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》

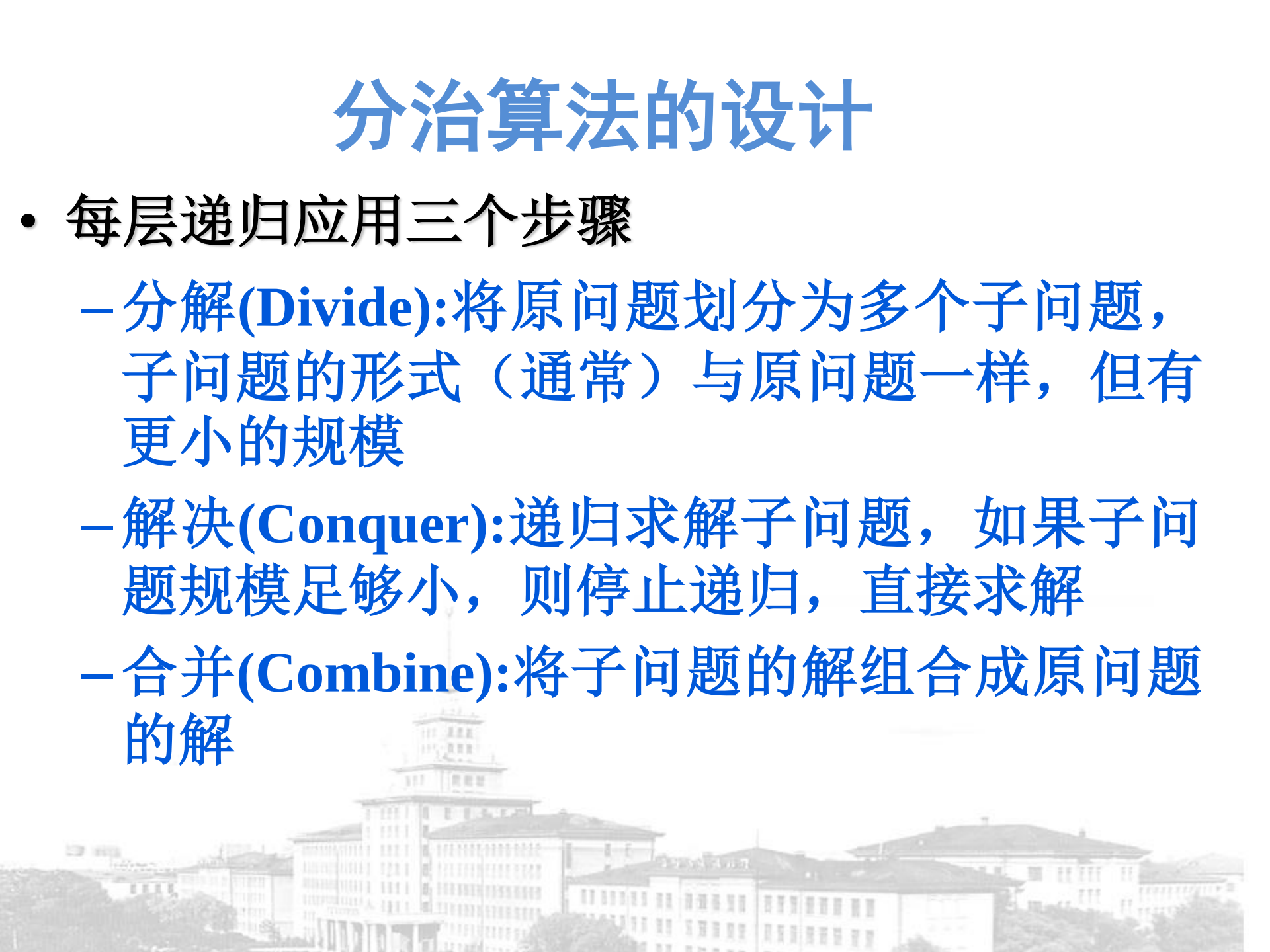
什么问题可以用“分治”解决呢？

– 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

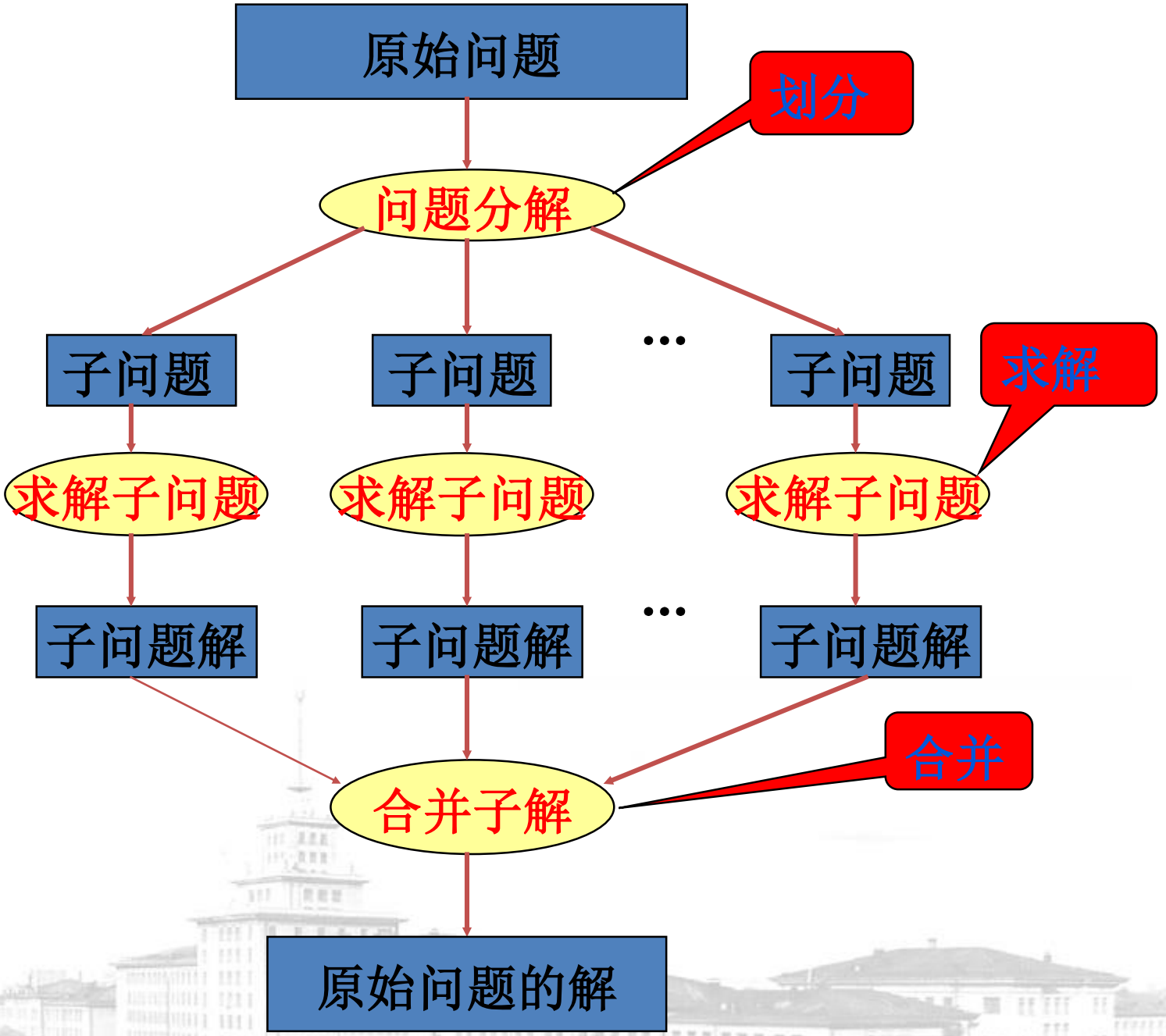
- 每层递归应用三个步骤
  - 分解(Divide):将原问题划分为多个子问题，子问题的形式（通常）与原问题一样，但有更小的规模
  - 解决(Conquer):递归求解子问题，如果子问题规模足够小，则停止递归，直接求解
  - 合并(Combine):将子问题的解组合成原问题的解

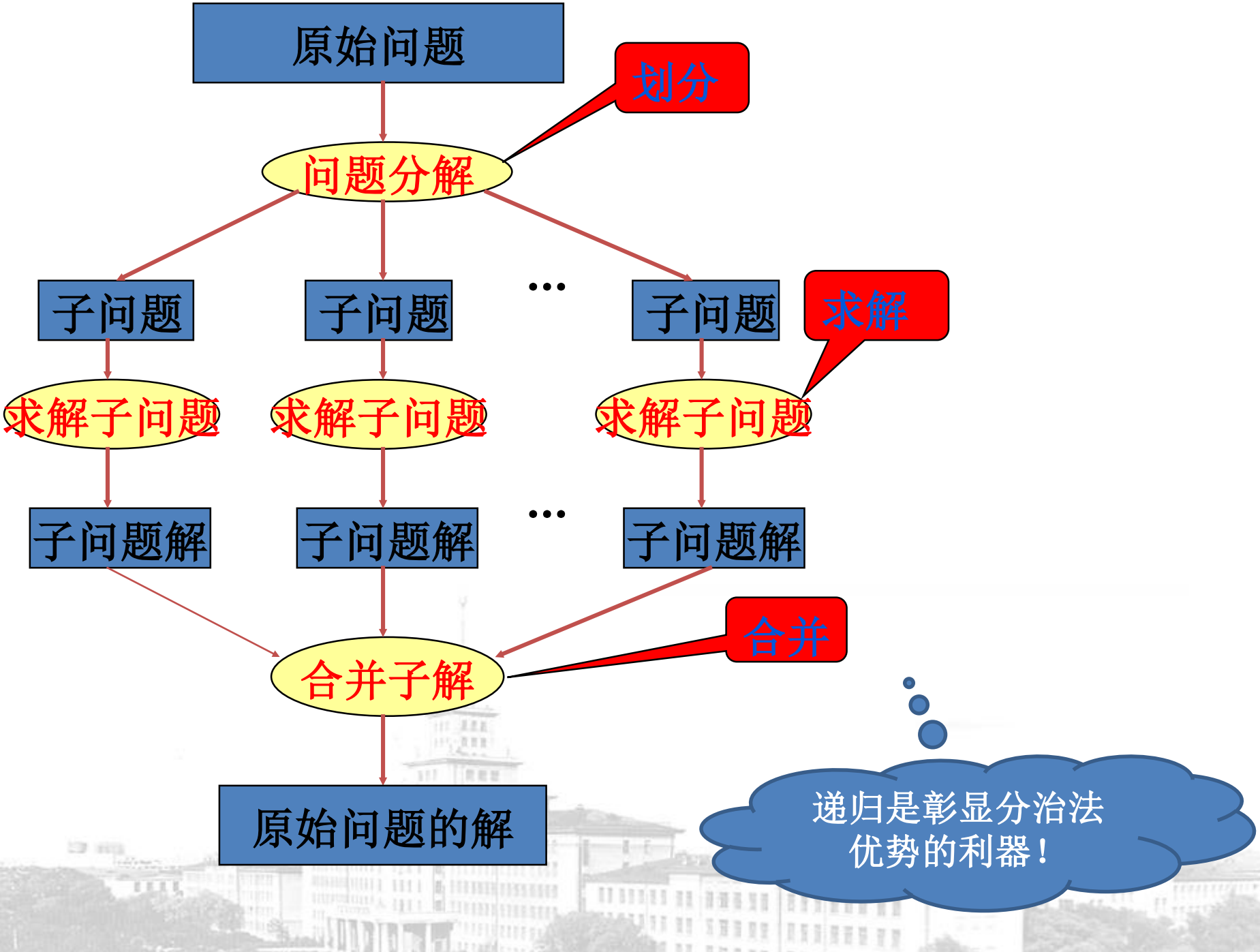


分治算法的设计

- 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 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$A[1,\ldots,n]$
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

