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 元素选取问题的线性时间算法
3.4 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》

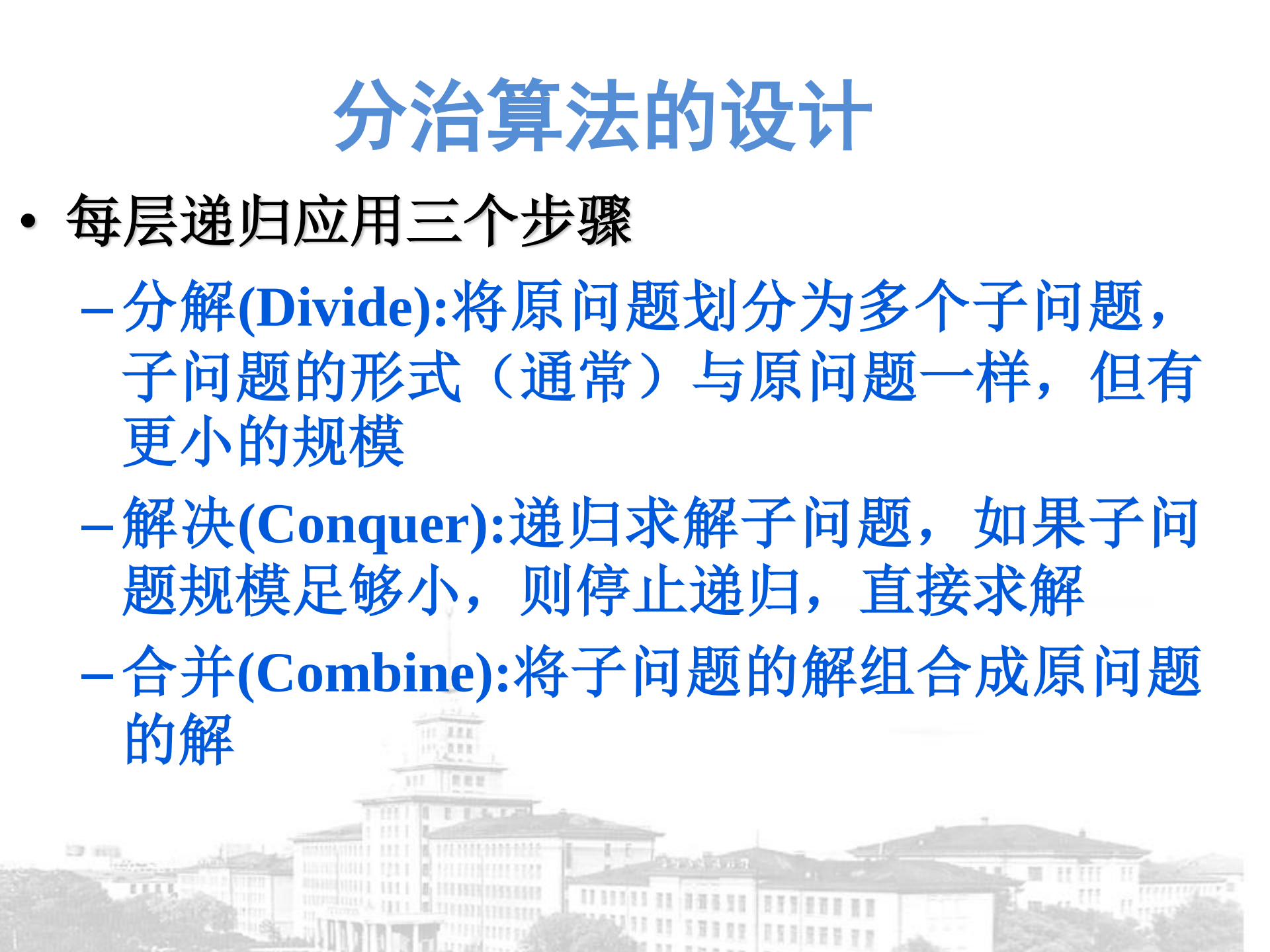
什么问题可以用“分治”解决呢？

