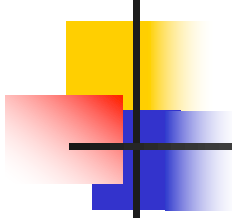


14 章 分治法

(Divide and Conquer)



提 纲

- 分治法基本思想
- 应用
 - 归并排序
 - 快速排序
 - 选择问题
- 复杂性的下限
 - 最大最小问题的下限
 - 排序算法的下限



14.1 分治法思想

分治法设计算法的思想是：

- 将问题分成(*divide*)多个子问题；
- 递归地(*conquer*)解决每个子问题；
- 将子问题的解合并(*combine*)成原问题的解；
- 分治法常常得到递归算法.
- Merge-Sort是用分治法设计算法的范例.
- 算法复杂性分析
 - Master method
 - Substitution method



Reminder: master method

- 递归方程: $T(n)=aT(n/b)+f(n)$, 式中 $a \geq 1$, $b > 1$, 为整数, $f(n) > 0$ 。
- 情形1. $f(n)=O(n^{\log a - \epsilon})$, $\epsilon > 0$, 为某一常数, $f(n)$ 的增长渐近地慢于 $n^{\log a}$ (慢 n^ϵ 倍).

Solution: $T(n)=\Theta(n^{\log a})$.

- 情形2: $f(n)=\Theta(n^{\log a} \lg^k n)$ $k \geq 0$ 为某一常数, $f(n)$ 和 $n^{\log a}$ 几乎有相同的渐近增长率.

Solution: $T(n)=\Theta(n^{\log a} \lg^{k+1} n)$.

- 情形3: $f(n)=\Omega(n^{\log a + \epsilon})$ $\epsilon > 0$ 为一常数. $f(n)$ 多项式地快于 $n^{\log a}$ (by an n^ϵ factor), $f(n)$ 满足以下规则性条件:

$af(n/b) \leq cf(n)$, $0 < c < 1$ 为常数.

Solution: $T(n) = \Theta(f(n))$.



适用条件

■ 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：
 - 该问题的规模缩小到一定的程度就可以容易地解决；
 - 该问题可以分解为若干个规模较小的相同问题；
 - 利用该问题分解出的子问题的解可以合并为该问题的解；
 - 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。
- 最后一条特征涉及到分治法的效率，如果各子问题是不独立的，则分治法要做许多不必要的工作，重复地解公共的子问题，此时虽然也可用分治法，但一般用**动态规划**较好。



步骤

分治法的基本步骤

- divide-and-conquer(P)

```
{  
  if ( | P | <= n0) adhoc(P); //解决小规模的问题  
  divide P into smaller subinstances P1,P2,...,Pk; //分解问题  
  for (i=1,i<=k,i++)  
    yi=divide-and-conquer(Pi); //递归的解各子问题  
  return merge(y1,...,yk); //将各子问题的解合并为原问题的解  
}
```

- 人们从大量实践中发现，在用分治法设计算法时，最好使子问题的规模大致相同。即将一个问题分成大小相等的 k 个子问题的处理方法是行之有效的。这种使子问题规模大致相等的做法是出自一种平衡(balancing)子问题的思想，它几乎总是比子问题规模不等的做法要好。



例14-1 [找出伪币]

- 现有16个硬币, 其中有一个是伪币, 并且伪币重量比真币轻。试用一台天平找出这枚伪币。
- 两两比较, 最坏情形需8次比较找出伪币。
- 分治法仅需4次比较找出伪币。



Compute a^n

- **Problem:** Compute a^n , where $n \in \mathbb{N}$.
- **Naive algorithm:** $\Theta(n)$.



■ Divide-and-conquer algorithm:

- $a^n = a^{n/2} \cdot a^{n/2}$ if n is even;
 $= a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a$ if n is odd.
- $T(n) = T(n/2) + \Theta(1)$
- Master方法, Case 2: $\log_b a = 0, k = 0$
- $T(n) = \Theta(\lg n)$



Fibonacci numbers

Recursive definition:

$$F_n = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2. \end{cases}$$

0 1 1 2 3 5 8 13 21 34 ...



Recursive definition:

$$F_n = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2. \end{cases}$$

0 1 1 2 3 5 8 13 21 34 ...

Naive recursive algorithm: $\Omega(\phi^n)$
(exponential time), where $\phi = (1 + \sqrt{5})/2$
is the *golden ratio*.