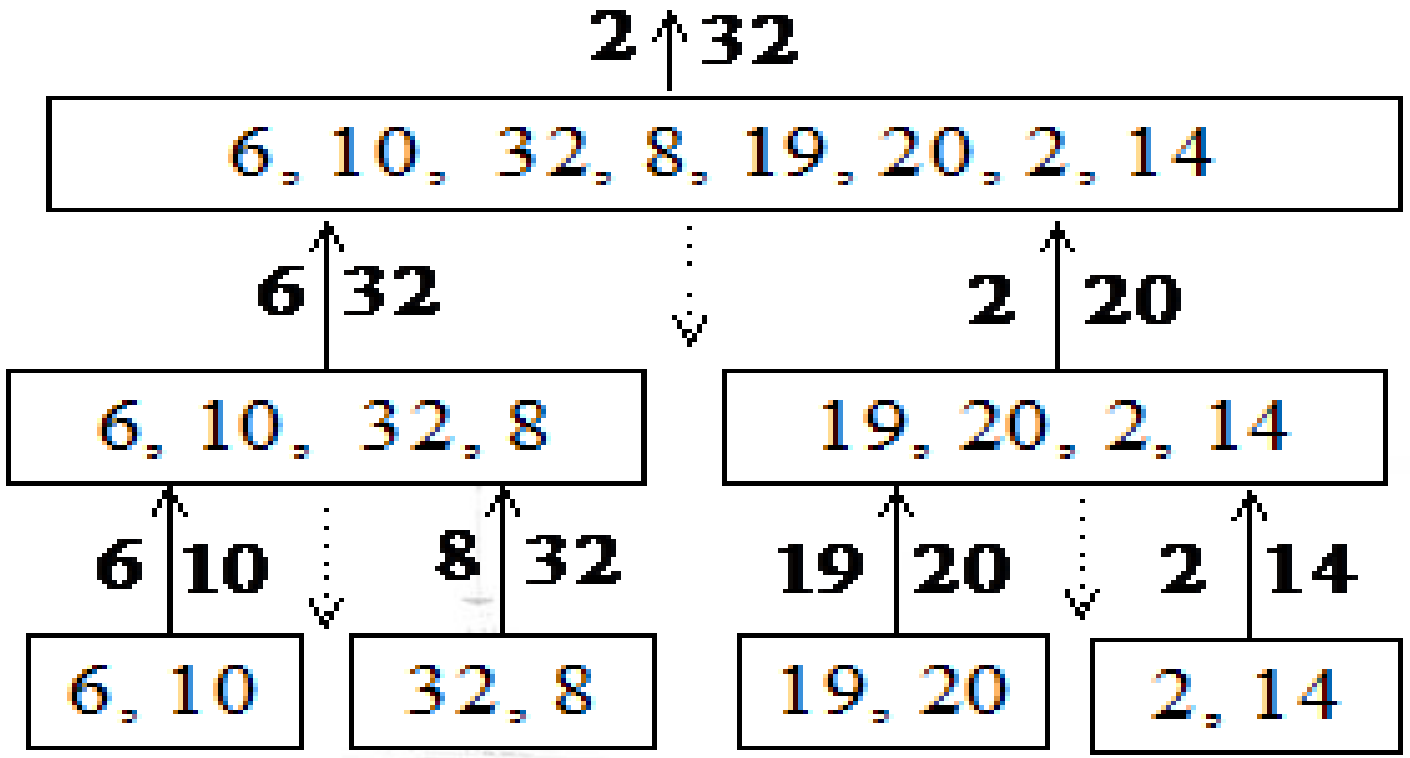
一、求最大值和最小值

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

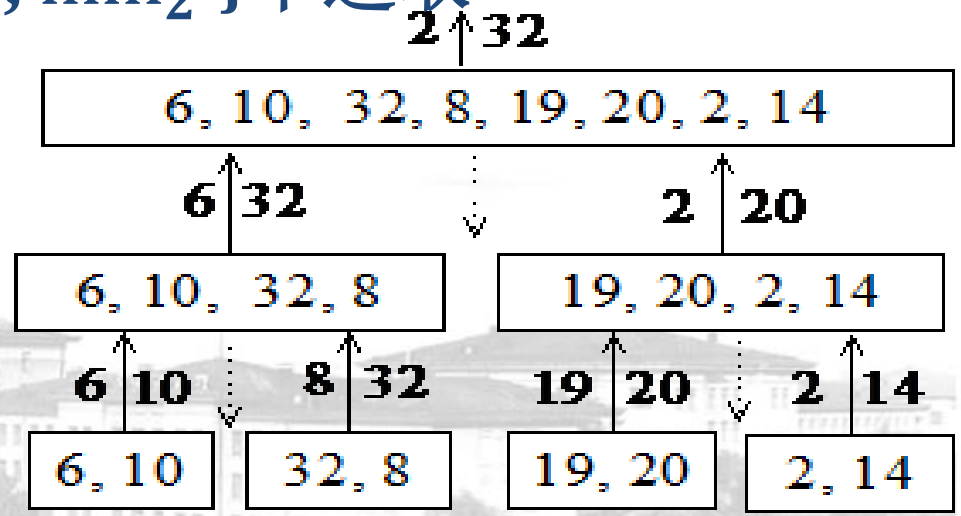
试给出一个仅需$\lceil 3n/2 \rceil -2$次比较操作的算法。

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. $m_1,M_1 \leftarrow$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，

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

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$A[1,\ldots,n]$
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

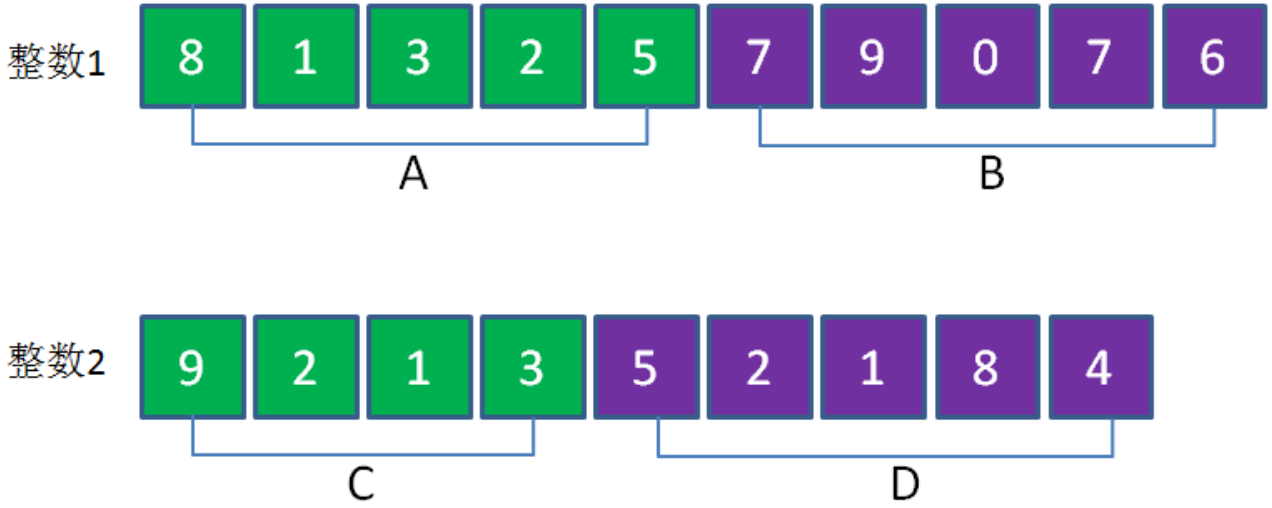
输出：X和Y的乘积

二、大整数乘法

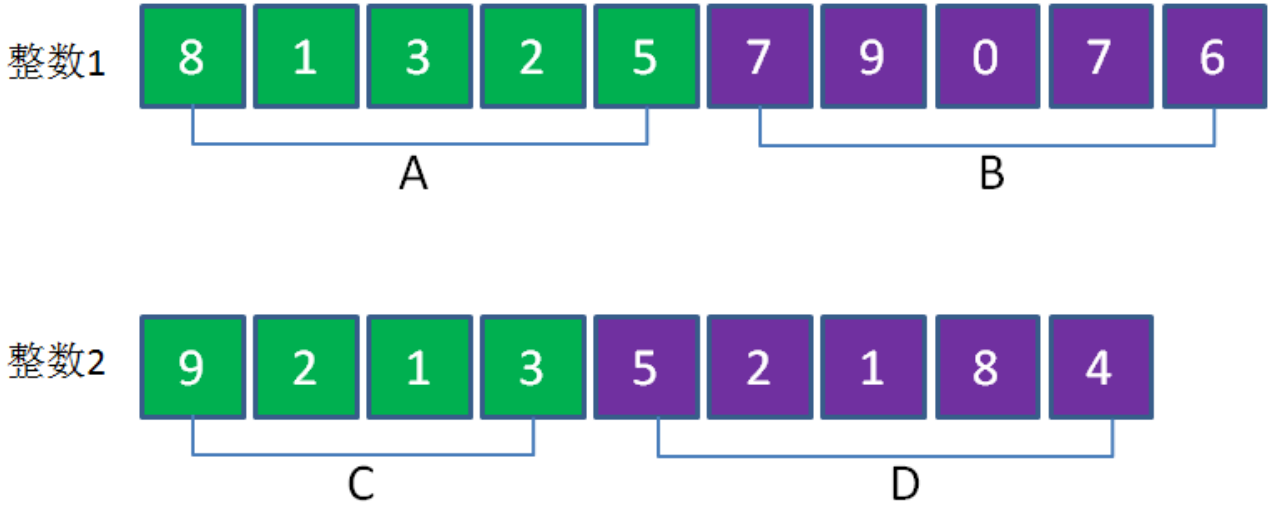
输入：n位二进制整数X和Y
输出：X和Y的乘积

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



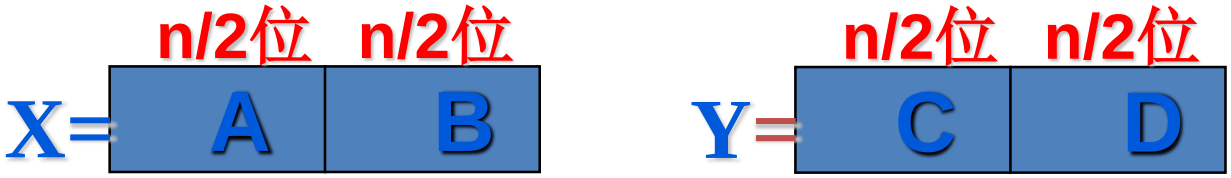
一个简单分治算法



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

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

一个简单分治算法



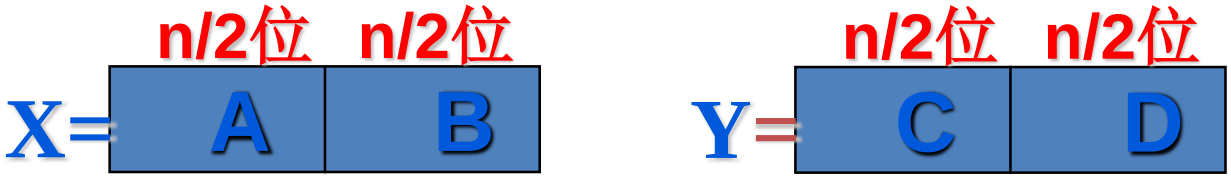
$$XY = (A2^{n/2} + B)(C2^{n/2} + D)$$
$$= AC2^{n} + (AD+BC)2^{n/2} + BD$$

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



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

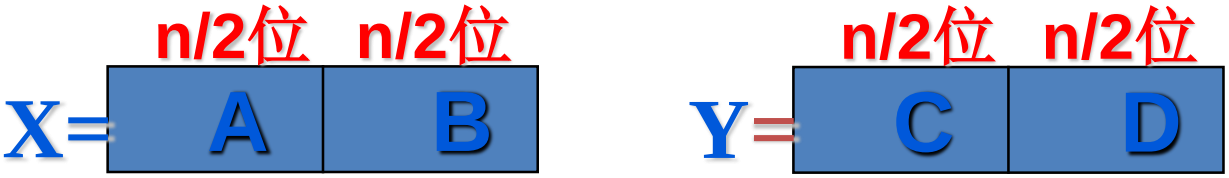
算法

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.....

