次单位根 $\omega = e^{2\pi i/n}$
输出: 点值表示 $A(\omega^0), \ldots, A(\omega^{n-1})$

1: **if** $n = 1$ **then**
2: **return** a_0
3: **end if**
4: 表示 $A(x) = A_{\mathsf{even}}(x^2) + xA_{\mathsf{odd}}(x^2)$
5: $\mathsf{FFT}(A_{\mathsf{even}}, \frac{n}{2}, \omega^2) \to (A_{\mathsf{even}}((\omega^2)^0), \ldots, A_{\mathsf{even}}((\omega^2)^{n/2-1}))$
6: $\mathsf{FFT}(A_{\mathsf{odd}}, \frac{n}{2}, \omega^2) \to (A_{\mathsf{odd}}((\omega^2)^0), \ldots, A_{\mathsf{odd}}((\omega^2)^{n/2-1}))$
7: **for** $j = 0$ **to** $n-1$ **do**
8: $A(\omega^j) = A_{\mathsf{even}}(\omega^{2j}) + \omega^j A_{\mathsf{odd}}(\omega^{2j})$ // $\Theta(n)$
9: **end for**
10: **return** $A(\omega^0), \ldots, A(\omega^{n-1})$

FFT 总结

定理

假设 $n = 2^k$。FFT 算法在 $\Theta(n \log n)$ 步内在每个 n 次单位根处求值一个 $n-1$ 次多项式。

运行时间

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

本质：选择具有特殊结构的 n 个点来加速 DFT 计算。

$$\boxed{a_0, a_1, \ldots, a_{n-1}} \xrightarrow{\Theta(n \log n)} \boxed{(\omega^0, y_0), \ldots, (\omega^{n-1}, y_{n-1})}$$

系数表示　　$\xleftarrow{???}$　　点值表示

回顾

我们首先开发了一种高层次的多项式乘法方法

$$\text{系数表示} \Rightarrow \text{点值表示}$$

点值表示使多项式乘法变得简单，但算法的输入输出形式是系数表示。

- 因此我们设计了 FFT：系数 $\Rightarrow$ 点值，时间仅为 $\Theta(n \log n)$，其中点 $\{x_i\}_n$ 是复数 n 次单位根 $(1, \omega, \omega^2, \ldots, \omega^{n-1})$。

$$\langle \text{值} \rangle = \mathsf{FFT}(\langle \text{系数} \rangle, \omega)$$

$$\underbrace{\boxed{a_0, a_1, \ldots, a_{n-1}}}_{\text{系数表示}} \underset{\text{插值}}{\overset{\text{求值}}{\rightleftharpoons}} \underbrace{\boxed{(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})}}_{\text{点值表示}}$$

插值

谜题的最后一块是逆运算——插值。令人惊讶的是：

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

插值因此以最简单优雅的方式得到解决，使用同样的 FFT 算法，但用 ω^{-1} 代替 ω！
这可能看起来像一个神奇的巧合，但当我们用线性代数的语言重新描述多项式运算时，它会变得更加合理。

逆离散傅里叶变换

点值 $\Rightarrow$ 系数

给定 n 个不同点 $x_0, \ldots, x_{n-1}$ 和值 $y_0, \ldots, y_{n-1}$，找到唯一的多项式 $A(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$，使其在给定点处具有给定值。

逆 DFT：傅里叶矩阵的逆 $F_n(\omega)^{-1}$

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & \omega^0 & \omega^{2\cdot 0} & \cdots & \omega^{(n-1)\cdot 0} \\ 1 & \omega^1 & \omega^{2\cdot 1} & \cdots & \omega^{(n-1)\cdot 1} \\ 1 & \omega^2 & \omega^{2\cdot 2} & \cdots & \omega^{(n-1)\cdot 2} \\ 1 & \omega^3 & \omega^{2\cdot 3} & \cdots & \omega^{(n-1)\cdot 3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{pmatrix}^{-1} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

