递归与分治

陈长建

计算机科学系

关于作业

- 编程语言不限，但需要自己实现核心部分
  - numpy.sort() ×

- 提交时间：9月29日

关于第一次实验

- 院楼103，10月13日（周日）上午8:30-12:00（现场验收）
- 第一次实验离线题（离线准备）
  - 1. 分治法查找最大最小值
  - 2. 分治法实现合并排序
  - 3. 实现题1-3 最多约数问题
- 在线题
  - OJ系统还在准备中

关于小班讨论

- 以组为单位,每组一题,每组人人参与,合理分工,ppt中标记分工,尽量都有代码演示,第8周上台报告

小班讨论课安排

- 讨论主题分为3类：
  - 算法分析练习
  - 算法实现练习
  - 《数学之美》等课外资料阅读
- 分组选题（每班分6组）
  - 每组选择1位组长，实行组长负责制
  - 每次讨论课，每组选1类题，即每次课参与讨论分析题、实现题与《 数学之美》三类题的各有2组
  - 每组在2次讨论课中，必须轮流选择3类题目

回顾 - 递归

- 直接或间接地调用自身的算法称为递归算法。用函数自身给出定义的函数称为递归函数。
- 完整的递归定义必须满足如下条件：
  - 包含一个基本部分，对于n的一个或多个值， F(n)必须是直接定义的（即边界条件）。
  - 在递归部分中，右侧所出现的所有F的参数都必须有一个比n小（递归方程）。

回顾 - 递归

- 阶乘函数可递归地定义为：

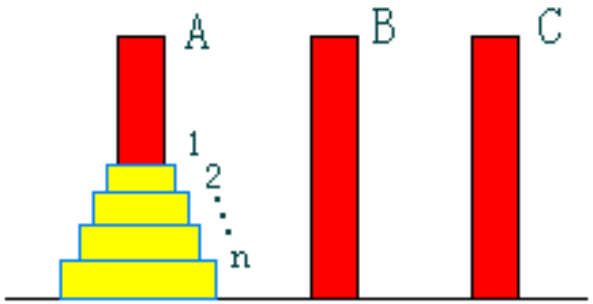
$$n! = \begin{cases} 1 & n = 0 \\ n(n-1)! & n > 0 \end{cases}$$

边界条件

递归方程

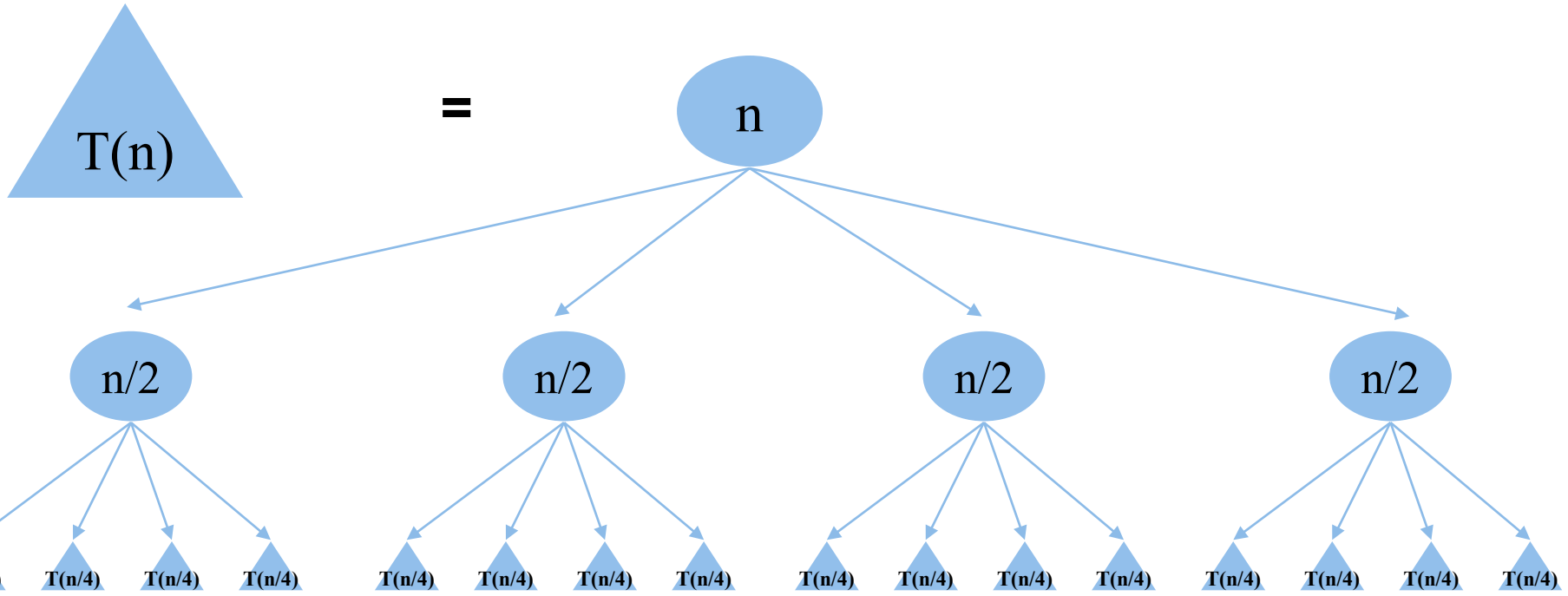
回顾 -汉诺（Hanoi）塔问题

- 设a,b,c是3个塔座。开始时，在塔座a上有一叠共n个圆盘，这些圆盘自下而上，由大到小地叠在一起。各圆盘从小到大编号为1,2,...,n,现要求将塔座a上的这一叠圆盘移到塔座c上，并仍按同样顺序叠置。在移动时应遵守以下移动规则：
- 规则1：每次只能移动1个圆盘；
- 规则2：任何时刻都不允许将较大圆盘压在较小的圆盘之上；
- 规则3：在满足移动规则1和2的前提下，可将圆盘移至a,b,c中任一塔座上。



回顾-分治

- 将求出的小规模的问题的解合并为一个更大规模的问题的解，自底向上逐步求出原来问题的解。



回顾 - 分治法的适用条件

- 分治法所能解决的问题一般具有以下几个特征：
  - 1. 该问题的规模缩小到一定的程度就可以容易地解决
  - 2. 该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质
  - 3. 利用该问题分解出的子问题解可以合并为该问题的解
  - 4. 该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子问题

回顾-二分搜索技术

- 给定已按升序排好序的n个元素a[0:n-1]，现要在这n个元素中找出一特定元素x。
  - 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。

