



哈尔滨工业大学

海量数据计算研究中心

Massive Data Computing Lab @ HIT

算法设计与分析

第三讲 分治法

哈尔滨工业大学
丁小欧

dingxiaoou@hit.edu.cn



本讲内容

- 3.1 分治法
- 3.2 分治法的简单实例
- 3.3 矩阵乘法
- 3.4 元素选取问题的线性时间算法
- 3.5*快速傅里叶变换

背景

- 分治法是一种古老而实用的策略
- “一个较大的力量分解为小的力量，这样小的力量无法与大的力量抗衡”
- 当前，将大区域化分为小块进行治理
- 省、市、县、镇等层层管理，每个地方都是有组织的

“分治”的背景

—治众如治寡，分数是也。——《孙子兵法》

- “分数”：各级组织的划分
- “分而治之”

—分治的要素：“天时地利人和”

- “农夫朴力而寡能，则上不失天时，下不失地利，中得人和而百事不废”——《荀子·王霸篇》

什么问题可以用“分治”解决呢？

– 有以下三个特点：

- 原问题可以分解为若干个规模更小的相同的子问题
- 子问题互相独立
- 子问题的解可以合并为原问题的解

分治算法的设计

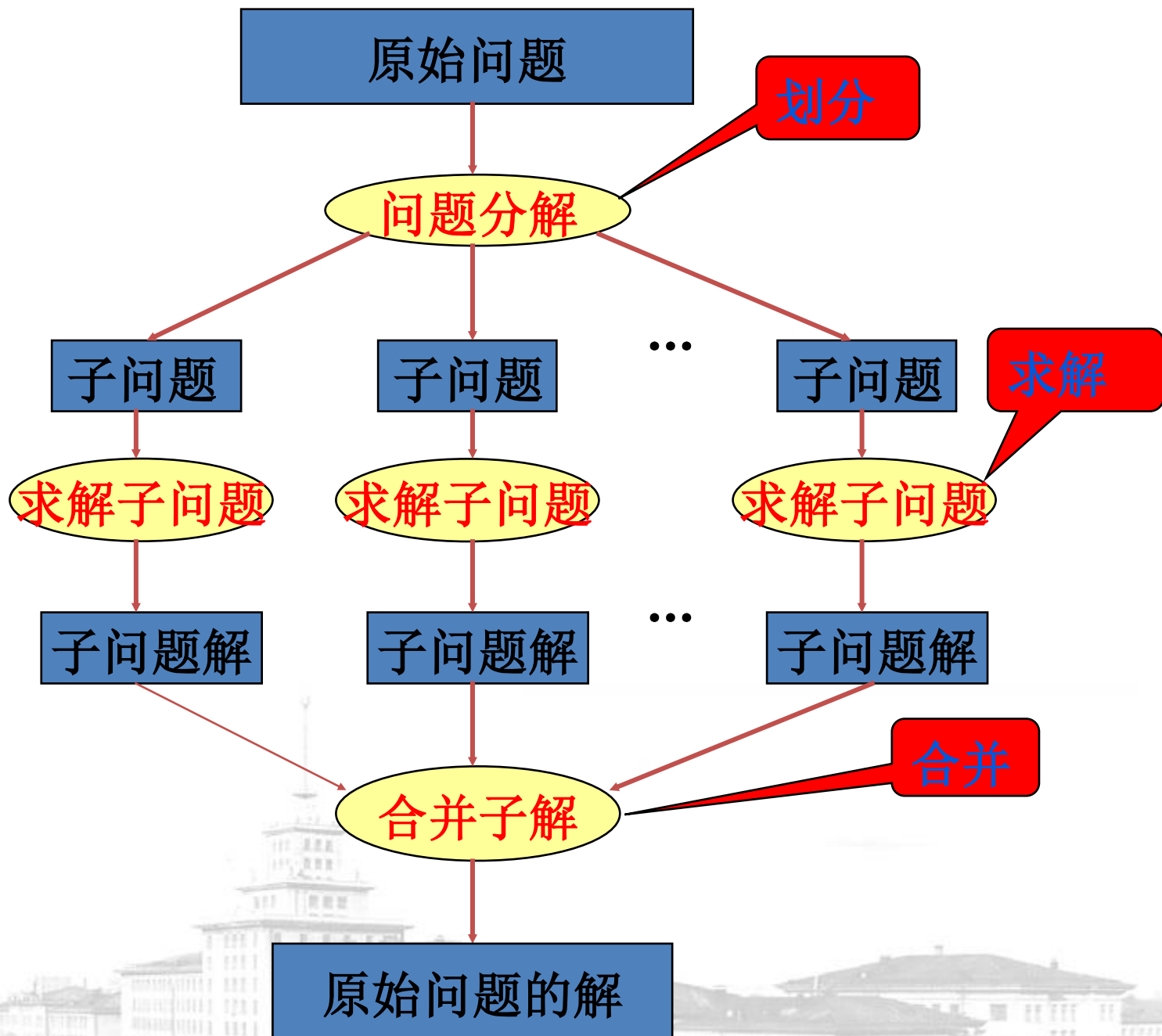
- 每层递归应用三个步骤
 - 分解(**Divide**):将原问题划分为多个子问题, 子问题的形式(通常)与原问题一样, 但有更小的规模
 - 解决(**Conquer**):递归求解子问题, 如果子问题规模足够小, 则停止递归, 直接求解
 - 合并(**Combine**):将子问题的解组合成原问题的解