傅里叶矩阵

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

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

关键事实

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

声明。F_n 和 G_n 互为逆矩阵
证明。检验 $F_n G_n$

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

求和引理

设 ω 为主 n 次单位根。则

$$\sum_{j=0}^{n-1} \omega^{kj} = \begin{cases} n & \text{如果} k = 0 \mod n \\ 0 & \text{否则} \end{cases}$$

- 如果 k 是 n 的倍数，则 $\omega^k = 1 \Rightarrow$ 级数求和为 n

结论

计算逆 FFT，应用相同的算法但使用

$$\omega^{-1} = e^{-2\pi i/n}$$

作为主 n 次单位根（并将结果除以 n）。

- 交换 $\langle a_0, \ldots, a_{n-1} \rangle$ 和 $\langle y_0, \ldots, y_{n-1} \rangle$ 的角色

插值问题解决

$\langle$值$\rangle = \mathsf{FFT}(\langle$系数$\rangle, \omega)$

$$\begin{pmatrix} A(\omega^0) \\ A(\omega^1) \\ A(\omega^2) \\ \vdots \\ A(\omega^{n-1}) \end{pmatrix} = \begin{pmatrix} 1 & \omega^0 & \omega^{2\cdot 0} & \cdots & \omega^{(n-1)\cdot 0} \\ 1 & \omega^1 & \omega^{2\cdot 1} & \cdots & \omega^{(n-1)\cdot 1} \\ 1 & \omega^2 & \omega^{2\cdot 2} & \cdots & \omega^{(n-1)\cdot 2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

$\langle$系数$\rangle = \frac{1}{n}\mathsf{FFT}(\langle$值$\rangle, \omega^{-1})$

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix} = \frac{1}{n} \begin{pmatrix} 1 & (\omega_0^{-1})^1 & (\omega_0^{-1})^2 & \cdots & (\omega_0^{-1})^{n-1} \\ 1 & (\omega_1^{-1})^1 & (\omega_1^{-1})^2 & \cdots & (\omega_1^{-1})^{n-1} \\ 1 & (\omega_2^{-1})^1 & (\omega_2^{-1})^2 & \cdots & (\omega_2^{-1})^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & (\omega_{n-1}^{-1})^1 & (\omega_{n-1}^{-1})^2 & \cdots & (\omega_{n-1}^{-1})^{n-1} \end{pmatrix} \begin{pmatrix} A(\omega^0) \\ A(\omega^1) \\ A(\omega^2) \\ \vdots \\ A(\omega^{n-1}) \end{pmatrix}$$

逆 FFT 总结

定理

假设 $n = 2^k$。逆 FFT 算法在 $\Theta(n \log n)$ 步内，给定每个 n 次单位根处的值，插值出一个 $n-1$ 次多项式。

运行时间。与 FFT 几乎相同的算法。

$$\boxed{a_0, a_1, \ldots, a_{n-1}} \underset{\Theta(n \log n)}{\overset{\Theta(n \log n)}{\rightleftharpoons}} \boxed{(\omega^0, y_0), \ldots, (\omega^{n-1}, y_{n-1})}$$

系数表示　　点值表示

多项式乘法

定理

两个 $n-1$ 次多项式可以在 $\Theta(n \log n)$ 步内相乘。

（用 0 填充使 n 成为 2 的幂）

[见原文第 65 页 - 多项式乘法流程图]

- 系数表示 $\xrightarrow{\text{2次 FFT } \Theta(n \log n)}$ 点值表示
- 点值乘法 $\Theta(n)$
- 点值表示 $\xrightarrow{\text{1次逆 FFT } \Theta(n \log n)}$ 系数表示

实际上，$2n-1$ 个点值元组就足够了。
但是，FFT 要求输入大小为 2^k，输出大小也是如此。

说明

标准 FFT。在 n 次单位根 $\omega^0, \omega^1, \ldots, \omega^{n-1}$ 处求值 $n-1$ 次 $A(x)$，通过在 $(n/2)$ 次单位根处求值 $n/2-1$ 次多项式 $A_{\mathsf{even}}(x)$ 和 $A_{\mathsf{odd}}(x)$。
我们选择多项式的次数作为输入大小，因为它决定了递归调用的深度。