二元比较树

1）不成功检索的最好、最坏和平均情况的计算时间均为 $\Theta(\log n)$ ——外结点处在最末的两级上；

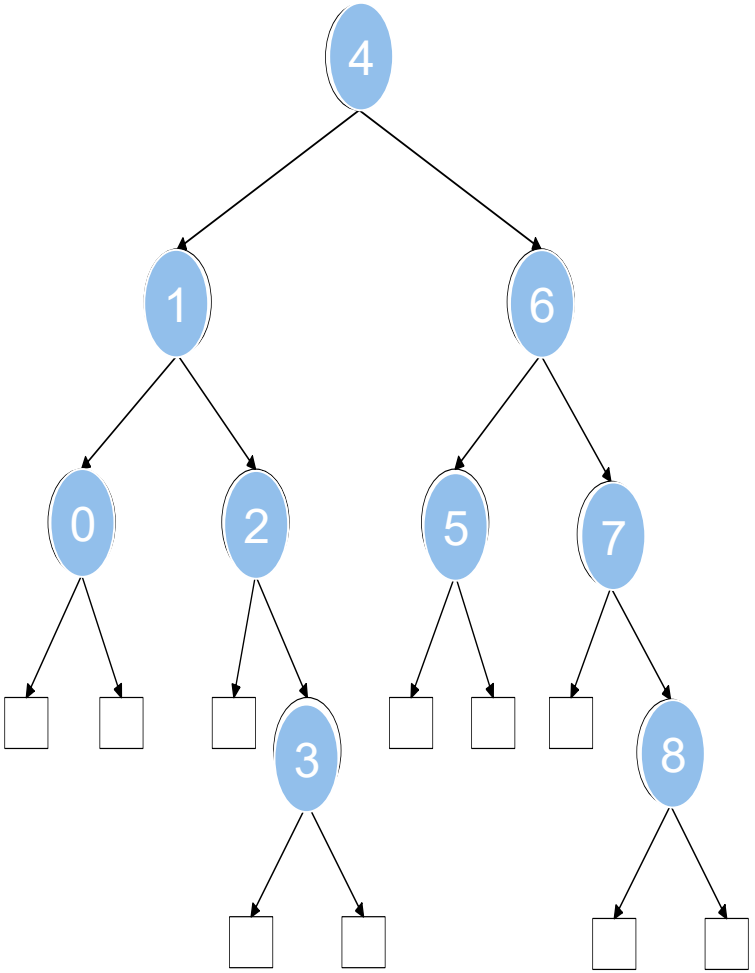
2）最好情况下的成功检索的计算时间为 $\Theta(1)$

最坏情况下的成功检索的计算时间为 $\Theta(\log n)$

（注：对数均以2为底）

二元比较树

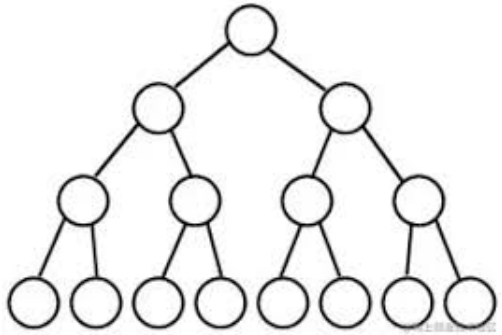
- 算法执行过程的主体是x与一系列中间元素a[middle]比较。可用一棵二元树描述这一过程，并称之为二元比较树。
- 构造：比较树由称为内结点和外结点的两种结点组成：
  - 内结点：表示一次元素比较，用圆形结点表示，存放一个middle值；代表一次成功检索；
  - 外结点：在二分检索算法中表示一种不成功检索的情况，用方形结点表示。
  - 路　径：表示一个元素的比较序列。



基于二元比较树分析时间复杂度

3）平均情况下的成功检索的计算时间分析

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

证明：$I=1*2^1+2*2^2+\ldots\ldots+k*2^k$

$E=(k+1)*2^{k+1}$

$E-I=(k+1)*2^{k+1}-(1*2^1+2*2^2+3*2^3\ \ldots\ldots+k*2^k)$

设 $F(k)=E-I$

$F(k)/2=(k+1)*2^k-(1*2^0+2*2^1+\ldots\ldots+k*2^{k-1})$

$F(k)-F(k)/2=(k+1)*2^{k+1}+2^0+2^1+\ldots.+2^{K-1}-(k+1)*2^K-k*2^k$

$=(K+1)*2^K+2^0+2^1+\ldots.+2^{K-1}-k*2^k$

$=2^0+2^1+\ldots.+2^{K-1}+2^k=n$

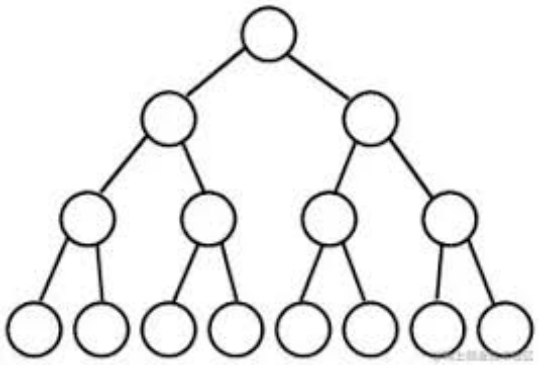
$F(k)=E-I=2n$ E=I+2n

基于二元比较树分析时间复杂度

- U(n)是平均情况下不成功检索的计算时间，则U(n) = E/(n+1)
- S(n)是平均情况下成功检索的计算时间，则S(n) = I/n+1
- 利用上述公式，可有: S(n) = (1+1/n)U(n) -1

  当n→∞, S(n) ∝ U(n) ，而U(n) =Θ(logn)

  所以 S(n) = Θ(logn)

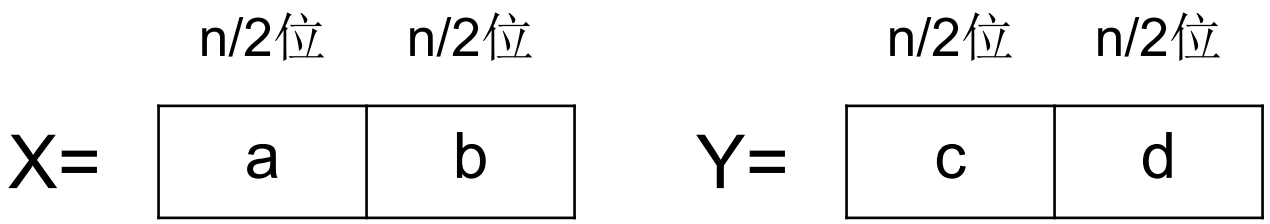


本章要点

- 算法总体思想
- 递归的概念
- 递归案例分析
- 分治法的概念
- 分治案例分析
  - 二分搜索、大整数法、棋盘覆盖、合并排序、快速排序、线性时间选择 ...

分治-例2 大整数乘法

- 请设计一个有效的算法，可以进行两个n位大整数的乘法运算
  - 小学的方法：$O(n^2)$
  - 分治法：



    - X = ab
    - Y = cd
    - $X = a \cdot 10^{n/2} + b$
    - $Y = c \cdot 10^{n/2} + d$
    - $XY = (a*c) \cdot 10^{n} + (a*d+b*c) \cdot 10^{n/2} + b*d$

时间复杂度分析

- XY =(a*c)•10^n + (a*d+b*c) •$10^{n/2}$ + b*d

$$T(n)=\begin{cases}O(1) & n=1\\ 4T(n/2)+O(n) & n>1\end{cases}$$

- T(n) = 4T(n/2) +O(n)
- = 4[4T(n/2^2)+O(n/2)] + O(n)=4^2T(n/2^2)+O(n)
- =…=O($4^{\log_2 n}$)+O(n)
- 对数运算规则 $a^{\log_b c} = c^{\log_b a}$
- T(n) = O($4^{\log_2 n}$) = O($n^{\log_2 4}$) = O(n^2)

×没有改进☹

乘法变换

- $XY = (a*c) \cdot 10^{n} + (a*d+b*c) \cdot 10^{n/2} + b*d$
- 变换：$a*d+b*c = (a+b)*(c+d) - (a*c) - (b*d)$
- $XY=(a*c) \cdot (10^{n} - 10^{n/2}) + (a+b)*(c+d) \cdot 10^{n/2} + (b*d) \cdot (1 - 10^{n/2})$
- 4次乘法减为3次乘法

两种变换方法

- $XY = a*c \bullet 10^{n} + ((a-c)*(b-d)+a*c+b*d) \bullet 10^{n/2} + b*d$
- $XY = a*c \bullet 10^{n} + ((a+c)*(b+d)-a*c-b*d) \bullet 10^{n/2} + b*d$

细节问题：两个XY的复杂度都是O(nlog3)，但考虑到a+c,b+d可能得到m+1位的结果，使问题的规模变大，故不选择第2种方案。

时间复杂度分析

$$T(n)=\begin{cases}O(1) & n=1\\ 3T(n/2)+O(n) & n>1\end{cases}$$

$T(n)=O(n^{\log 3})=O(n^{1.585})$ ✓**较大的改进☺**

大整数乘法总结

- 小学的方法：$O(n^2)$ ✗效率太低
- 分治法: $O(n^{1.585})$ ✓较大的改进
- 更快的方法：
  - 如果将大整数分成更多段，用更复杂的方式把它们组合起来，将有可能得到更优的算法。
  - 最终，这个思想导致了快速傅利叶变换(Fast Fourier Transform)的产生。该方法也可以看作是一个复杂的分治算法，对于大整数乘法，它能在O(nlogn)时间内解决。
- 是否能找到线性时间算法？目前为止还没有结果。

时间复杂度分析思考

- 两个问题用两种分析方法
- 它们是否是通用的?
- 如何选择使用哪种方法分析?