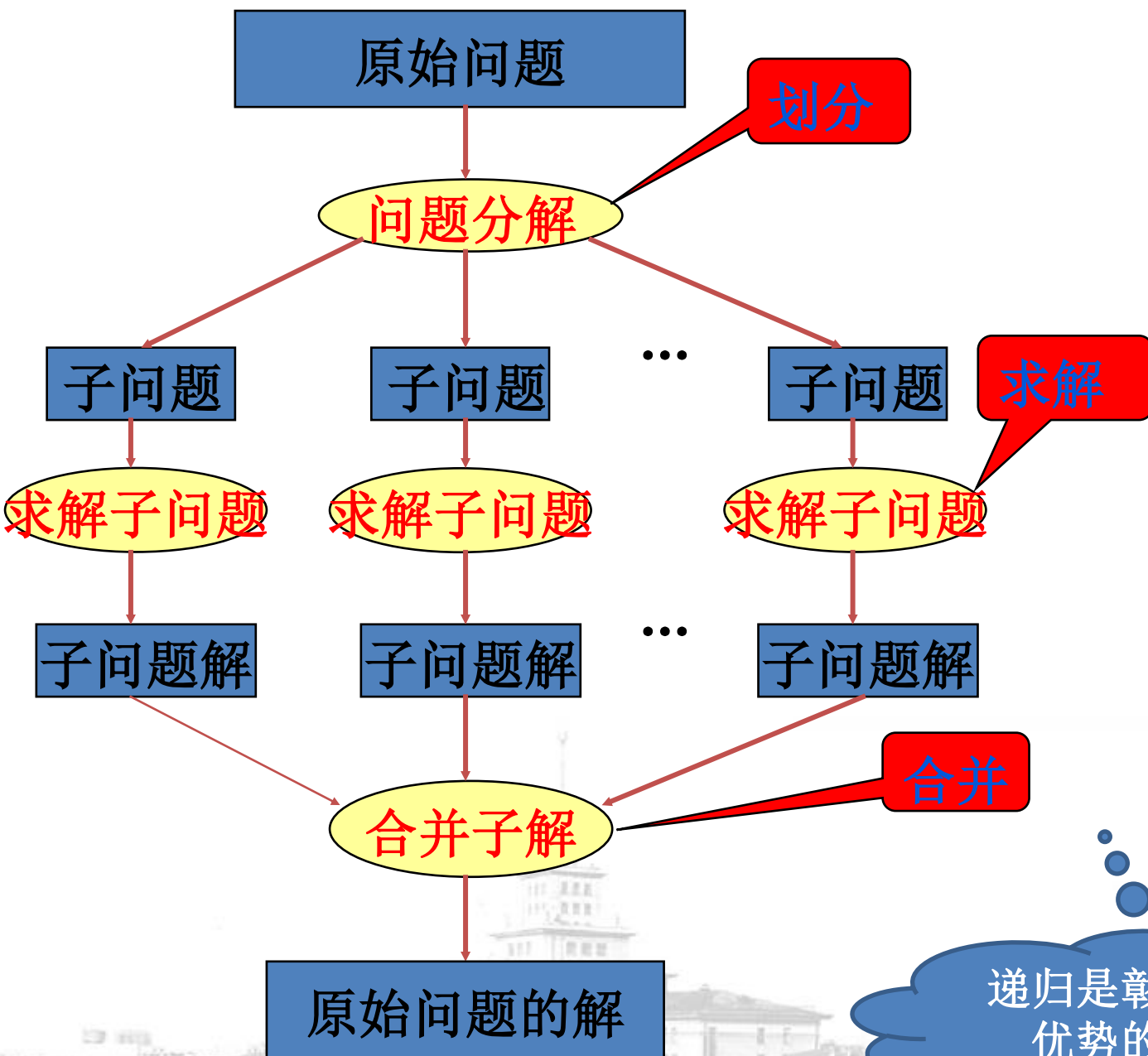


分治算法的设计

- 每层递归应用三个步骤
 - 分解(**Divide**):将原问题划分为多个子问题, 子问题的形式(通常)与原问题一样, 但有更小的规模
 - 解决(**Conquer**):递归求解子问题, 如果子问题规模足够小, 则停止递归, 直接求解
 - 合并(**Combine**):将子问题的解组合成原问题的解

Divide-and-Conquer 算法

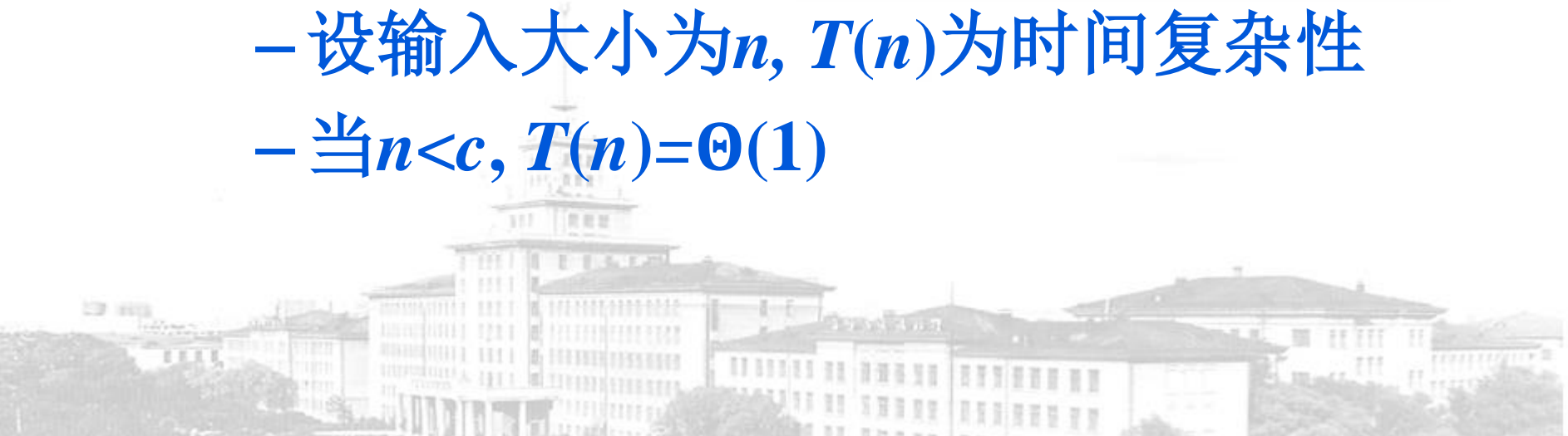




递归是彰显分治法优势的利器！

分治算法的分析

- 分析过程
 - 建立递归方程
 - 求解
- 递归方程的建立方法
 - 设输入大小为 n , $T(n)$ 为时间复杂性
 - 当 $n < c$, $T(n) = \Theta(1)$



– 划分阶段的时间复杂性

- 划分问题为 a 个子问题。
- 每个子问题大小为 n/b 。
- 划分时间可直接得到= $D(n)$

– 递归求解阶段的时间复杂性

- 递归调用
- 求解时间= $aT(n/b)$

– 合并阶段的时间复杂性

- 时间可以直接得到= $C(n)$



—概括:

- $T(n) = \theta(1)$ if $n < c$
- $T(n) = aT(n/b) + D(n) + C(n)$ if $n \geq c$

—求解递归方程 $T(n)$

- 使用第二章的方法



本讲内容

3.1 分治法

3.2 分治法的简单实例

3.3 矩阵乘法

3.4 元素选取问题的线性时间算法

3.5*快速傅里叶变换

一、同时求最大值和最小值

输入：数组 $A[1, \dots, n]$

输出： A 中的 \max 和 \min

通常，直接扫描需要 $2n-2$ 次比较操作

对于只确定最小值：其下界为 $n-1$ 次比较

引申为：体育比赛，除了冠军以外，每个队都至少要输掉一场比赛

一、求最大值和最小值

输入：数组 $A[1, \dots, n]$

输出： A 中的 \max 和 \min

通常，直接扫描需要 $2n-2$ 次比较操作

思考：能否用分治算法求解？

进一步思考：

- 可将数组中的元素分解为 $n/2$ 个子问题，直到 $n/2$ 中的元素 ≤ 2 为止
- 每个子问题与原问题性质相同，可递归求每个子问题的解，且每个子问题的解都是独立
- 子问题的解可以整合为原问题的解

一、求最大值和最小值

输入：数组 $A[1, \dots, n]$

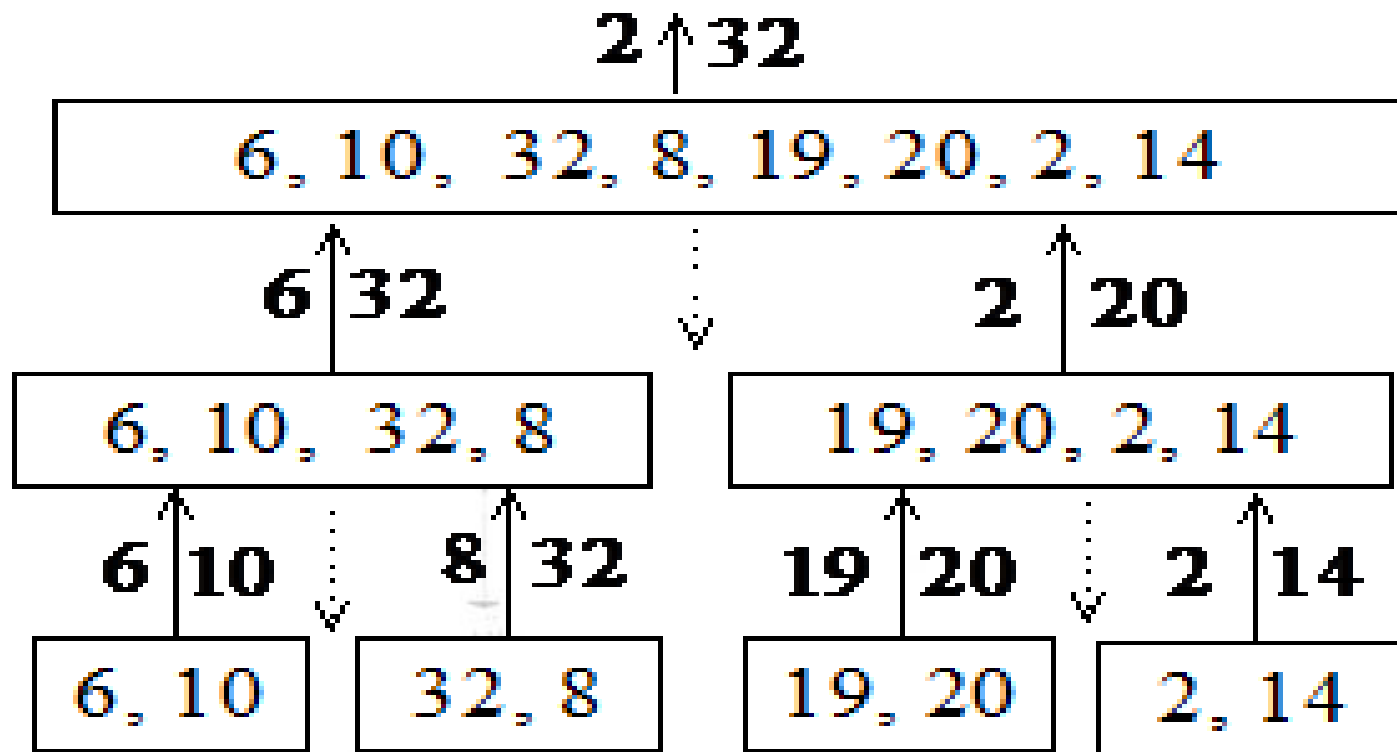
输出： A 中的 \max 和 \min

通常，直接扫描需要 $2n-2$ 次比较操作

试给出一个仅需 $\lceil 3n/2 \rceil - 2$ 次比较操作的算法。

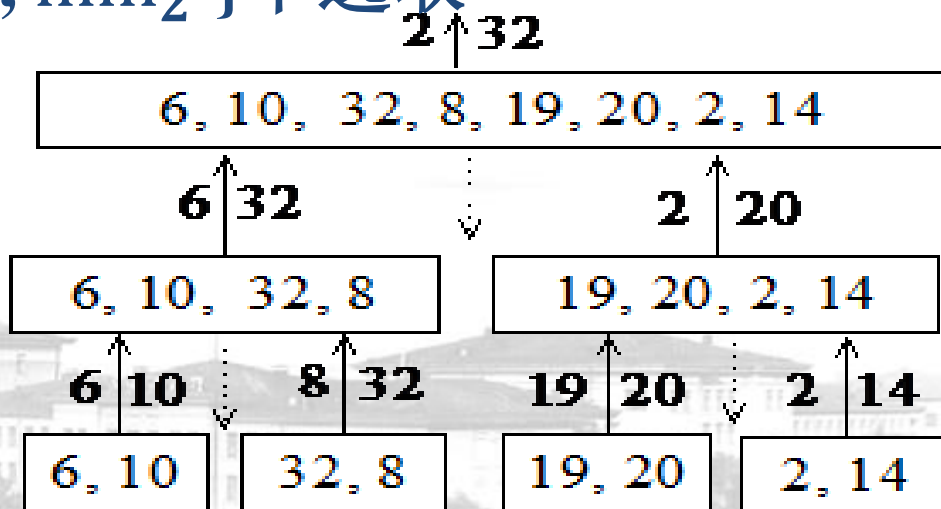


基本思想



算法大致步骤

- 将数组A从中间划分为两个子数组 A_1 和 A_2 ;
- 递归地在 A_1 中找到最大数 \max_1 和最小数 \min_1 ;
- 递归地在 A_2 中找到最大数 \max_2 和最小数 \min_2 ;
- 原问题的 \max 从 $\max\{\max_1, \max_2\}$ 中选取;
- 原问题的 \min 从 $\min\{\min_1, \min_2\}$ 中选取



算法

算法MaxMin(A)