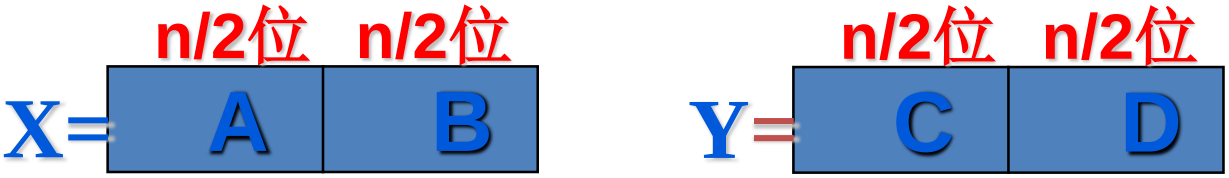
是的，不过我们的努力方向并没有白费。只要在这个思路上再动一动脑筋，是可以找到优化方法的。

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



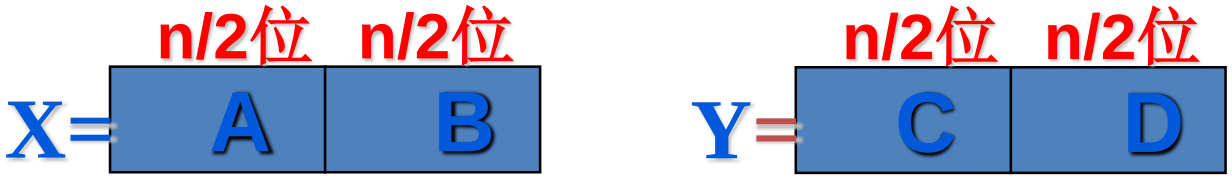
$$XY = (A2^{n/2} + B)(C2^{n/2} + D)$$
$$= AC2^{n} + (AD+BC)2^{n/2} + BD$$

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！

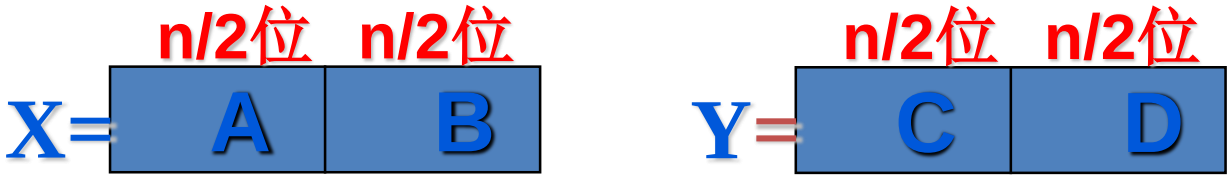
$$AD+BC=(AD + AC+BD+BC-AC-BD)$$

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

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

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



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

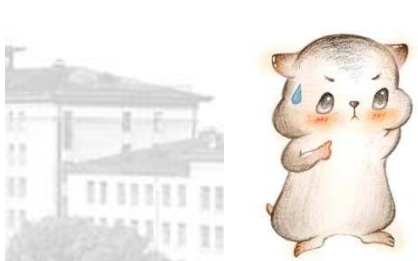
$$= AC2^{n} + (AD+BC)2^{n/2} + BD$$

思想：减少乘法的次数！

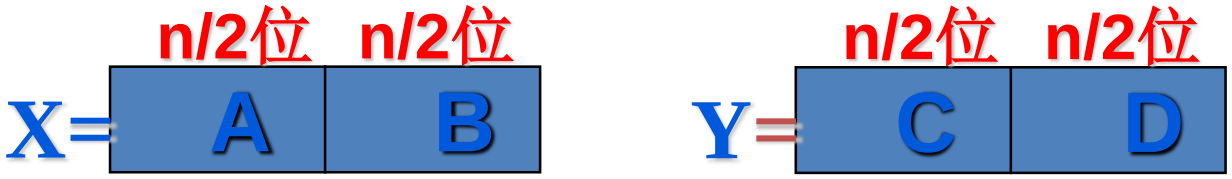
$$AD+BC=(AD + AC+BD+BC-AC-BD)$$

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

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



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!

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*

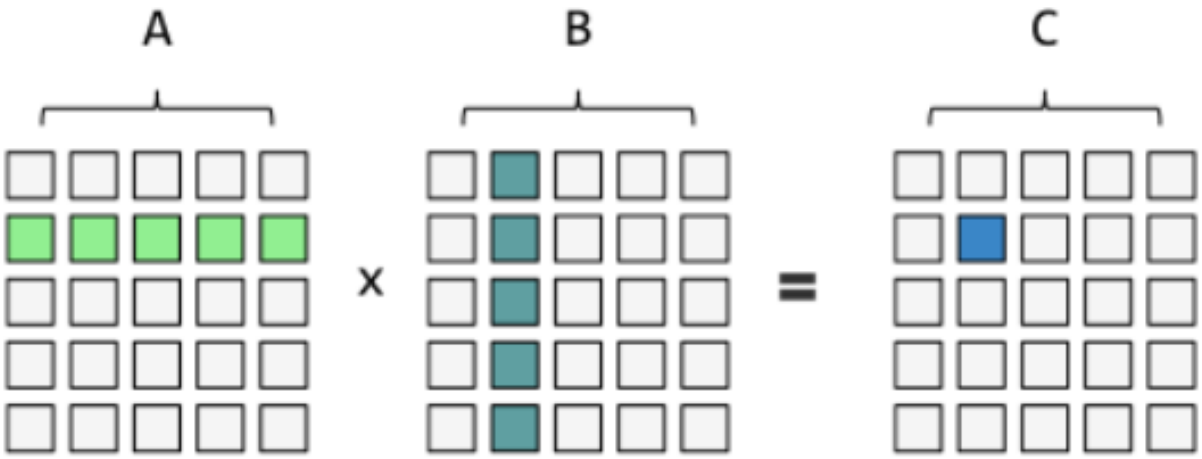
$$\mathsf{M}(n) = O(n \log n).$$

本讲内容

3.1 分治法
3.2 分治法的简单实例
3.3 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



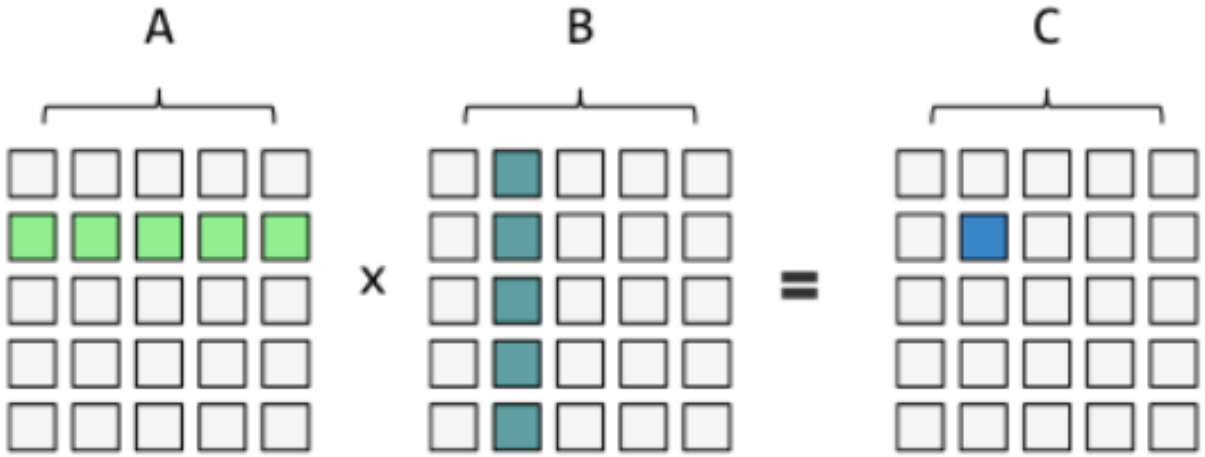
三、矩阵乘法

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的时间复杂性为$O(n^3)$，

面试提问：能否给出一个复杂性为$O(n^{2.81})$的算法？

矩阵乘法初步算法设计

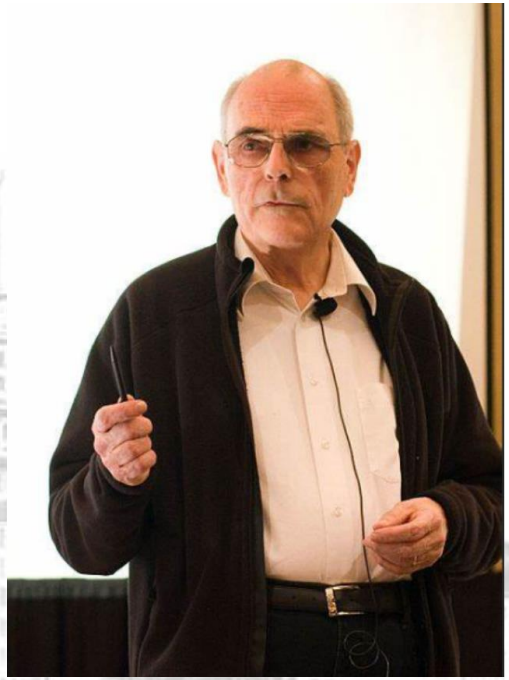
- 划分：把C=AB中的每一个矩阵分为规模相同的4个子矩阵($n/2 \times n/2$)
- 则$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$
- 其中$C_{11} = A_{11}\ B_{11} + A_{12}\ B_{21}$
- $C_{12}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

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—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

$$
\begin{aligned}
S_1 &= B_{12} - B_{22}, \\
S_2 &= A_{11} + A_{12}, \\
S_3 &= A_{21} + A_{22}, \\
S_4 &= B_{21} - B_{11}, \\
S_5 &= A_{11} + A_{22}, \\
S_6 &= B_{11} + B_{22}, \\
S_7 &= A_{12} - A_{22}, \\
S_8 &= B_{21} + B_{22}, \\
S_9 &= A_{11} - A_{21}, \\
S_{10} &= B_{11} + B_{22}.
\end{aligned}
$$

矩阵乘法：Strassen算法

- 递归地计算$n/2 \times n/2$矩阵的乘法：

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 = A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \\
P_2 &= S_2 \cdot B_{22} = A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \\
P_3 &= S_3 \cdot B_{11} = A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \\
P_4 &= A_{22} \cdot S_4 = A_{22} \cdot B_{21} - A_{22} \cdot B_{11} \\
P_5 &= S_5 \cdot S_6 = A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \\
P_6 &= S_7 \cdot S_8 = A_{12} \cdot B_{21} + A12 \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} \\
P_7 &= S_9 \cdot S_{10} = A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12}
\end{aligned}
$$

矩阵乘法：Strassen算法

- 递归地计算$n/2 \times n/2$矩阵的乘法：

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 = A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \\
P_2 &= S_2 \cdot B_{22} = A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \\
P_3 &= S_3 \cdot B_{11} = A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \\
P_4 &= A_{22} \cdot S_4 = A_{22} \cdot B_{21} - A_{22} \cdot B_{11} \\
P_5 &= S_5 \cdot S_6 = A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \\
P_6 &= S_7 \cdot S_8 = A_{12} \cdot B_{21} + A12 \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} \\
P_7 &= S_9 \cdot S_{10} = A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12}
\end{aligned}
$$

只有中间一列需要计算，一共计算7次矩阵积: P_1 - P_7.

矩阵乘法：Strassen算法

- 通过P_i计算C_{11}，C_{12}，C_{21}，C_{21}

$$
\begin{aligned}
C_{11} &= P_5 + P_4 - P_2 + P_6 \\
C_{12} &= P_1 + P_2 \\
C_{21} &= P_3 + P_4 \\
C_{22} &= P_5 + P_1 - P_3 - P_7
\end{aligned}
$$

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_{21}:$\Theta(n^2)$

$$T(n)=\Theta(1) \quad \text{if } n=1$$
$$= 7T(n/2)+\Theta(n^2) \quad \text{if } n>1$$

$$T(n)=\Theta(nlg7)=\Theta(n^{2.81})$$

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 × 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]

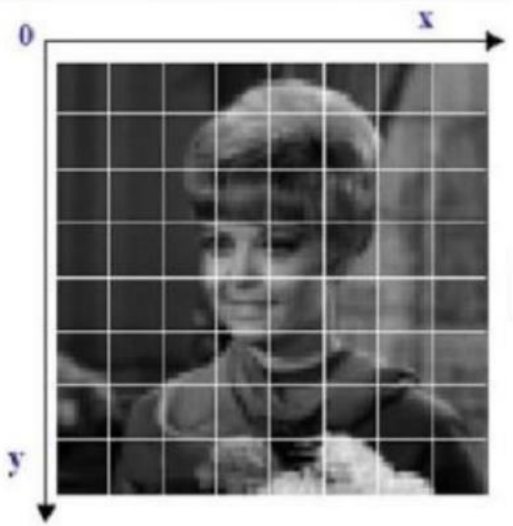
Strassen算法伪代码续（课后熟练）

```
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
```

矩阵乘法的应用

- **计算机的数字、图片的存储方式就是矩阵**
- 计算机科学中许多数学任务都是通过矩阵乘法来处理的，例如机器学习、计算机图形的创建，各种模拟或数据压缩。而计算机计算乘法的速度要远远慢于加法



101	89	110	101	98	98	103	123
100	67	95	98	89	89	99	110
101	56	78	89	87	93	95	97
96	65	76	78	85	89	91	92
67	66	67	78	74	83	86	86
55	54	65	76	76	78	82	83
97	34	56	56	73	74	79	79
56	45	54	56	65	68	73	75