标准 FFT 可以很容易地扩展到在 $2n$ 次单位根 $\omega^0, \omega^1, \ldots, \omega^{2n-1}$ 处求值 $n-1$ 次多项式 $A(x)$，通过在 n 次单位根处求值 $n/2-1$ 次多项式 $A_{\mathsf{even}}(x)$ 和 $A_{\mathsf{odd}}(x)$。
我们仍然选择多项式的次数作为输入大小，递推关系类似，

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

总体复杂度在渐近意义上没有变化。

FFT 的扩展

FFT 在复数域 $\mathbb{C}$ 中工作，根可能是复数 $\rightsquigarrow$ 精度损失不可避免
有时我们只需要在有限域中工作，例如 $\mathbb{F} = \mathbb{Z}/p$，即模素数 p 的整数。

- 当 n 整除 $p-1$ 时，主 n 次单位根存在，所以我们有 $p = \xi n + 1$，其中 ξ 是正整数。
- 特别地，设 ω 是主 $(p-1)$ 次单位根，则可以通过令 $\alpha = \omega^{\xi}$ 找到 n 次单位根 α

扩展 $\Rightarrow$ 数论变换 (NTT)：通过将离散傅里叶变换特化到 $\mathbb{F}$ 上得到。

- 无精度损失 & 更快
- 在 SNARK 的实现中广泛使用，例如 `libsnark`

[见原文第 67 页 - libfqfft 库截图]

FFT 的应用

- 光学、声学、量子物理、电信、雷达、控制系统、信号处理、语音识别、数据压缩、图像处理、地震学、质谱分析...
- 数字媒体。[DVD, JPEG, MP3, H.264]
- 医学诊断。[MRI, CT, PET 扫描, 超声波]
- Poisson 方程的数值解。
- Shor 的量子因式分解算法。

"FFT 是 [20 世纪] 真正伟大的计算发展之一。它改变了科学和工程的面貌，以至于说没有 FFT 我们所知的生活将大不相同并不夸张。"

—*Charles van Loan*

Gilbert Strang 将 FFT 描述为" 我们这一生最重要的数值算法"。

傅里叶分析

傅里叶定理 [Fourier, Dirichlet, Riemann]

任何（足够光滑的）周期函数都可以表示为一系列正弦波的和。

[见原文第 69 页 - 锯齿波傅里叶级数图示]

欧拉公式。

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

正弦波。正弦和余弦的和 = 复指数的和。

傅里叶变换

[见原文第 70 页 - 时域 vs 频域图示]

FFT 的信号处理观点

时域和频域之间的快速转换方法

FFT 的算法观点

快速乘法和求值多项式的方法

FFT：简史

Gauss。分析小行星谷神星的周期运动（用拉丁文）
Runge-König (1924)。奠定理论基础。
Danielson-Lanczos (1942)。高效算法，X 射线晶体学。
Cooley-Tukey (1965)。监测苏联核试验和追踪潜艇。重新发现并推广 FFT。

[见原文第 71 页 - Cooley-Tukey 论文截图]

重要性直到数字计算机出现后才被充分认识。

Cooley-Tukey 算法的故事

James Cooley 和 John Tukey 独立地重新发现了这些早期算法，并在 1965 年发表了一个更通用的 FFT，适用于 n 是合数而不一定是 2 的幂的情况，并分析了 $O(n \log n)$ 的复杂度。Tukey 在肯尼迪总统科学顾问委员会的一次会议上提出了这个想法，会议的讨论主题涉及通过在苏联周围设置传感器来检测核试验。为了分析这些传感器的输出，需要一个 FFT 算法。在与 Tukey 的讨论中，Richard Garwin 认识到该算法不仅适用于国家安全问题，还适用于广泛的问题，包括他当时直接感兴趣的一个问题：确定氦-3 的 3D 晶体中自旋取向的周期性。Garwin 将 Tukey 的想法交给了在 IBM Watson 实验室工作的 Cooley 进行实现。Cooley 和 Tukey 在相对较短的六个月内发表了论文。由于该想法的可专利性受到质疑，算法进入了公共领域，通过接下来十年的计算革命，使 FFT 成为数字信号处理中不可或缺的算法之一。