– 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

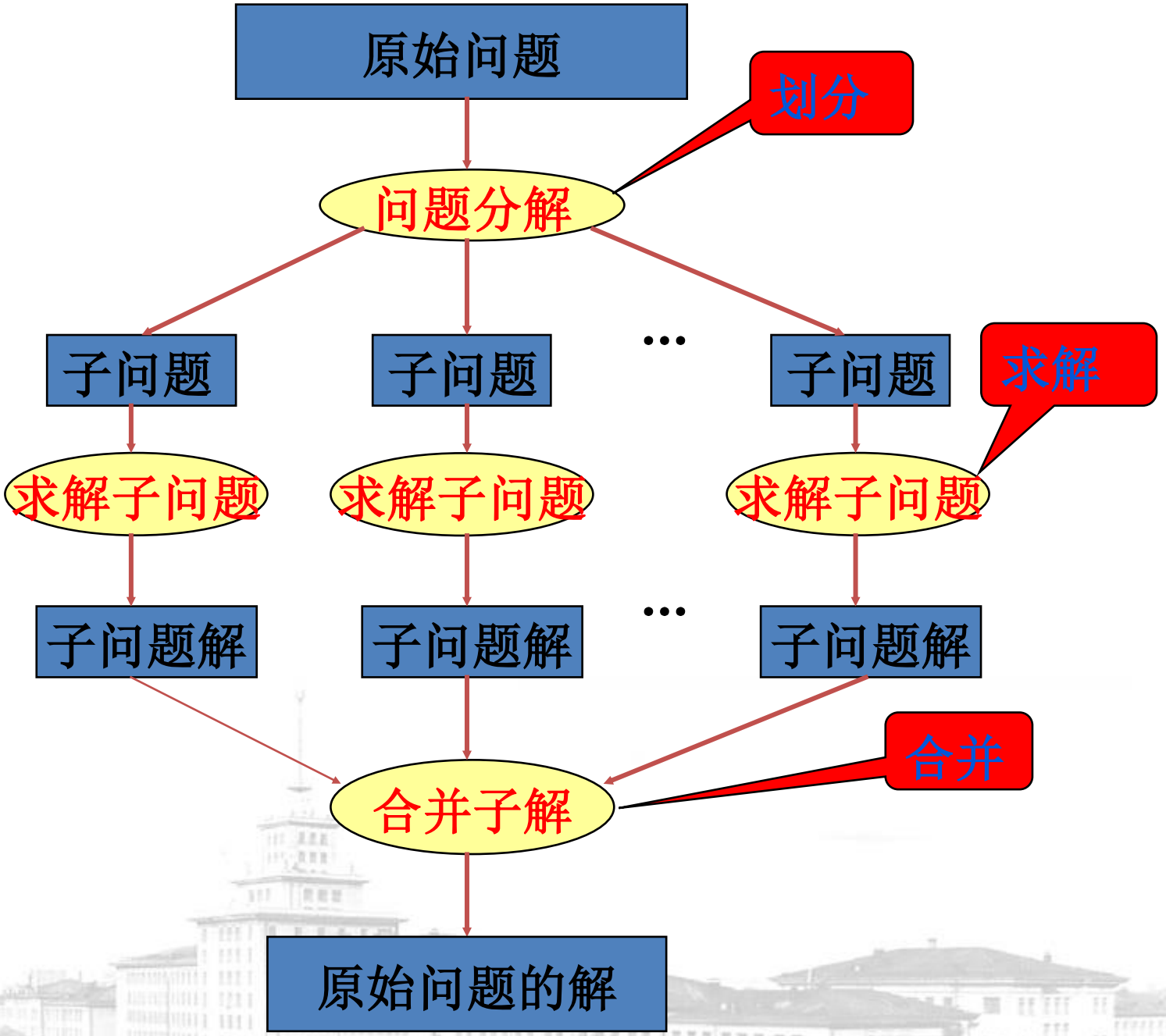
- 每层递归应用三个步骤
  - 分解(Divide):将原问题划分为多个子问题，子问题的形式（通常）与原问题一样，但有更小的规模
  - 解决(Conquer):递归求解子问题，如果子问题规模足够小，则停止递归，直接求解
  - 合并(Combine):将子问题的解组合成原问题的解