矩阵乘法的持续探索

- 1969 年，德国数学家 Volker Strassen 开发了一种算法，首次将 4×4 矩阵乘法的求解从 64 步减少到 49 步，震动了数学界

$$\begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} \times \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix} = \begin{bmatrix} c_{1,1} & c_{1,2} \\ c_{2,1} & c_{2,2} \end{bmatrix}$$

Standard algorithm

$$h_1 = a_{1,1}\ b_{1,1}$$
$$h_2 = a_{1,1}\ b_{1,2}$$
$$h_3 = a_{1,2}\ b_{2,1}$$
$$h_4 = a_{1,2}\ b_{2,2}$$
$$h_5 = a_{2,1}\ b_{1,1}$$
$$h_6 = a_{2,1}\ b_{1,2}$$
$$h_7 = a_{2,2}\ b_{2,1}$$
$$h_8 = a_{2,2}\ b_{2,2}$$

$$c_{1,1} = h_1 + h_3$$
$$c_{1,2} = h_2 + h_4$$
$$c_{2,1} = h_5 + h_7$$
$$c_{2,2} = h_6 + h_8$$

Strassen's algorithm

$$h_1 = (a_{1,1} + a_{2,2})\ (b_{1,1} + b_{2,2})$$
$$h_2 = (a_{2,1} + a_{2,2})\ b_{1,1}$$
$$h_3 = a_{1,1}\ (b_{1,2} - b_{2,2})$$
$$h_4 = a_{2,2}\ (-b_{1,1} + b_{2,1})$$
$$h_5 = (a_{1,1} + a_{1,2})\ b_{2,2}$$
$$h_6 = (-a_{1,1} + a_{2,1})\ (b_{1,1} + b_{1,2})$$
$$h_7 = (a_{1,2} - a_{2,2})(b_{2,1} + b_{2,2})$$

$$c_{1,1} = h_1 + h_4 - h_5 + h_7$$
$$c_{1,2} = h_3 + h_5$$
$$c_{2,1} = h_2 + h_4$$
$$c_{2,2} = h_1 - h_2 + h_3 + h_6$$

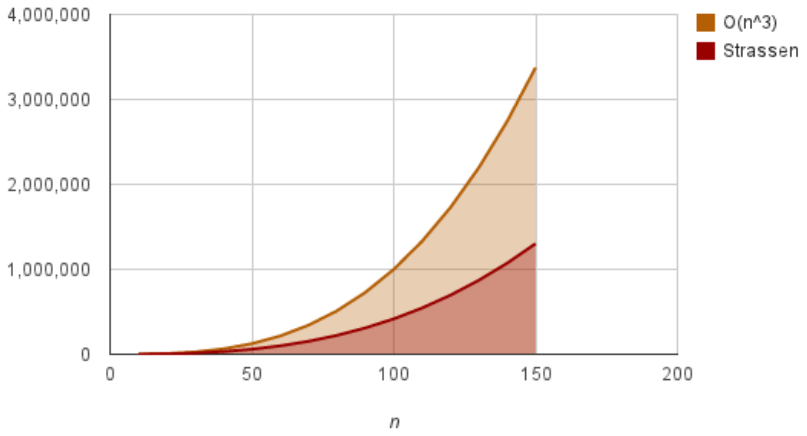
矩阵乘法的延伸知识

5×5 矩阵乘法（n=4）以前要计算 80 步，而 AlphaTensor 新算法只需 76 步；当n=5时，AlphaTensor 将求解从原来的 98 步减少到 96 步。4×4 矩阵乘法由 Strassen 减少到 49 步，AlphaTensor 则将其优化到 47 步。这样的效率是由 AlphaTensor 生成的 70 多个矩阵乘法的算法实现的

Size (n, m, p)	Best method known	Best rank known	AlphaTensor rank Modular	AlphaTensor rank Standard
(2,2,2)	(Strassen, 1969)[2]	7	7	7
(3,3,3)	(Laderman, 1976)[15]	23	23	23
(4,4,4)	(Strassen, 1969)[2] (2,2,2) ⊗ (2,2,2)	49	47	49
(5,5,5)	(3,5,5) + (2,5,5)	98	96	98
(2,2,3)	(2,2,2) + (2,2,1)	11	11	11
(2,2,4)	(2,2,2) + (2,2,2)	14	14	14
(2,2,5)	(2,2,2) + (2,2,3)	18	18	18
(2,3,3)	(Hopcroft and Kerr, 1971)[16]	15	15	15
(2,3,4)	(Hopcroft and Kerr, 1971)[16]	20	20	20
(2,3,5)	(Hopcroft and Kerr, 1971)[16]	25	25	25
(2,4,4)	(Hopcroft and Kerr, 1971)[16]	26	26	26
(2,4,5)	(Hopcroft and Kerr, 1971)[16]	33	33	33
(2,5,5)	(Hopcroft and Kerr, 1971)[16]	40	40	40
(3,3,4)	(Smirnov, 2013)[18]	29	29	29
(3,3,5)	(Smirnov, 2013)[18]	36	36	36
(3,4,4)	(Smirnov, 2013)[18]	38	38	38
(3,4,5)	(Smirnov, 2013)[18]	48	47	47
(3,5,5)	(Sedoglavic and Smirnov, 2021)[19]	58	58	58
(4,4,5)	(4,4,2) + (4,4,3)	64	63	63
(4,5,5)	(2,5,5) ⊗ (2,1,1)	80	76	76

矩阵乘法的应用

- **计算机的数字、图片的存储方式就是矩阵**
- 计算机科学中许多数学任务都是通过矩阵乘法来处理的，例如机器学习、计算机图形的创建，各种模拟或数据压缩。而计算机计算乘法的速度要远远慢于加法
- 考虑到 GPU 每天要进行万亿次矩阵计算，每一步看似很小的增量改进，都能很大地提升计算效率！



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

中位数问题定义

Input: 由n个数构成的多重集合X

Output: $x \in X$使得 $-1 \le |\{y \in X \mid y<x\}| - |\{y \in X \mid y>x\}| \le 1$

中位数问题定义

Input: 由n个数构成的多重集合X

Output: $x \in X$使得 $-1 \le |\{y \in X \mid y<x\}| - |\{y \in X \mid y>x\}| \le 1$

一个数集中最多有一半的数值小于中位数，也最多有一半的数值大于中位数。如果大于和小于中位数的数值个数均少于一半，那么数集中必有若干值等同于中位数。

中位数选取问题的复杂度





[Blum et al. *STOC*'72 & *JCSS*'73]

– A "shining" paper by five authors:
  - Manuel Blum (Turing Award 1995)
  - Robert W. Floyd (Turing Award 1978)
  - Vaughan R. Pratt
  - Ronald L. Rivest (Turing Award 2002)
  - Robert E. Tarjan (Turing Award 1986)

– 从***n***个数中选取中位数需要的比较操作的次数介于 **1.5*n***到 **5.43*n***之间

比较操作次数的上下界

- 上界
  - $3n + o(n)$ by Schonhage, Paterson, and Pippenger (*JCSS* 1975).
  - $2.95n$ by Dor and Zwick (*SODA* 1995, *SIAM Journal on Computing* 1999).
- 下界
  - $2n+o(n)$ by Bent and John (*STOC* 1985)
  - $(2+2^{-80})n$ by Dor and Zwick (*FOCS* 1996, *SIAM Journal on Discrete Math* 2001).

线性时间选择

–本节讨论如何在$O(n)$时间内从n个不同的数中选取第i大的元素

–中位数问题也就解决了，因为选取中位数即选择第$n/2$-大的元素

Input: n个(不同)数构成的集合X,整数i,其中$1 \le i \le n$

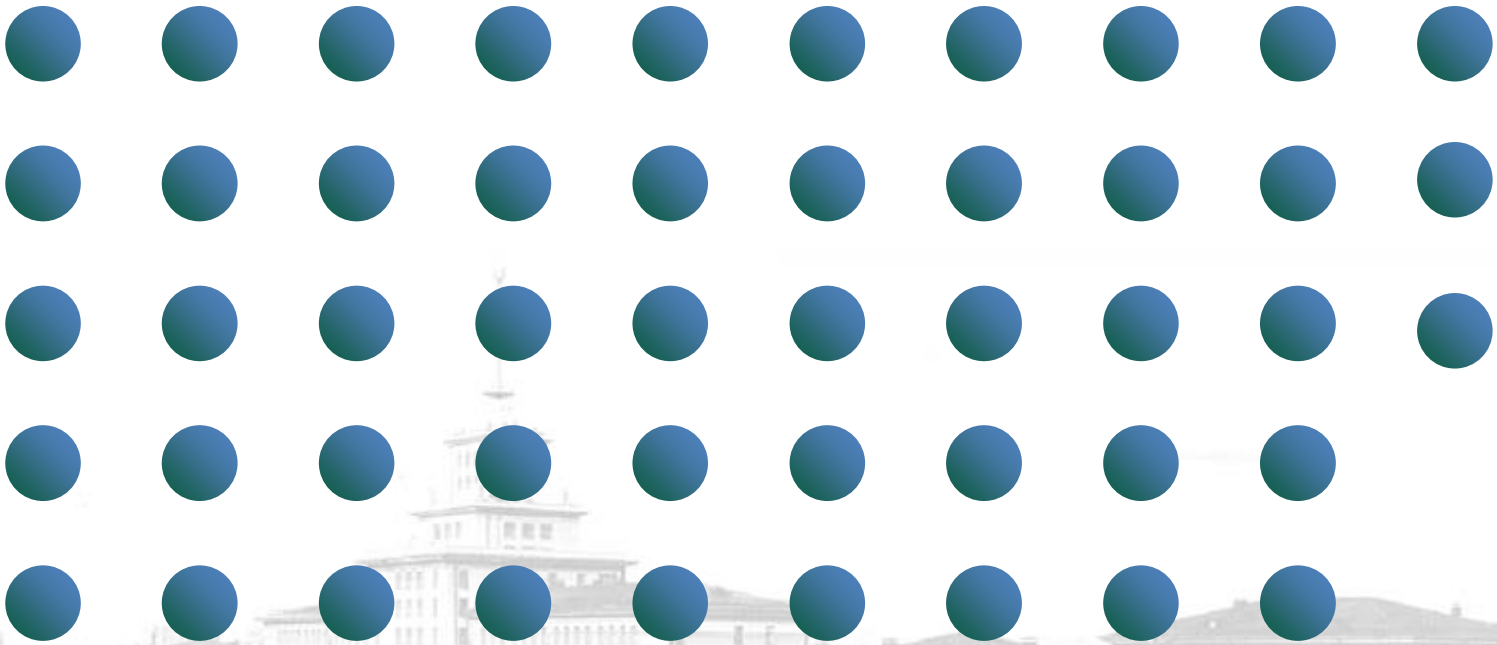
Output: $x \in X$使得X中恰有i-1个元素小于x

求解步骤

第一步： 分组，每组5个数

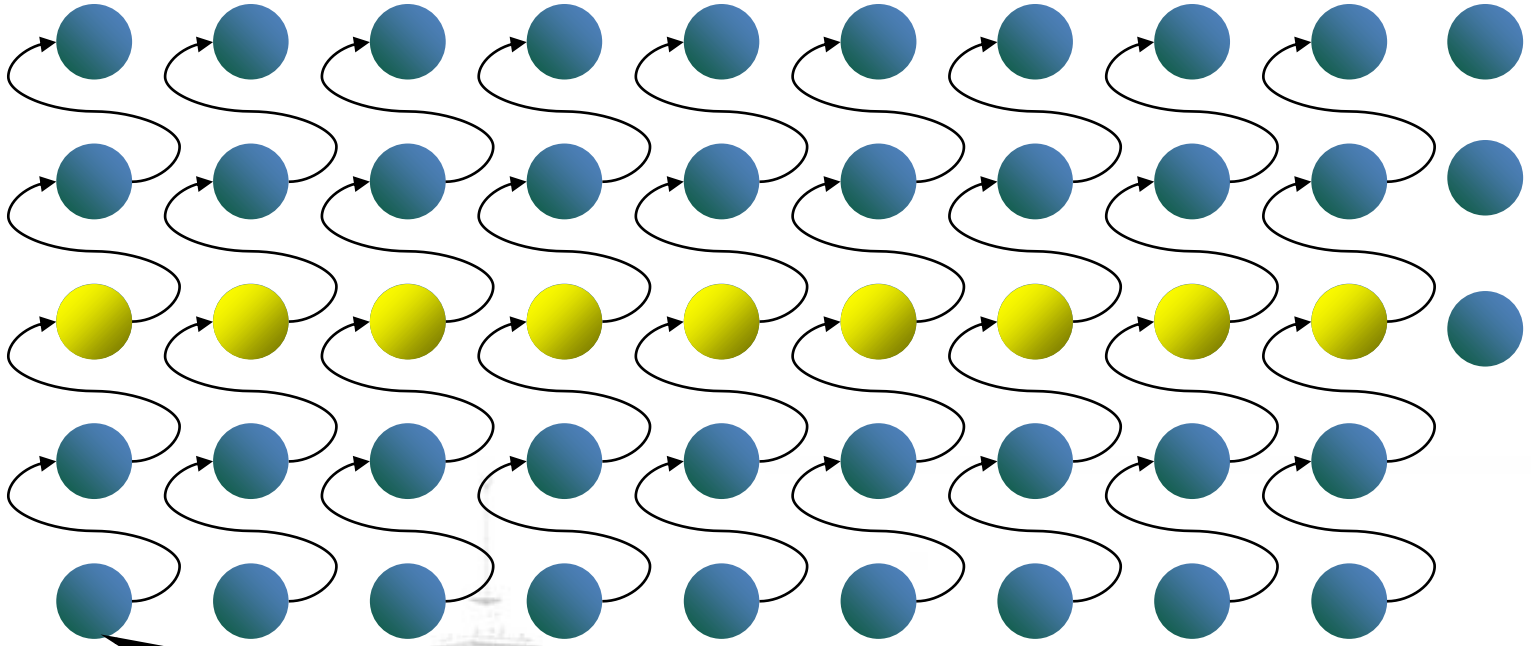
最后一组可能少于5个数，一共有$\lceil n/5 \rceil$组

(至多只有一组由剩下的n mod 5 个数组成)



