吉林大学

第四章 分治法

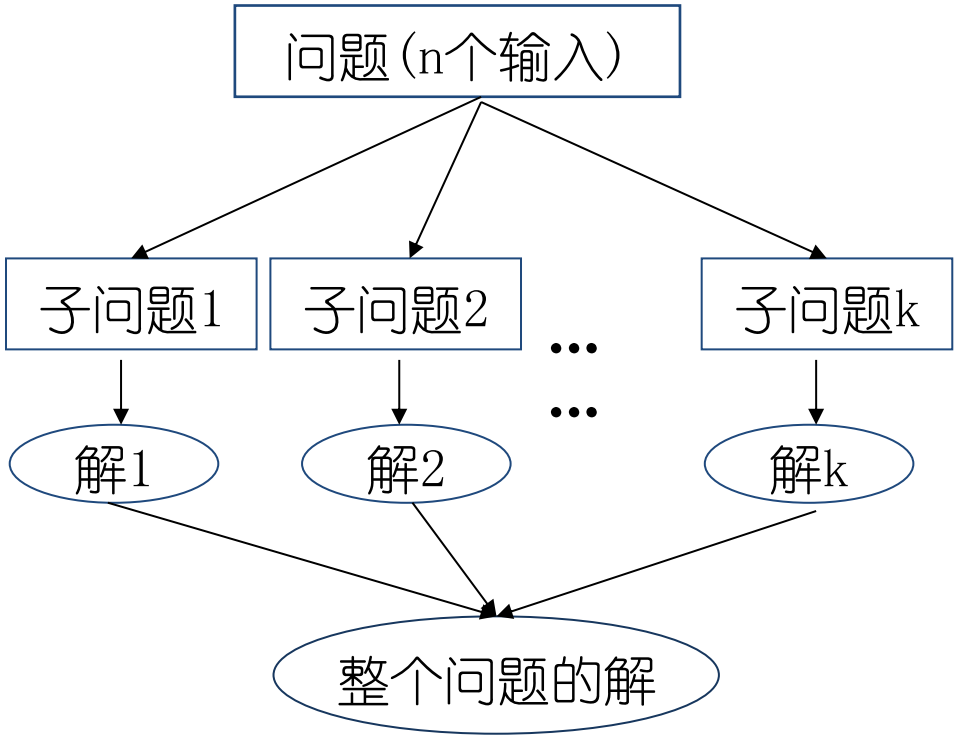
目录

4.1 一般方法

- 分治法适用的问题
- 分治法的求解思想
- 分治策略DANDC的抽象化控制
- 分治策略DANDC的计算时间

分治法适用的问题

- n取值相当大，直接求解往往非常困难，甚至无法求出。
- 将n个输入分解成k个不同子集，得到k个不同的可独立求解的子问题。
- 在求出子问题的解之后，能找到适当的方法把它们合并成整个问题的解。



分治法的求解思想

- 思想：将整个问题分成若干个小问题后分而治之。
  - 分(Divide)：子问题与原问题具有相同的特征，但规模更小。
  - 治(Conquer)：反复使用分治策略，直到可以直接求解子问题为止。
  - 合(Combine)：将子问题的解组合成原问题的解。

分治法天然适合递归实现

子问题的个数K?

- 显然有K>1, K越大越好么?
- 蚯蚓的故事
  - 蚯蚓一家这天很无聊，小蚯蚓想了想，把自己切成两段，打羽毛球去了。蚯蚓妈妈觉得这方法法不错，就把自己切成四段，打麻将去了。没过一会，蚯蚓爸爸就把自己切成了肉末。蚯蚓妈妈哭着说："你怎么那么傻，切得那么碎会死的。"蚯蚓爸爸弱弱地说："……突然想踢足球……"

分与合的代价

思考：子问题的规模相近或不相近是否会影响算法效率？

分治策略DANDC的抽象化控制

```
procedure DANDC(p,q)
global  n, A(1:n); integer  m, p, q;
 //1≤p≤q≤n//
if  SMALL(p,q)
    then  return(G(p,q))
    else  m←DIVIDE(p,q)
    return(COMBINE( DANDC(p,m),
    DANDC(m+1,q)))
endif
end DANDC
```

原问题为A(1:n)，调用函数DANDC(1,n)

- SMALL：判断输入规模q-p+1是否足够小，可直接求解
- G：求解该规模问题的函数
- DIVIDE：分割函数，p≤m≤q，原问题被分为A(p:m)和A(m+1:q)两个子问题
- COMBINE：合并函数，将两个子问题的解合并为原问题的解

分治策略DANDC的计算时间

- 假设所分成的两个子问题的输入规模大致相等，则DANDC的计算时间可表示为：

$$T(n)=\begin{cases} g(n) & n\text{足够小} \\ 2T(n/2)+f(n) & \text{否则} \end{cases}$$

- T(n) 是输入规模为n的分治策略的计算时间
- g(n)是对足够小的输入规模能直接计算出答案的时间
- f(n) 是COMBINE函数合成原问题解的计算时间

4.2 二分查找

- 问题描述
- 分治法建模
- 算法BINSRCH描述
- 算法正确性证明
- 算法实例
- 算法的计算时间分析
- 二元比较树
- 算法BINSRCH的时间复杂度
- 算法BINSRCH的变型
- 以比较为基础查找的时间下界

问题描述

- 查找问题：已知一个非降序排列$a_1, a_2, \ldots, a_n$（$\mathbf{a_1 \le a_2 \le \ldots \le a_n}$），判定给定元素x是否在其中。若是，则找出x位置，并将下标赋给变量j；否则，j置成0。

分治法建模

- 将查找问题表示为：$I=(n, a_1, \ldots, a_n, x)$
  - 分：选取一个下标k，可得到三个子问题
    - $I_1=(k-1, a_1, \ldots, a_{k-1}, x)$
    - $I_2=(1, a_k, x)$
    - $I_3=(n-k, a_{k+1}, \ldots, a_n, x)$
  - 治：
    - 若$x = a_k$，$j \leftarrow k$，算法结束
    - 若$x < a_k$，在子问题I_1中求解
    - 若$x > a_k$，在子问题I_3中求解
  - 合：
    - 当前子问题的解就是I的解