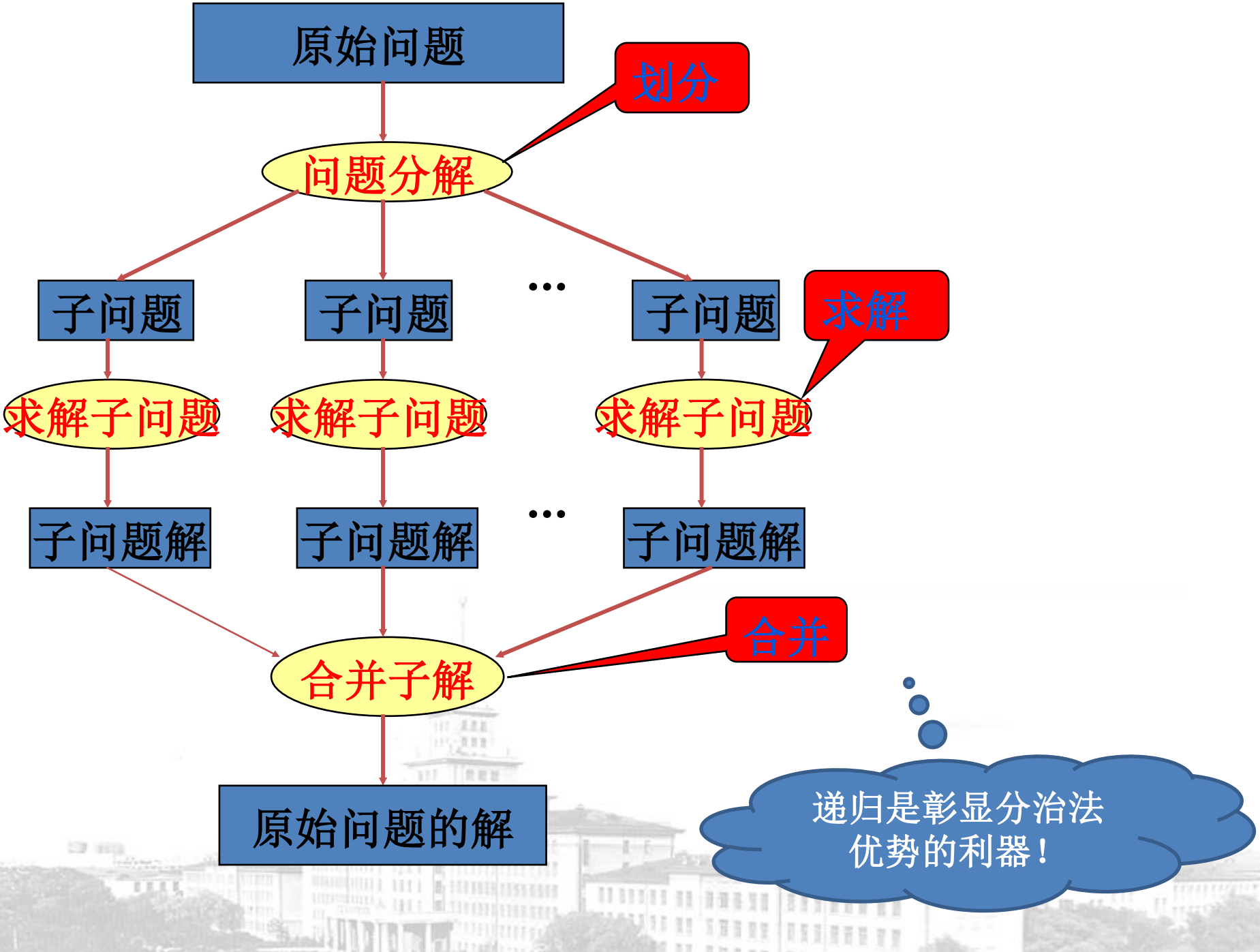


分治算法的设计

- 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


原始问题
划分
问题分解
子问题
子问题
…
子问题
求解
求解子问题
求解子问题
求解子问题
子问题解
子问题解
…
子问题解
合并
合并子解
原始问题的解


原始问题
划分
问题分解
子问题
子问题
…
子问题
求解
求解子问题
求解子问题
求解子问题
子问题解
子问题解
…
子问题解
合并
合并子解
原始问题的解
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 元素选取问题的线性时间算法