第二步：寻找每一组的中位数

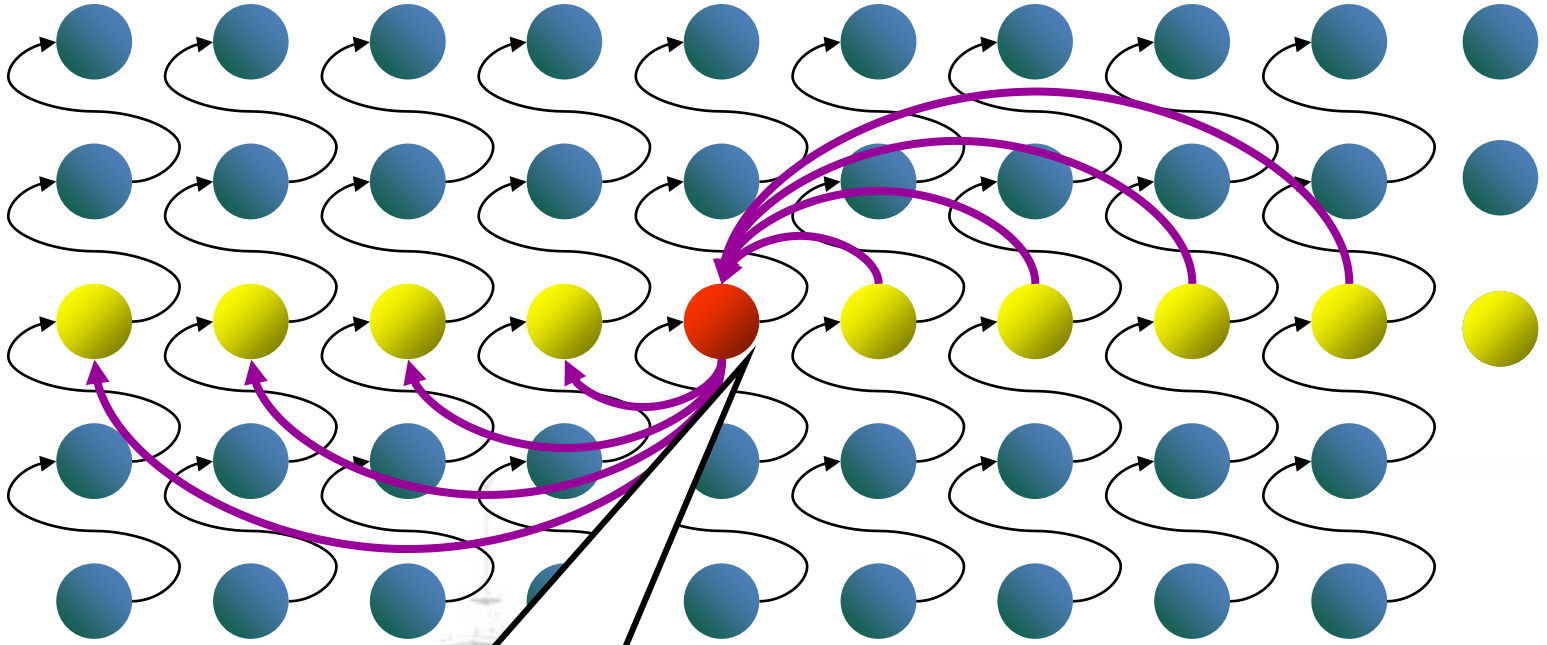
将每组数分别用InsertSort排序
确定每组有序元素的中位数



排序每组数时，比较操作的次数为5(5-1)/2=10次
总共需要10*⌈*n*/5⌉次比较操作

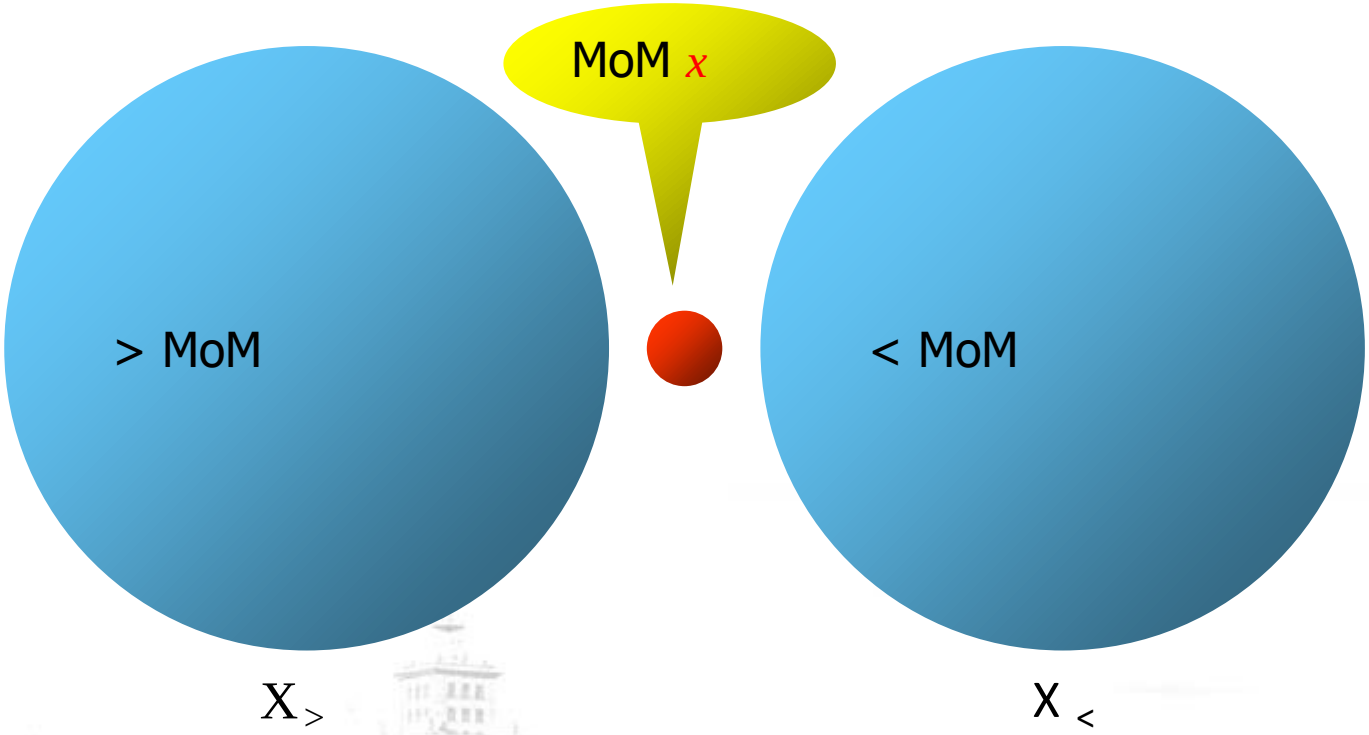
第三步：递归调用算法求得这些中位数的中位数(MoM)

(如果有偶数个中位数，为方便，规定x是较小的中位数)



时间复杂性：$T(\lfloor n/5 \rfloor)$

第四步:用中位数的中位数(MoM)对数组进行划分，让x是第k小的元素，且有n-k个元素划分在高区



时间复杂性$O(n)$

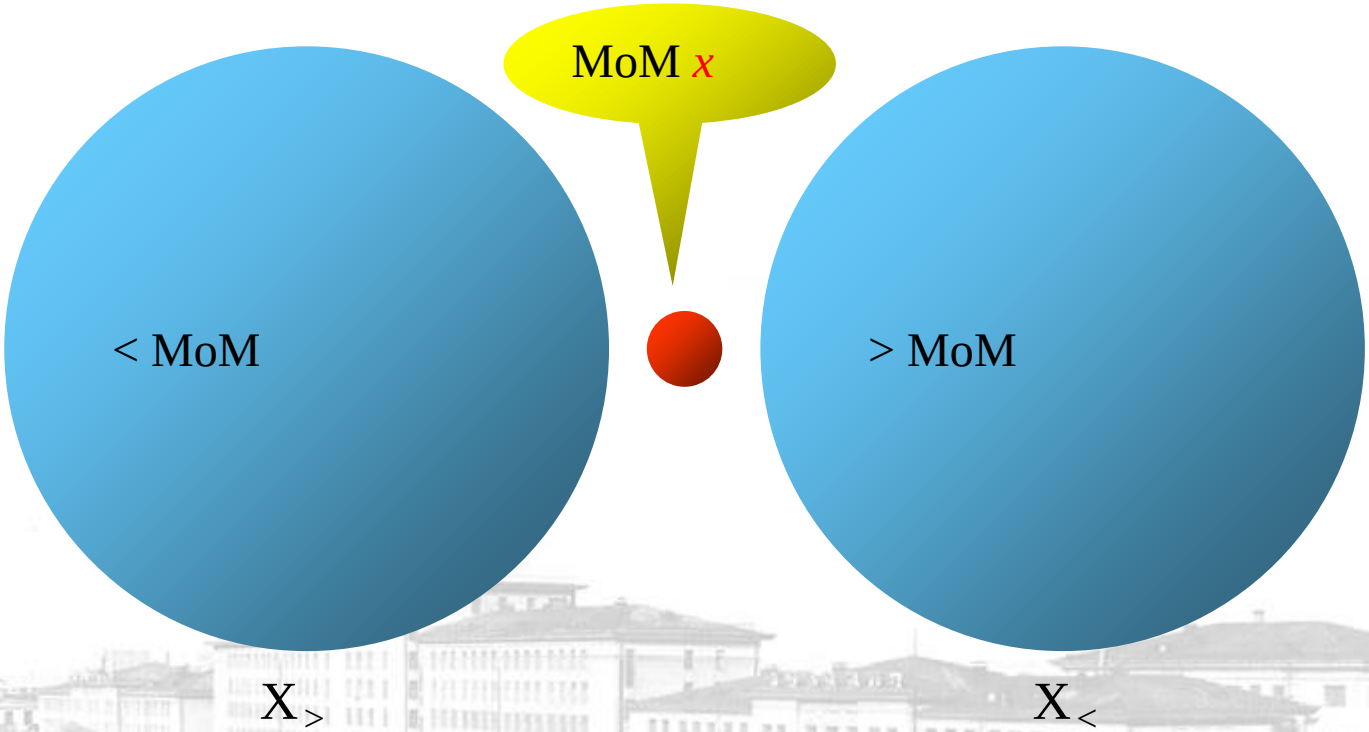
第五步:递归

设x是中位数的中位数(MoM),划分完成后其下标为k

如果$i=k$,则返回x

如果$i<k$,则在第一个部分递归选取第i-大的数

如果$i>k$,则在第三个部分递归选取第$(i-k)$-大的数



算法Select(*A*,*i*)