对所求解的问题（或子问题）所选的下标都是中间元素的下标

思考1：k= ？

思考2：合并策略是否正确？即为什么可以忽略掉其他子问题？

算法描述

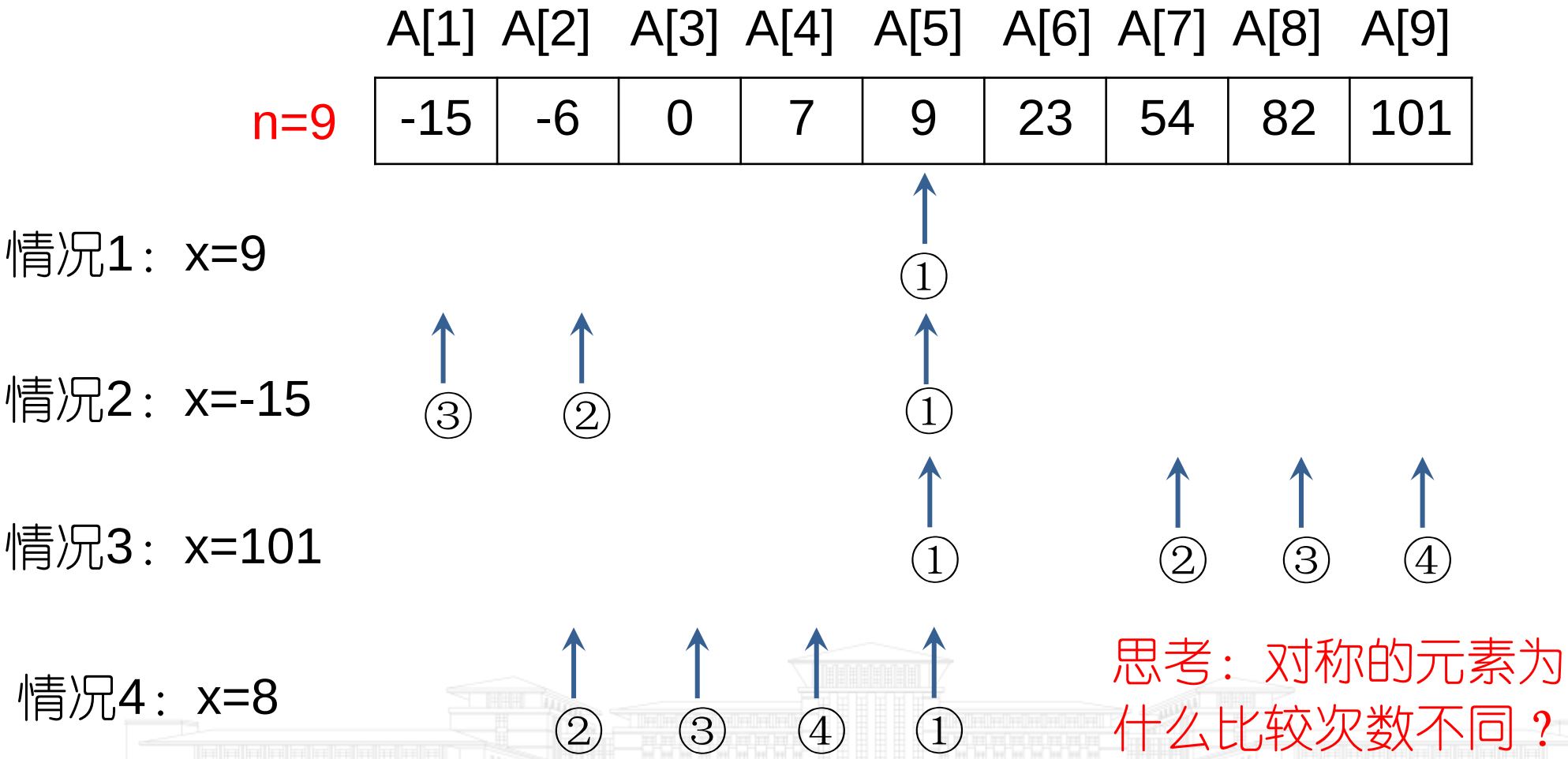
```
procedure BINSRCH(A, n, x, j)
  integer  low, high, mid, j, n;
  low←1;    high←n
  while low≤high do
          mid← |(low+high)/2|      //取中间值
        case
          :x<A(mid):  high←mid-1 //*在前一半寻找*/
          :x>A(mid):  low←mid+1  //*在后一半寻找*/
          :else:   j←mid;  return;    //*检索成功, 结束*/
        endcase
    repeat
  j←0                                  //查找失败
end BINSRCH
```

算法正确性证明

证明思路：
1检验到参数的每类取值
2检验到算法的每个分支

- 如果n=0，则不进入循环，j=0，算法终止。
- 否则就会进入循环与数组A中的元素进行比较。
- 成功情况：
  - 若x=A[mid]，则j=mid，查找成功，算法终止。
  - 否则，若x<A(mid)，则x根本不可能在mid~high,查找范围缩小到low~mid-1。反之，x>A(mid)时亦然。
- 不成功情况：
  - 因为low，high和mid都是整形变量，按算法所述方式缩小查找范围总可以在有限步内使low>high，说明x不在A中，退出循环，j=0，算法终止。

算法实例


A[1] A[2] A[3] A[4] A[5] A[6] A[7] A[8] A[9]
n=9
-15 -6 0 7 9 23 54 82 101
情况1：x=9
①
情况2：x=-15
③ ② ①
情况3：x=101
① ② ③ ④
情况4：x=8
② ③ ④ ①
思考：对称的元素为
什么比较次数不同？

算法的计算时间分析

- A有n个元素，x和A中元素比较
  - 成功查找：n种情况
  - 不成功查找：n+1种情况
- 分析算法时间复杂度
  - 成功查找：最好、平均、最坏情况下
  - 不成功查找：最好、平均、最坏情况下

算法的计算时间分析

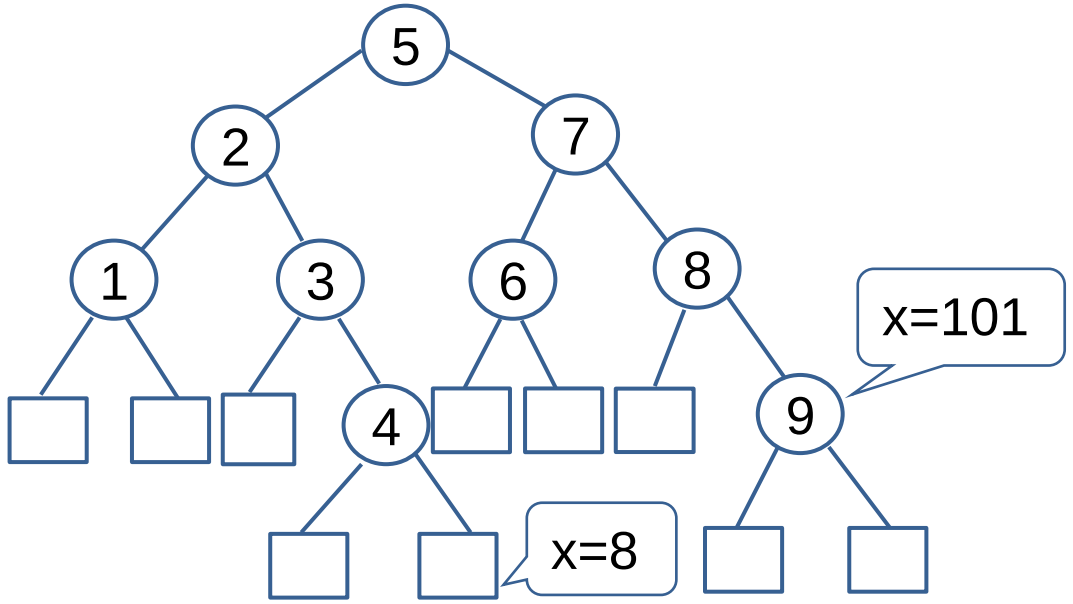
- 最好: 用最少的比较，就能找到x所在位置；
- 最坏：用最多的比较，才能找到x 所在位置；
- 平均：成功的所有情况都要考虑

不论检索是否成功，算法的执行过程都是x与一系列中间元素A(mid)的比较过程，可以用一棵二元比较树来描述算法的执行过程。

二元比较树

- 描述二分查找算法执行过程的二元比较树由内结点和外结点组成
  - 内结点记录一个mid值，表示一次元素的比较
  - 外结点表示不成功查找的一种情况
  - 一条路径表示一个元素的比较序列

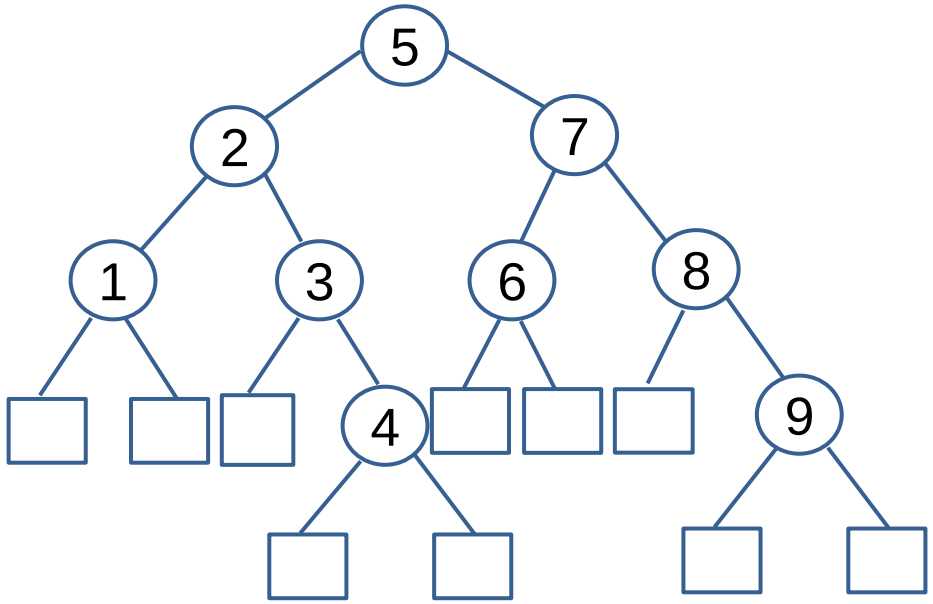
	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
n=9	-15	-6	0	7	9	23	54	82	101