3.4 快速傅里叶变换

一、同时求最大值和最小值

输入：数组$A[1,\ldots,n]$
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$A[1,\ldots,n]$
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

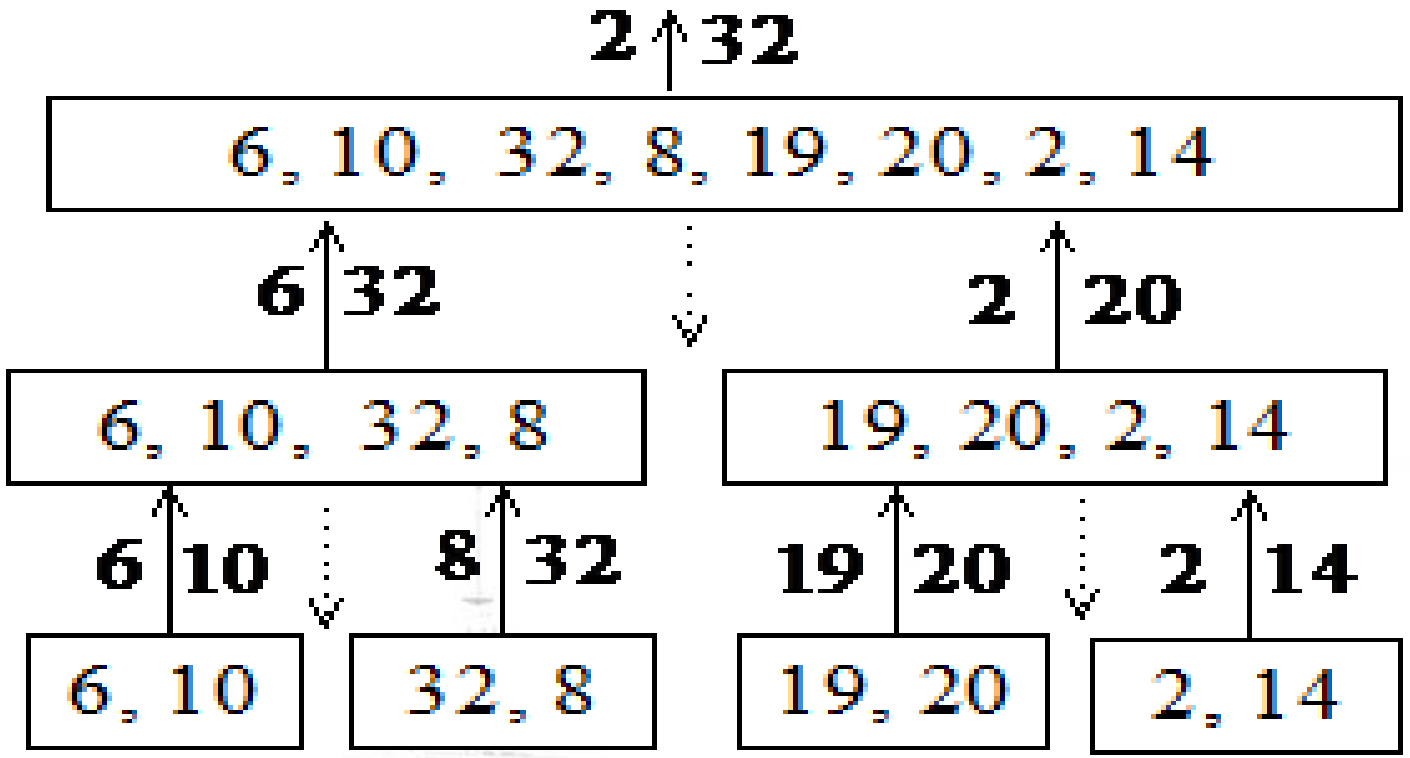
一、求最大值和最小值

输入：数组$A[1,\ldots,n]$
输出：A中的max和min

通常，直接扫描需要$2n-2$次比较操作

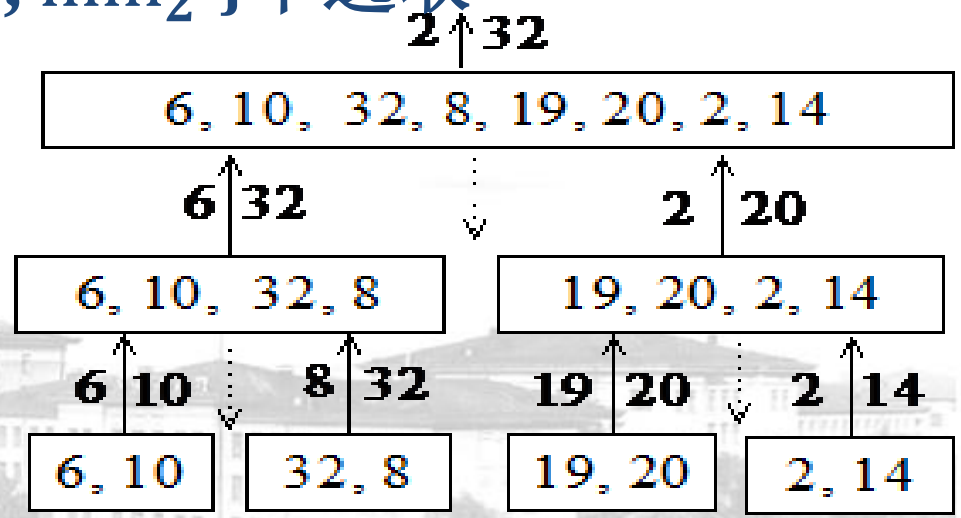
试给出一个仅需$\lceil 3n/2-2 \rceil$次比较操作的算法。

基本思想


2 32
6, 10, 32, 8, 19, 20, 2, 14
6 32
2 20
6, 10, 32, 8
19, 20, 2, 14
6 10
8 32
19 20
2 14
6, 10
32, 8
19, 20
2, 14

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$A[i,\ldots,j]$
输出:数组$A[i,\ldots,j]$中的max和min

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. $m_1,M_1 \leftarrow$MaxMin($A[i:k]$);
6. $m_2,M_2 \leftarrow$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

$T(1)=0$

$T(2)=1$

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

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

$= \ldots$

$=2^{k-1}T(2)+2^{k-1}+2^{k-2}+\ldots+2^2+2$ $\quad n=2^k$

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

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

$=3n/2-2$

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，