小结-分治法时间复杂度分析

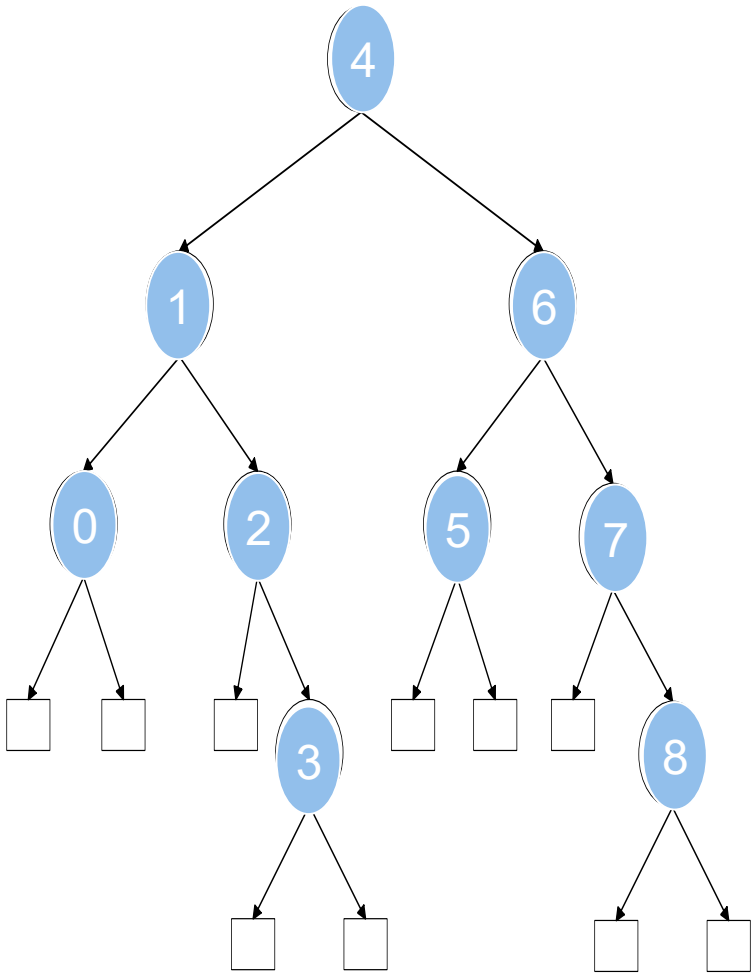
- **代换法**（大数乘法） $T(n)=\begin{cases}O(1) & n=1\\ 4T(n/2)+O(n) & n>1\end{cases}$
  - T(n) = 4T(n/2) = 4^2T(n/2^2)=…=O($4^{\log_2 n}$)
  - 简单、但局限于特定情形
- **主定理法**

$$T(n)=\begin{cases}c & n=1\\ aT\left(\frac{n}{b}\right)+cn & n>1\end{cases}$$

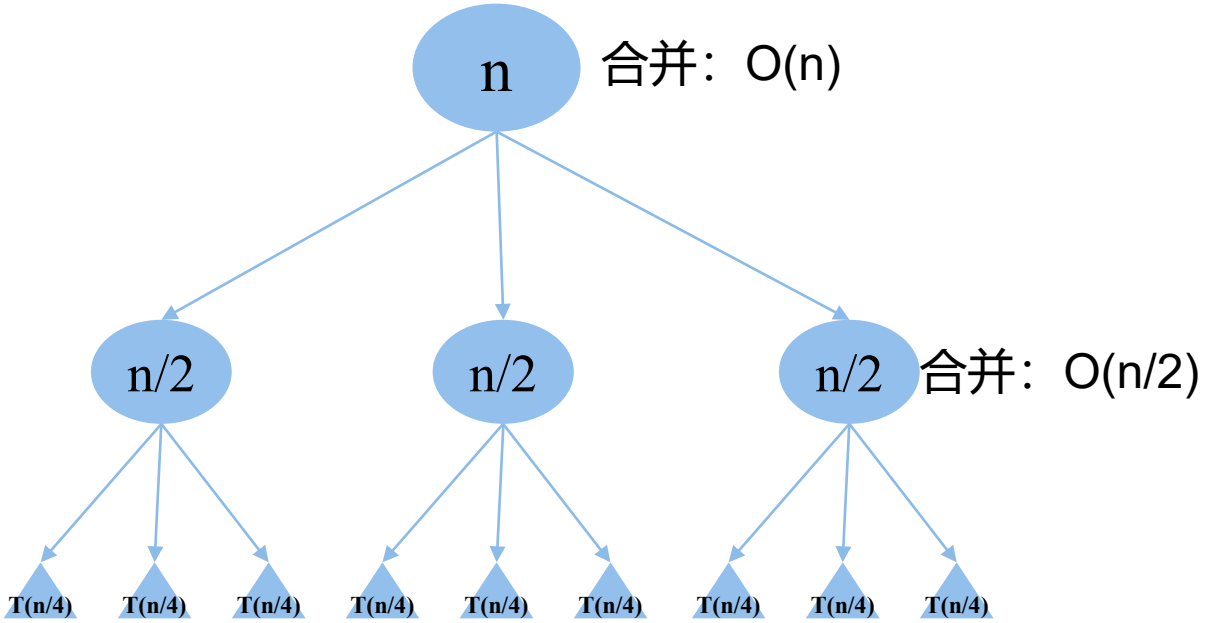
$$T(n)=\begin{cases}\Theta(n) & a<b\\ \Theta(n\log_b n) & a=b\\ \Theta\left(n^{\log_b a}\right) & a>b\end{cases}$$

小结-分治法时间复杂度分析

- **递归树**方法
  - 二分搜索算法
  - 分治过程都可以表示成树
  - 更加通用
  - 但是更为复杂



递归树方法分析大数乘法



- **递归树**方法
  - 计算量：叶子节点树+合并时间总数
  - 叶子节点数：2^h，其中$h=\log_3 n$
  - 合并时间：$O(n)+3O(n/2)+\ldots=O(n)$
  - 总时间：$2^{\log_3 n}+O(n)=O(n^{\log_3})$