Input: 数组*A*[1:*n*], 1≤ *i* ≤*n*

Output: *A*[1:*n*]中的第*i*-大的数

```
1.  for  j←1   to  n/5                                          第一步
2.       InsertSort(A[(j-1)*5+1 : (j-1)*5+5]);                  第二步
3.       swap(A[j], A[[(j-1)*5+3]);
4.  x ←Select(A[1: n/5],  n/10 );                               第三步
5.  k ←partition(A[1:n], x);        快速排序的确定性划分算法     第四步
6.  if        k=i  then   return x;                             第五步
7.  else if  k>i  then   return Select(A[1:k-1],i);
8.  else                 return Select(A[k+1:n],i-k);
```

算法分析

算法Select(*A*,*i*)

Input: 数组*A*[1:*n*], 1≤ *i* ≤*n*

Output: *A*[1:*n*]中的第*i*-大的数

```
1.  for j←1  to  n/5
2.      InsertSort(A[(j-1)*5+1 : (j-1)*5+5]);
3.      swap(A[j], A[[(j-1)*5+3]);
4.  x ←Select(A[1: n/5],  n/10 );
5.  k ←partition(A[1:n], x);
6.  if       k=i  then   return x;
7.  else if  k>i  then   retrun Select(A[1:k-1],i);
8.  else                 retrun Select(A[k+1:n],i-k);
```

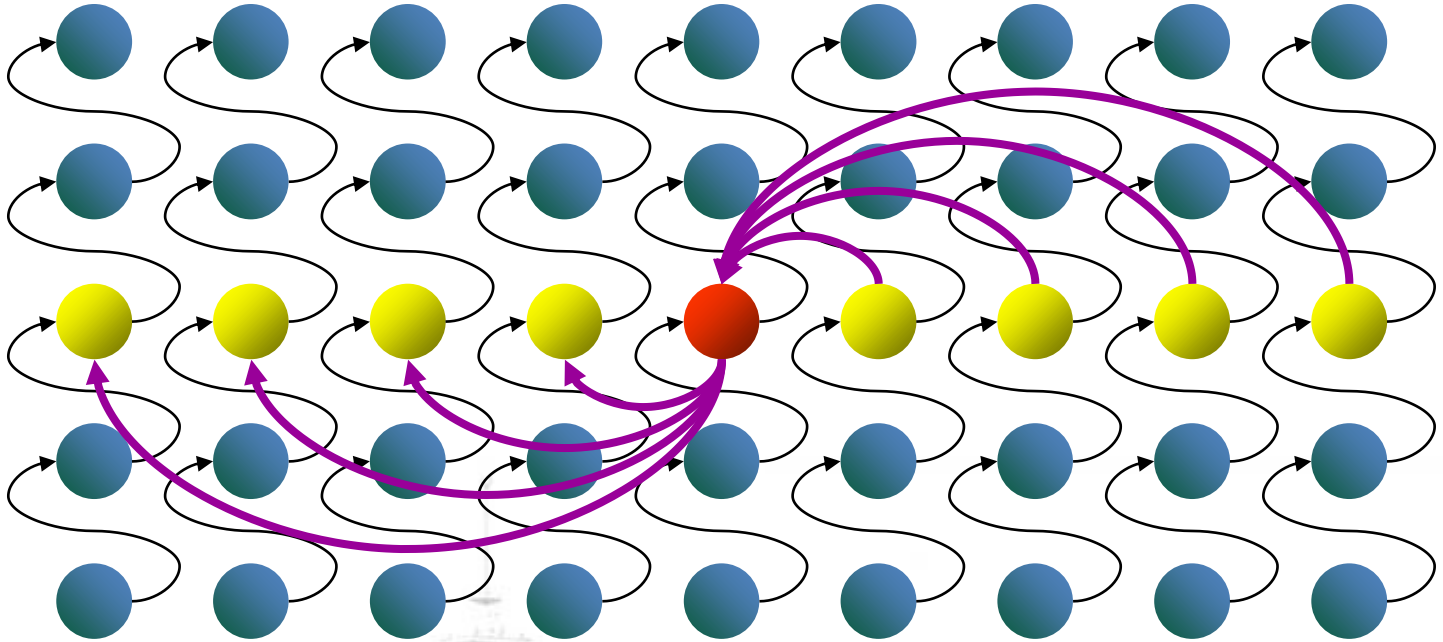
O(*n*)

T(⌈*n*/5⌉)

O(*n*)

???

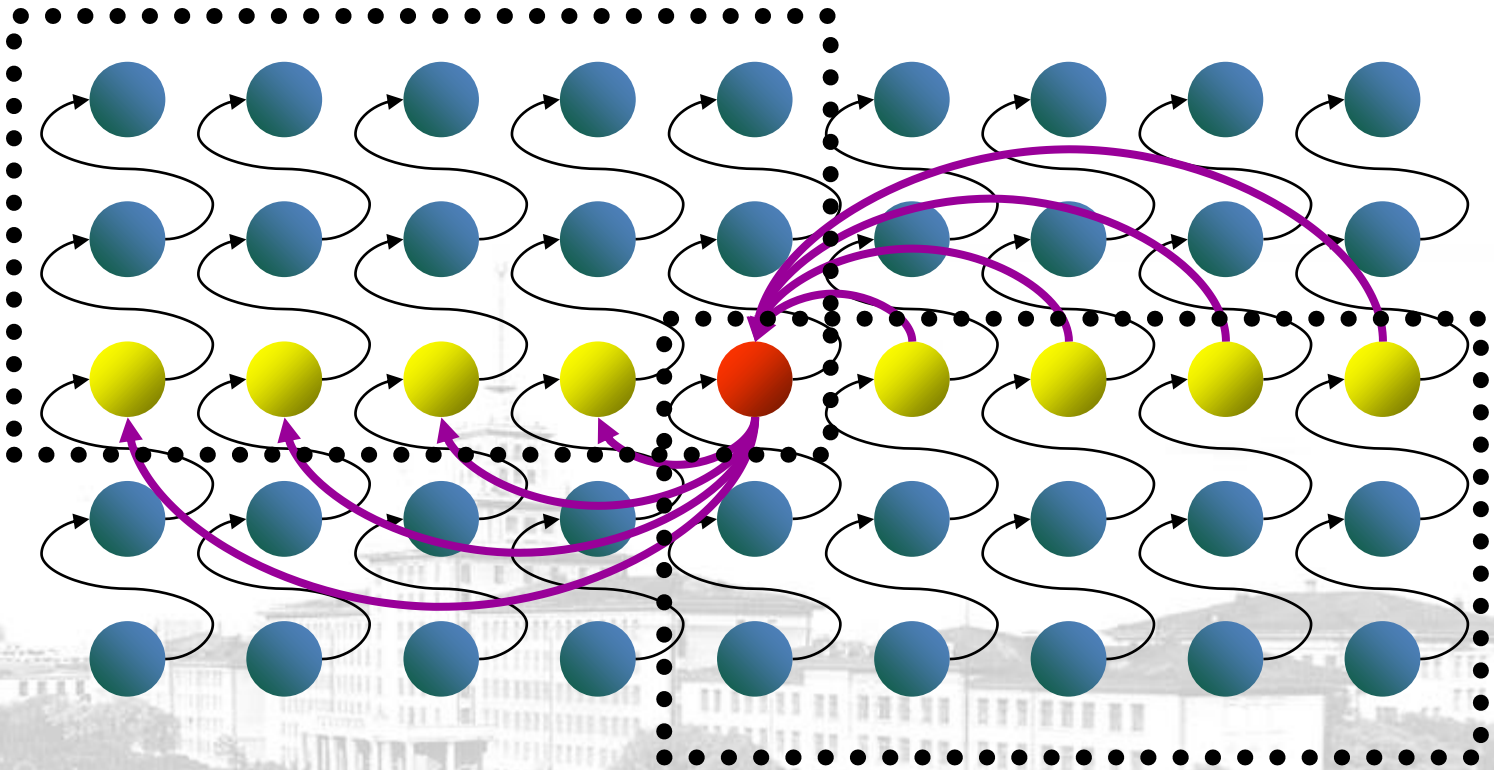
观察第五步的处理过程



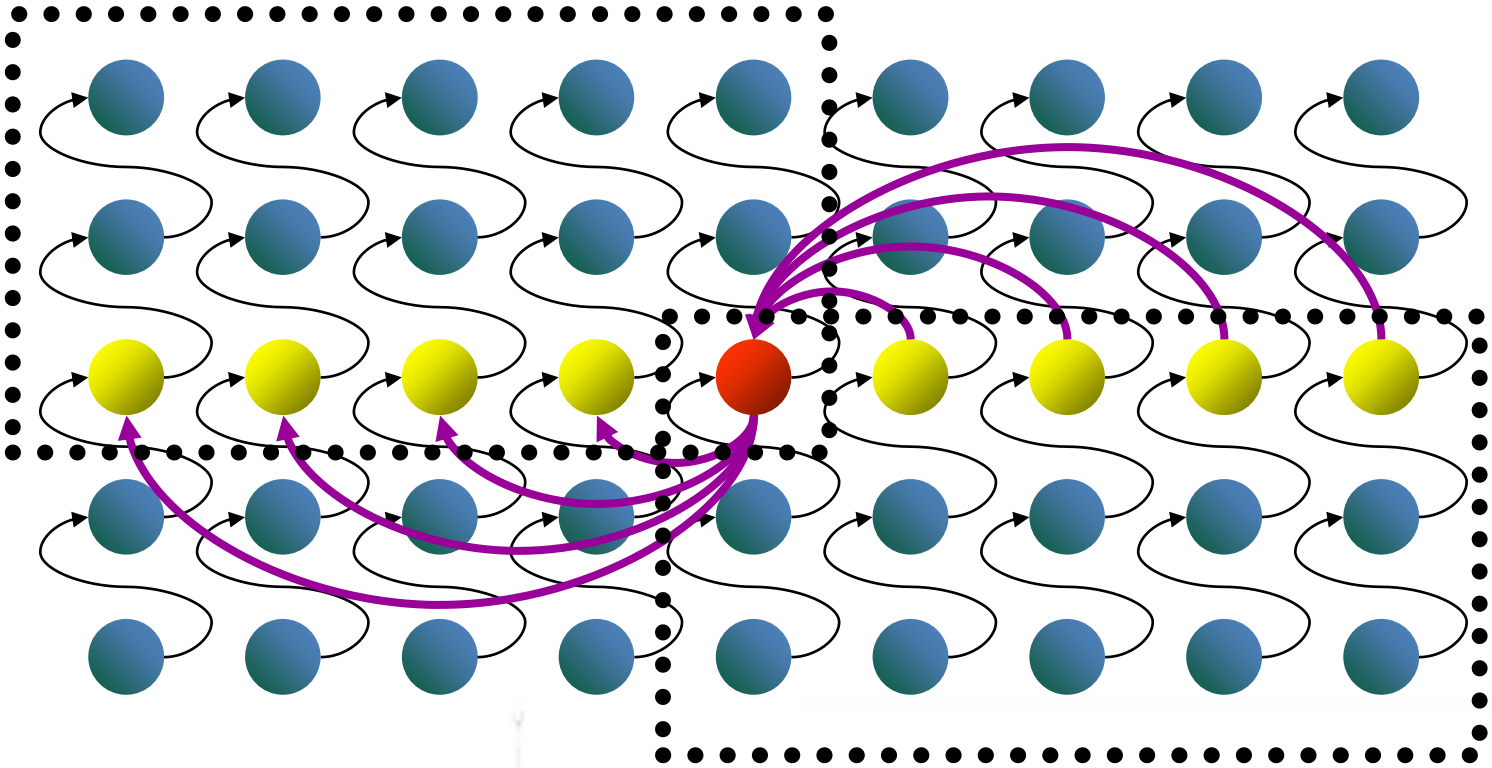
第五步至少删除了[3n/10]-6个数：

思考:在x的右边，每个5元素列中有3个元素大于x；在x的左边，每个5元素列有3个元素小于x

在第2步找到的中位数中，至少有一半大于等于x. 在所有 $\left\lfloor \frac{n}{5} \right\rfloor$ 组中，至少一半的组有3个元素大于x，除了最后一组和包含x的组，则大于x的元素个数至少是：$3\left(\left\lfloor \frac{1}{2}\left\lfloor \frac{n}{5} \right\rfloor \right\rfloor - 2\right) \geq \frac{3n}{10} - 6$



第五步至少删除了[3*n*/10]-6个数



$n-\lceil 3n/10\rceil +6 \le 7n/10+6$

如果时间复杂度是输入规模的递增函数
则第五步的时间开销不超过T(7n/10+6)

算法的分析

$$T(n)\leq \begin{cases} \Theta(1) & \text{if } n\leq c \\ T(\lceil n/5\rceil)+T(7n/10+6)+O(n) & \text{if } n>c \end{cases}$$

$$T(n)=O(n)$$

算法的分析

假设：任何少于140个元素的输入要O(1)时间

$$T(n)\leq \begin{cases} O(1) & \text{if } n\leq 140 \\ T(\lceil n/5\rceil)+T(7n/10+6)+O(n) & \text{if } n>140 \end{cases}$$

证明思路：找$T(n)\leq cn$ 成立的c，并且找到a，使得对于所有n, $O(n)$项所对应的函数(用来描述算法运行时间中的非递归部分)的上界是an

$$T(n) \leq c\lceil n/5\rceil + c\left(\frac{7n}{10}+6\right)+an$$
$$\leq c\frac{n}{5}+\mathrm{c}+\frac{7nc}{10}+6c+an$$
$$= \frac{9nc}{10}+7c+an = cn+(-\frac{cn}{10}+7c+an)$$

当n>70时，有$c \geq 10a\frac{n}{n-70}$. 因假设$n>140$,有$\frac{n}{n-70}\leq 2$.

