《计算机算法设计与分析》

第三章 分治法

马丙鹏

2020年09月22日

中国科学院大学
University of Chinese Academy of Sciences

第三章 分治法

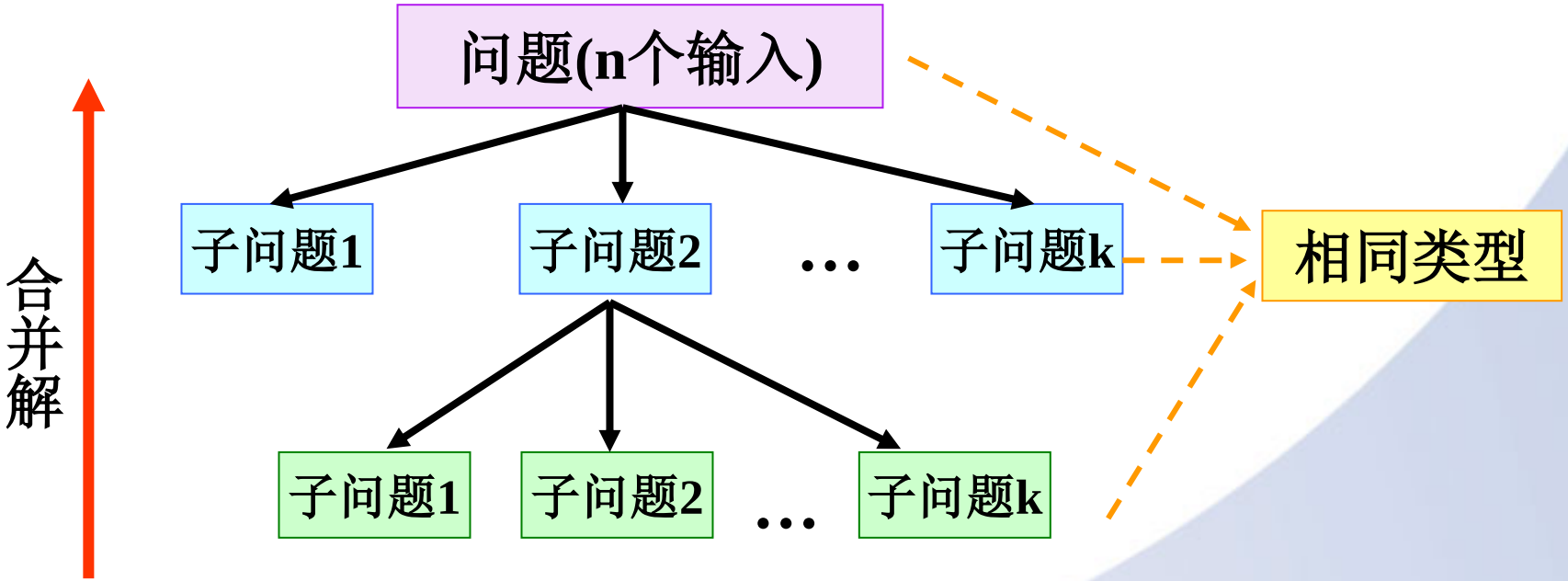
- 3.1 一般方法
- 3.2 二分检索
- 3.3 找最大和最小元素
- 3.4 归并排序
- 3.5 快速排序
- 3.6 选择问题
- 3.7 斯特拉森矩阵乘法

3.1 一般方法

- 基本思想
  - 为解决一个大问题，可以
    - ① 把它分解成两个或多个更小的问题；
    - ② 分别解决每个小问题；
    - ③ 把各小问题的解答组合起来，可得到原问题的解。
  - 小问题通常与原问题相似或同质，因而可以递归地使用分而治之策略解决。

3.1 一般方法

■ 基本思想



3.1 一般方法

■ 分治策略的抽象化控制

```
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)  //p≤m＜q//
      return(COMBINE(DANDC(p, m),
                     DANDC(m+1, q)))
  endif
end DANDC
```

- k=2: 二分是最常用的分解策略；
- SMALL(p, q)：布尔函数，判断输入规模q-p+1是否足够小而无需再进一步分就可求解；
- G(p, q)：当SMALL(p, q)为真时，对输入规模为q-p+1的子问题求解；
- DIVIDE(p, q)：当规模q-p+1还较大时，对规模为q-p+1的子问题进一步分解，返回值为[p, q]区间进一步的分割点；
- COMBINE(x, y)：子结果的合并函数，将区间[p, m]和[m+1, q]上的子问题的解合并成区间[p, q]上的“较完整”的解。当p=1, q=n时，就得到整个问题的解。