思考：二元比较树的结构受问题实例影响么？

算法的执行过程实质上就是x与一系列中间元素A(mid)的比较过程。

二元比较树分析



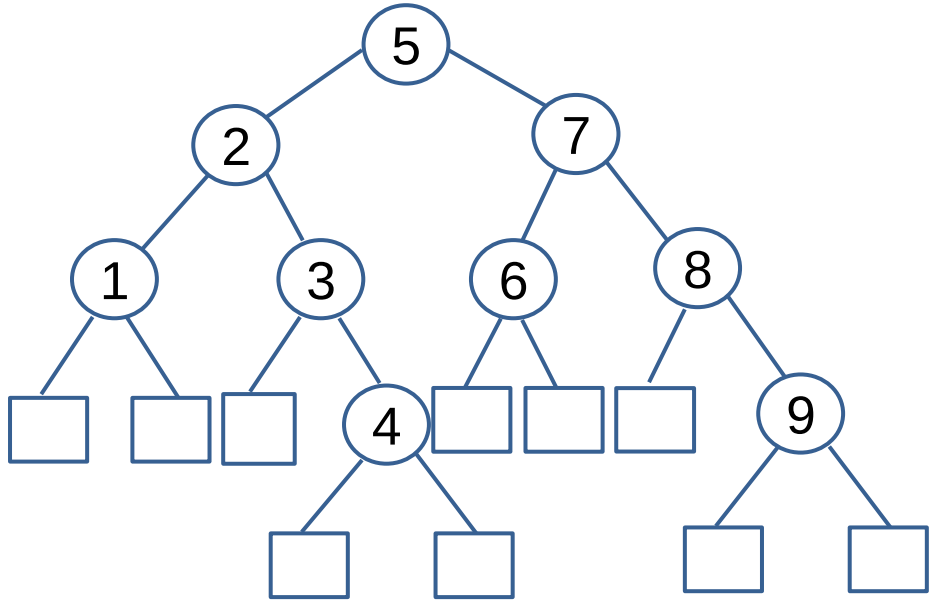
- 二元比较树分析：
  - 令根在1级，内节点最大级数为k
  - 成功：元素比较次数等于内节点级数，最少1次，最多k
  - 失败：元素比较次数等于外节点级数–1，最少k-1次，最多k次

思考：平均比较次数？

- n=9：
  - 成功比较总数=1+2+2+3+3+3+3+4+4=25
  - 失败比较总数=3+3+3+4+4+3+3+3+4+4=34

	最好	最坏	平均
成功	1	4	25/9
失败	3	4	34/10

二元比较树分析



- 内节点个数n与k之间的关系：
  - 内节点最大级数为k的二元比较树，其k-1层必为满。故$2^{k-1} \le n < 2^k$，则有

$$k-1 \le \log n \Rightarrow k \le \log n + 1, \text{又} k > \log n$$
$$\Rightarrow \log n < k \le \log n + 1 \Rightarrow k = \lfloor \log n \rfloor + 1$$

$k=\Theta(\log n)$

算法BINSRCH的时间复杂度分析

定理4.1：若n在区域$[2^{k-1}, 2^k)$中，则对于一次成功的查找，二分查找算法至多作k次比较，而对于一次不成功的查找，或者作k-1次比较或者作k次比较。

$k=\Theta(\log n)$

- 证明
  - 考察二分查找算法执行过程的二元比较树：成功查找在内结点终止，不成功查找在外结点终止。当$2^{k-1} \le n < 2^k$时，内结点级数≤k(根在1级)，外结点级数在k和k+1级。又知成功查找所需要的元素比较次数=级数，不成功查找所需要的元素比较次数=级数−1。定理得证。

思考：基于该定理可知算法在哪些情况下的时间复杂度？

成功：最好、最坏
失败：最好、最坏、平均

成功查找平均情况下的时间复杂度？

思想：利用内外节点到根的距离和之间的一种简单关系，
由不成功查找的平均比较数求出成功查找的平均比较数。

- 内部路径长度I：根到所有内节点的距离之和；
- 外部路径长度E：根到所有外节点的距离之和；
- 容易证明E=I+2n（见《数据结构》教材）；
- 成功查找的平均比较次数S(n)=(I/n)+1;
- 不成功查找的平均比较次数U(n)=E/(n+1);

$$E=(n+1)U(n)=I+2n=n(S(n)-1)+2n$$

$$\Rightarrow S(n)=\frac{n+1}{n}U(n)-1=(1+\frac{1}{n})U(n)-1$$

S(n)=Θ(U(n)) =Θ(logn)

算法BINSRCH的时间复杂度和空间复杂度

- 时间

计算时间	最好情况	最坏情况	平均情况
成功查找	Θ(1)	Θ(logn)	Θ(logn)
失败查找	Θ(logn)	Θ(logn)	Θ(logn)

- 空间
  - 用n个位置存放数组A，还有low, high, mid, x, j五个变量需要存储，共需空间 n+5= Θ(n)

算法结束后的思考

- 二分思想下的查找算法的时间复杂度是固定不变的么？时间复杂度会受到算法描述细节的影响吗？
- BINSRCH算法是解决查找问题的最优算法么？是否可能进一步优化？

算法BINSRCH的变型

- case语句用if语句来替换。一次关键字比较不是三种分支，而是两种分支。
- 最坏情况下，x总被比较两次。

```
procedure BINSRCH(A, n, x, j)
  integer  low, high, mid, j, n;
  low←1;    high←n
  while low≤high do
      mid ←⌊(low+high)/2⌋
      if x<A(mid)
        then high←mid-1
        else if x>A(mid)
                then low←mid+1
                else  j←mid; return;
             endif
      endif
repeat
j←0
end BINSRCH
```

算法BINSRCH的变型

最好、最坏和平均情况时间对于成功和不成功的查找都是Θ(logn)。

```
procedure BINSRCH1(A, n, x, j)
  integer  low, high, mid, j, n;
  low←1;    high←n+1 //high总比可能的取值大1
  while low<high-1 do
          mid←[(low+high)/2]
          if x<A(mid) //在循环中只比较一次
              then high←mid
              else low←mid //x>=A(mid)
          endif
    repeat
  if x=A(low) then j←low //x出现
                   else j←0 //x不出现
end BINSRCH1
```

思考：二分思想下的查找算法的时间复杂度是固定不变的么？时间复杂度会受到算法实现细节的影响吗？

以比较为基础查找的时间下界

- 对于已排序的n个元素，查找某元素x是否出现时，是否存在以比较为基础的查找算法，在最坏情况下该算法的计算时间比二分查找算法的计算时间更低？
  - 只允许进行元素间的比较；
    不允许对它们实施运算；
    则在这种条件下设计的算法都称为“以比较为基础的算法”