Bottom-up:

- Compute $F_0, F_1, F_2, \dots, F_n$ in order, forming each number by summing the two previous.
- Running time: $\Theta(n)$.



■ Recursive squaring

Theorem:
$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

Algorithm: Recursive squaring.
Time = $\Theta(\lg n)$.



例14-2 [金块问题]

- 有若干金块试用一台天平找出其中最轻和最重的金块.
- n 个数中找出最大和最小的数.
- 直接求解1:
 - 先找出最大值, 然后在剩下的 $n-1$ 个数中再找出最小值.
 - 需 $2n-3$ 次比较.



直接求解2:

- $\text{min} \leftarrow a[0]; \text{max} \leftarrow a[0];$
- For $i \leftarrow 1$ to $n-1$ do
 - if $a[i] < \text{min}$
 - $\text{min} \leftarrow a[i]$
 - else if $a[i] > \text{max}$ $\text{max} \leftarrow a[i]$
- 当输入数组未排好序时,算法要做 $2(n-1)$ 次比较.当输入为逆序时,算法要做 $n-1$ 次比较.



分治法求解

```
Max-min(A[0,n-1],max,min)
if n=1 max←min←a[0],return;
if n=2 {if a[0]≥a[1] max←a[0],min←a[1]
        else max←a[1],min←a[0];}
else m←n/2
      Max-min(A[0,m-1],max1,min1),
      Max-min(A[m,n-1],max2,min2),
      max←max(max1,max2),
      min←min(min1,min2),
      return.
```




例14-2 [金块问题] (续)

- 设 $c(n)$ 为使用分治法所需要的比较次数,假设 $n=2^k$ 则有:
 $c(1)=0, n==1$
 $c(2)=1, n==2$
 $c(n)=2c(n/2)+2, n > 2 .$
- 迭代展开可得: $c(n)= (3n/2)-2$ 。
- 分治法比直接求解法少用了**25%**次比较.
- 程序**14.1**是该算法的迭代实现(对任意 n 都适用).

程序14-1 找出最小值和最大值的非递归程序

```
template<class T>
bool MinMax(T w[], int n, T& Min, T& Max)
{
    // 寻找w[0:n-1]中的最小和最大值
    // 如果少于一个元素，则返回 false
    // 特殊情形： n <= 1
    if (n < 1) return false;
    if (n == 1) {Min = Max = 0;
                return true;}

    //对Min 和Max进行初始化
    int s; // 循环起点
```



程序14-1 找出最小值和最大值的非递归程序

```
if (n % 2) { // n 为奇数
    Min = Max = 0;
    s = 1;}
else { // n 为偶数, 比较第一对
    if (w[0] > w[1]) {
        Min = 1;
        Max = 0;}
    else {Min = 0;
        Max = 1;}
    s = 2;}
```

程序14-1 找出最小值和最大值的非递归程序

// 比较余下的数对

```
for (int i = s; i < n; i += 2) {
```

 // 寻找w[i] 和w[i+1]中的较大者

 // 然后将较大者与 w[Max]进行比较

 // 将较小者与 w[Min]进行比较

```
    if (w[i] > w[i+1]) {
```

```
        if (w[i] > w[Max]) Max = i;
```

```
        if (w[i+1] < w[Min]) Min = i + 1;}
```

```
    else {
```

```
        if (w[i+1] > w[Max]) Max = i + 1;
```

```
        if (w[i] < w[Min]) Min = i;}
```

```
    }
```

```
return true;
```

```
}
```



算法14.1分析

- 当 n 为奇数时, $n=2m+1$,比较 m 对相邻元素,
比较次数为 $3*m=3*(n-1)/2$

$$=3n/2-3/2=[3n/2]-1/2-3/2$$

$$=[3n/2]-2$$

[]表示向上取整.