3.1 一般方法

■ DANDC的计算时间

- 若所分成的两个子问题的输入规模大致相等，则DANDC总的计算时间可用递归关系式表示，如下：

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

- 注：
  - T(n)：输入规模为n的DANDC计算时间
  - g(n)：对足够小的输入规模直接求解的计算时间
  - f(n)：COMBINE对两个子区间的子结果进行合并的时间
  - 该公式针对具体问题有各种不同的变形

第三章 分治法

- 3.1 一般方法
- 3.2 二分检索
- 3.3 找最大和最小元素
- 3.4 归并排序
- 3.5 快速排序
- 3.6 选择问题
- 3.7 斯特拉森矩阵乘法

3.2 二分检索

- 问题描述
  - 已知一个按非降次序排列的元素表$a_1, a_2, \ldots, a_n$,
  - 判定某给定的元素x是否在该表中出现。
    - 若是，则找出x在表中的位置并返回其所在下标;
    - 若非，则返回0值。

3.2 二分检索

- 分治求解策略分析：
  - 定义问题的形式描述：$I=(n, a_1, a_2, \ldots, a_n, x)$
  - 问题分解：选取下标k，由此将I分解为3个子问题：

    $I1=(k-1, a_1, a_2, \ldots, a_{k-1}, x)$ ← 规模较大

    $I2=(1, a_k, x)$ ← 规模足够小

    $I3=(n-k, a_{k+1}, a_{k+2}, \ldots, a_n, x)$ ← 规模较大
  - 对于I2，若$a_k=x$，则有解k，对I1、I3不用再求解；
  - 否则
    - 若$x<a_k$，则只在I1中求解，对I3不用求解；
    - 若$x>a_k$，则只在I3中求解，对I1不用求解；
  - I1、I3上的求解可再次采用分治方法划分后求解(递归过程)

3.2 二分检索

- 二分检索算法
  - 给定一个按非降次序排列的元素数组A(1:n)，n≥1，判断x是否出现。
  - 若是，置j，使得x=A(j)
  - 若非，j=0

算法3.3 二分检索

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

3.2 二分检索

■ 算法轨迹示意

□例3.1：设A(1:9)=(-15，-6，0，7，9，23，54，82，101)，在A中检索x=101，-14，82。

表1 运行轨迹示例

x=101			x=-14			x=82		
low	high	mid	low	high	mid	low	high	mid
1	9	5	1	9	5	1	9	5
6	9	7	1	4	2	6	9	7
8	9	8	1	1	1	8	9	8
9	9	9	2	1	找不到			找到
		找到						

成功的检索　　不成功的检索　　成功的检索

3.2 二分检索

- 空间性能分析
  - n+5个空间位置(A, x, j, mid, low, high)——O(n)
- 时间性能分析
  - 对于计算时间，需要分别考虑以下几种情况：
    - 成功检索的最好情况、平均情况、最坏情况
    - 不成功检索的最好情况、平均情况、最坏情况
    - 最好、最坏、平均检索情况——与数据配置有关
  - 成功检索：所检索的x出现在A中。
    - 成功检索情况共有n种: x恰好是A中的某个元素，A中共有n个元素，故有n种可能的情况

3.2 二分检索

- 时间性能分析
  - 不成功检索：所检索的x不出现在A中。
    - 不成功检索情况共有n+1种：x<A(1),或A(i)<x<A(i+1), 1≤i<n-1或x>A(n)
  - 成功/不成功检索的最好情况：
    - 最少几次能达到目的
    - 执行步数最少，计算时间最短
    - 成功检索的最好情况：若x恰好是A中的某个元素，最少几次确定x在A中的出现位置：1次
    - 不成功检索的最好情况：若x不是A中的任何元素，最少几次能判断出这一结论

3.2 二分检索

- 时间性能分析
  - 成功/不成功检索的最坏情况：
    - 最多几次能达到目的
    - 执行步数最多，计算时间最长
  - 成功/不成功检索的平均情况：
    - 典型数据配置下的执行情况，为一般情况下计算时间的平均值

3.2 二分检索

■ 实例分析(例3.1)

□频率计数特征

1. while循环体外语句的频率计数为1
2. 集中考虑while循环中x与A中元素的比较运算——其它运算的频率计数与比较运算有相同的数量级。
3. case语句的分析：假定只需一次比较就可确定case语句控制是三种情况的哪一种。

算法3.3 二分检索

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

3.2 二分检索

- 实例分析(例3.1)
  - 频率计数特征
    - 查找每个元素所需的元素比较次数如下：

	A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	A(7)	A(8)	A(9)
	-15	-6	0	7	9	23	54	82	101
成功：	3	2	3	4	1	3	2	3	4
不成功：	3	3	3	4	4	3	3	3	4	4