小结-分治法时间复杂度分析

- 递归树方法
  - 二分搜索算法
- 主定理法

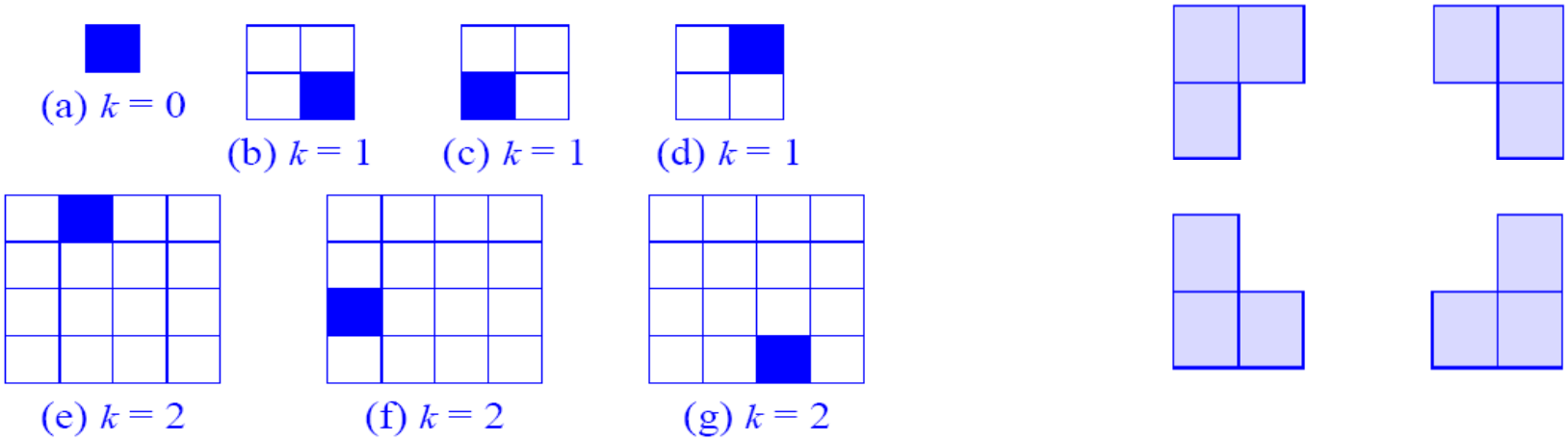
$$T(n)=\begin{cases} c & n=1 \\ aT\left(\frac{n}{b}\right)+cn & n>1 \end{cases}$$

$$T(n)=\begin{cases} \Theta(n) & a<b \\ \Theta(n\log_b n) & a=b \\ \Theta\left(n^{\log_b a}\right) & a>b \end{cases}$$

- 代换法（数学归纳）

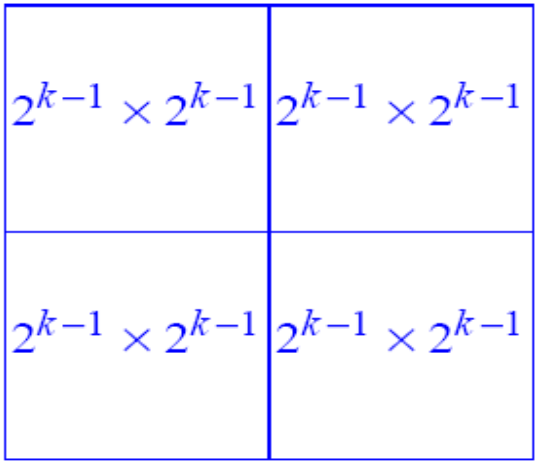
分治-例3 棋盘覆盖

- 在一个$2^k \times 2^k$ 个方格组成的棋盘中，恰有一个方格与其他方格不同，称该方格为一特殊方格，且称该棋盘为一特殊棋盘。在棋盘覆盖问题中，要用图示的4种不同形态的L型骨牌覆盖给定的特殊棋盘上除特殊方格以外的所有方格，且任何2个L型骨牌不得重叠覆盖。

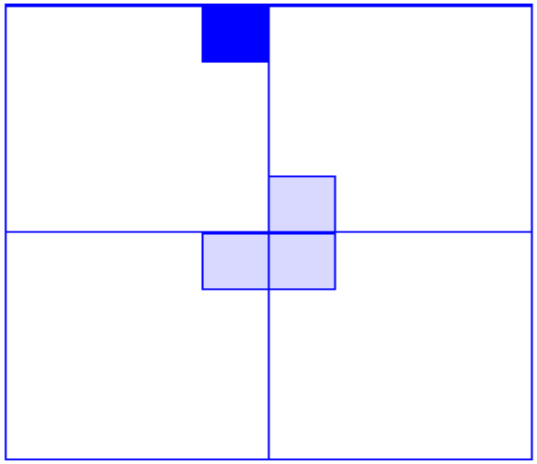


棋盘覆盖的分治方法

- 当k>0时，将2k×2k棋盘分割为4个2k-1×2k-1 子棋盘(a)所示。 特殊方格必位于4个较小子棋盘之一中，其余3个子棋盘中无特殊方格。为了将这3个无特殊方格的子棋盘转化为特殊棋盘，可以用一个L型骨牌覆盖这3个较小棋盘的会合处，如 (b)所示，从而将原问题转化为4个较小规模的棋盘覆盖问题。递归地使用这种分割，直至棋盘简化为棋盘1×1。



(a) Partitioning


(b) Triomino placed

时间复杂度分析

$$T(k)=\begin{cases}O(1) & k=0\\ 4T(k-1)+O(1) & k>0\end{cases}$$

设M(k)为chessBoard算法在计算覆盖一个2^kx2^k棋盘所需时间：

当k>1时，M(k)=4M(k-1)，　M(0)=1

M(k)=4M(k-1)　　替换M(k-1)=4M(k-2)

=4[4M(k-2)]=4^2 M(k-2)

=……

=4^kM(k-k) =4^k

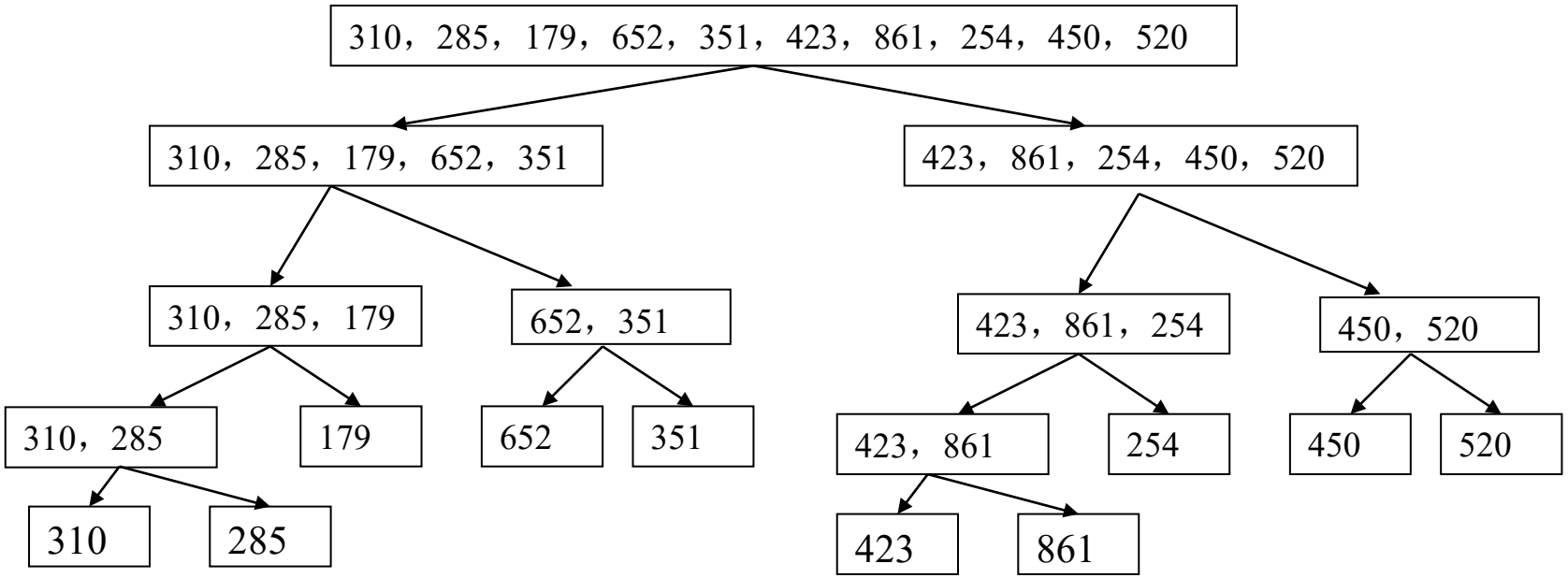
T(K)=O(4^K)