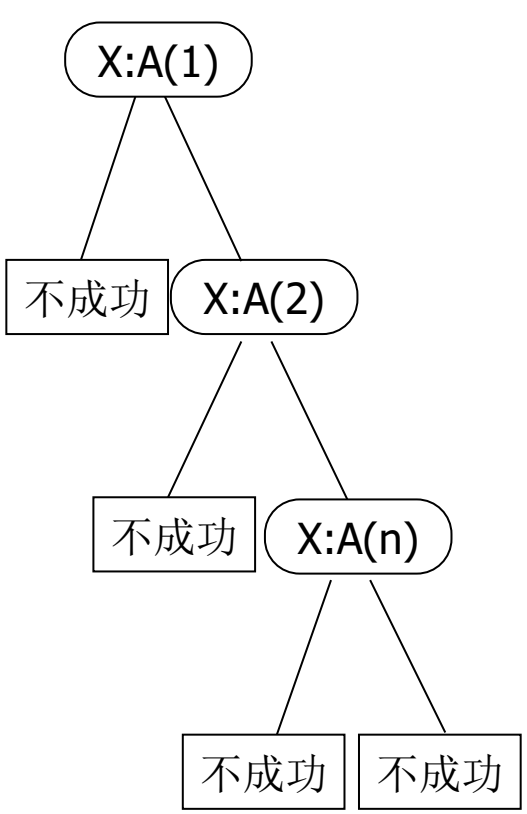
以比较为基础查找的时间下界

- 对于以比较为基础的检索算法的分析，采取构建二元比较树的方式。

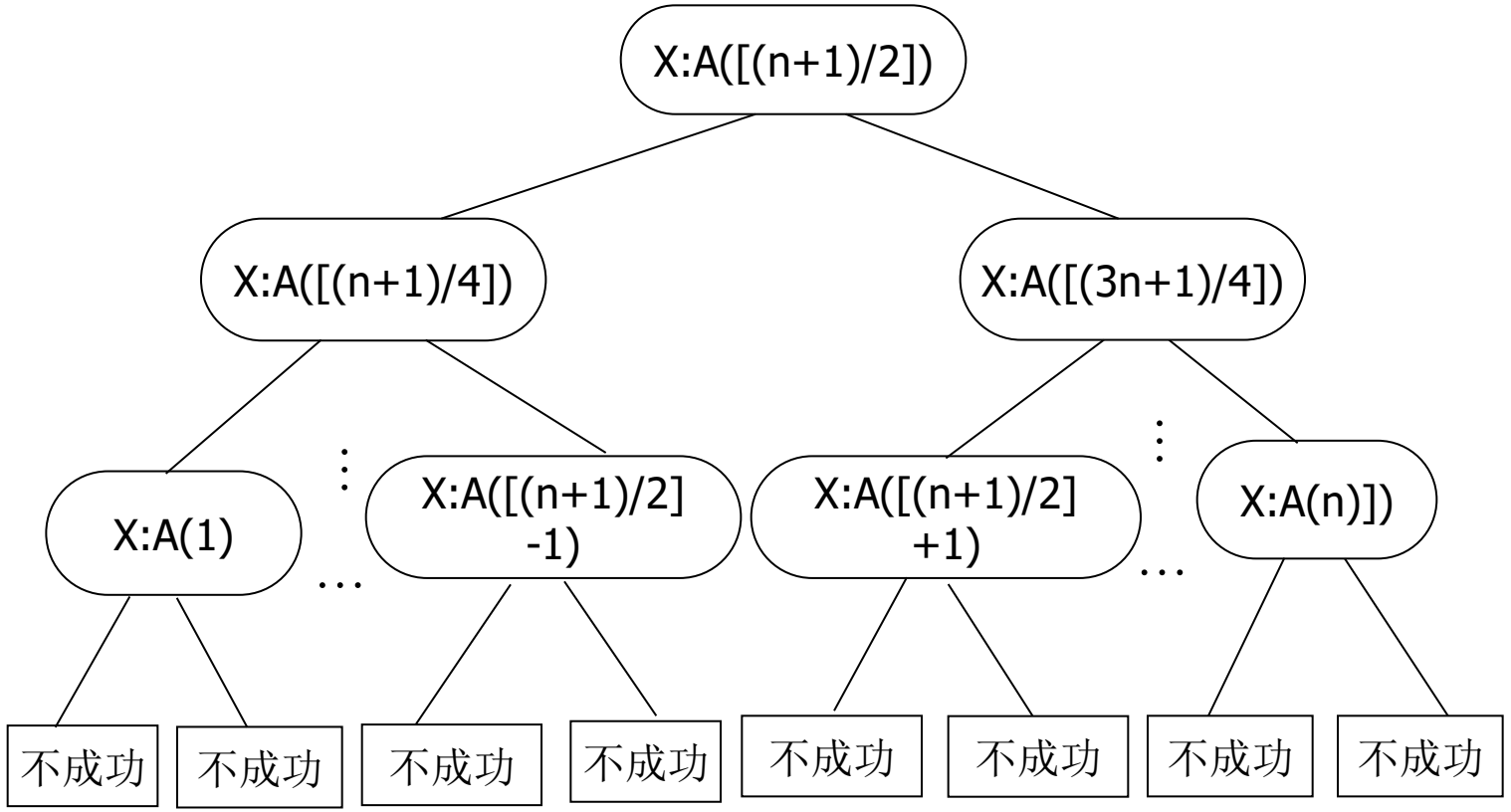
任何以比较为基础的检索算法，在执行过程中都可以用二元比较树来描述。

每个内节点表示一次元素比较，因此任何比较树中必须包含n个内节点，且分别与n个不同的值对应。每个外节点对应一次不成功检索。

［问题描述］有n个如下关系的元素： A(1)<A(2)…<A(n),查找某一给定元素X是否在A中出现。



线性查找



二分查找

以比较为基础查找的时间下界

定理4.2: 设A(1:n)含有n(n≥1)个不同的元素，排序为A(1)<A(2)<…<A(n)。又设以比较为基础去判断是否x∈A(1:n)的任何算法在最坏情况下所需的最小比较次数是FIND(n)，那么FIND(n)≥ [log(n+1)]

- 证明：对于任何以比较为基础的检索算法，都可以用二元树来表示，设树高为k。在这所有的树中都必定有n个内节点与x在A中可能的n种出现情况相对应。一棵二元树的所有内节点所在的级数小于或等于级数k，则最多有2^k-1个内节点。因此n<= 2^k-1 ，则有k≥log(n+1)，进一步k≥ [log(n+1)]

  所以有FIND(n)=k≥ [log(n+1)] 。证毕。

不存在其最坏情况下计算时间低于Θ (logn)的算法。
二分查找是解决查找问题最坏情况的最优算法

思考题

- 证明E=I+2n。
- 证明BINSRCH1的最好、最坏和平均情况时间对于成功和不成功的查找都是Θ(logn)。
- 三分（多分）查找会得到更优的结果吗？

4.3 归并排序

- 问题描述及一般方法思想(直接插入法)
- 归并排序算法思想、描述及实例
- 合并算法思想、描述及实例
- 归并排序算法的计算时间
- 归并排序算法的缺点及改进思想
- 以比较为基础的排序算法时间下界

问题描述及一般方法思想

- 问题描述：给定一个含有n个元素的集合，要求把它们按非降次序排列。
- 一般方法(直接插入法)

```
for j←2 to n do
    将A(j)放入已排序A(1:j-1)中
repeat
```

最好情况：O(n)
最坏情况：$O(n^2)$

分治策略设计归并排序算法

- 分：将A(1),…A(n)平均分为2个子集：A(1),…,A([n/2])和A([n/2]+1),…,A(n)
- 治：递归调用，将2个子集排序
- 合：将2个排好序的子集合并为一个有序集合

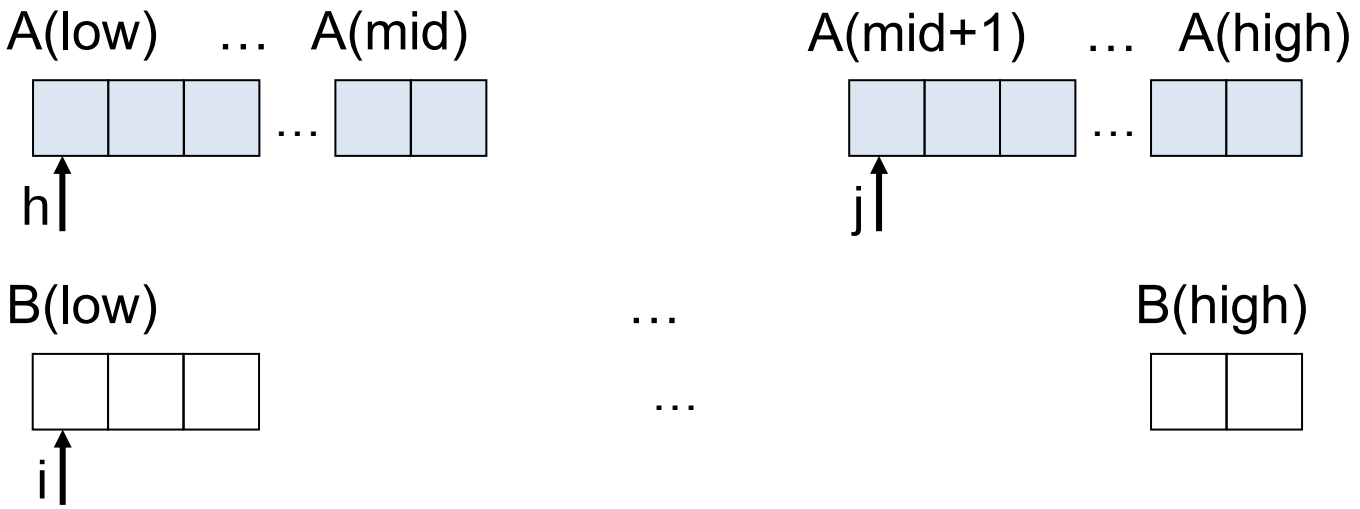
MERGESORT算法描述

```
Procedure MERGESORT(low,high)
int  low, high, mid;
if  low<high
    then   mid= ⌊(low+high)/2⌋
               call  MERGESORT(low,mid)
               call  MERGESORT(mid+1,high)
               call  MERGE(low,mid,high)
endif
end  MERGESORT
```