3.2 二分检索

- 实例分析(例3.1)
  - 成功检索
    - 最好：1次
    - 最坏：4次
    - 平均：(3+2+3+4+1+3+2+3+4)/9≈2.77次
  - 不成功检索
    - 最好：3次
    - 最坏：4次
    - 平均：(3+3+3+4+4+3+3+3+4+4)/10 = 3.4次

3.2 二分检索

■ 二元比较树

□算法执行过程的主体是x与一系列中间元素A(mid)比较。可用一棵二元树描述这一过程，并称之为二元比较树。

●结点：分为内结点和外结点两种

➢内结点：代表一次元素比较

用圆形结点表示

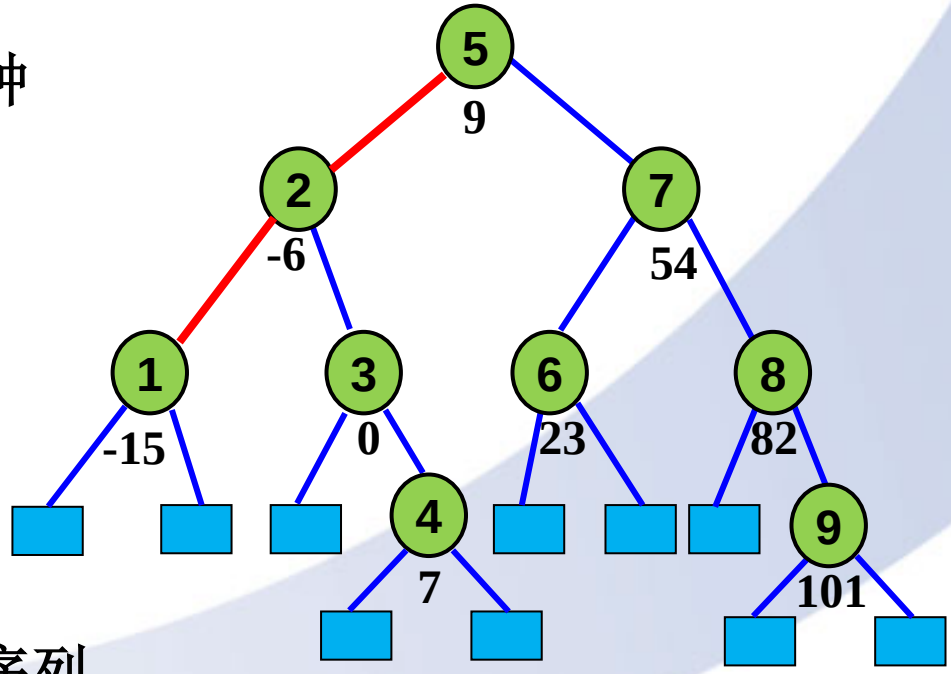
存放一个 mid值(下标)