当取$c \geq 20a$时，$(-\frac{cn}{10}+7c+an)\leq 0$ $T(n)=O(n)$

对此算法的思考—

–本算法中输入元素被分为每组5个元素

- 如果被分为7个元素，此算法还是线性时间吗？
- 如果被分为3个元素，此算法还是线性时间吗？

对此算法的思考一

–本算法中输入元素被分为每组5个元素

- 如果被分为k个元素：
- 比 MoM x 更小（或者更大）的元素至少有：

$$\left\lceil \frac{k}{2} \right\rceil \left(\left\lceil \frac{1}{2} \left\lceil \frac{n}{k} \right\rceil \right\rceil - 2 \right) \geq \frac{n}{4} - k$$

- Select算法在最坏情况下至多递归了 $n - \left(\frac{n}{4} - k\right) = \frac{3n}{4} + k$ 个元素，递归方程有：
- $T(n) \leq T(\lceil \frac{n}{k} \rceil) + T(3n/4+k) + O(n)$
- 猜想 $T(n) \leq cn$, $T(n) \leq c\lceil \frac{n}{k} \rceil + c(3n/4+k) + O(n)$
- $\leq c(n/k+1) + 3cn/4 + ck + O(n)$
- $\leq cn/k + 3cn/4 + c + ck + O(n)$
- $= cn(1/k+3/4) + c(1+k) + O(n)$
- 当k >4时，最后式子≤cn

当k =3时，$T(n)=\Omega(n \lg n)$

对此算法的思考二

–递归方程中140的选取有何奥妙？

– 证明如果$n \geq 140$,则至少$\left\lfloor \frac{n}{4} \right\rfloor$个元素大于中位数的中位数$x$，至少$\left\lfloor \frac{n}{4} \right\rfloor$个元素小于$x$

对此算法的思考二

– 递归方程中140的选取有何奥妙？

– 证明如果$n \geq 140$,则至少$\left\lceil \frac{n}{4} \right\rceil$个元素大于中位数的中位数$x$，至少$\left\lceil \frac{n}{4} \right\rceil$个元素小于$x$

MoM至少大于$\frac{3n}{10}-6$个元素，同时至少小于$\frac{3n}{10}-6$个元素.

$\frac{3n}{10}-6 \geq \frac{2.5n}{10}+\frac{0.5n}{10}-6$，当$n \geq 140$时，$\frac{0.5n}{10} \geq 7$

$\geq \frac{2.5n}{10}+7-6$

$= \frac{2.5n}{10}+1 \geq \left\lceil \frac{n}{4} \right\rceil$

BFPRT 算法延伸知识

–最坏情况下线性时间查找中位数的算法是由Manuel Blum，Robert W. Floyd,Vaughan Pratt，Ron Rivest 与 Robert Tarjan 提出的

- **Manuel Blum (1995图灵奖)，计算复杂性理论奠基人之一**
- Robert W. Floyd **(1978图灵奖)**，发明了最短路径算法，且是堆排算法的发明者之一
- Vaughan Pratt，KMP 算法的发明者之一（其中的 P ）导师是著名的 Donald Ervin Knuth
- Ron Rivest(2002年图灵奖)，RSA 算法的发明者之一（其中的 R 就是他）
- Robert Tarjan (1986年图灵奖)，发明了强连通分量算法

BFPRT 算法延伸知识

–最坏情况下线性时间查找中位数的算法是由Manuel Blum，Robert W. Floyd,Vaughan Pratt，Ron Rivest 与 Robert Tarjan 提出的

-Median of medians（BFPRT算法的维基百科）
https://en.wikipedia.org/wiki/Median_of_medians

-BFPRT算法论文：《Time Bounds for Selection》
http://people.csail.mit.edu/rivest/pubs/BFPRT73.pdf

BFPRT 算法也可称“五星算法“

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

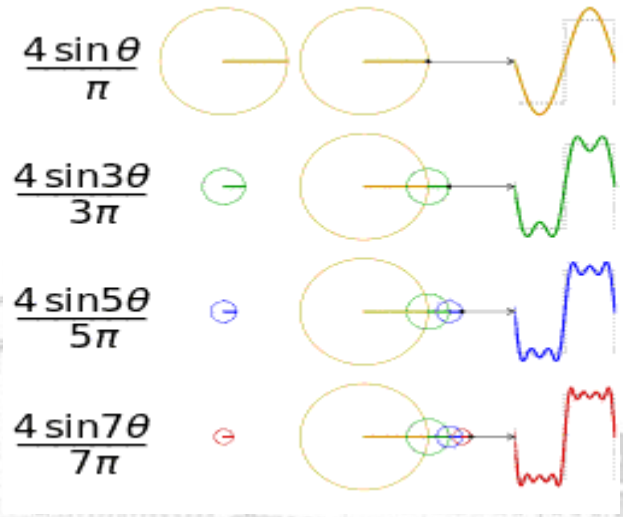
问题定义

输入：$a_0,a_1,\ldots,a_{n-1}$，　$n=2^k$, a_i是实数，$(0\le i\le n-1)$

输出：$A_0,A_1,\ldots,A_{n-1}$

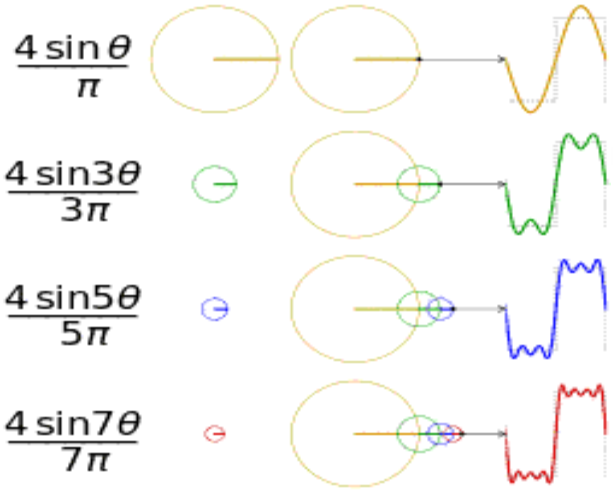
$A_j=\sum_{k=0}^{n} a_k e^{\frac{2jk\pi}{n}i}$,其中j=0,1,,…,n-1,

e是自然对数的底数，i是虚数单位



傅里叶级数？

傅里叶级数是把类似波的函数表示成简单的正弦波的方式。它能将任何周期性函数信号分解成一个（可能由无穷个元素组成的）简单震荡函数集合，即正弦函数和余弦函数



傅里叶变换？

- 傅里叶变换(**Fourier transform FT**)将一个函数(通常是一个时间的函数，或一个信号)分解成它的组成频率，例如用组成音符的音量和频率表示一个音乐和弦。
- **傅里叶变换是频域表示和将频域表示与时间函数相关联的数学运算**。其本质是一种线性积分变换，用于信号在时域（或空域）和频域之间的变换，在物理学和工程学中有许多应用。
- 傅里叶变换就像化学分析，确定物质的基本成分；信号来自自然界，也可对其进行分析，**确定其基本频率成分**

傅里叶级数vs变换？

- **傅里傅立叶级数对应的是周期信号，而傅立叶变换则对应的是一个时间连续可积信号（不一定是周期信号）**
- 傅立叶级数要求信号在一个周期内能量有限，而后者则要求在整个区间能量有限
- 傅立叶级数的对应ω是离散的，而傅立叶变换则对应ω是连续的。
- 傅立叶级数是周期信号的另一种时域的表达方式,也就是正交级数,它是不同的频率的波形的时域叠加。而傅立叶变换则是完全的频域分析
- 傅里叶级数适用于**对周期性现象**做数学上的分析，傅里叶变换可以看作傅里叶级数的极限形式，也可以看作是对周期现象进行数学上的分析，同时也适用于**非周期性现象**的分析。

问题定义

输入：$a_0,a_1,\ldots,a_{n-1}$，　$n=2^k$, a_i是实数，$(0\le i\le n-1)$

输出：$A_0,A_1,\ldots,A_{n-1}$

$A_j=\sum_{k=0}^{n} a_k e^{\frac{2jk\pi}{n}i}$,其中j=0,1,,…,n-1,

e是自然对数的底数，i是虚数单位

蛮力法利用定义计算每个A_j，时间复杂度为$\Theta(n^2)$

优化该算法的意义

输入：$a_0,a_1,\ldots,a_{n-1}$，　$n=2^k$, a_i是实数 , $(0\le i\le n-1)$

输出：$A_0,A_1,\ldots,A_{n-1}$

$A_j=\sum_{k=0}^{n} a_k e^{\frac{2jk\pi}{n}i}$,其中$j=0,1,,\ldots,n-1$,

e是自然对数的底数，i是虚数单位

蛮力法利用定义计算每个A_j，时间复杂度为$\Theta(n^2)$

降低时间复杂度对于嵌入式应用领域（受限于采用的芯片算力）很有价值！

快速傅里叶变换

令 $\beta_n = e^{\frac{2\pi}{n}i}$

$$A_j = a_0 + a_1\beta_n^{\,j} + a_2\beta_n^{\,2j} + \ldots + a_{n-1}\beta_n^{\,(n-1)j} \qquad 0 \le j \le n-1$$

$$= [a_0 + a_2\beta_n^{\,2j} + a_4\beta_n^{\,4j} \ldots + a_{n-2}\beta_n^{\,(n-2)j}] +$$

$$[a_1 + a_3\beta_n^{\,2j} + a_5\beta_n^{\,4j} \ldots + a_{n-1}\beta_n^{\,(n-2)j}]\,\beta_n^{\,j}$$

快速傅里叶变换

令 $\beta_n = e^{\frac{2\pi}{n}i}$

$$A_j = a_0 + a_1\beta_n^{\,j} + a_2\beta_n^{\,2j} + \ldots + a_{n-1}\beta_n^{\,(n-1)j} \qquad 0 \le j \le n-1$$

$$= [a_0 + a_2\beta_n^{\,2j} + a_4\beta_n^{\,4j} \ldots + a_{n-2}\beta_n^{\,(n-2)j}] +$$

$$[a_1 + a_3\beta_n^{\,2j} + a_5\beta_n^{\,4j} \ldots + a_{n-1}\beta_n^{\,(n-2)j}]\,\beta_n^{\,j}$$

FFT中重要性质（消去引理）：$\beta_n^k = \beta_{2n}^{2k}$

快速傅里叶变换

令 $\beta_n = e^{\frac{2\pi}{n}i}$

$$A_j = a_0 + a_1\beta_n^{\,j} + a_2\beta_n^{\,2j} + \ldots + a_{n-1}\beta_n^{\,(n-1)j}$$

$$= [a_0 + a_2\beta_n^{\,2j} + a_4\beta_n^{\,4j} \ldots + a_{n-2}\beta_n^{\,(n-2)j}] + [a_1 + a_3\beta_n^{\,2j} + a_5\beta_n^{\,4j} \ldots + a_{n-1}\beta_n^{\,(n-2)j}]\,\beta_n^{\,j}$$

$$= \left[a_0 + a_2\beta_{n/2}^{\,j} + a_4\beta_{n/2}^{\,2j} + \ldots + a_{n-2}\beta_{n/2}^{\frac{n-2}{2}j}\right] + \left[a_1 + a_3\beta_{n/2}^{\,j} + a_5\beta_{n/2}^{\,2j} + \ldots + a_{n-1}\beta_{n/2}^{\frac{n-2}{2}j}\right]\beta_n^{\,j}$$