- 当 n 为偶数时, $n=2m$,比较 $m-1$ 对相邻元素
比较次数为 $1+3*(m-1)=1+3*(n-2)/2$

$$=1+3n/2-3$$

$$=[3n/2]-2$$

- 该算法所用比较次数最少

例14.3 大整数的乘法

- 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
- 直接方法： $O(n^2)$ ✖效率太低

- 分治法：

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

$$X = \overset{n/2\text{位}}{\boxed{A}} \overset{n/2\text{位}}{\boxed{B}} \quad Y = \overset{n/2\text{位}}{\boxed{C}} \overset{n/2\text{位}}{\boxed{D}}$$

- 复杂度分析

- $T(n) = \begin{cases} O(1) & n = 1 \\ 4T(n/2) + O(n) & n > 1 \end{cases}$
- $T(n) = O(n^2)$ ✖没有改进☹

算法改进

- 为了降低时间复杂度，必须减少乘法的次数。为此，我们把XY写成另外的形式：
 - $XY = ac \cdot 2^n + ((a-c)(b-d) + ac + bd) \cdot 2^{n/2} + bd$ 或
 - $XY = ac \cdot 2^n + ((a+c)(b+d) - ac - bd) \cdot 2^{n/2} + bd$
- 复杂性：
 - 这两个算式看起来更复杂一些，但它们仅需要3次 $n/2$ 位乘法[ac、bd和 $(a \pm c)(b \pm d)$]，于是
 - $$T(n) = \begin{cases} O(1) & n = 1 \\ 3T(n/2) + O(n) & n > 1 \end{cases}$$
 - $T(n) = O(n^{\log_3}) = O(n^{1.59})$ ✓较大的改进😊
- 细节问题：两个XY的复杂度都是 $O(n^{\log_3})$ ，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。



更快的方法

- 直接方法: $O(n^2)$ ——效率太低
- 分治法: $O(n^{1.59})$ ——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。
 - 最终的, 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 $O(n \log n)$ 时间内解决。
 - 是否能找到线性时间的算法? 目前为止还没有结果。



例14-4 [矩阵乘法]

两个 $n \times n$ 阶的矩阵A与B的乘积是另一个 $n \times n$ 阶矩阵C，C的元素为：

$$C(i, j) = \sum_k A(i, k)B(k, j)$$

```
template <class T>
void MatrixMulti( T** A, T**B, T** C, int n)
{
    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++){
            T sum=0;
            for (int k=0; k<n; k++)
                sum+=A[i][k]*B[k][j]
            c[i][j]=sum;
        }
}
```

每一个C(i, j)都用此公式计算,则计算C 所需要的运算次数为 n^3 次乘法和 $n^2(n-1)$ 次加法.



矩阵乘法-分治法

- 2×2 分块矩阵乘法

$$\left[\begin{array}{c|c} r & s \\ \hline t & u \end{array} \right] = \left[\begin{array}{c|c} a & b \\ \hline c & d \end{array} \right] \cdot \left[\begin{array}{c|c} e & f \\ \hline g & h \end{array} \right]$$

$$C = A \cdot B$$

- a,b,c,d是A的4个 $(n/2)$ 阶子矩阵;e,f,g,h是B的4个 $(n/2)$ 阶子矩阵.

- 
- 8个 $(n/2)$ 阶矩阵乘法 and 4个 n 阶矩阵加法

$$\left. \begin{array}{l} r = ae + bg \\ s = af + bh \\ t = ce + dh \\ u = cf + dg \end{array} \right\} \begin{array}{l} \text{recursive} \\ 8 \text{ mults of } (n/2) \times (n/2) \text{ submatrices} \\ 4 \text{ adds of } (n/2) \times (n/2) \text{ submatrices} \end{array}$$

- 分治法得到的算法分析
- $T(n) = 8T(n/2) + \Theta(n^2)$
- $\Theta(n^2)$ – 4个 $(n/2)$ 阶矩阵加法的计算时间



■ $n^{\log a} = n^{\log 8} = n^3 \Rightarrow \text{CASE 1} \Rightarrow T(n) = \Theta(n^3)$

- 从分治法没得到好处！
- 原因：矩阵分割和再组合花费的时间开销太大。
- 如何采用分治思想设计算法来提高算法性能？



例14-3 Strassen矩阵乘法算法

Strassen算法计算以下7个 $n/2$ 阶矩阵: P_1, \dots, P_7 , 需7次 $n/2$ 阶矩阵乘法.