分治-例4 合并排序

- 给定一n个元素的集合A，按照某种方法将A中的元素按降序或增序排列。
- 常见内排序方法：
  - 冒泡排序，计数排序，插入排序，归并排序，快速排序，堆排序， …

合并排序基本思想

- 将待排序元素分成大小大致相同的2个子集合，分别对2个子集合进行排序，最终将排好序的子集合合并成为所要求的排好序的集合。
  - 例：A=（310，285，179，652，351，423，861，254，450，520）



合并排序算法

```
public static void mergeSort(int[ ] a, int left,  int right)
  {
  if (left<right)
    {//至少有2个元素
      int i=(left+right)/2;  //计算中分点
      mergeSort(a, left, i); //在第一个子集合上分类(递归)
      mergeSort(a, i+1, right); //在第二个子集合上分类(递归)
      merge(a, b, left, i, right);  //合并到数组b
      copy(a, b, left, right);    //复制回数组a
      }
  }
```

归并过程

A=（310，285，179，652，351，423，861，254，450，520）

首先进入左分枝的划分与归并。首先形成的划分状态是：

（310 |||| 285 ||| 179 || 652，351 | 423，861，254，450，520）

第一次归并：（285，310 ||| 179 || 652，351 | 423，861，254，450，520）

第二次归并：（179，285，310 || 652，351 | 423，861，254，450，520）

第三次归并：（179，285，310 || 351，652 | 423，861，254，450，520）

第四次归并：（179，285，310，351，652 | 423，861，254，450，520）

最后合并：（179，254，285，310，351，423，450，520，652，861）

归并过程

算法返回到mergeSort首次递归调用的后续语句处

进入右分枝的划分与归并过程

(179，285，310，351，652 | 423 |||| 861||| 254 || 450，520)

第一次归并：(179，285，310，351，652 | 423，861 ||| 254 || 450，520)

第二次归并：(179，285，310，351，652 | 254， 423，861 || 450，520)

第三次归并：(179，285，310，351，652 | 254， 423，861 || 450，520)

第四次归并：(179，285，310，351，652 | 254， 423，450，520，861)

最后合并结果：

(179，254，285，310，351，423，450，520，652，861)