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

一、求最大值和最小值

输入：数组$A[1,\ldots,n]$
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

给出了一个仅需$\lceil 3n/2 \rceil - 2$ 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……

```
          9 3 2 8 1
X           2 0 3 4
-----------------------
        3 7 3 1 2 4
      2 7 9 8 4 3
    0 0 0 0 0 0
  1 8 6 5 6 2
-----------------------
  1 8 9 7 3 3 5 5 4
```

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，这样做法确实可以实现功能，
你说说它的时间复杂度是多少？

```
        9 3 2 8 1
X           2 0 3 4
-----------------------
      3 7 3 1 2 4
    2 7 9 8 4 3
  0 0 0 0 0 0
1 8 6 5 6 2
-----------------------
1 8 9 7 3 3 5 5 4
```

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）。

```
          9 3 2 8 1
X           2 0 3 4
-----------------------
        3 7 3 1 2 4
      2 7 9 8 4 3
    0 0 0 0 0 0
  1 8 6 5 6 2
-----------------------
  1 8 9 7 3 3 5 5 4
```

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)。

```
          9 3 2 8 1
X           2 0 3 4
-----------------------
        3 7 3 1 2 4
      2 7 9 8 4 3
    0 0 0 0 0 0
  1 8 6 5 6 2
-----------------------
  1 8 9 7 3 3 5 5 4
```