代表成功检索情况

➢外结点：用方形结点表示，

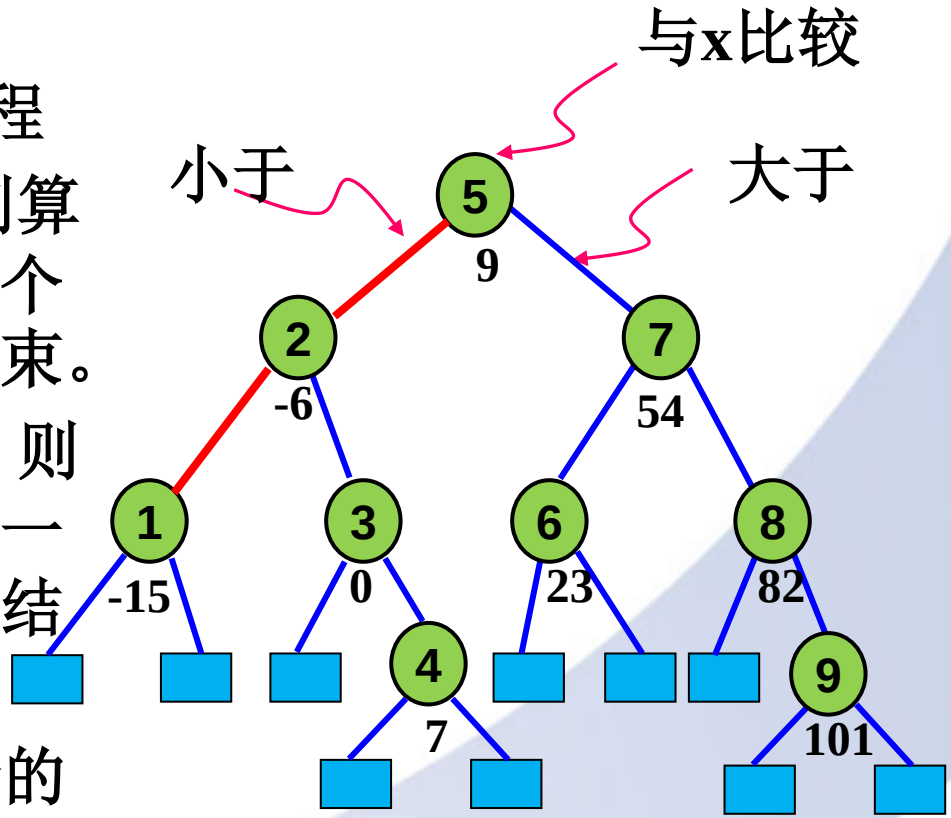
表示不成功检索情况

●路径：代表检索中元素的比较序列



3.2 二分检索

- 二元比较树
  - 二元比较树的查找过程
    - 若x在A中出现，则算法的执行过程在一个圆形的内结点处结束。
    - 若x不在A中出现，则算法的执行过程在一个方形的外结点处结束
    - 外结点不代表元素的比较，因为比较过程在该外结点的上一级的内结点处结束。



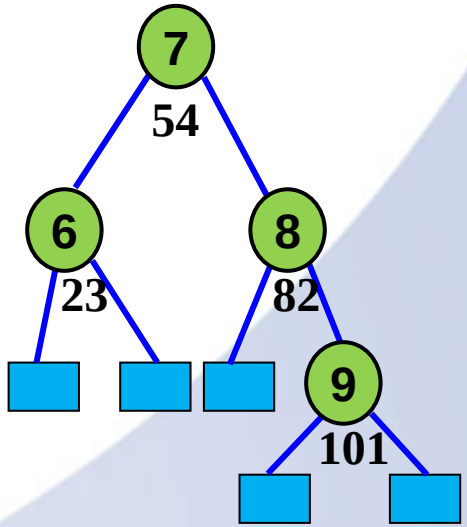
3.2 二分检索

■ 二元比较树

- 定理3.2 若n在区域$[2^{k-1}, 2^k)$中，则对于一次成功的检索，BINSRCH至多做k次比较；对于一次不成功的检索，或者做k-1次比较，或者做k次比较。
- 证明要点：
  - 二分检索中，每次mid取中值，故其二元比较树是一种“结点平衡”的树
  - 当$2^{k-1} \le n < 2^k$时，结点分布情况为：
    内结点：1至k级
    外结点：k级或k+1级，
  - 外部结点在相邻的两级上：最深一层或倒数第2层

3.2 二分检索

- 二元比较树
  - 比较过程：
    - 成功检索在内结点处结束，不成功检索在外结点处结束。
    - 成功检索在i级的某个内结点终止时，所需要的元素比较次数是i，等于根到该内结点的路径长度＋1。
    - 不成功检索在i级的外部结点终止所需的元素比较次数是i-1，等于根到该外结点的路径长度。



3.2 二分检索

- **BINSRCH计算复杂度的理论分析**
  - 不成功
    - 不成功检索的最好、最坏和平均情况的计算时间均为**Θ(logn)**——外结点处在最末的两级上；

$$n \in [2^{k-1}, 2^k) \Rightarrow k \approx \log n$$

  - 成功
    - 最好情况下的成功检索的计算时间为**Θ(1)**
    - 最坏情况下的成功检索的计算时间为**Θ(logn)**

3.2 二分检索

- **BINSRCH计算复杂度的理论分析**
  - 平均
    - 利用外部结点和内部结点到根距离和之间的关系进行推导：
      - ✓由根到所有内结点的距离之和称为内部路径长度，记为I；
      - ✓由根到所有外部结点的距离之和称为外部路径长度，记为E。

则有，**E=I+2n** ·····①

3.2 二分检索

- **BINSRCH计算复杂度的理论分析**
  - 平均
    - 记U(n)是平均情况下不成功检索的计算时间，
      则U(n) = E/(n+1) ·····②
    - S(n)是平均情况下成功检索的计算时间，
      则S(n) = I/n+1 ·····③

利用① ② ③ ， 可有:

$S(n) = (1+1/n)U(n) - 1$

当$n \to \infty$, $S(n) \propto U(n)$， 而$U(n) = \Theta(\log n)$

所以 $S(n) = \Theta(\log n)$

3.2 二分检索

- BINSRCH计算复杂度的理论分析
  - 总结

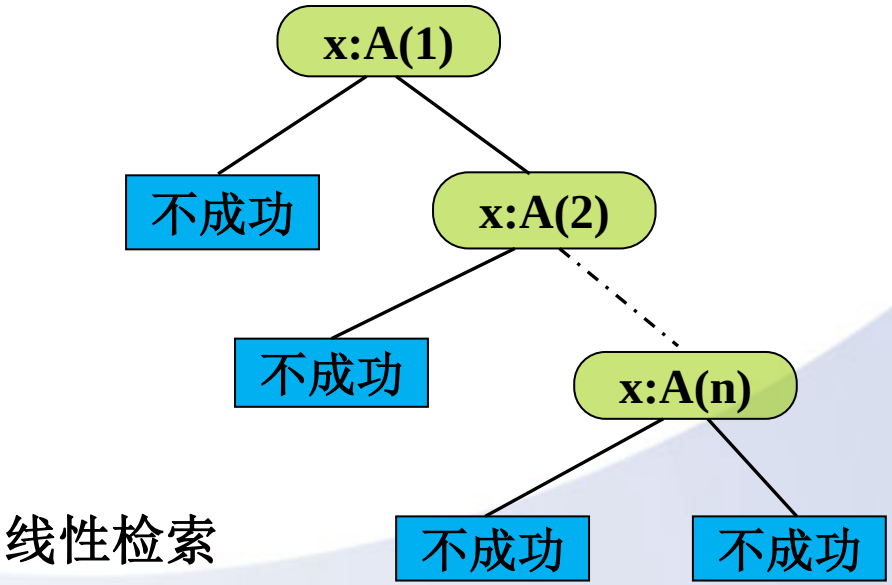
计算时间	最好情况	平均情况	最坏情况
成功的检索	Θ(1)	Θ(logn)	Θ(logn)
不成功的检索	Θ(logn)	Θ(logn)	Θ(logn)

3.2 二分检索

- 以比较为基础检索的时间下界
  - 问题：设n个元素的数组A(1:n)有A(1)< A(2) < …< A(n)。检索给定元素x是否在A中出现。
  - 定理3.2给出了二分检索算法的时间下界，是否预计还存在有以元素比较为基础的另外的检索算法，它在最坏情况下比二分检索算法在计算时间上有更低的数量级?
  - 以比较为基础的算法：
    - 假定算法中只允许进行元素间的比较，而不允许对它们实施其它运算。

3.2 二分检索

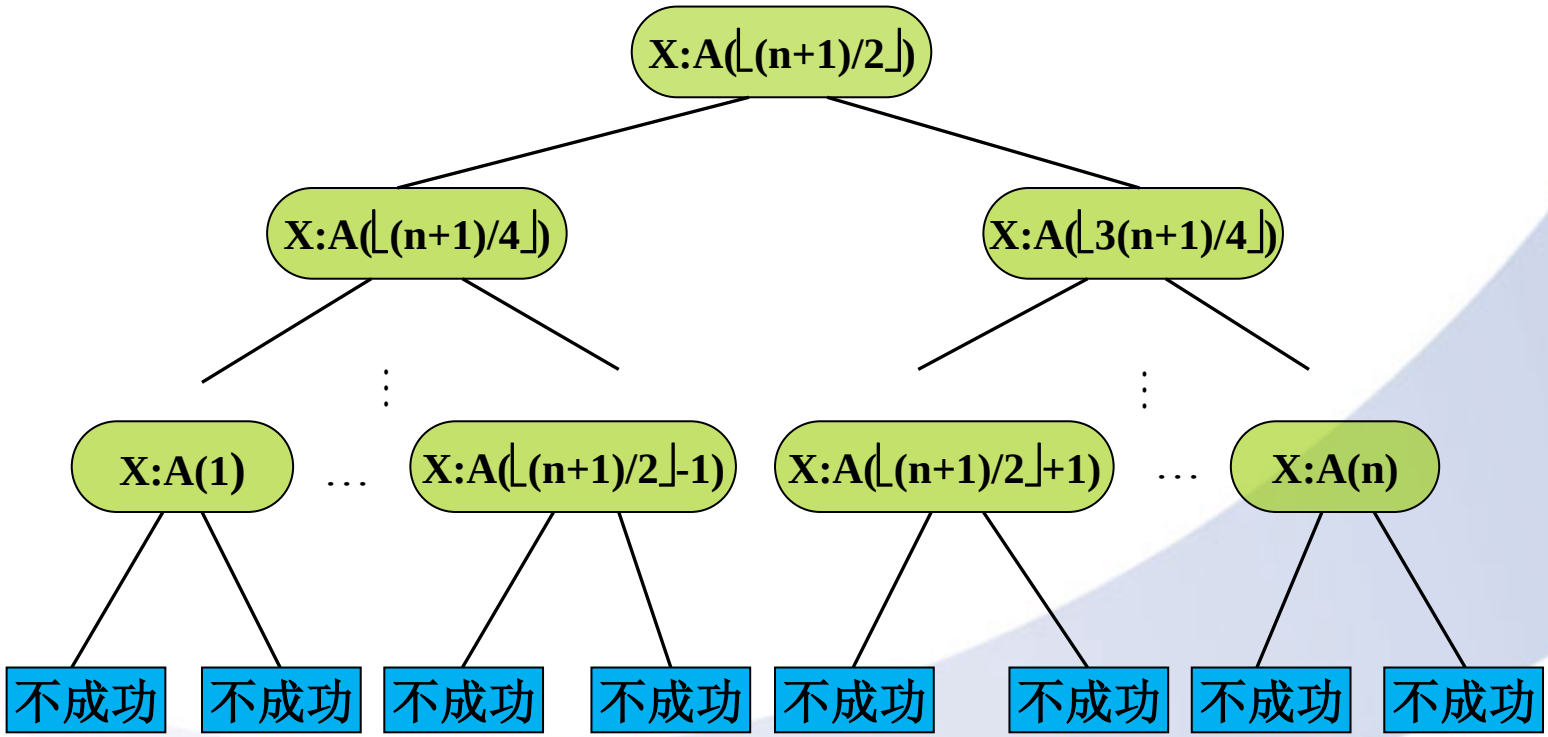
- 以比较为基础检索的时间下界
  - 任何以比较为基础的检索算法，其执行过程都可以用二元比较树来描述。
  - 线性检索与二分检索的比较树