递归调用，分别对两个子问题排序

合并两个已排好的子集，得到原问题的解

MERGE算法思想



(1) 如果h ≤mid 且 j ≤high，那么B(i) ←min(A(h),A(j))

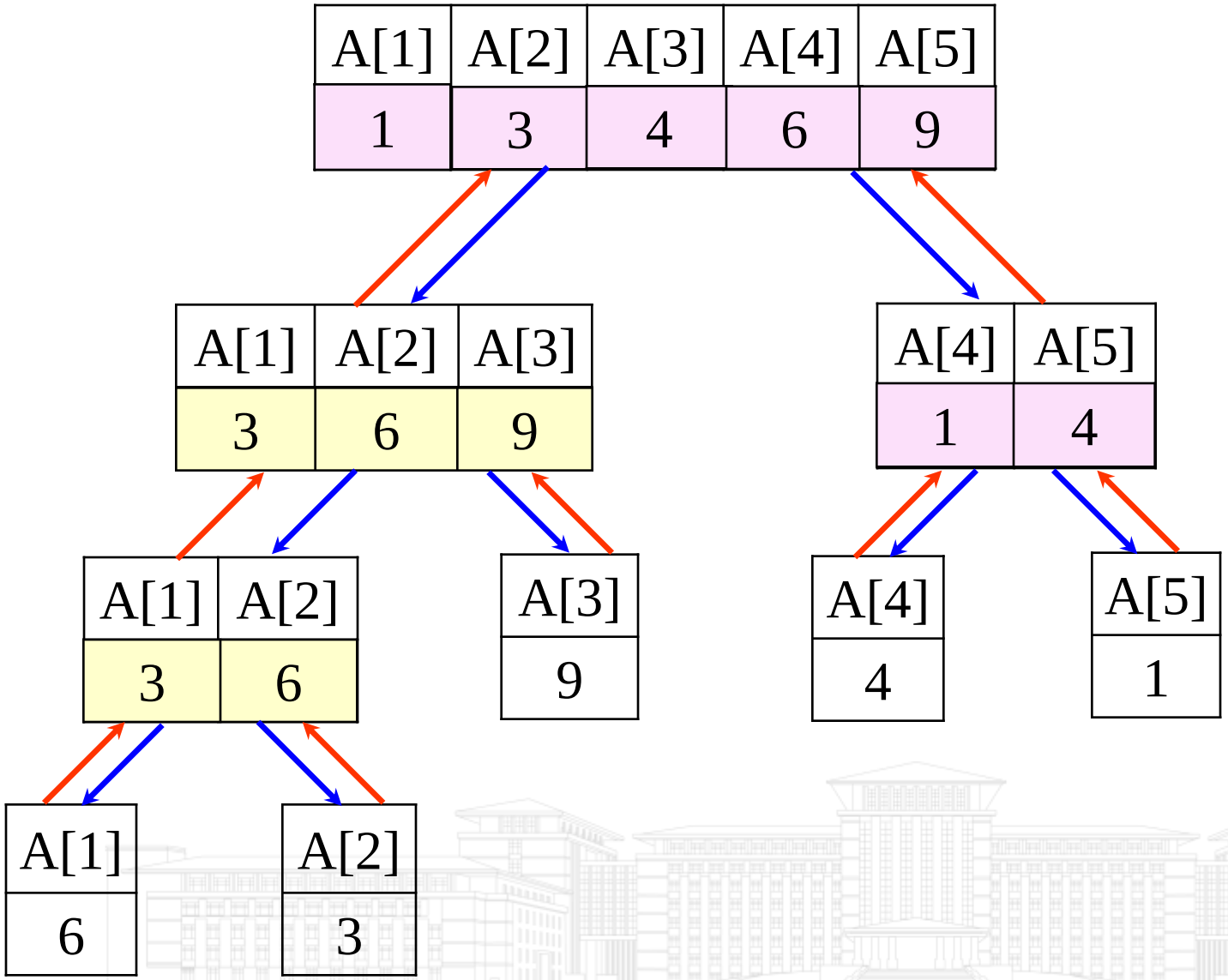
(2) 如果h >mid 且 j ≤high，那么B(i) ←A(j)

(3) 如果h ≤mid 且 j >high，那么B(i) ←A(h)

MERGE算法描述

```
procedure MERGE(low, mid, high)
integer h, i, j, k, low, mid, high; global A(low : high);   local B(low : high)
h ← low;  i ← low;  j ← mid+1;
// 处理两个已排好序的集合
while h≤mid  and  j≤high do
      if A(h) ≤A(j)  then B(i) ← A(h); h ← h+1
                     else  B(i) ← A(j);  j ← j+1; endif
      i ← i+1
repeat
//剩余元素处理过程
if h>mid  then for k←j to high do B(i)←A(k);i←i+1;repeat
                else for k←h to mid do B(i)←A(k);i←i+1;repeat
for k ← low to high do   A(k) ← B(k)
end  MERGE
```