输入: 数组 $A[i, \dots, j]$

输出: 数组 $A[i, \dots, j]$ 中的min和max

1. If $j-i+1 = 1$ Then 输出 $A[i], A[i]$, 算法结束
2. If $j-i+1 = 2$ Then
3. If $A[i] < A[j]$ Then 输出 $A[i], A[j]$; 算法结束
4. $k \leftarrow (j-i+1)/2$
5. $m_1, M_1 \leftarrow \text{MaxMin}(A[i:k]);$
6. $m_2, M_2 \leftarrow \text{MaxMin}(A[k+1:j]);$
7. $m \leftarrow \min(m_1, m_2);$
8. $M \leftarrow \max(M_1, M_2);$
9. 输出 m, M

算法复杂性

$$T(1)=0$$

$$T(2)=1$$

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

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

$$= \dots$$

$$=2^{k-1}T(2)+2^{k-1}+2^{k-2}+\dots+2^2+2$$

$$n=2^k$$

$$=2^{k-1}+2^{k-1}-1$$

$$=2^k+2^{k/2}-2$$

$$=3n/2-2$$



算法复杂性

观察 $T(n) = 3n/2 - 2$ ，证明在最坏条件下，同时找到 n 个元素的最大值和最小值的比较次数下界是 $\lceil 3n/2 \rceil - 2$.

提醒：考虑有多少个数有成为最大值或最小值的可能，然后分析每一次比较会如何影响这些计数

如果 n 是奇数，

$$\begin{aligned} 1 + \frac{3(n-3)}{2} + 2 &= \frac{3n}{2} - \frac{3}{2} \\ &= \left(\left\lceil \frac{3n}{2} \right\rceil - \frac{1}{2} \right) - \frac{3}{2} \\ &= \left\lceil \frac{3n}{2} \right\rceil - 2 \end{aligned}$$

如果 n 是偶数，

$$\begin{aligned} 1 + \frac{3(n-2)}{2} &= \frac{3n}{2} - 2 \\ &= \left\lceil \frac{3n}{2} \right\rceil - 2 \end{aligned}$$

算法复杂性

观察 $T(n) = 3n/2 - 2$ ，证明在最坏条件下，同时找到 n 个元素的最大值和最小值的比较次数下界是 $\lceil 3n/2 \rceil - 2$.

提醒：考虑有多少个数有成为最大值或最小值的可能，然后分析每一次比较会如何影响这些计数

如果 n 是奇数，

$$\begin{aligned} 1 + \frac{3(n-3)}{2} + 2 &= \frac{3n}{2} - \frac{3}{2} \\ &= \left(\left\lceil \frac{3n}{2} \right\rceil - \frac{1}{2} \right) - \frac{3}{2} \\ &= \left\lceil \frac{3n}{2} \right\rceil - 2 \end{aligned}$$

如果 n 是偶数，

$$\begin{aligned} 1 + \frac{3(n-2)}{2} &= \frac{3n}{2} - 2 \\ &= \left\lceil \frac{3n}{2} \right\rceil - 2 \end{aligned}$$

每个元素都参与了与最大值最小值的比较，比较了两次。将数组成对处理，两两相比，并将大的与最大值比较，小的与最小值比较。每对元素需要比较的次数是：两两比较一次，大的与最大值比较一次，小的与最小值比较一次，共三次。

一、求最大值和最小值

输入：数组 $A[1, \dots, n]$

输出： A 中的 \max 和 \min

通常，直接扫描需要 $2n-2$ 次比较操作

体现了分治的优势

给出了一个需 $\lceil 3n/2 \rceil - 2$ 次比较操作的算法。



二、大整数乘法

输入： n 位二进制整数 X 和 Y

输出： X 和 Y 的乘积



大整数乘法问题背景

- 加法和乘法运算被当做基本运算处理
 - 以进行运算的整数能在计算机硬件对整数的表示范围内可被直接处理为前提
- 计算机硬件无法直接处理很大的整数
 - 能否分而治之？
 - 解决大整数处理问题，并提高计算效率



大整数乘法-朴素方法

输入：整数 X 和 Y

输出： X 和 Y 的乘积

下面考你一道算法题：

给你两个很大很大的整数，比如
超过 100 位的整数，如何求出它
们的乘积？



大整数乘法-朴素方法

输入：整数 X 和 Y

输出： X 和 Y 的乘积

有了！我刚刚学过大整数相加的实现方法，我可以沿用这个思路，像小学生一样列出竖式求解……



$$\begin{array}{r} 9\ 3\ 2\ 8\ 1 \\ X 2\ 0\ 3\ 4 \\ \hline 3\ 7\ 3\ 1\ 2\ 4 \\ 2\ 7\ 9\ 8\ 4\ 3 \\ 0\ 0\ 0\ 0\ 0\ 0 \\ 1\ 8\ 6\ 5\ 6\ 2 \\ \hline 1\ 8\ 9\ 7\ 3\ 3\ 5\ 5\ 4 \end{array}$$



大整数乘法-朴素方法

输入：整数 X 和 Y

输出： X 和 Y 的乘积

OK，这样做确实可以实现功能，你说说它的时间复杂度是多少？



[illegible]

大整数乘法-朴素方法

输入：整数 X 和 Y

输出： X 和 Y 的乘积

由于两个大整数的所有数位都需要一一彼此相乘，如果整数 A 的长度为 m ，整数 B 的长度为 n ，那么时间复杂度就是 $O(m*n)$ 。



$$\begin{array}{r} 9\ 3\ 2\ 8\ 1 \\ X 2\ 0\ 3\ 4 \\ \hline 3\ 7\ 3\ 1\ 2\ 4 \\ 2\ 7\ 9\ 8\ 4\ 3 \\ 0\ 0\ 0\ 0\ 0\ 0 \\ 1\ 8\ 6\ 5\ 6\ 2 \\ \hline 1\ 8\ 9\ 7\ 3\ 3\ 5\ 5\ 4 \end{array}$$



大整数乘法-朴素方法

输入：整数 X 和 Y

输出： X 和 Y 的乘积

如果两个大整数的长度接近，那么
时间复杂度也可以写为 $O(n^2)$ 。



$$\begin{array}{r} 9\ 3\ 2\ 8\ 1 \\ X 2\ 0\ 3\ 4 \\ \hline 3\ 7\ 3\ 1\ 2\ 4 \\ 2\ 7\ 9\ 8\ 4\ 3 \\ 0\ 0\ 0\ 0\ 0\ 0 \\ 1\ 8\ 6\ 5\ 6\ 2 \\ \hline 1\ 8\ 9\ 7\ 3\ 3\ 5\ 5\ 4 \end{array}$$

大整数乘法-朴素方法

输入：整数 X 和 Y

输出： X 和 Y 的乘积

所以... 有没有优化方法，可以让
时间复杂度低于 $O(n^2)$ 呢？



大整数乘法-朴素方法

输入：整数 X 和 Y

输出： X 和 Y 的乘积

没有别的办法了吧.....