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

大整数乘法-朴素方法

输入：整数X和Y

输出：X和Y的乘积

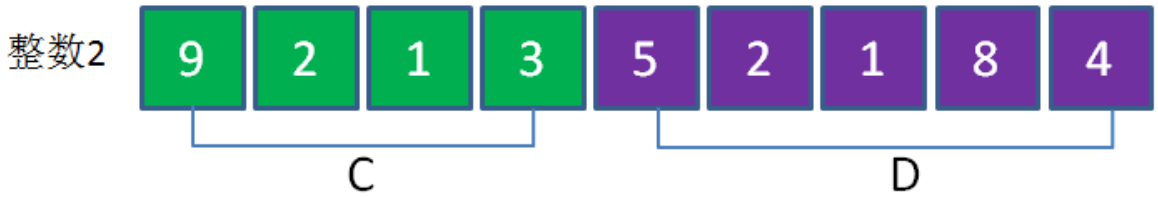
二、大整数乘法

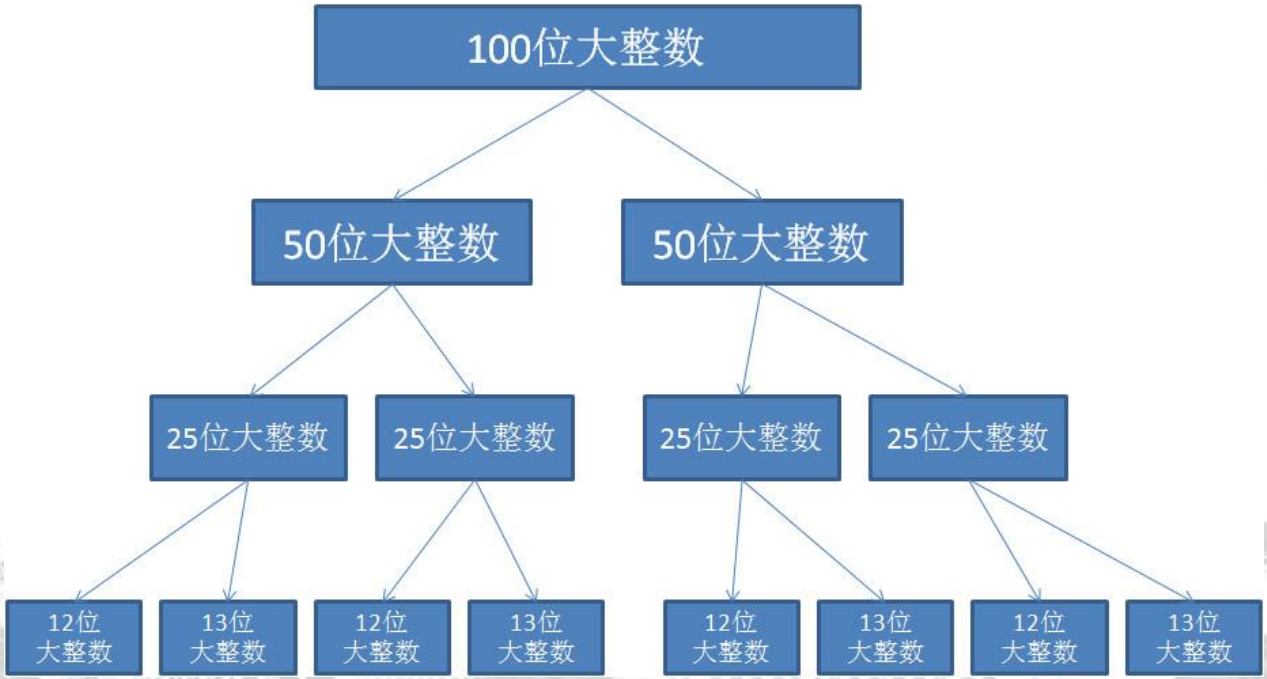
输入：n位二进制整数X和Y
输出：X和Y的乘积

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。

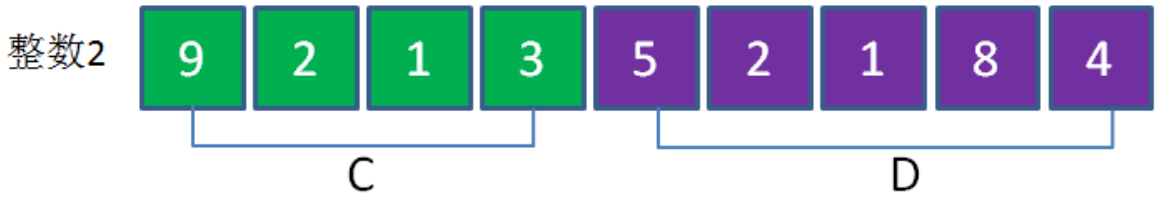
一个简单分治算法


整数1
8
1
3
2
5
7
9
0
7
6
A
B


整数2
9
2
1
3
5
2
1
8
4
C
D


100位大整数
50位大整数
50位大整数
25位大整数
25位大整数
25位大整数
25位大整数
12位
大整数
13位
大整数
12位
大整数
13位
大整数
12位
大整数
13位
大整数
12位
大整数
13位
大整数

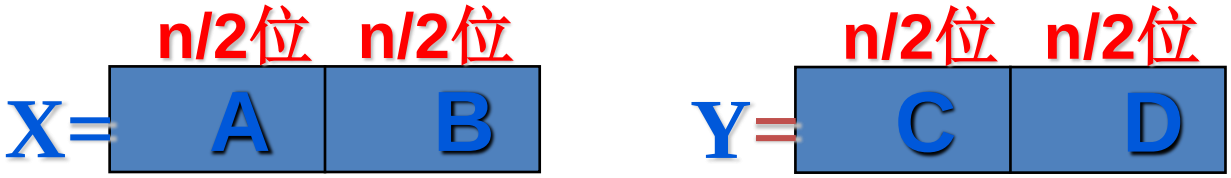