第一项内形如$a_0, a_2, a_4, \ldots, a_{n-2}$的离散傅里叶变换

第二项内形如$a_1, a_3, a_5, \ldots, a_{n-1}$的离散傅里叶变换

令 $\beta_n = e^{\frac{2\pi}{n}i}$

$$A_j = a_0 + a_1\beta_n^{\,j} + a_2\beta_n^{\,2j} + \ldots + a_{n-1}\beta_n^{\,(n-1)j} \qquad 0 \le j \le n-1$$

$$= \left[a_0 + a_2\beta_{n/2}^{\,j} + a_4\beta_{n/2}^{\,2j} + \ldots + a_{n-2}\beta_{n/2}^{\frac{n-2}{2}j} \right] +$$

$$\left[a_1 + a_3\beta_{n/2}^{\,j} + a_5\beta_{n/2}^{\,2j} + \ldots + a_{n-1}\beta_{n/2}^{\frac{n-2}{2}j} \right] \beta_n^{\,j}$$

$$B_j = a_0 + a_2\beta_{n/2}^{\,j} + a_4\beta_{n/2}^{\,2j} + \ldots + a_{n-2}\beta_{n/2}^{\,((n-2)/2)j} \qquad 0 \le j \le (n-2)/2$$

$$C_j = a_1 + a_3\beta_{n/2}^{\,j} + a_5\beta_{n/2}^{\,2j} + \ldots + a_{n-1}\beta_{n/2}^{\,((n-2)/2)j} \qquad 0 \le j \le (n-2)/2$$

令 $\beta_n = e^{\frac{2\pi}{n}i}$

$$A_j=a_0+a_1\beta_n^{\,j}+a_2\beta_n^{\,2j}+\ldots+a_{n-1}\beta_n^{\,(n-1)j} \qquad 0\le j\le n-1$$

$$B_j=a_0+a_2\beta_{n/2}^{\,j}+a_4\beta_{n/2}^{\,2j}+\ldots+a_{n-2}\beta_{n/2}^{\,((n-2)/2)j} \qquad 0\le j\le (n-2)/2$$

$$C_j=a_1+a_3\beta_{n/2}^{\,j}+a_5\beta_{n/2}^{\,2j}+\ldots+a_{n-1}\beta_{n/2}^{\,((n-2)/2)j} \qquad 0\le j\le (n-2)/2$$

FFT中重要性质（折半引理）

$$\beta_{n/2}^{kj} = e^{\frac{2\pi i}{n/2}kj} = e^{\frac{2\pi i}{n/2}kj-2k\pi i} = e^{\frac{2\pi i}{n/2}k(j-n/2)} = \beta_{n/2}^{k(j-n/2)}$$

令 $\beta_n = e^{\frac{2\pi}{n}i}$

$$A_j=a_0+a_1\beta_n^{\,j}+a_2\beta_n^{\,2j}+\ldots+a_{n-1}\beta_n^{\,(n-1)j} \qquad 0\le j\le n-1$$

$$B_j=a_0+a_2\beta_{n/2}^{\,j}+a_4\beta_{n/2}^{\,2j}+\ldots+a_{n-2}\beta_{n/2}^{\,((n-2)/2)j} \qquad 0\le j\le (n-2)/2$$

$$C_j=a_1+a_3\beta_{n/2}^{\,j}+a_5\beta_{n/2}^{\,2j}+\ldots+a_{n-1}\beta_{n/2}^{\,((n-2)/2)j} \qquad 0\le j\le (n-2)/2$$

FFT中重要性质（折半引理）

$$\beta_{n/2}^{kj} = e^{\frac{2\pi i}{n/2}kj} = e^{\frac{2\pi i}{n/2}kj-2k\pi i} = e^{\frac{2\pi i}{n/2}k(j-n/2)} = \beta_{n/2}^{k(j-n/2)}$$

折半引理保证了递归子问题的规模是原问题的一半！

分治算法过程

划分： 将输入拆分成$a_0,a_2,\ldots,a_{n-2}$和$a_1,a_3,\ldots,a_{n-1}$.

递归求解： 递归计算$a_0,a_2,\ldots,a_{n-2}$的变换$B_0,B_1,\ldots,B_{n/2-1}$
递归计算$a_1,a_3\ldots,a_{n-1}$的变换$C_0,C_1,\ldots,C_{n/2-1}$

合并： $A_j= B_j+C_j\cdot\beta_n^{\,j}$ $(j<n/2)$
$A_j=B_{j-n/2}+C_{j-n/2}\cdot\beta_n^{\,j}$ $(n/2\le j<n-1)$

算法

算法FFT

输入：$a_0,a_1,\ldots,a_{n-1}$，$n=2^k$

输出：$a_0,a_1,\ldots,a_{n-1}$的傅里叶变换$A_0,\ldots,A_{n-1}$

1. $\beta \leftarrow \exp(2\pi i/n)$;
2. If (n=2) Then
3. $A_0 \leftarrow a_0+a_1$;
4. $A_1 \leftarrow a_0-a_1$;
5. 输出A_0,A_1,算法结束;
6. $B_0,B_1,\ldots,B_{n/2-1} \leftarrow \mathrm{FFT}(a_0,a_2,\ldots,a_{n-2}, n/2)$;
7. $C_0,C_1,\ldots,C_{n/2-1} \leftarrow \mathrm{FFT}(a_1,a_3,\ldots,a_{n-1}, n/2)$;
8. For j=0 To n/2-1
9. $A_j \leftarrow B_j+C_j\cdot\beta^j$;
10. $A_{j+n/2} \leftarrow B_j+C_j\cdot\beta^{j+n/2}$;
11. 输出$A_0,A_1,\ldots,A_{n-1}$,算法结束;

算法分析

$$T(n)=\Theta(1) \quad \text{If} \quad n=2$$
$$T(n)=2T(n/2)+\Theta(n) \quad \text{If} \quad n>2$$

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

算法延伸知识

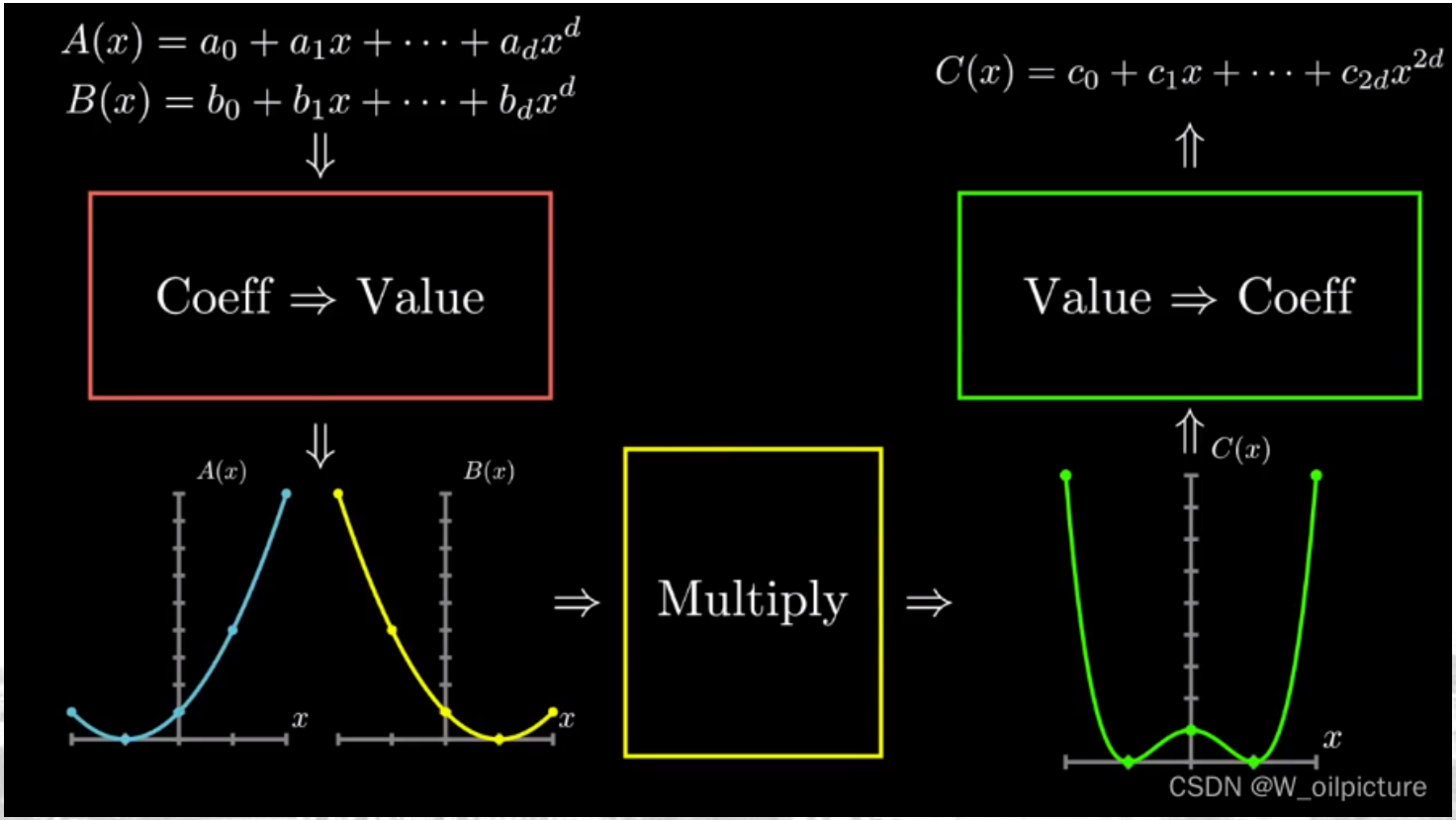
- 1994年，Gilbert Strang将FFT描述为“我们一生中最重要的数值算法”，并被IEEE杂志《计算科学与工程》列入20世纪十大算法之一，它深远的影响了我们世界与日常生活。
- 在我们日常生活中很多设备里面都有它的影子，比如手机、比如photoshop，比如数字音响等
- https://www.math.wustl.edu/~victor/mfmm/fourier/fft.c

算法延伸知识

- 阅读《算法导论》第30章 前两节，了解快速傅里叶变换如何应用于**加速**多项式相乘问题
  - **多项式的系数表示法和点值表示法**
  - 多项式的系数表示和点值表示可以**互相转换**
  - FFT其实是一个用O（n log 2 n）时间将一个用系数表示的多项式转换成它的点值表示的算法
  - https://www.zhihu.com/zvideo/1572580452883202048 快速傅里叶变换(FFT):有史以来最巧妙的算法

算法延伸知识

- 阅读《算法导论》第30章 前两节，了解快速傅里叶变换如何应用于**加速**多项式相乘问题
  - **多项式的系数表示法和点值表示法**
  - 多项式的系数表示和点值表示可以**互相转换**



算法延伸知识

```
def FFT(P):
    # P - [p_0, p_1, ..., p_{n-1}] coeff representation
    n = len(P) # n is a power of 2
    if n == 1:
        return P
    ω = e^{2πi/n}
    P_e, P_o = [p_0, p_2, ..., p_{n-2}], [p_1, p_3, ..., p_{n-1}]
    y_e, y_o = FFT(P_e), FFT(P_o)
    y = [0] * n
    for j in range(n/2):
        y[j] = y_e[j] + ω^j y_o[j]
        y[j + n/2] = y_e[j] - ω^j y_o[j]
    return y
```

$$\text{FFT}\quad P(x):[p_0,p_1,\ldots,p_{n-1}],\quad \omega=e^{\frac{2\pi i}{n}}:[\omega^0,\omega^1,\ldots,\omega^{n-1}]$$

$$n=1\Rightarrow P(1)$$

$$\text{FFT}\quad P_e(x^2):[p_0,p_2,\ldots,p_{n-2}],\quad [\omega^0,\omega^2,\ldots,\omega^{n-2}]$$

$$y_e=[P_e(\omega^0),P_e(\omega^2),\ldots,P_e(\omega^{n-2})]$$

$$\text{FFT}\quad P_o(x^2):[p_1,p_3,\ldots,p_{n-1}],\quad [\omega^0,\omega^2,\ldots,\omega^{n-2}]$$

$$y_o=[P_o(\omega^0),P_o(\omega^2),\ldots,P_o(\omega^{n-2})]$$

$$P(\omega^j)=y_e[j]+\omega^j y_o[j])$$

$$P(\omega^{j+n/2})=y_e[j]-\omega^j y_o[j])$$

$$j\in\{0,1,\ldots(n/2-1)\}$$

$$y=[P(\omega^0),P(\omega^1),\ldots,P(\omega^{n-1})]$$