呵呵，没关系，回家等通知去吧！



大整数乘法-朴素方法

输入：整数 X 和 Y

输出： X 和 Y 的乘积

大黄，要实现大整数相乘，有没有时间复杂度低于 $O(n^2)$ 的方法呀？

还真有一种方法。对于大整数的相乘，我们可以使用「分治法」来简化问题的规模。



二、大整数乘法

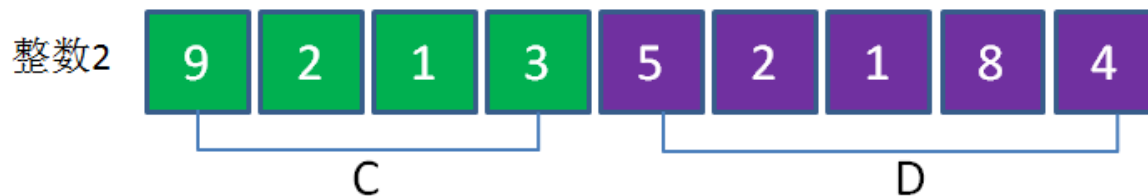
输入： n 位二进制整数 X 和 Y

输出： X 和 Y 的乘积

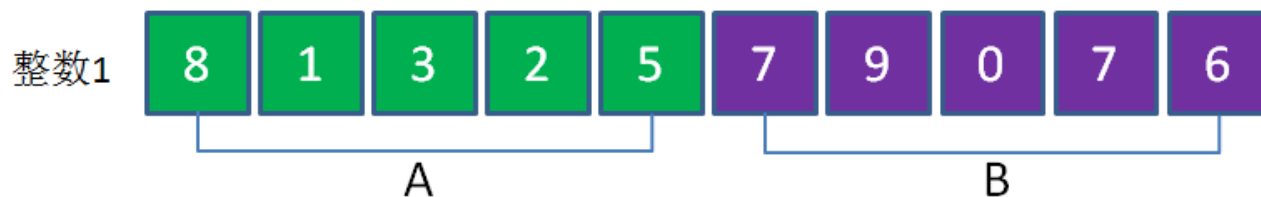
通常，计算 $X * Y$ 时间复杂性为 $O(n^2)$ ，
我们给出一个复杂性为 $O(n^{1.59})$ 的算法。



一个简单分治算法



一个简单分治算法



$$X = \overset{n/2\text{位}}{A} \overset{n/2\text{位}}{B} \quad Y = \overset{n/2\text{位}}{C} \overset{n/2\text{位}}{D}$$

$$\begin{aligned} XY &= (A2^{n/2} + B)(C2^{n/2} + D) \\ &= AC2^n + (AD+BC)2^{n/2} + BD \end{aligned}$$

一个简单分治算法

$$X = \overset{n/2\text{位}}{A} \overset{n/2\text{位}}{B} \quad Y = \overset{n/2\text{位}}{C} \overset{n/2\text{位}}{D}$$

$$\begin{aligned} XY &= (A2^{n/2} + B)(C2^{n/2} + D) \\ &= AC2^n + (AD+BC)2^{n/2} + BD \end{aligned}$$

算法

1. 划分产生A,B,C,D;
2. 计算n/2位乘法AC、AD、BC、BD;
3. 计算AD+BC;
4. AC左移n位, (AD+BC)左移n/2位;
5. 计算XY。

原本长度为n的大整数的1次乘积, 被转化成了长度为n/2的大整数的4次乘积 (AC, AD, BC, BD)

一个简单分治算法

$$X = \overset{n/2\text{位}}{A} \overset{n/2\text{位}}{B} \quad Y = \overset{n/2\text{位}}{C} \overset{n/2\text{位}}{D}$$

$$\begin{aligned} XY &= (A2^{n/2} + B)(C2^{n/2} + D) \\ &= AC2^n + (AD+BC)2^{n/2} + BD \end{aligned}$$

算法

1. 划分产生A,B,C,D;
2. 计算 $n/2$ 位乘法AC、AD、BC、BD;
3. 计算AD+BC;
4. AC左移 n 位, (AD+BC)左移 $n/2$ 位;
5. 计算XY。

时间复杂性

$$T(n) = 4T(n/2) + \theta(n)$$

$$T(n) = \theta(n^2)$$

一个简单分治算法

$$X = \overset{n/2\text{位}}{A} \overset{n/2\text{位}}{B} \quad Y = \overset{n/2\text{位}}{C} \overset{n/2\text{位}}{D}$$

$$\begin{aligned} XY &= (A2^{n/2} + B)(C2^{n/2} + D) \\ &= AC2^n + (AD+BC)2^{n/2} + BD \end{aligned}$$

哎我去，闹了半天，时间复杂度还是 $O(n^2)$ 啊！我白高兴一场……

是的，不过我们的努力方向并没有白费。只要在这个思路再动一动脑筋，是可以找到优化方法的。



时间复杂性

$$T(n) = 4T(n/2) + \theta(n)$$

$$T(n) = \theta(n^2)$$

一个简单分治算法

$$X = \overset{n/2\text{位}}{A} \overset{n/2\text{位}}{B} \quad Y = \overset{n/2\text{位}}{C} \overset{n/2\text{位}}{D}$$

$$\begin{aligned} XY &= (A2^{n/2} + B)(C2^{n/2} + D) \\ &= AC2^n + (AD+BC)2^{n/2} + BD \end{aligned}$$

我们之前使用分治法，在大方向上是没错的，但是两个大整数的相乘转化为四个较小整数的相乘，性能的瓶颈仍旧在乘法上面。



时间复杂性

$$T(n) = 4T(n/2) + \theta(n)$$

$$T(n) = \theta(n^2)$$

一个优化的分治算法

$$X = \overset{n/2\text{位}}{A} \overset{n/2\text{位}}{B} \quad Y = \overset{n/2\text{位}}{C} \overset{n/2\text{位}}{D}$$

$$\begin{aligned} XY &= (A2^{n/2} + B)(C2^{n/2} + D) \\ &= AC2^n + (AD+BC)2^{n/2} + BD \end{aligned}$$