$$P_1 = a \cdot (f - h)$$

$$P_2 = (a + b) \cdot h$$

$$P_3 = (c + d) \cdot e$$

$$P_4 = d \cdot (g - e)$$

$$P_5 = (a + d) \cdot (e + h)$$

$$P_6 = (b - d) \cdot (g + h)$$

$$P_7 = (a - c) \cdot (e + f)$$



例14-3

- 8次 $(n/2)$ 阶矩阵加/减法得到乘积矩阵C:

$$r = P_5 + P_4 - P_2 + P_6$$

$$s = P_1 + P_2$$

$$t = P_3 + P_4$$

$$u = P_5 + P_1 - P_3 - P_7$$

例如:

$$r = P_5 + P_4 - P_2 + P_6$$

$$= (a+d)(e+h) + d(g-e) - (a+b)h + (b-d)(g+h)$$

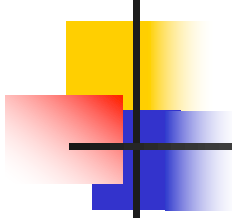
$$= ae + ah + de + dh + dg - de - ah - bh$$

$$+ bg + bh - dg - dh = ae + bg$$



Strassen's idea:

- 合并分块矩阵以减少乘法次数
- 2×2 矩阵相乘可用7次乘法，所以仅需7次递归调用.
- $T(n) = 7T(n/2) + \Theta(n^2)$
- Master 方法:
 $\log_b a = \log_2 7$, case 1: $\varepsilon = \log_2 7 - 2.5$
- $T(n) = \Theta(n^{\log_2 7}) = \Theta(n^{2.81})$

- 
- **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。
 - 在**Strassen**之后又有许多算法改进了矩阵乘法的计算时间复杂性。
 - **Best to date** (of theoretical interest only): $\Theta(n^{2.376})$ (Don Coppersmith, Shmuel Winograd, Matrix Multiplication via Arithmetic Progressions, STOC 1987: 1-6).
 - 是否能找到 $O(n^2)$ 的算法? 目前为止还没有结果。



14. 2应用

- Defective Chessboard
- 归并排序 (Merge Sort)
- 快速排序 (Quick Sort)
- 选择 (Selection Problem)

Defective Chessboard

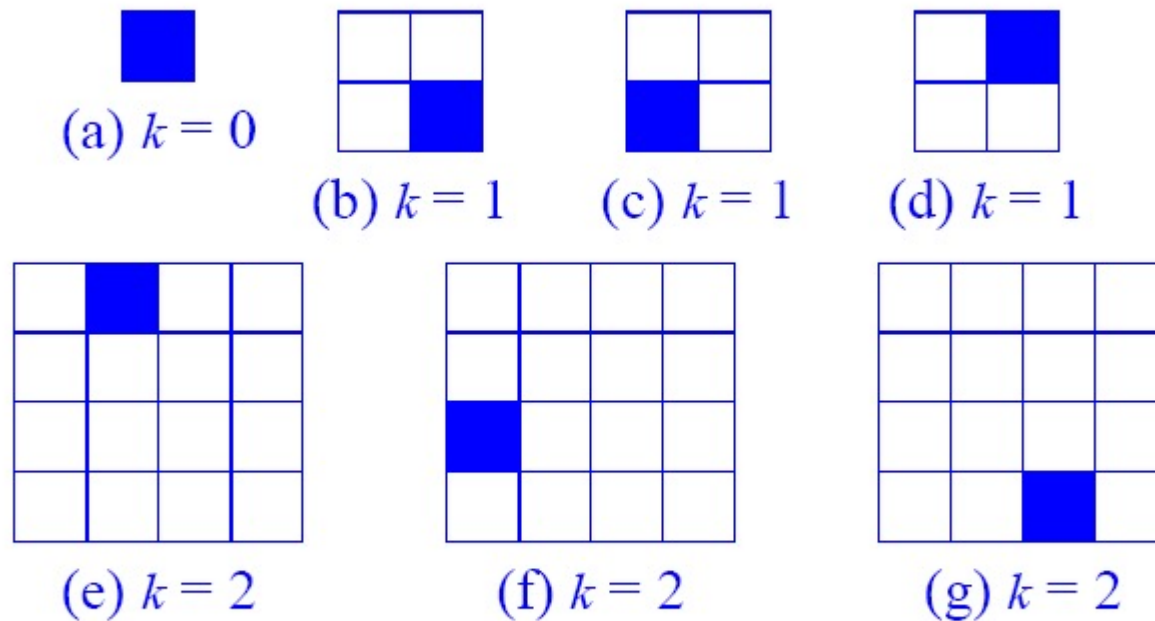
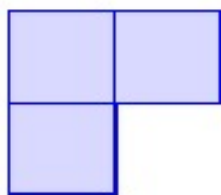