一个简单分治算法



$$XY = (A2^{n/2} + B)(C2^{n/2} + D)$$

$$= AC2^{n} + (AD+BC)2^{n/2} + BD$$

一个简单分治算法



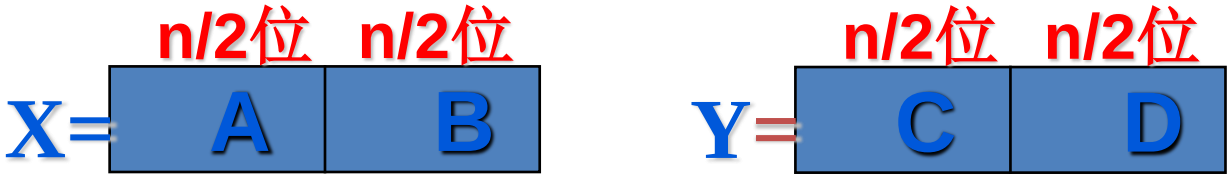
$$XY = (A2^{n/2} + B)(C2^{n/2} + D)$$
$$= AC2^{n} + (AD+BC)2^{n/2} + BD$$

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



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

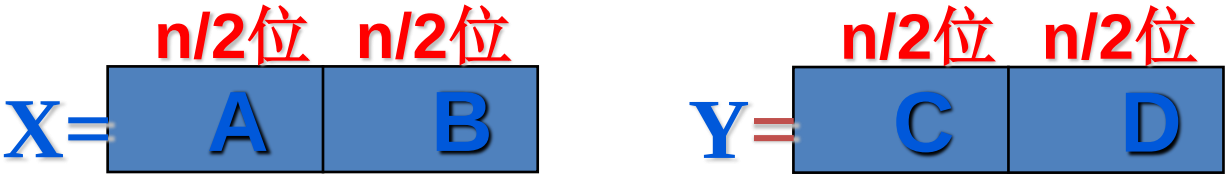
算法

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



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

哎我去，闹了半天，时间复杂度还是O(n^2)啊！我白高兴一场.....