思想：减少乘法的次数！

$$\begin{aligned} AD+BC &= (AD + AC + BD + BC - AC - BD) \\ &= (A+B)(C+D) - AC - BD \end{aligned}$$

$$XY = AC2^n + ((A+B)(C+D) - AC - BD) 2^{n/2} + BD$$

原来的4次乘法和3次加法转化成了3次乘法和6次加减法

一个优化的分治算法

$$X = \overset{n/2\text{位}}{A} \overset{n/2\text{位}}{B} \quad Y = \overset{n/2\text{位}}{C} \overset{n/2\text{位}}{D}$$

$$\begin{aligned} XY &= (A2^{n/2} + B)(C2^{n/2} + D) \\ &= AC2^n + (AD+BC)2^{n/2} + BD \end{aligned}$$

思想：减少乘法的次数！

$$\begin{aligned} AD+BC &= (AD + AC + BD + BC - AC - BD) \\ &= (A+B)(C+D) - AC - BD \end{aligned}$$

$$XY = AC2^n + ((A+B)(C+D) - AC - BD) 2^{n/2} + BD$$

你骗人，最后的式子里明明
包含 5 次乘法呀？



傻孩子，AC 出现了两次，BD 也出现了两次，这两个乘积分别只需要计算一次就行，所以总共只需要计算 3 次乘法呀！

一个优化的分治算法

$$X = \overset{n/2\text{位}}{A} \overset{n/2\text{位}}{B} \quad Y = \overset{n/2\text{位}}{C} \overset{n/2\text{位}}{D}$$

$$\begin{aligned} XY &= (A2^{n/2} + B)(C2^{n/2} + D) \\ &= AC2^n + (AD+BC)2^{n/2} + BD \end{aligned}$$

$$AD+BC = (A+B)(C+D) - AC - BD$$

1. 划分产生A,B,C,D;
2. 计算A+B和C+D;
3. 计算 $n/2$ 位乘法AC、BD、 $(A+B)(C+D)$;
4. 计算 $(A+B)(C+D) - AC - BD$;
5. AC左移 n 位, $(A+B)(C+D) - AC - BD$ 左移 $n/2$ 位;
6. 计算XY

此算法的分析

- 建立递归方程

$$T(n)=\theta(1) \quad \text{if } n=1$$

两个大整数被拆分为三个较小部分的乘积， $f(n)$ 是6次加法运算的规模= $O(n)$

$$T(n)=3T(n/2)+O(n) \quad \text{if } n>1$$

- 使用Master定理

$$a=3, b=2, O(n^{\log_2 3 - \varepsilon}) = O(n), \varepsilon = 1$$

$$T(n) = O(n^{\log 3}) = O(n^{1.59})$$

此算法的分析

- 建立递归方程

$$T(n) = \theta(1) \quad \text{if } n=1$$

$$T(n) = 3T(n/2) + O(n) \quad \text{if } n > 1$$

- 使用Master定理

$$T(n) = O(n^{\log 3}) = O(n^{1.59})$$

2 和 1.59 之间的差距看似不大，但是当整数长度非常大的时候，两种方法的性能将是天壤之别！



关于大整数乘法的延伸知识

- 半个世纪的猜测终被证明
 - 1971年以来，数学科学家猜测（假设未被证实），超级大整数相乘极限速度将是 $N\log(N)$ ，且无法被超越
 - 2019年后，澳大利亚新南威尔士大学(UNSW)的数学家设计算法逼近了这一理论极限

两个十亿位的整数相乘，若是采用常规算法，大约需要几个月才能算出它们的结果。但是应用该新算法，仅需**30秒**！

关于大整数乘法的延伸知识

— 阅读论文 《Integer multiplication in time $O(n \log n)$ 》

<https://hal.science/hal-02070778v2/file/nlogn.pdf>



算法理论的魅力持续激励着当今的计算机人！

Integer multiplication in time $O(n \log n)$

DAVID HARVEY AND JORIS VAN DER HOEVEN

ABSTRACT. We present an algorithm that computes the product of two n -bit integers in $O(n \log n)$ bit operations, thus confirming a conjecture of Schönhage and Strassen from 1971. Our complexity analysis takes place in the multitape Turing machine model, with integers encoded in the usual binary representation. Central to the new algorithm is a novel “Gaussian resampling” technique that enables us to reduce the integer multiplication problem to a collection of multidimensional discrete Fourier transforms over the complex numbers, whose dimensions are all powers of two. These transforms may then be evaluated rapidly by means of Nussbaumer’s fast polynomial transforms.

1. INTRODUCTION

Let $M(n)$ denote the time required to multiply two n -bit integers. We work in the multitape Turing model, in which the time complexity of an algorithm refers to the number of steps performed by a deterministic Turing machine with a fixed, finite number of linear tapes [35]. The main results of this paper also hold in the Boolean circuit model [41, Sec. 9.3], with essentially the same proofs.

For functions $f(n_1, \dots, n_k)$ and $g(n_1, \dots, n_k)$, we write $f(n) = O(g(n))$ to indicate that there exists a constant $C > 0$ such that $f(n) \leq Cg(n)$ for all tuples $n = (n_1, \dots, n_k)$ in the domain of f . Similarly, we write $f(n) = \Omega(g(n))$ to mean that $f(n) \geq Cg(n)$ for all n in the domain of f , and $f(n) = \Theta(g(n))$ to indicate that both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ hold. From Section 2 onwards we will always explicitly restrict the domain of f to ensure that $g(n) > 0$ throughout this domain. However, in this Introduction we will slightly abuse this notation: when writing for instance $f(n) = O(n \log n \log \log n)$, we tacitly assume that the domain of f has been restricted to $[n_0, \infty)$ for some sufficiently large threshold n_0 .

Schönhage and Strassen conjectured in 1971 that the true complexity of integer multiplication is given by $M(n) = \Theta(n \log n)$ [40], and in the same paper established their famous upper bound $M(n) = O(n \log n \log \log n)$. In 2007 their result was sharpened by Fürer to $M(n) = O(n \log n K^{\log^* n})$ [12, 13] for some unspecified constant $K > 1$, where $\log^* n$ denotes the iterated logarithm, i.e., $\log^* x := \min\{k \geq 0 : \log^k x \leq 1\}$. Prior to the present work, the record stood at $M(n) = O(n \log n 4^{\log^* n})$ [22].