用树结构描述归并分类过程

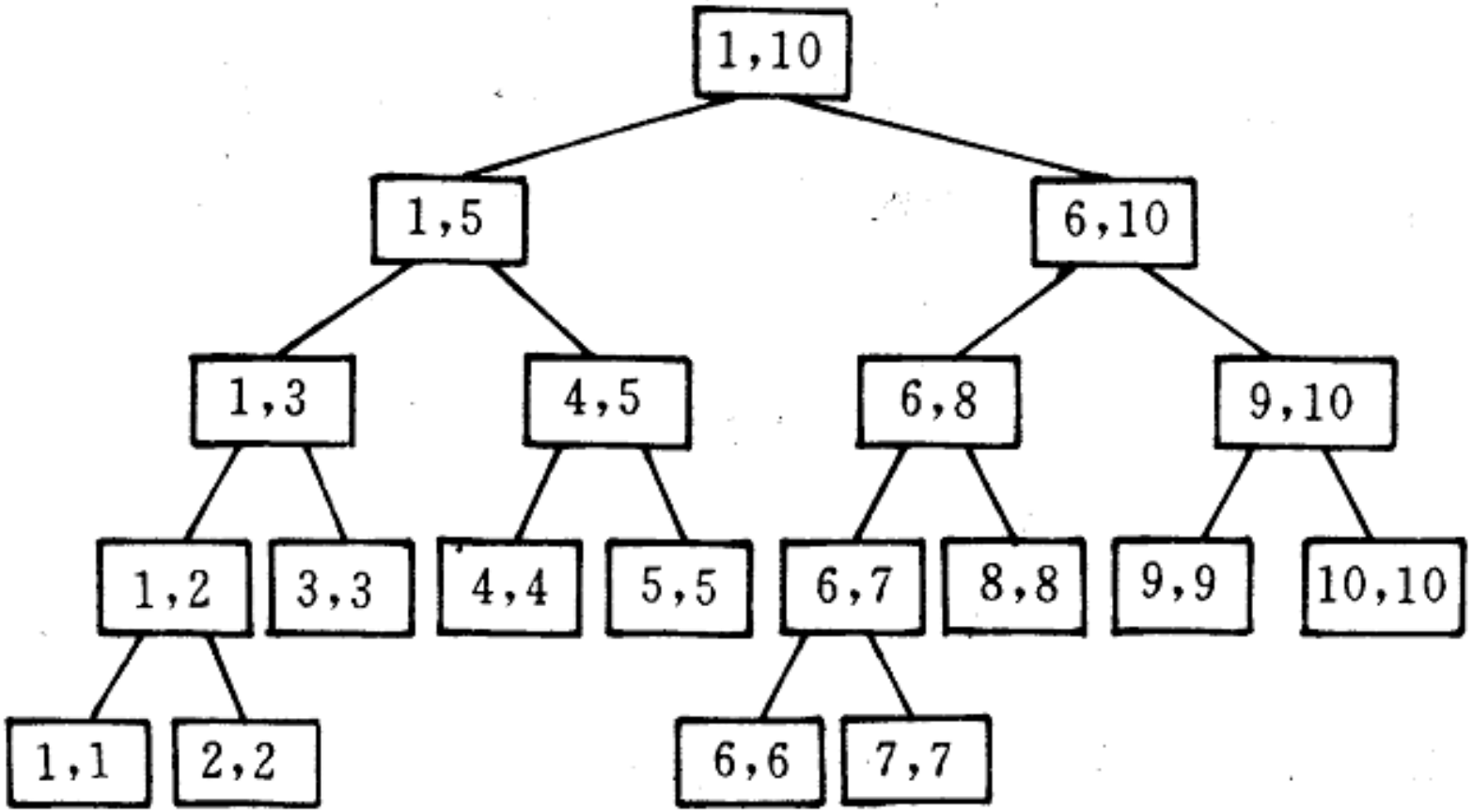


图 2.4　用树表示 MERGESORT(1,10)的调用

用树结构描述归并分类过程

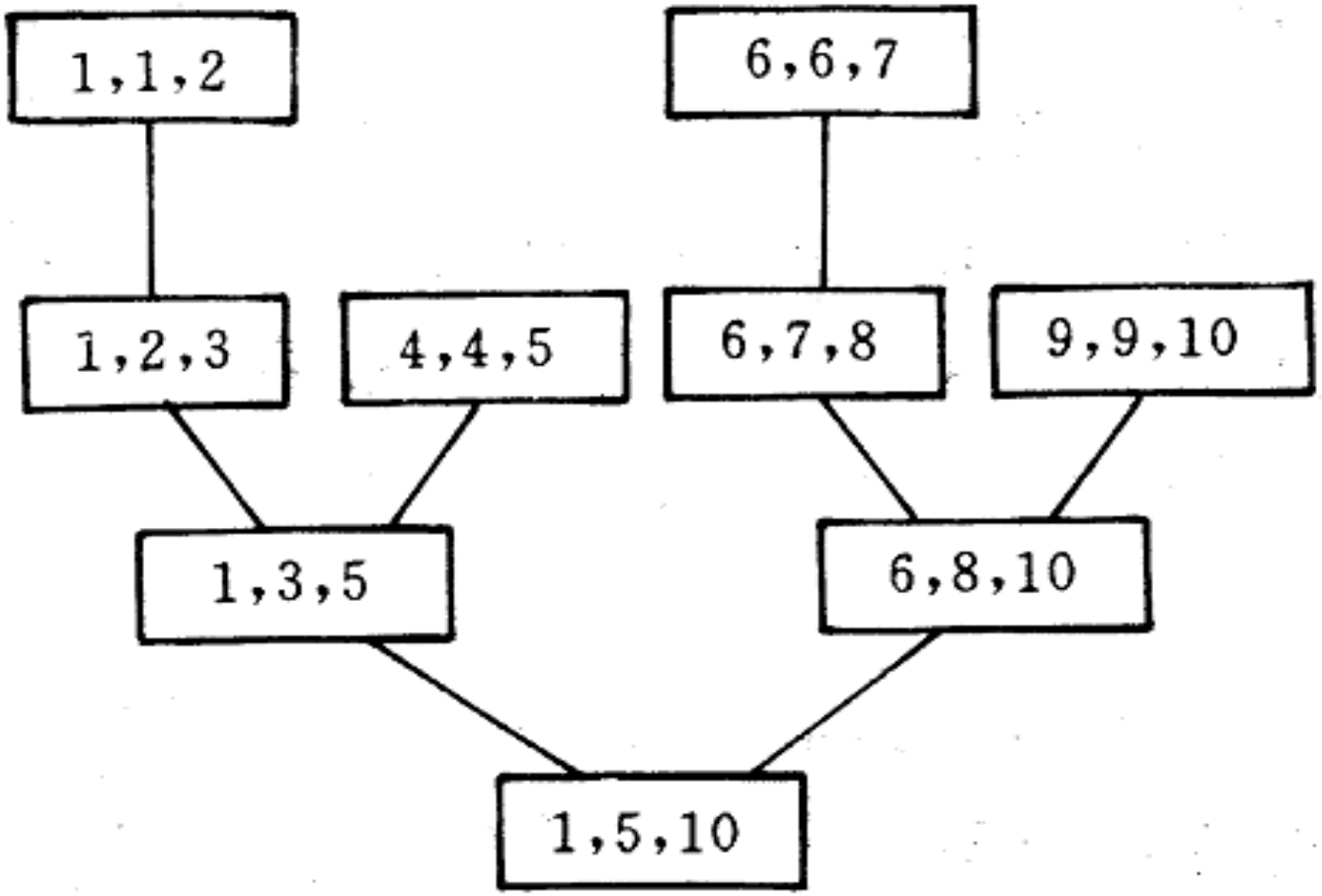


图 2.5　用树表示对 MERGE 的调用

合并排序算法的问题

```
public static void mergeSort(int[ ] a, int left,  int right)
  {
  if (left<right)
    {//至少有2个元素
      int i=(left+right)/2;  //计算中分点
      mergeSort(a, left, i); //在第一个子集合上分类(递归)
      mergeSort(a, i+1, right); //在第二个子集合上分类(递归)
      merge(a, b, left, i, right);  //合并到数组b
      copy(a, b, left, right);    //复制回数组a
      }
  }
```

递归层次太深

数据频繁移动

合并排序算法的改进

- 改进后归并排序算法：
  - 首先将每两个相邻的大小为 1 的子序列归并，然后对上一次归并所得到的大小为 2 的子序列进行相邻归并，如此反复，直至最后归并到一个序列，归并过程完成。
  - 通过轮流地将元素从 a 归并到b,并从b 归并到a，可以消除复制过程。

消去递归后的归并分类算法

```
public static void mergeSort(int[ ] a)
  {
    int[ ] b = new int[a.length];
    int s=1;
    while(s<a.length){
       mergePass(a, b, s); //合并大小为s的相邻子数组到数组b
       s+=s;
       mergePass(b, a, s); //合并到数组a
       s+=s;
       }
  }
```

mergePass算法：

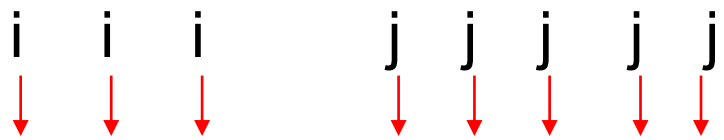
```
public static void mergePass(int[ ] x, int[ ] y, int s)
  { //合并大小为s的相邻子数组
    int i = 0;
    while(i<x.length - 2*s)
      {//合并大小为s的相邻2段子数组
       merge(x, y, i, i+s-1, i+2*s-1);
       i=i+2*s;
       }
     //剩下的元素少于2s
       if(i+s<x.length)
           merge(x, y, i, i+s-1, x.length-1);
       else   //复制到y
          for(int j=i; j<x.length; j++)
             y[j]=x[j];
       }
```

merge算法：

```
public static void merge(int[ ] c, int[ ] d, int l, int m, int r)
  { //合并c[l : m ] 和 c[m+1 : r] 到d[l : r]
    int i =l, j=m+1, k=l;
    while ( (i<=m) && (j<=r) )
      if ( c[ i ] <= c[ j ] )
          d[k++] = c[i++];
      else d[k++] = c[j++];
      if(i>m)
          for(int q=j;q<=r;q++)
              d[k++] = c[q];
      else
          for(int q=i;q<=m;q++)
              d[k++] = c[q];
      }
```

Merge算法示例

- (4, 5, 8, 9)|(1, 2, 6, 7) → (1, 2, 4, 5, 6, 7, 8, 9)
  - 参数: l = 0; m = 3; r = 7



  - (4, 5, 8, 9) | (1, 2, 6, 7)

d[ ] (1 2 4 5 6 7 8 9)

性能分析

- 1）空间特性
  - 2n+3个空间位置
- 2) 算法**merge**的时间与两数组元素的总数成正
  - (可表示为: cn, n为元素个数, c为某正常数)
- 3) 算法**mergeSort**的时间用递推关系式表示如下:

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

证明

- 化简：若$n=2^k$,则有,

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

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

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

$=2^kT(1)+kcn$

$=an+cnlogn$

- $k=logn$

若$2^k<n<2^{k+1}$,则有$T(n)\leq T(2^{k+1})$。

所以得：$T(n) = O(nlogn)$

以比较为基础分类的时间下界

任何以关键字比较为基础的分类算法，其最坏情况下的时间下界都为：$\Omega(n\log n)$

利用二元比较树证明。

假设参加分类的n个关键字A(1),A(2),...,A(n)互异。任意两个关键字的比较必导致A(i)<A(j)或A(i)>A(j)的结果。

以二元比较树描述元素间的比较过程：

- 若A(i)<A(j)，进入下一级的左分支
- 若A(i)>A(j)，进入下一级的右分支

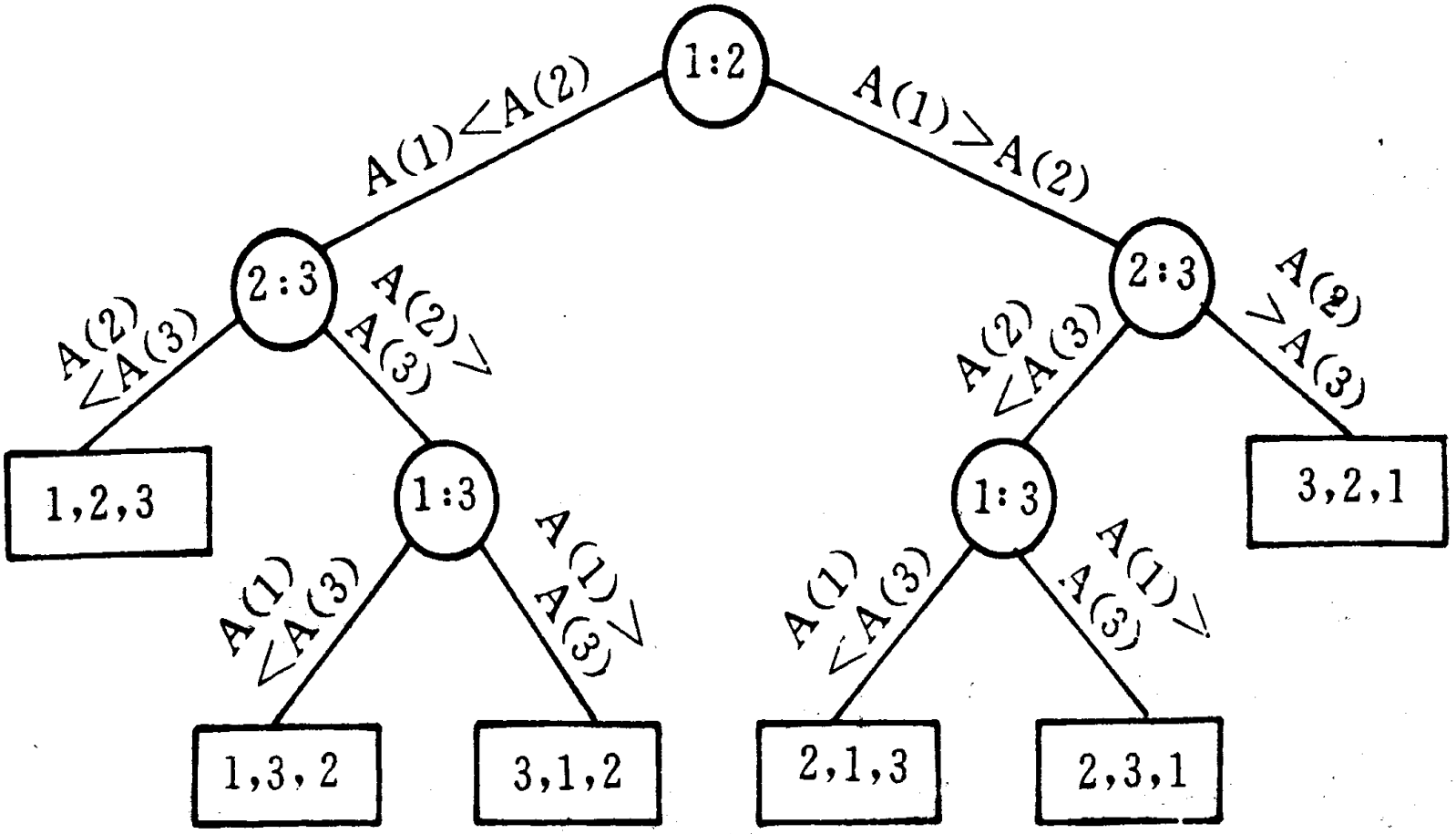


图 2.6　对三个关键字分类的比较树

算法在**外部结点**终止。

从根到某外结点的**路径**代表某个特定输入情况下一种唯一的**分类排序序列**。路径长度表示生成该序列代表的分类表所需要的**比较次数**。而**最长的路径**代表算法在最坏情况下的执行情况，该路径的长度即是算法在最坏情况下所作的比较次数。

故，以比较为基础的分类算法的最坏情况下界等于该算法对应的比较树的**最小高度**。

① 由于n个关键字有n!种可能的排列，所以二元比较树中将有n!个外部结点：每种排列对应于某种特定输入情况下的分类情况，每个外部结点表示一种可能的分类序列。

② 设一棵二元比较树的所有内结点的级数均小于或等于k，则该树中最多有2^k个外结点。

记算法在最坏情况下所作的比较次数为T(n)，则有T(n)=k：生成外结点所代表的分类序列所需的比较次数等于该外结点所在的级数-1；

根据①和②的分析，有：$n! \leqslant 2^{T(n)}$

化简：

当n>1时，有$n! \geqslant n(n-1)(n-2)\cdots(\lceil n/2 \rceil) \geqslant (n/2)^{n/2}$

当n≥4时，有 $T(n) \geqslant (n/2)\log(n/2) \geqslant (n/4)\log n$

故，任何以比较为基础的分类算法的最坏情况的时间下界为：

$\Omega(n\log n)$

分治-例5 快速排序

- 任何一个基于比较来确定两个元素相对位置的排序算法需要Ω(nlogn)计算时间。如果我们能设计一个需要□ (n1ogn)时间的排序算法，则在渐近的意义上，这个排序算法就是最优的。许多排序算法都是追求这个目标。
- 下面介绍快速排序算法，它在平均情况下需要(nlogn)时间。这个算法是于1962年由C.A.R.Hoare提出的。

基本思想

- 快速排序是一种基于划分的排序方法；
- 划分：选取待分类集合A中的某个元素t，按照与t的大小关系重新整理A中元素，使得整理后的序列中所有在t以前出现的元素均小于等于t，而所有出现在t以后的元素均大于等于t。这一元素的整理过程称为划分（Partitioning）。元素t称为划分元素。
- 快速排序：通过反复地对待排序集合进行划分达到排序目的的排序算法。

划分过程的算法描述

用元素 a[p] 划分集合 a[p : r]

```
public static int partition (int p, int r)
{
   int i = p,  j = r + 1;
   Comparable x = a[p];
   // 将>= x的元素交换到左边区域
   // 将<= x的元素交换到右边区域
  while (true) {
      while (a[++i].compareTo(x) < 0); // i由左向右移
      while (a[--j].compareTo(x) > 0);  // j由右向左移
      if (i >= j) break;
      MyMath.swap(a, i, j);
       }
      a[p] = a[j];
      a[j] = x;
    return j;   //划分元素在位置j
  }
```

例子 划分实例

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	i	j
A:	65	70	75	80	85	60	55	50	45	+∞	2	9
		|.....|										
A:	65	45	75	80	85	60	55	50	70	+∞	3	8
			|.....|									
A:	65	45	50	80	85	60	55	75	70	+∞	4	7
				|.....|								
A:	65	45	50	55	85	60	80	75	70	+∞	5	6
					|..|						
A:	65	45	50	55	60	85	80	75	70	+∞	6	5
交换划分元素 →	|.....|											
A:	60	45	50	55	65	85	80	75	70	+∞		

划分元素定位于此

分析

- 经过一次“划分”后，实现了对集合元素的调整：其中一个子集合的所有元素均小于等于另外一个子集合的所有元素。
- 按同样的策略对两个子集合进行分类处理。当子集合分类完毕后，整个集合的分类也完成了。这一过程避免了子集合的归并操作。这一分类过程称为快速分类。

快速排序算法实现

通过反复使用划分算法 **partition** 实现对集合元素的排序。

以a[p]为基准元素将a[p:r]划分成3段：

a[p:q-1]，a[q]，a[q+1:r]，

使得：a[p:q-1]中任何元素小于等于a[q]，

a[q+1:r]中任何元素大于等于a[q]，

下标q在划分过程中确定。

```
public  static void qSort(int p, int r)
 {
    if (p<r) {
    int q=partition(p,r);
     qSort (p,q-1); //对左半段排序
     qSort (q+1,r); //对右半段排序
     }
 }
```