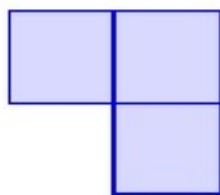
Figure 14.3 Example defective chessboards



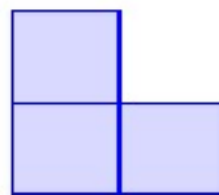
Defective Chessboard



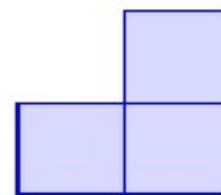
(a)



(b)



(c)

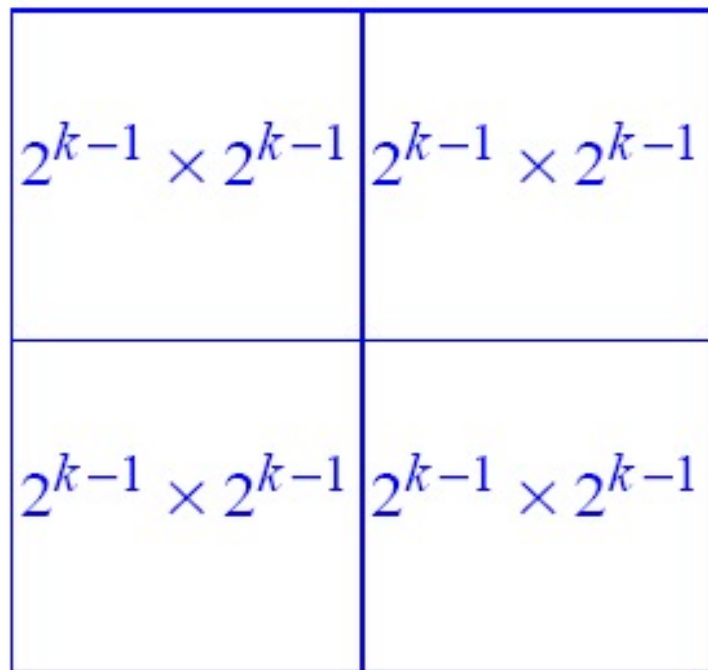


(d)

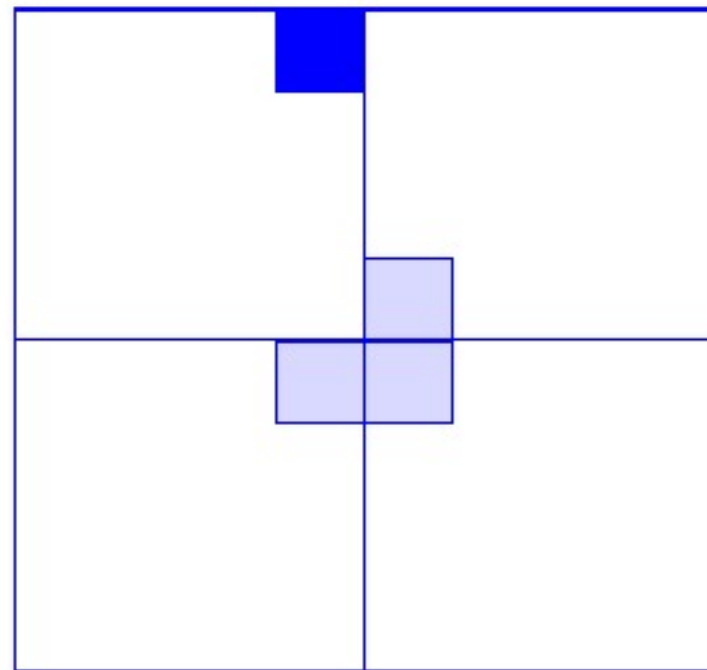
Figure 14.4 Triominoes with different orientations