The main result of this paper is a verification of the upper bound in Schönhage and Strassen’s conjecture, thus closing the remaining $4^{\log^* n}$ gap:

Theorem 1.1. *There is an integer multiplication algorithm achieving*

$$M(n) = O(n \log n).$$

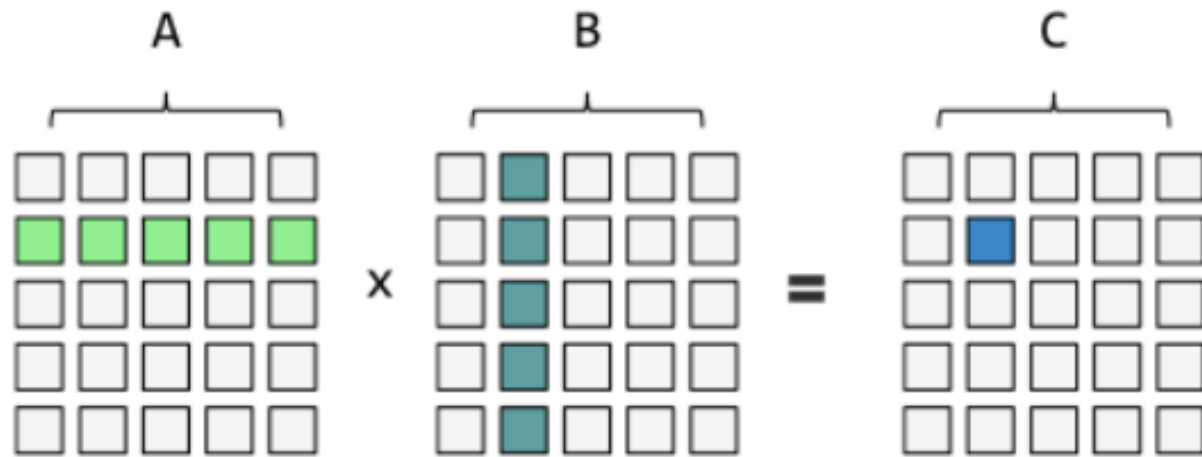
本讲内容

- 3.1 分治法
- 3.2 分治法的简单实例
- 3.3 矩阵乘法
- 3.4 元素选取问题的线性时间算法
- 3.5*快速傅里叶变换

三、矩阵乘法

输入：两个 $n \times n$ 矩阵 A 和 B

输出： A 和 B 的乘积



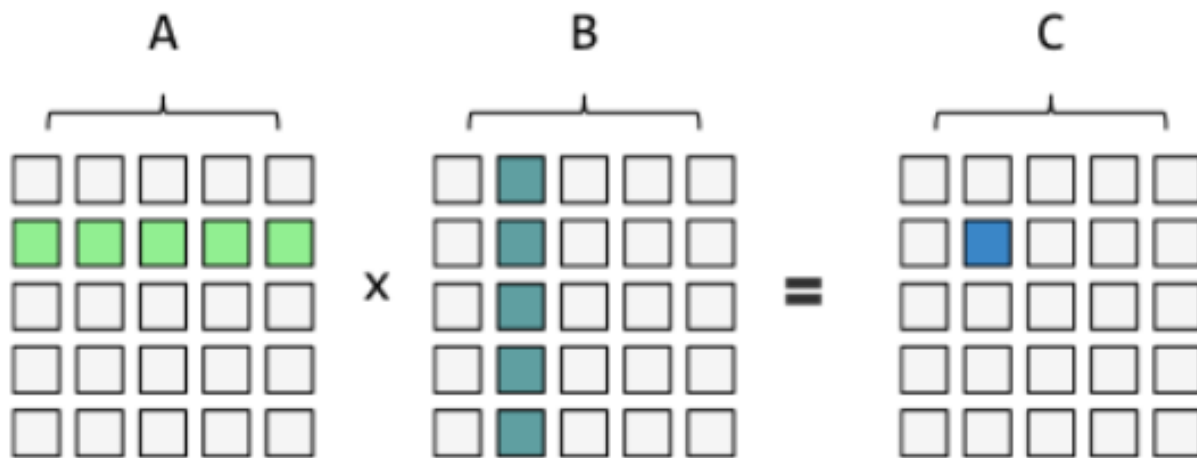
三、矩阵乘法

输入：两个 $n \times n$ 矩阵 A 和 B

输出： A 和 B 的乘积

$m \times p$ 的矩阵 A ， $p \times n$ 的矩阵 B ，完成 $C = AB$ 需要做的乘法数：

$m=p=n=N$ ，对于每个行向量 r ，共有 N 行；对于每个列向量 c ，共有 N 列，计算内积，共有 N 次乘法计算，即 $\Theta(N^3)$



三、矩阵乘法

输入：两个 $n \times n$ 矩阵 A 和 B

输出： A 和 B 的乘积

通常，计算 AB 的时间复杂度为 $O(n^3)$,

面试提问：能否给出一个复杂度为 $O(n^{2.81})$ 的算法？

所以... 有没有优化方法，可以让
时间复杂度低于 $O(n^3)$ 呢？



矩阵乘法初步算法设计

- 划分：把 $C=AB$ 中的每一个矩阵分为规模相同的4个子矩阵($n/2 \times n/2$)
- 则
$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$
- 其中 $C_{11} = A_{11} B_{11} + A_{12} B_{21}$
- $C_{12} = A_{11} B_{12} + A_{12} B_{22}$
- $C_{21} = A_{21} B_{11} + A_{22} B_{21}$
- $C_{22} = A_{21} B_{12} + A_{22} B_{22}$

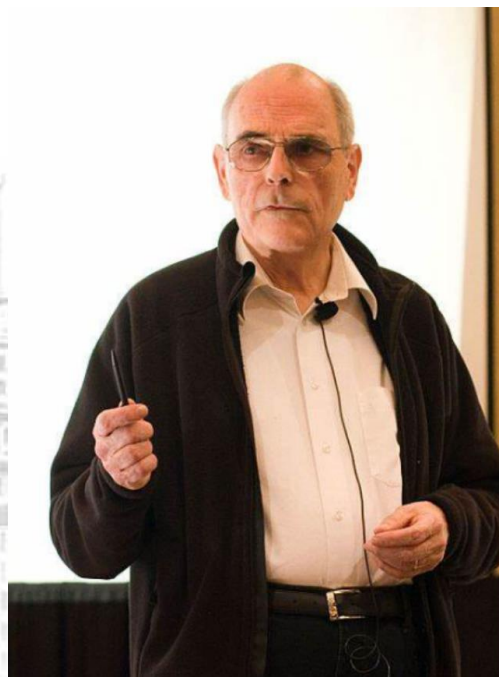


矩阵乘法初步算法设计

- 划分：把 $C=AB$ 中的每一个矩阵分为规模相同的4个子矩阵($n/2 \times n/2$)
- 则
$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$
- 其中 $C_{11} = A_{11} B_{11} + A_{12} B_{21}$
- $C_{12} = A_{11} B_{12} + A_{12} B_{22}$
- $C_{21} = A_{21} B_{11} + A_{22} B_{21}$
- $C_{22} = A_{21} B_{12} + A_{22} B_{22}$
- 需要求解8个($n/2 \times n/2$)子问题,
- 4次矩阵加法及合并矩阵C的时间: $O(n^2)$
- $T(n) = 8T(n/2) + O(n^2)$, $T(n) = O(n^3)$