FFT 实践

西方最快傅里叶变换 (FFTW)。[Frigo 和 Johnson]

- 优化的 C 库。
- 特性：DFT, DCT, 实数, 复数, 任意大小, 任意维度。
- 获得 1999 年 Wilkinson 奖。

实现细节。

- 不执行预定算法，而是评估你的硬件并使用专用编译器生成针对问题" 形状" 优化的算法。
- 核心算法是 Cooley-Tukey 的非递归版本。
- $\Theta(n \log n)$，即使对于素数大小也是如此。

[见原文第 73 页 - FFTW logo]

整数乘法，重访

整数乘法。给定两个 n 位整数 $a = a_{n-1} \dots a_1 a_0$ 和 $b = b_{n-1} \dots b_1 b_0$，计算它们的乘积 ab。

1. 构造两个多项式（基 2 表示 $\Rightarrow a = A(2),\ b = B(2)$）

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

2. 通过 FFT 计算 $C(x) = A(x)B(x)$，求值 $C(2) = ab$

运行时间：$\Theta(n \log n)$ 次复数算术运算。

实践。GMP 根据 n 的大小使用暴力法、Karatsuba 和 FFT。

[见原文第 74 页 - GMP logo]

地球上最快的大数库

总结 (1/3)

分治法的概念

主要思想。将问题归约为子问题

原则。子问题应与原问题类型相同，并可独立求解。

- 直接划分：将原问题分成大小大致相同的子问题
  - FindMinMax, 归并排序
- 复杂划分
  - 一般选择：使用中位数的中位数作为枢轴（找到枢轴本身需要努力）
  - 最近点对：分析中线周围条带
  - 凸包：有时很难以平衡方式划分（凸包）

总结 (2/3)

实现。递归或迭代（注意可以直接求解的最小子问题）
时间复杂度

- 找到递推关系和初始值，求解递推关系

分治算法的递推关系

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

- a：子问题数量，n/b：子问题规模
- $f(n)$：划分和合并的代价

总结 (3/3)

优化技巧 1。减少子问题数量：当 $f(n)$ 不是很大时，$h(n) = n^{\log_b a}$ 主导整体复杂度 $\Rightarrow$ $T(n) = \Theta(h(n))$

- 减少 a 可以立即降低 $T(n)$ 的阶
- 当子问题相关时 $\rightsquigarrow$ 利用关系通过组合其他子问题的解来解决某些子问题

示例

- 幂算法：子问题相同
- 简单代数技巧：整数乘法（$f(n)$ 仍然很低）
- 利用依赖性：矩阵乘法（$f(n)$ 可能增加但不影响阶）

优化技巧 2。减少划分和合并的代价 $f(n)$：添加全局预处理

- 最近点对

重要的分治算法

- 搜索算法：二分搜索
- 排序算法：快速排序、归并排序
- 选择算法：找最小/最大值、一般选择算法
- 最近点对、凸包
- 快速幂算法
- 矩阵乘法：Strassen 算法
- 整数、多项式乘法：FFT

从前慢

随着算法不断拨快生活的节奏，内心越希望可以偶尔慢下来，松下烹茶、雨夜听琴，享受一份从容安宁。

——我的心声

从前慢

随着算法不断拨快生活的节奏，内心越希望可以偶尔慢下来，松下烹茶、雨夜听琴，享受一份从容安宁。

——我的心声

[木心照片]
木心

[《云雀叫了一整天》]
云雀叫了一整天

从前慢

随着算法不断拨快生活的节奏，内心越希望可以偶尔慢下来，松下烹茶、雨夜听琴，享受一份从容安宁。

——我的心声

[木心照片]
木心

[《云雀叫了一整天》]
云雀叫了一整天

从前慢，慢的不仅是车、马与邮件，还有在等待中默默酝酿的心。一封薄薄的信，装着一份炽热的情，翻过山淌过水送至一生唯一的爱人手中，信上的折痕都是那么饱含爱意...

——网易云音乐评论