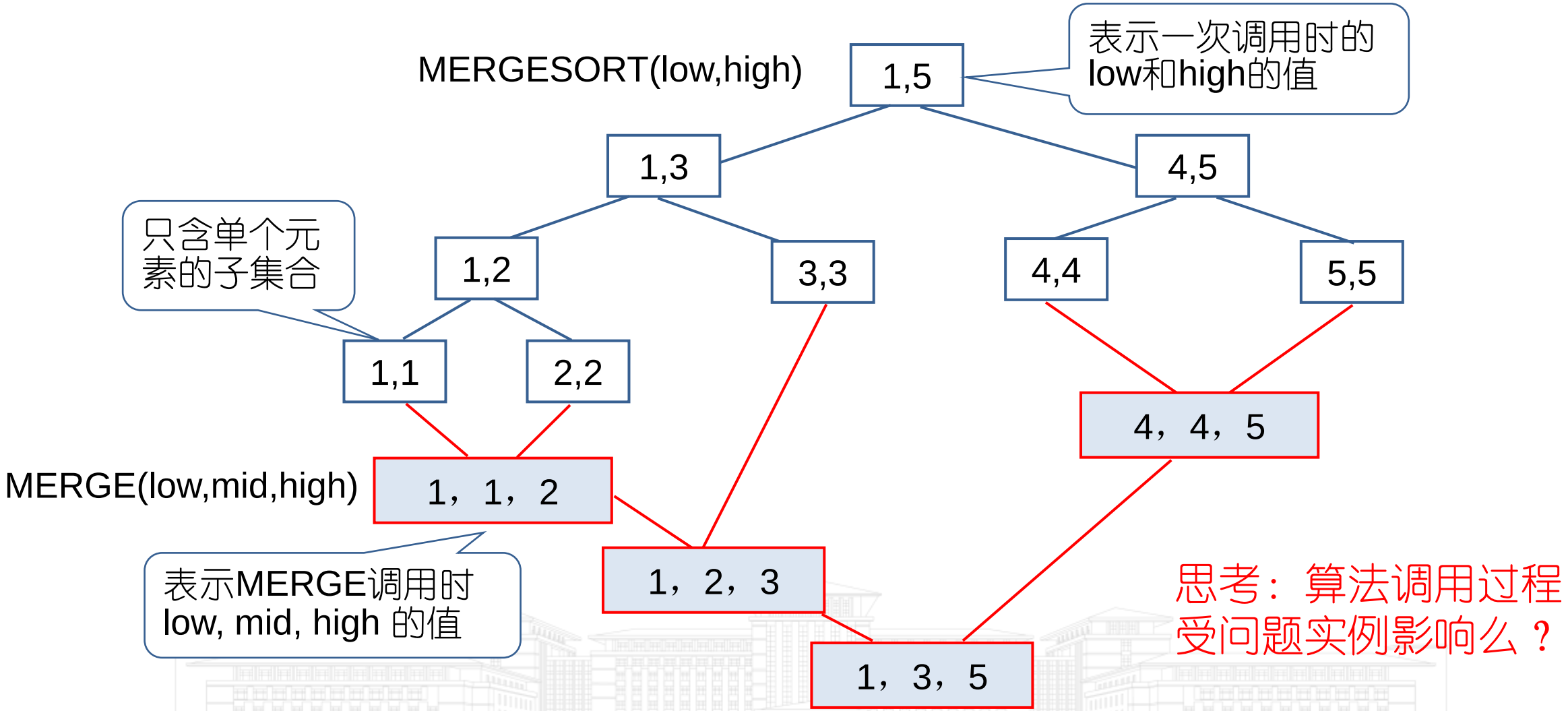
算法实例


A[1] A[2] A[3] A[4] A[5]
1 3 4 6 9
A[1] A[2] A[3]
3 6 9
A[4] A[5]
1 4
A[1] A[2]
3 6
A[3]
9
A[4]
4
A[5]
1
A[1]
6
A[2]
3

算法实例

A[1]	A[2]	A[3]	A[4]	A[5]
6	3	9	4	1

n=5



思考：算法调用过程受问题实例影响么？

归并排序的计算时间

$$T(n)=\begin{cases} a & n=1，a是常数 \\ 2T(n/2)+cn & n>1，c是常数 \end{cases}$$

归并2个子数组所需的元素比较次数在n/2到n-1之间

解：

当 $n=2^k$时，可得

$T(n) = 2(2T(n/4)+cn/2)+cn=2^2T(n/4)+2cn$

$= 2^2(2T(n/8)+cn/4)+2cn$

$= 2^3T(n/8)+3cn$

……

$= 2^kT(1)+kcn$

$= an+cnlogn$

如果$2^k<n \le 2^{k+1}$，显然有$T(n)\le T(2^{k+1})$

则有T(n)=O(nlogn)

归并排序算法的优化

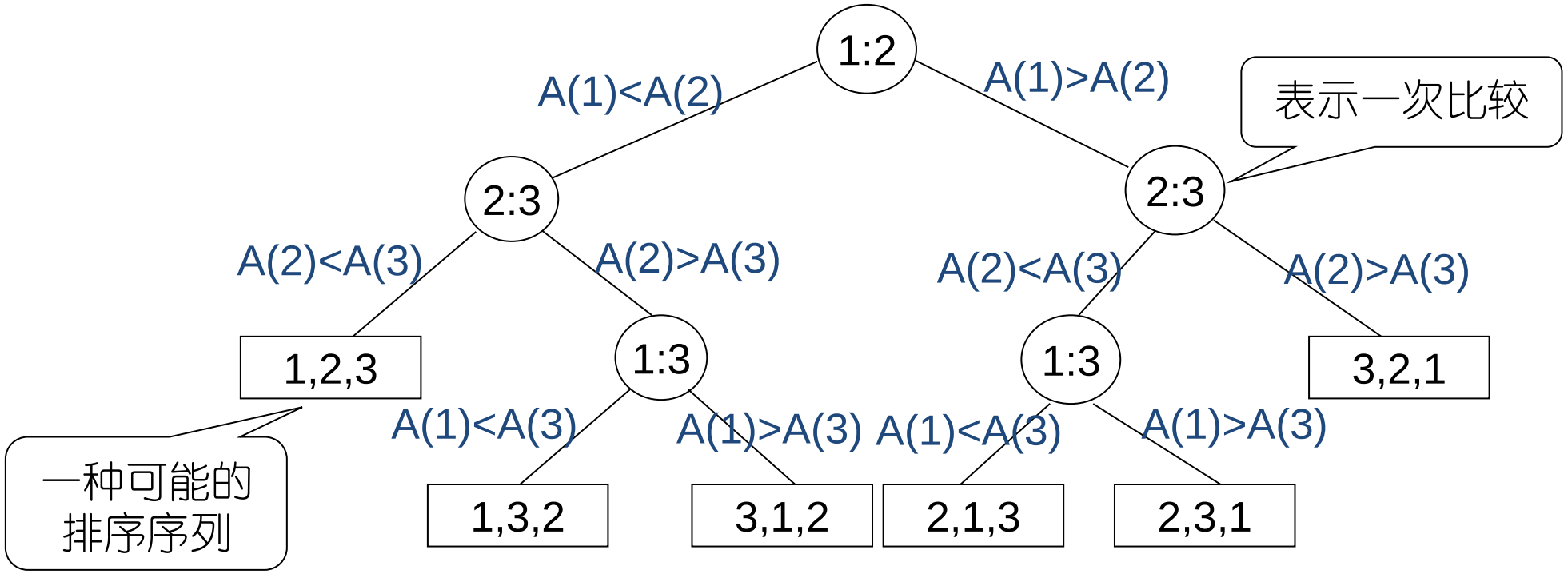
- 时间：递归处理消耗了很多时间。
  - 当子集合的元素个数很少时，采用直接插入算法减少时间消耗。
- 空间：辅助数组B增加了算法空间，每次调用MERGE时，B中结果复制到A中，消耗了一部分时间。
  - 用一个链接数组LINK(1:n)代替B，LINK中元素为A中元素的指针，它指向下一个元素所在的下标位置。

思考：能否采用自底向上的设计方式来取消对系统栈空间的需要？

以比较为基础排序的时间下界

- 任何以关键字比较为基础的排序算法，最坏情况下的时间下界都是Ω(nlogn)，因此从数量级角度看，归并算法是最坏情况下的最优算法。

- 假设参加排序的n个关键字A(1),…,A(n)各不相同。采用二元比较树记录关键字比较。



对3个关键字排序的二元比较树

考虑任何排序方法所对应的二元比较树
设该树的高度为k

- 则内节点的级数最大为k
- 则外节点的级数最大为k+1
- k+1级上的外节点最多有2^k个
- 从而有$n! \leq 2^k$
- 由于：一个级数为k+1的外节点，需要比较k次可以到达，如果设T(n)为该算法最坏情况下的比较次数，则有T(n)=k

往证T(n) = Ω(nlogn)

- 则必有$2^{T(n)} \geq n!$

$2^{T(n)} \geq n! \geq n(n-1)(n-2)\ldots(\lfloor n/2 \rfloor) \geq (n/2)^{n/2}$

$T(n) \geq (n/2)\log(n/2) \geq (n/4)\log n$

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

n≥4时，有
$n/2 \geq n^{1/2}$

任何以关键字比较为基础的排序算法，最坏情况下的时间下界都是Ω(nlogn)。

4.4 斯特拉森矩阵乘法

- 矩阵相乘问题
- 分治法求解矩阵相乘问题
- 斯特拉森矩阵乘法思想
- 斯特拉森矩阵乘法时间复杂度

矩阵相乘问题

- 假定矩阵A和B的级数n是2的幂，否则，添加适当的全零行和全零列。
- $A_{n\times n}$和$B_{n\times n}$的乘积矩阵$C_{n\times n}$中的元素C[i,j]定义为:

$$C[i][j]=\sum_{k=1}^{n}A[i][k]B[k][j]$$

- 直接求解的时间复杂度
  - 令乘法运算为基本运算
  - 一个元素C[i][j]需要做n次乘法和n-1次加法
  - 矩阵C一共有n^2个元素

$O(n^3)$

1969年之前

- 做过很多尝试，以改进矩阵乘法的效率
- 设计了一些改进算法，可惜在计算时间上，始终囿界于n^3这个数量级
- 1969年，德国人Volker Strassen利用分治法+处理技巧，计算矩阵相乘
- 计算时间为$O(n^{2.81})$

正常计算两个2×2矩阵A和B的乘积C时，

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

8次乘法，
4次加法

斯特拉森发现，可以减少乘法的次数

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

$m_1 = (a_{11} + a_{22})(b_{11} + b_{22})$

$m_2 = (a_{21} + a_{22})b_{11}$

$m_3 = a_{11}(b_{12} - b_{22})$

$m_4 = a_{22}(b_{21} - b_{11})$

$m_5 = (a_{11} + a_{12})b_{22}$

$m_6 = (a_{21} - a_{11})(b_{11} + b_{12})$

$m_7 = (a_{12} - a_{22})(b_{21} + b_{22})$

7次乘法，
18次加减法

蛮力算法：8次乘法，4次加减法

斯特拉森算法：
7次乘法，18次加减法

这并不能吸引我们用斯特拉森算法来计算2 ×2矩阵乘积。

该算法的重要性在于：

当矩阵的阶很大时，算法的效率才能显现出来!