快速排序分析

- 记录比较和交换是从两端向中间进行
  - 关键字较大的记录一次就能交换到后面单元
  - 关键字较小的记录一次就能交换到前面单元
  - 记录每次移动的距离较大，因而总比较和移动次数较少。
- 快速排序算法的性能取决于划分的对称性。
  - 可以设计出采用随机选择策略的快速排序算法。
  - 在快速排序算法的每一步中，当数组还没有被划分时，可以在a[p:r]中随机选出一个元素作为划分基准
  - 这样可以使划分基准的选择是随机的，从而可以期望划分是较对称的。

中值快速排序

- 中值快速排序（median-of-three quick sort）是快速排序一种变化，这种算法有更好的平均性能。在快速排序中总是选择a [1]做为支点，而在中值快速排序算法中，取{a[1],a[(1+r)/2],a[r]} 中大小居中的那个元素作为支点。
- 实现中值快速排序算法的一种最简单的方式就是首先选出中值元素并与a[1] 进行交换，然后利用快速排序算法完成排序。

时间复杂度分析

- 统计的对象：元素的比较次数，记为：C(n)
- 两点假设

①参加分类的n个元素各不相同

② **partition**中的划分元素t是随机选取的（针对平均情况的分析）

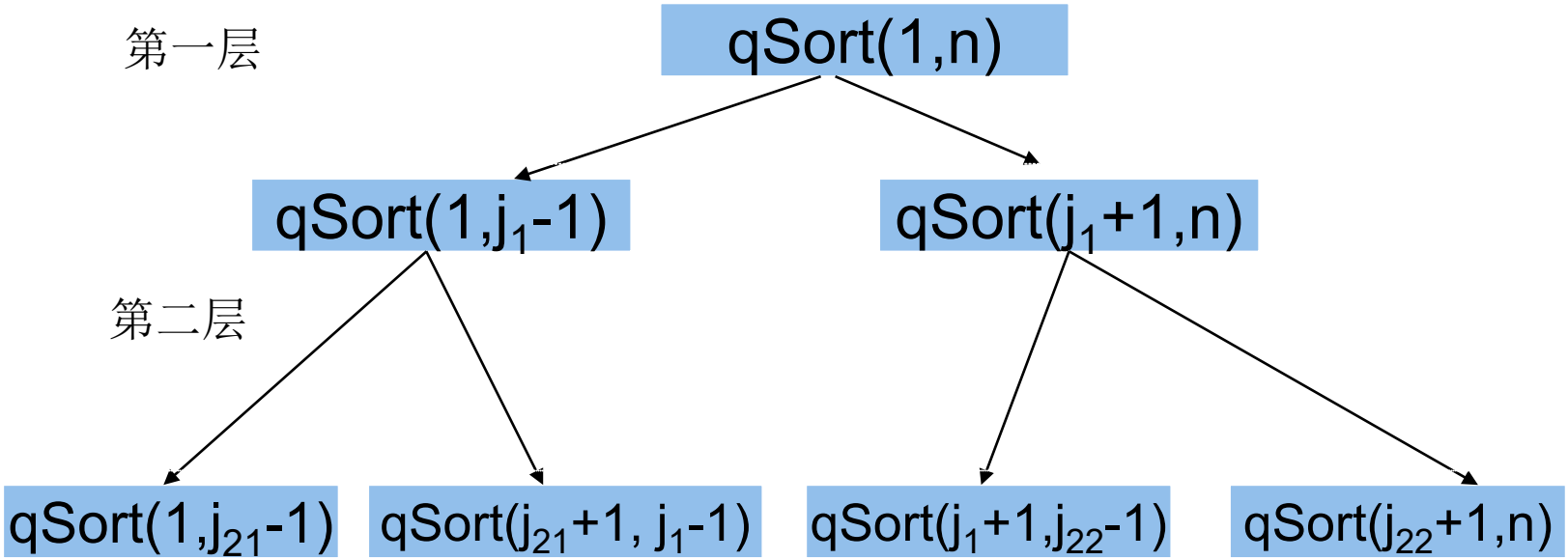
➢ 随机选取划分元素：

在划分区间[p,r]随机生成某一坐标：i← **random**(p,r)；

调换a[p]与a[i]

➢ 作用：将随机指定的划分元素的值依旧调换到a[p]位置。 之后，算法主体不变，仍从a[p]开始执行划分操作。

递归层次



第三层

设在任一级递归调用上，调用partition处理的所有元素总数为r，则，初始时r=n，以后的每级递归上，由于删去了上一级的划分元素，故r比上一级至少1：

理想情况，第一级少1，第二级少2，第三级少4，…；

最坏情况，每次仅减少1（如集合元素已经按照递增或递减顺序排列）

最坏情况分析

- 记最坏情况下的元素比较次数是$C_w(n)$;
- **partition**一次调用中的元素比较数是r – p+1，故每级递归调用上，元素的比较次数等于该级所处理的待分类元素个数。
- 最坏情况下，每级递归调用的元素总数仅比上一级少1，故Cw(n)是r由n到2的累加和。

即：

$$C_w(n) = \sum_{2 \le i \le n} i = O(n^2)$$

最好情况分析

- 记最好情况下的元素比较次数是$C_{best}(n)$；
- 最好情况下，每级递归调用的元素总数仅为上一级的1/2，故$C_{best}(n)$满足递推式：

$$C_{best}(n)=\begin{cases}0 & n=1\\ 2C_{best}(n/2)+n & n>1\end{cases}$$

$$n>1,\quad C_{best}(n)=2C_{best}(n/2)+n$$

$$=2(2C_{best}(n/2^2)+n/2)+n$$

$$=2^2C_{best}(2^{k-2})+2n=\ldots\ldots$$

$$=2^kC_{best}(1)+kn$$

$$当n=2^k,C_{best}(n)=n\log n$$

平均情况分析

- 记平均情况下的元素比较次数是$C_A(n)$；
  - 平均情况是指集合中的元素以任意一种顺序排列，且任选所有可能的元素作为划分元素进行划分和分类，在这些所有可能的情况下，算法执行性能的平均值。
  - 设调用**partition**(p,r)时，所选取划分元素t 恰好是a[p:r] 中的第i小元素（$1 \leqslant i \leqslant r-p$）的概率相等。则经过一次划分，所留下的待分类的两个子数组恰好是a[p:j-1]和a[j+1:p-1]的概率是：1/(r-p), $p \leqslant j < r$。则有，

$$C_A(n) = n + 1 + \frac{1}{n} \sum_{1 \leq k \leq n} (C_A(k-1) + C_A(n-k))$$

  - n+1是partition第一次调用时所需的元素比较次数。
  - $C_A(0) = C_A(1) = 0$

平均情况分析

- 化简上式可得：

$C_A(n)/(n+1) = C_A(n-1)/n + 2/(n+1)$

$= C_A(n-2)/(n-1) + 2/n + 2/(n+1)$

$= C_A(n-3)/(n-2) + 2/(n-1) + 2/n + 2/(n+1)$

…

$= C_A(1)/2 + 2\sum_{3\le k\le n+1} 1/k$

由于 $\sum_{3\le k\le n+1} 1/k \le \int_2^{n+1} \frac{dx}{x} < \log_e(n+1)$

所以得，$C_A(n) < 2(n+1)\log_e(n+1) = O(n\log n)$

各种排序算法的比较

方法	最坏复杂性	平均复杂性
冒泡排序	n^2	n^2
计数排序	n^2	n^2
插入排序	n^2	n^2
选择排序	n^2	n^2
堆排序	nlogn	nlogn
归并排序	nlogn	nlogn
快速排序	n^2	nlogn