矩阵乘法一个优化算法

- 思想：计算机对加法的处理效率大于乘法，应尽量降低做乘法的次数
- Strassen算法通过减少矩阵相乘的次数，实现算法复杂性的降低



矩阵乘法：Strassen算法

- 将输入矩阵A，B和输出矩阵C分解为规模为 $n/2 \times$

$n/2$ 子矩阵
$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

- 建立10个规模为 $n/2 \times n/2$ 的子矩阵

$$S_1 = B_{12} - B_{22},$$

$$S_2 = A_{11} + A_{12},$$

$$S_3 = A_{21} + A_{22},$$

$$S_4 = B_{21} - B_{11},$$

$$S_5 = A_{11} + A_{22},$$

$$S_6 = B_{11} + B_{22},$$

$$S_7 = A_{12} - A_{22},$$

$$S_8 = B_{21} + B_{22},$$

$$S_9 = A_{11} - A_{21},$$

$$S_{10} = B_{11} + B_{22}.$$

矩阵乘法：Strassen算法

- 递归地计算 $n/2 \times n/2$ 矩阵的乘法：

$$P_1 = A_{11} \cdot S_1 = A_{11} \cdot B_{12} - A_{11} \cdot B_{22}$$

$$P_2 = S_2 \cdot B_{22} = A_{11} \cdot B_{22} + A_{12} \cdot B_{22}$$

$$P_3 = S_3 \cdot B_{11} = A_{21} \cdot B_{11} + A_{22} \cdot B_{11}$$

$$P_4 = A_{22} \cdot S_4 = A_{22} \cdot B_{21} - A_{22} \cdot B_{11}$$

$$P_5 = S_5 \cdot S_6 = A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22}$$

$$P_6 = S_7 \cdot S_8 = A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22}$$

$$P_7 = S_9 \cdot S_{10} = A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12}$$



矩阵乘法：Strassen算法

- 递归地计算 $n/2 \times n/2$ 矩阵的乘法：

$$P_1 = A_{11} \cdot S_1 = A_{11} \cdot B_{12} - A_{11} \cdot B_{22}$$

$$P_2 = S_2 \cdot B_{22} = A_{11} \cdot B_{22} + A_{12} \cdot B_{22}$$

$$P_3 = S_3 \cdot B_{11} = A_{21} \cdot B_{11} + A_{22} \cdot B_{11}$$

$$P_4 = A_{22} \cdot S_4 = A_{22} \cdot B_{21} - A_{22} \cdot B_{11}$$

$$P_5 = S_5 \cdot S_6 = A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22}$$

$$P_6 = S_7 \cdot S_8 = A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22}$$

$$P_7 = S_9 \cdot S_{10} = A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12}$$

只有中间一列需要计算，一共计算7次矩阵积： P_1 至 P_7

矩阵乘法：Strassen算法

- 通过 P_i 计算 C_{11} , C_{12} , C_{21} , C_{22}

$$C_{11} = P_5 + P_4 - P_2 + P_6$$

$$C_{12} = P_1 + P_2$$

$$C_{21} = P_3 + P_4$$

$$C_{22} = P_5 + P_1 - P_3 - P_7$$



Strassen算法复杂性分析

- 分解子矩阵: $\Theta(1)$
- 创建10个规模为 $n/2 \times n/2$ 的子矩阵: $\Theta(n^2)$
- 递归计算7个子矩阵矩阵乘积: $7T(n/2)$
- 进行加减运算, 计算出 C 的子矩阵 C_{11} , C_{12} , C_{21} , C_{22} : $\Theta(n^2)$

$$\begin{aligned} T(n) &= \Theta(1) & \text{if } n=1 \\ &= 7T(n/2) + \Theta(n^2) & \text{if } n>1 \end{aligned}$$

$$T(n) = \Theta(n^{\lg 7}) = \Theta(n^{2.81})$$

Strassen算法伪代码（课后熟练）

STRASSEN(A, B)

n = A.rows

if n == 1

return a[1, 1] * b[1, 1]

let C be a new $n \times n$ matrix

A[1, 1] = A[1..n / 2][1..n / 2]

A[1, 2] = A[1..n / 2][n / 2 + 1..n]

A[2, 1] = A[n / 2 + 1..n][1..n / 2]

A[2, 2] = A[n / 2 + 1..n][n / 2 + 1..n]

B[1, 1] = B[1..n / 2][1..n / 2]

B[1, 2] = B[1..n / 2][n / 2 + 1..n]

B[2, 1] = B[n / 2 + 1..n][1..n / 2]

B[2, 2] = B[n / 2 + 1..n][n / 2 + 1..n]

Strassen算法伪代码续（课后熟练）

STRASSEN(A, B)

...

$$S[1] = B[1, 2] - B[2, 2]$$

$$S[2] = A[1, 1] + A[1, 2]$$

$$S[3] = A[2, 1] + A[2, 2]$$

$$S[4] = B[2, 1] - B[1, 1]$$

$$S[5] = A[1, 1] + A[2, 2]$$

$$S[6] = B[1, 1] + B[2, 2]$$

$$S[7] = A[1, 2] - A[2, 2]$$

$$S[8] = B[2, 1] + B[2, 2]$$

$$S[9] = A[1, 1] - A[2, 1]$$

$$S[10] = B[1, 1] + B[1, 2]$$

Strassen算法伪代码续（课后熟练）

STRASSEN(A, B)

...

P[1] = STRASSEN(A[1, 1], S[1])

P[2] = STRASSEN(S[2], B[2, 2])

P[3] = STRASSEN(S[3], B[1, 1])

P[4] = STRASSEN(A[2, 2], S[4])

P[5] = STRASSEN(S[5], S[6])

P[6] = STRASSEN(S[7], S[8])

P[7] = STRASSEN(S[9], S[10])

C[1..n / 2][1..n / 2] = P[5] + P[4] - P[2] + P[6]

C[1..n / 2][n / 2 + 1..n] = P[1] + P[2]

C[n / 2 + 1..n][1..n / 2] = P[3] + P[4]

C[n / 2 + 1..n][n / 2 + 1..n] = P[5] + P[1] - P[3] - P[7]

return C