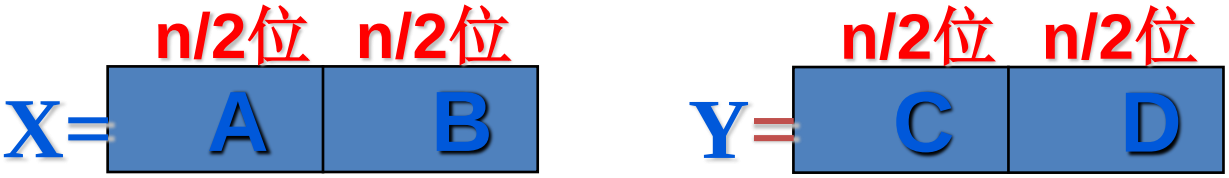
是的，不过我们的努力方向并没有白费。只要在这个思路上再动一动脑筋，是可以找到优化方法的。

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



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

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

$$XY = (A2^{n/2} + B)(C2^{n/2} + D)$$

$$= AC2^{n} + (AD+BC)2^{n/2} + BD$$

思想：减少乘法的次数！

$$AD+BC=(AD + AC+BD+BC-AC-BD)$$

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

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

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

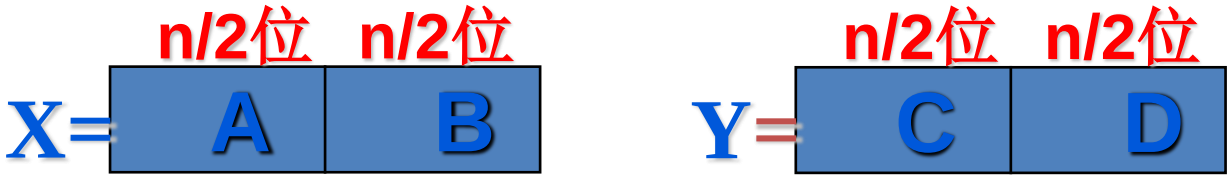
一个优化的分治算法

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

思想：减少乘法的次数！

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

一个优化的分治算法



$$XY = (A2^{n/2} + B)(C2^{n/2} + D)$$

$$= AC2^{n} + (AD+BC)2^{n/2} + BD$$

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

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

$T(n)=\theta(1)$ 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)

$T(n)=3T(n/2)+O(n)$ 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

$T(n)=\theta(1)$ if $n=1$

$T(n)=3T(n/2)+O(n)$ 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!

大整数乘法算法的思考

- X和Y必须是2^n位
- （如果不是，则在分解过程中出现奇数，乘法中的次幂有所不同，无法变成3次乘法。解决：补齐位数，即在数前（高位）补0）

大整数乘法算法的思考

- 试着编程实现大整数乘法的简单和优化算法
  - 提醒：大整数加法函数
  - 提醒：在用数组存放大数时，倒序存储！

（乘法加法运算可能产生进位，倒序存储可以让进位存储在数组的末尾）

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(nlogn)** 》

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 $\mathsf{M}(n)$ 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 $f(n_1, \ldots, n_k)$ and $g(n_1, \ldots, n_k)$, we write $f(n) = O(g(n))$ to indicate that there exists a constant $C > 0$ such that $f(n) \leqslant Cg(n)$ for all tuples $n = (n_1, \ldots, n_k)$ in the domain of f. Similarly, we write $f(n) = \Omega(g(n))$ to mean that $f(n) \geqslant Cg(n)$ 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 $\mathsf{M}(n) = \Theta(n \log n)$ [40], and in the same paper established their famous upper bound $\mathsf{M}(n) = O(n \log n \log\log n)$. In 2007 their result was sharpened by Fürer to $\mathsf{M}(n) = O(n \log n\, K^{\log^* n})$ [12, 13] for some unspecified constant $K > 1$, where $\log^* n$ denotes the iterated logarithm, i.e., $\log^* x := \min\{k \geqslant 0 : \log^{\circ k} x \leqslant 1\}$. Prior to the present work, the record stood at $\mathsf{M}(n) = O(n \log n\, 4^{\log^* n})$ [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*

$$\mathsf{M}(n) = O(n \log n).$$

本讲内容

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