线性检索

3.2 二分检索

- 以比较为基础检索的时间下界
  - 线性检索与二分检索的比较树



3.2 二分检索

- 以比较为基础检索的时间下界
  - 线性检索与二分检索的比较树
    - 每个内结点表示一次元素的比较。
      每棵比较树中恰好含有n个内结点，分别与n个不同i值相对应；
    - 每个外结点对应一次不成功的检索，
      并恰好又n+1个外结点对应于n+1中不成功检索的情况。

3.2 二分检索

- 以比较为基础的有序检索问题最坏情况的时间下界
  - 定理3.3 设A(1:n)含有n(n≥1)个不同的元素，排序为A(1)< A(2) < …< A(n)。又设用以比较为基础的算法去判断是否x ∈ A(1:n)，则这样的任何算法在最坏情况下所需的最小比较次数FIND(n)有：

$$FIND(n) \geq \lceil \log(n+1) \rceil$$

3.2 二分检索

- 以比较为基础的有序检索问题最坏情况的时间下界
  - 证明：
    - 从模拟求解检索问题算法的比较树可知，FIND(n)不大于树中由根到一个叶子的最长路径的距离。
    - 而所有树中必定有n个内结点与x在A中的n中可能的出现相对应。
    - 如果一棵二元树的所有内结点所在的级数小于或等于k，则最多有2^k-1个内结点。
    - 故$n \leq 2^k-1$,即

$$\text{FIND}(n) = k \geq \lceil \log(n+1) \rceil$$

3.2 二分检索

- 以比较为基础的有序检索问题最坏情况的时间下界
  - 结论
    - 任何一种以比较为基础的算法，在最坏情况下的计算时间都不低于O(logn)。
    - 因此，不可能存在最坏情况比二分检索数量级还低的算法。
    - 二分检索是解决检索问题的最优的最坏情况算法。

第三章 分治法

- 3.1 一般方法
- 3.2 二分检索
- 3.3 找最大和最小元素
- 3.4 归并排序
- 3.5 快速排序
- 3.6 选择问题
- 3.7 斯特拉森矩阵乘法
- 3.8 快速傅立叶变换
- 3.9 最接近点对问题

3.3 找最大和最小元素

- 查找问题：
  - 在元素表中确定某给定的元素的位置或判定其不出现：二分检索，顺序查找
  - 在元素表单独或同时找出其中的最大和(或)最小
  - 在元素表找出其中第一小(大)和第二小(大)元素
  - 找元素表中的第k小元素：排序或选择
  - …
- 本节问题：
  - 给定含有n个元素的集合，找出最大和最小元素。

3.3 找最大和最小元素

- 直接找最大和最小元素
  - 算法的性能：该算法最好、最坏、平均情况下均需要做2(n-1)次元素比较

算法3.5 直接找最大和最小元素

```
procedure STRAITMAXMIN(A, n, max, min)
//将A中的最大值置于max，最小值置于min//
    integer i, n
    max←min←A(1)
    for i←2 to n do
        if A(i) > max then max←A(i) endif
        if A(i) < min then  min←A(i)  endif
    repeat
end STRAITMAXMIN
```

3.3 找最大和最小元素

- 直接找最大和最小元素
  - STRAITMAXMIN算法的一种简单改进形式

```
procedure STRAITMAXMIN1(A, n, max, min)
    integer i, n
    max←min←A(1)
    for i←2 to n do
        if A(i) > max then    max←A(i)    endif
          else if A(i) < min then    min←A(i)   endif
    repeat
end STRAITMAXMIN1
```