Strassen算法的分治思想

将矩阵A，B和C中每一矩阵都分块成4个大小相等的子矩阵。

以n×n矩阵A为例：

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

每个子矩阵是(n/2) ×(n/2)的矩阵

若n不能被2整除，怎么办？

增加适当的全零行和全零列

分治法求矩阵相乘问题

8次(n/2) ×(n/2)矩阵乘法
4次(n/2) ×(n/2)矩阵加法

- 将A和B都分成4个(n/2)*(n/2)的方形矩阵。

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

- 上述分治法求解的时间复杂度 $T(n) = \begin{cases} b & n = 2 \\ 8T(n/2) + dn^2 & n > 2 \end{cases}$

$O(n^3)$

4次子矩阵相加

子问题个数多

思考：当前分治法没有提高求解效率的原因是什么？

斯特拉森矩阵乘法思想

- 斯特拉森在分治法的基础上，利用技巧减少了子问题的个数
  - 用7个乘法和10个加(减)法来算出7个(n/2)*(n/2)的中间矩阵
  - 用8个加(减)法算出子问题的解C_{ij}

$$M_1 = A_{11}(B_{12} - B_{22})$$
$$M_2 = (A_{11} + A_{12})B_{22}$$
$$M_3 = (A_{21} + A_{22})B_{11}$$
$$M_4 = A_{22}(B_{21} - B_{11})$$
$$M_5 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$M_6 = (A_{12} - A_{22})(B_{21} + B_{22})$$
$$M_7 = (A_{11} - A_{21})(B_{11} + B_{12})$$

$$C_{11} = M_5 + M_4 - M_2 + M_6$$
$$C_{12} = M_1 + M_2$$
$$C_{21} = M_3 + M_4$$
$$C_{22} = M_5 + M_1 - M_3 - M_7$$

斯特拉森矩阵乘法时间复杂度

$$T(n)=\begin{cases} O(1) & n=2 \\ 7T(n/2)+O(n^2) & n>2 \end{cases}$$

令n=2^k

$$T(n) = 7T(n/2) + an^2$$

$$= 7(7T(n/4) + a(n/2)^2) + an^2$$

$$= 7^2T(n/4) + an^2(\frac{7}{4} + 1)$$

$$= 7^3T(n/8) + an^2((\frac{7}{4})^2 + \frac{7}{4} + 1)$$

$$= 7^4T(n/16) + an^2((\frac{7}{4})^3 + (\frac{7}{4})^2 + \frac{7}{4} + 1)$$

$$= \ldots$$

$$= 7^kT(n/2^k) + an^2((\frac{7}{4})^{k-1} + \ldots + \frac{7}{4} + 1)$$

$$T(n) = 7^k T(n/2^k) + an^2((\frac{7}{4})^{k-1} + \ldots + \frac{7}{4} + 1)$$

$$= 7^k + an^2 \frac{(\frac{7}{4})^k - 1}{\frac{7}{4} - 1} \leq 7^k + cn^2(\frac{7}{4})^k$$

$$= 7^{\log n} + cn^2(\frac{7}{4})^{\log n} = n^{\log 7} + cn^2 n^{(\log 7 - \log 4)}$$

$$= n^{\log 7} + cn^{\log 7} = (1+c)n^{\log 7} = O(n^{\log 7})$$

$$\approx O(n^{2.81})$$

总结

- 分治法不一定总会提高算法效率
- 减少子问题个数是降低时间复杂度的一个有效途径

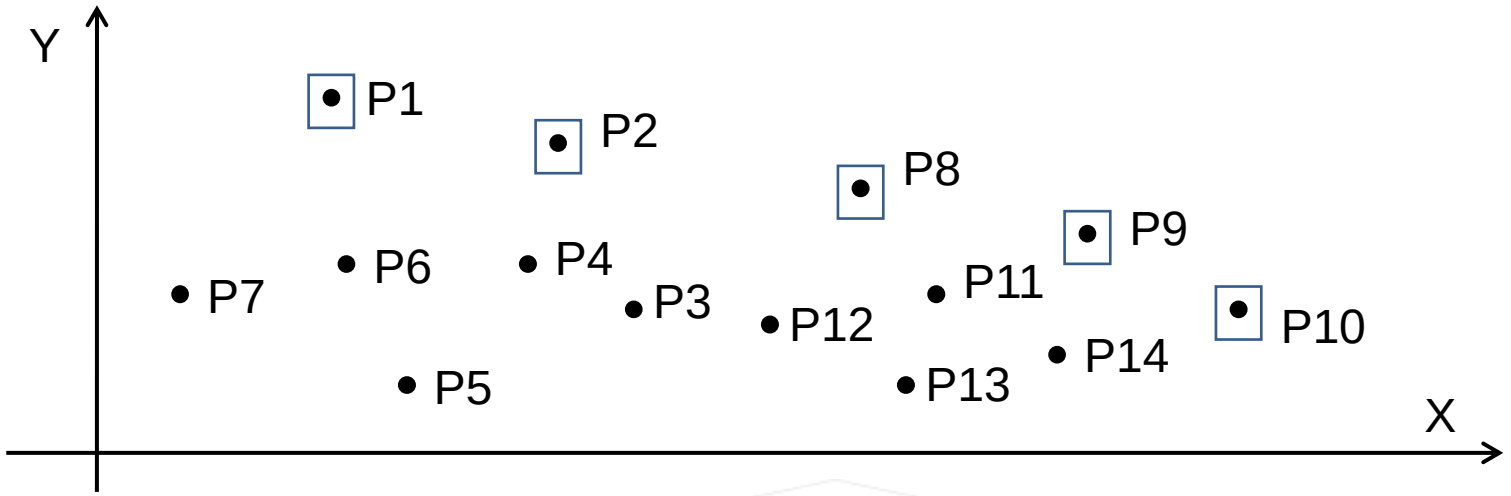
Hopcroft和Kerr已经证明(1971)，计算2个2×2矩阵的乘积，7次乘法是必要的。因此，要想进一步改进矩阵乘法的时间复杂性，就不能再基于计算2×2矩阵的7次乘法这样的方法了。或许应当研究3×3或5×5矩阵的更好算法。

4.5 二维极大点问题

- 问题描述
- 分治法设计思想
- 分治法描述
- 时间复杂度

问题描述

- 支配规则：在二维空间中，如果$x_1>x_2$，且$y_1>y_2$，那么称点(x_1,y_1)支配点(x_2,y_2)
- 极大点：如果一个点没有被其他点支配，则称其为极大点。
- 求极大点问题：已知有n个点的集合，找出其中的所有极大点。