3.3 找最大和最小元素

- 直接找最大和最小元素
  - STRAITMAXMIN算法的一种简单改进形式
    - 最好情况：按递增次序排列，元素比较次数为n-1次
    - 最坏情况：按递减次序排列，元素比较次数为2(n-1)次
    - 平均情况：元素比较次数为3(n-1)/2次

3.3 找最大和最小元素

- 分治求解策略
  - 记问题的一个实例为：

    $I = (n, A(1), \ldots, A(n))$
  - 采用二分法将I分成两个子集合处理

    $I1 = (\lfloor n/2 \rfloor, A(1), \ldots, A(\lfloor n/2 \rfloor))$, 和

    $I2 = (n-\lfloor n/2 \rfloor, A(\lfloor n/2 \rfloor+1), \ldots, A(n))$
  - 则有，

    $MAX(I) = \max(MAX(I1), MAX(I2))$

    $MIN(I) = \min(MIN(I1), MIN(I2))$
  - 采用递归的设计策略，得到以下算法：

3.3 找最大和最小元素

■ 一种递归求解策略

算法3.6 递归求取最大和最小元素

```
procedure MAXMIN(i, j, fmax, fmin)
  //A(1:n)是含有n个元素的数组，参数i，j是整数，1≤i，j≤n //
 //该过程把A(i:j)中的最大和最小元素分别赋给fmax和fmin //
 integer i, j; global n, A(1:n)
 case
   :i=j:  fmax←fmin←A(i)
   :i=j-1: if A(i)<A(j) then fmax←A(j); fmin←A(i)
           else fmax←A(i); fmin←A(j) endif
    :else: mid←⌊(i + j)/2⌋      //取中
          call MAXMIN(i, mid, gmax, gmin)
          call MAXMIN(mid+1, j, hmax, hmin)
          fmax←max(gmax, hmax); fmin←min(gmin, hmin)
  end case
end MAXMIN
```