- 直接比较每一对点，时间复杂度：$O(n^2)$

分治法设计思想

- 分：设计中位线L，将整个点集分为两个子集S_L和S_R
- 治：分别找出S_L和S_R的极大点
- 合：基于支配规则合并S_L和S_R的极大点

- 思考1：怎样确定中位线？
- 思考2：怎样设计递归出口？
- 思考3：怎样实现支配规则？

问题分析

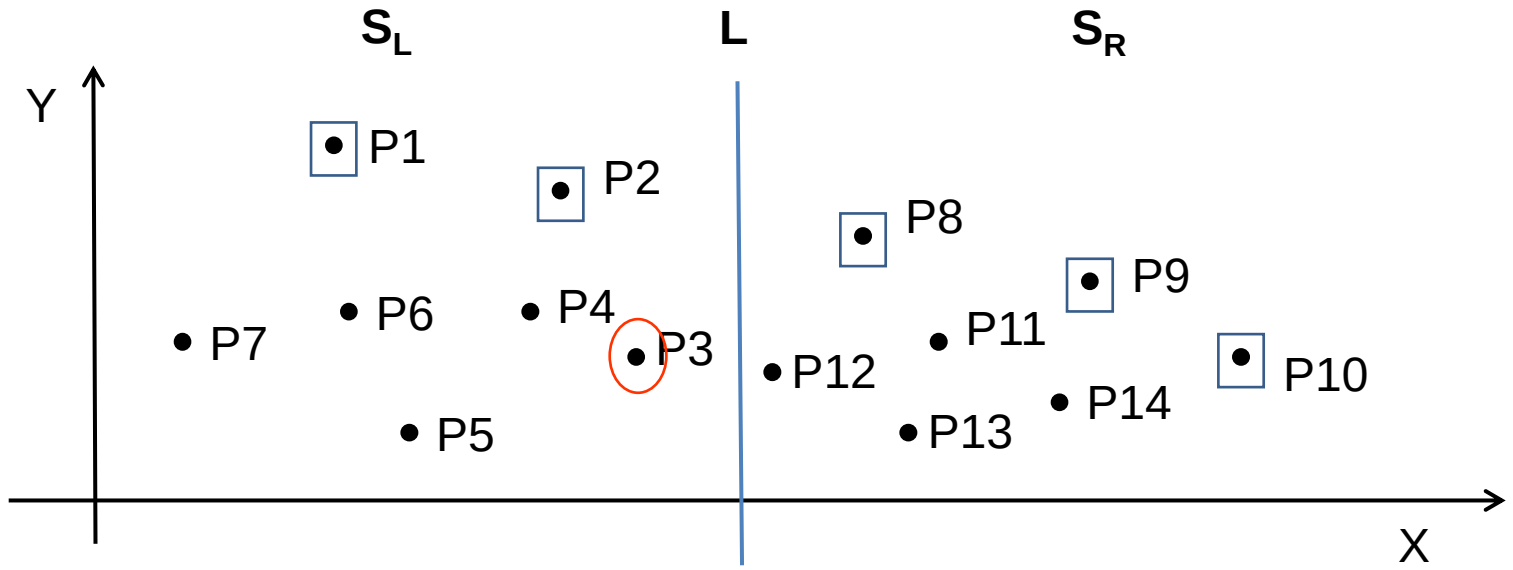
可以是任何曲线，要选择最简单的方式，有利于提高效率。

- 中位线：
  - 垂直于X轴的中位线L
  - 对集合S中的所有点按x值排序后，寻找第n/2点位置
- 递归出口：
  - 如果集合S中只有一个点，那么该点为极大点
- 基于支配规则合并：
  - 对于S_L中的极大点p，如果y_p小于S_R中极大点的y值，则p被支配，舍弃掉

S_L中的点的x值总是小于S_R中的点的x值

分治法描述：求二维极大点问题

- input：n个平面点的集合S
- output：S的极大点
- step1：如果S只有一个点，输出该点为极大点；否则，寻找中位线L，将S划分为两个子集合S_L和S_R，使每个子集含有n/2个点。
- step2：递归找出S_L和S_R的极大点。
- step3：判断S_L的所有极大点$p \in S_L$，如果y_p值小于S_R中极大点的y值，那么舍弃点p。



$$T(n)=\begin{cases}1 & n=1\\ 2T(n/2)+\text{中位线L的计算时间}+\text{基于支配规则合并的时间} & n>1\end{cases}$$

治

分：O(nlog(n))

合：O(n)

Ω(nlogn)

预处理：提前排序

思考1：能够优化中位线的计算时间？是否有必要每次求解子问题都重新排序？

时间复杂度

$$T(n)=\begin{cases}1 & n=1\\ 2T(n/2)+O(n)+O(n) & n>1\end{cases}$$

治　分　合

O(nlogn)

工作量放到递归过程之外，降低重复处理

- 具有预排序的分治策略找极大点的时间复杂度是O(nlogn)
  - 预排序O(nlogn)
  - 分治法O(nlogn)
- 优于直接方法的时间复杂度$O(n^2)$

4.6 分治法的优化

- 效率低的原因
- 优化策略1
- 优化策略2

效率低的原因

- 一些情况下，分治法比蛮力法(直接求解)明显改进了效率，但是也并不是处处有效
  - 原因1：子问题个数多，发生在治的环节
  - 原因2：递归过程内工作量过多，发生在分/合的环节

优化思路：变多为少

优化策略1

- 针对子问题个数多的问题
  - 寻找子问题之间的依赖关系，如果一个子问题的解可以通过其他子问题的解简单运算得到，那么不必重新递归计算该子问题的解，而是通过上述简单运算获得。
    - 斯特拉森矩阵相乘问题：矩阵相乘减少一个子问题个数
    - 归并排序问题：减少小规模下子问题个数

优化策略2

- 针对递归过程内工作量过多的问题
  - 把某些工作提取到递归过程之外预处理，从而减少递归内部的调用工作量
    - 二维极大点问题：预排序x值，求子问题时查询该序列，确定中位线

4.7主定理

- 什么是主定理
- 理解主定理
- 分治法的效率分析

什么是主定理（求解时间复杂度）

- 主定理：设a ≥1,b>1为常数，f(n)为函数，T(n)为非负整数，且 $T(n)=aT(n/b)+f(n)$，则T(n)有如下渐近界：

1.若∃常数$\varepsilon>0$,使得$f(n)=O(n^{\log_b a-\varepsilon})$，则$T(n)=\Theta(n^{\log_b a})$

f(n)的数量级低于 $n^{\log_b a}$

2.若$f(n)=\Theta(n^{\log_b a})$，则$T(n)=\Theta(n^{\log_b a}\log n)$

f(n)的数量级等于 $n^{\log_b a}$

3.若∃常数ε，使得$f(n)=\Omega(n^{\log_b a+\varepsilon})$,

且∃常数$c<1$,且n足够大时，有$af(n/b)\le cf(n)$,

则$T(n)=\Theta(f(n))$

f(n)的数量级高于 $n^{\log_b a}$

说明a个小规模的f函数，可以将规模放大

理解主定理

1：斯特拉森矩阵的计算时间 a=7,b=2,f(n)=Θ(n^2)

$$T(n)=\begin{cases} O(1) & n=2 \\ 7T(n/2)+O(n^2) & n>2 \end{cases}$$

O(nlog7) =O($n^{2.81}$)

2：归并排序的计算时间 a=2,b=2,f(n)=Θ(n)

$$T(n)=\begin{cases} a & n=1\text{，a是常数} \\ 2T(n/2)+cn & n>1\text{，c是常数} \end{cases}$$

O(nlogn)

分治法的效率分析

- 主定理：　在通用分治递归式T(n)=aT(n/b)+f(n)中，有

$$f(n) = \Theta(n^d)$$

a是子问题的个数

合并所需的时间规模

$$T(n) = \begin{cases} \Theta(n^d), \text{当} a < b^d \text{时} \\ \Theta(n^d \log n), \text{当} a = b^d \text{时} \\ \Theta(n^{\log_b^a}), \text{当} a > b^d \text{时} \end{cases}$$

b是原问题与子问题的规模比例

4.8 小结

- 分治法特点
  - 分-治-合
  - 子问题独立：两个子问题不会共享一个问题域更小的公共子问题
  - 自顶向下将原问题分解成子问题

4.8 小结

- 4.1 一般方法
  - 掌握分治法适用的问题特点、求解思想和计算时间等基本内容。能掌握分治法求解问题的一般过程。
- 4.2 二分查找
- 4.3 归并排序
  - 掌握分治算法的一般设计思想，算法正确性证明、时间复杂度分析和求解的一般方法。

4.8 小结

- 4.4 斯特拉森矩阵乘法
- 4.5 二维极大点问题
- 4.6 分治法的优化
  - 深入理解分治法适用的问题特征，掌握复杂问题求解办法。分析分治法时间复杂度的影响因素，掌握分治算法的优化技巧。掌握分治算法优化的一般原理和策略。
- 4.7 主定理
  - 理解主定理。

能够识别出适合分治法的可计算性问题、独立设计算法和分析算法复杂度。

本 章 结 束