基本问题：只有一个元素

基本问题：只有二个元素

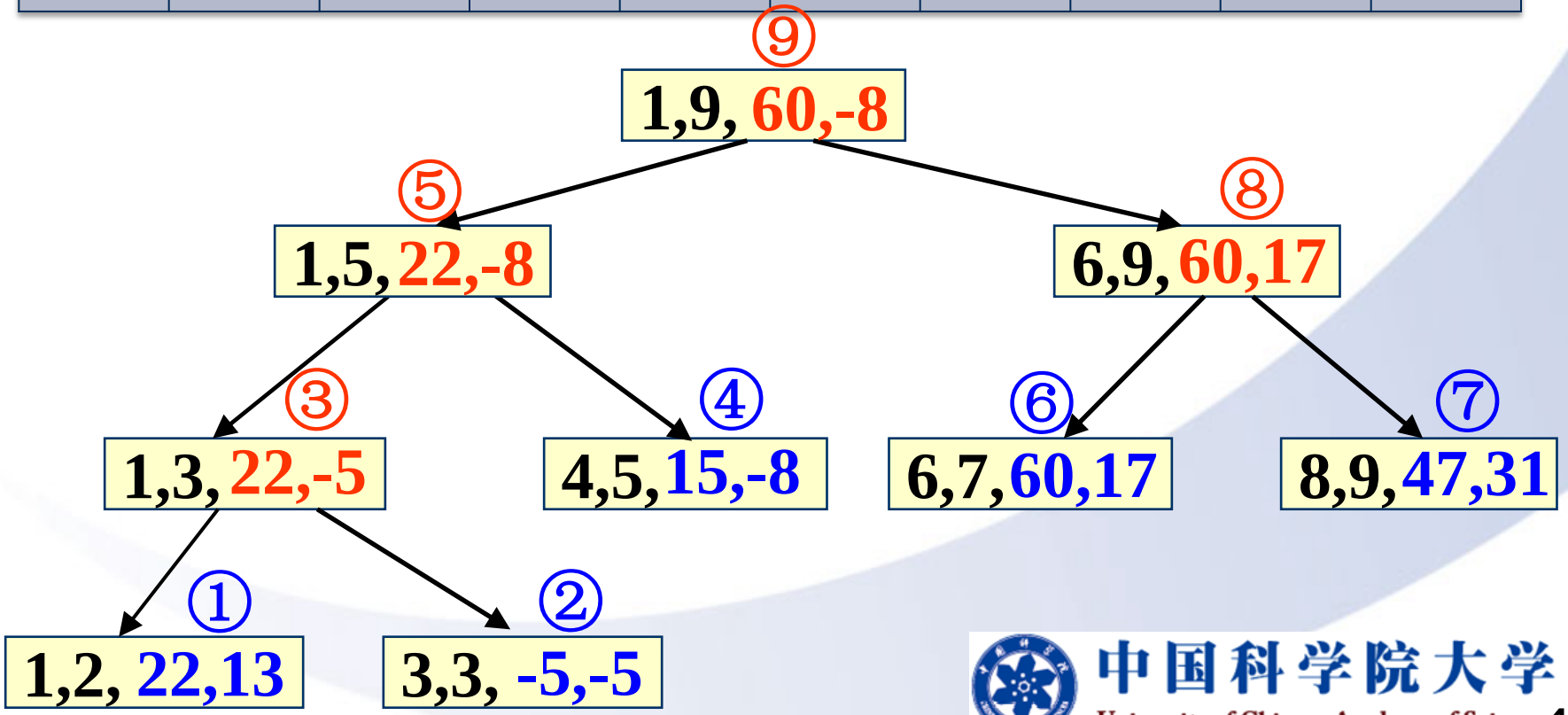
分治

合并

3.3 找最大和最小元素

■ 一种递归求解策略

i	1	2	3	4	5	6	7	8	9
a[i]	22	13	-5	-8	15	60	17	31	47



3.3 找最大和最小元素

- 一种递归求解策略
  - 性能分析(T(n)表示元素比较数)

$$T(n)=\begin{cases}0 & n=1\\ 1 & n=2\\ T(\lfloor n/2\rfloor)+T(\lceil n/2\rceil)+2 & n>2\end{cases}$$

  - 当n是2的幂时($n=2^k$),化简上式有，

$$T(n)=2T(n/2)+2=2(2T(n/4)+2)+2$$
$$=\ldots=2^{k-1}T(2)+\sum_{1\le i\le k-1}2^i=2^{k-1}+2^k-2$$
$$=3n/2-2$$

3.3 找最大和最小元素

- 一种递归求解策略
  - 性能分析
    - 与STRAITMAXMIN算法相比，比较次数减少了25%，(3n/2-2 : 2n-2)。
    - 已经达到了以元素比较为基础的找最大最小元素的算法计算时间的下界：$\lceil 3n/2 \rceil - 2$
    - 存在的问题——递归调用造成
      - ✓空间占用量大，有⌊logn⌋ + 1级的递归，入栈参数i, j, fmax, fmin和返回地址五个值
      - ✓时间可能不比预计的理想，如果元素A(i)和A(j)的比较与i和j的比较相差不大时，算法MAXMIN不可取

3.3 找最大和最小元素

- 一种递归求解策略
  - 性能分析

假设元素的比较与i和j的比较时间相同(整型数)。

又设对case语句调整，使得仅需一次i和j的比较就能够确定是哪种情况。

用i ≥j-1来代替i=j和i=j-1两次比较

```
case
 :i≥j-1:if A(i)<A(j) then fmax←A(j); fmin←A(i)
        else fmax←A(i); fmin←A(j)  //两个元素的情况
       endif

 :else: …
```

3.3 找最大和最小元素

■ 一种递归求解策略

□性能分析

记此时MAXMIN的频率计数为C(n)，$n=2^k$ (k为正整数)。则有，

$$C(n)=\begin{cases}2 & n=2\\ 2C(n/2)+3 & n>2\end{cases}$$

化简得，

$$\begin{aligned}C(n)&=2C(n/2)+3\\&=4C(n/4)+6+3\\&\cdots\\&=2^{k-1}C(2)+3\sum_{0\le i\le k-2}2^i\\&=2^k+3\times 2^{k-1}-3\\&=5n/2-3\end{aligned}$$

3.3 找最大和最小元素

- 一种递归求解策略
  - 性能分析

按照同样的假设，重新估算STRAITMAXMIN算法的比较次数为：3(n-1)。(包括for循环的比较)

- STRAITMAXMIN与MAXMIN在计算时间上的差异越来越小1/4 (25%)=>1/6(16.7%)
- 考虑MAXMIN算法递归调用的开销，此时MAXMIN算法的效率可能还不如STRAITMAXMIN算法。

3.3 找最大和最小元素

- 一种递归求解策略
  - 性能分析结论
    - 如果A中的元素间的比较代价远比整型数(下标)的比较昂贵，则分治方法将产生一个效率较高的算法；
    - 反之，可能仅得到一个低效的算法。
    - 分治策略只能看成一个较好的但并不总是成功的算法设计指导。

作业-课后练习4

- 问题描述
  - 写一个三分检索算法，首先检查n/3处的元素是否等于某个x的值，然后检查2n/3处的元素。这样或者找到x，或者把集合缩小到原来的1/3。
  - 分析算法在各种情况下的计算复杂度。

End