Defective Chessboard

残缺棋盘的问题要求用三格板 (triominoes) 覆盖残缺棋盘。在此覆盖中，两个三格板不能重叠，三格板不能覆盖残缺方格，但必须覆盖其他所有的方格



(a) Partitioning



(b) Triomino placed

Figure 14.5 Partitioning a $2^k \times 2^k$ board



```
void TileBoard(int tr, int tc, int dr, int dc, int size)
```

```
{// 覆盖残缺棋盘
```

```
    if (size == 1) return;
```

```
    int t = tile++, // 所使用的三格板的数目
```

```
    s = size/2; // 象限大小
```

```
// 覆盖左上象限
```

```
    if (dr < tr + s && dc < tc + s)
```

```
        // 残缺方格位于本象限
```

```
        TileBoard(t r, tc, dr, dc, s);
```

```
    else {// 本象限中没有残缺方格
```

```
        // 把三格板t 放在右下角
```

```
        Board[tr + s - 1][tc + s - 1] = t;
```

```
        // 覆盖其余部分
```

```
        TileBoard(t r, tc, tr+s-1, tc+s-1, s);}
```

```
// 覆盖右上象限
```

```
    if (dr < tr + s && dc >= tc + s)
```

```
        // 残缺方格位于本象限
```

```
        TileBoard(t r, tc+s, dr, dc, s);
```

```
    else {// 本象限中没有残缺方格
```

```
        // 把三格板t 放在左下角
```

```
        Board[tr + s - 1][tc + s] = t;
```

```
        // 覆盖其余部分
```



```
TileBoard(tr, tc+s, tr+s-1, tc+s, s);}
```

```
// 覆盖左下象限
```

```
if (dr >= tr + s && dc < tc + s)
```

```
// 残缺方格位于本象限
```

```
TileBoard(tr+s, tc, dr, dc, s);
```

```
else { // 把三格板t 放在右上角
```

```
Board[tr + s][tc + s - 1] = t;
```

```
// 覆盖其余部分
```

```
TileBoard(tr+s, tc, tr+s, tc+s-1, s);}
```

```
// 覆盖右下象限
```

```
if (dr >= tr + s && dc >= tc + s)
```

```
// 残缺方格位于本象限
```

```
TileBoard(tr+s, tc+s, dr, dc, s);
```

```
else { // 把三格板t 放在左上角
```

```
Board[tr + s][tc + s] = t;
```

```
// 覆盖其余部分
```

```
TileBoard(tr+s, tc+s, tr+s, tc+s, s);}
```

```
}
```

```
void OutputBoard(int size)
```

```
{
```

```
for (int i = 0; i < size; i++) {
```

```
for (int j = 0; j < size; j++)
```

```
cout << setw (5) << Board[i][j];
```

```
cout << endl;
```

```
}
```

```
}
```



Defective Chessboard

■ $n=4^k$

- 需 $(n-1)/3$ 个3-方块填满棋盘
- 算法的时间复杂度:
- $t(n)=4t(n/4)+c, \log_b a=1, \text{case } 1: \epsilon=0.5$
- $t(n)=\Theta(n)$



14.2.2 归并排序

- 我们采用平衡分割法来分割 n 个元素,即将 n 个元素分为 A 和 B 两个集合,其中 A 集合中含有 n/k 个元素, B 中包含其余的元素.
- 然后递归地使用分治法对 A 和 B 进行排序.
- 当 A 或 B 内元素数 $< k$ 时使用插入排序.
- 然后采用一个称为归并 (merge) 的过程, 将已排好序的 A 和 B 合并成一个集合.



例14.5

- 考虑8个元素,值分别为[10,4,6,3,8,2,5,7]。
- 1) $k=2$ 时排序;
 - $A_1=[10,4, 6, 3]$ $A_2=[8,2,5,7]$
 - $A_{11}=[10,4]$ $A_{12}=[6, 3]$ $A_{21}=[8,2]$ $A_{22}=[5,7]$
 - $A_{111}=[10]$ $A_{112}=[4]$ $A_{121}=[6]$ $A_{122}=[3]$ $A_{211}=[8]$
 $A_{212}=[2]$ $A_{221}=[5]$ $A_{222}=[7]$
- 2) $k=4$ 时排序,



图14-6 分治排序算法的伪代码

```
template<class T>
void sort( T E, int n)
{ //对E中的n 个元素进行排序, k为全局变量
    if (n >= k) {
        i = n/k;
        j = n-i;
        令A 包含E中的前 i 个元素
        令 B 包含E中余下的 j 个元素
        sort(A,i);
        sort(B,j);
        merge(A,B,E,i,j); //把A 和 B 合并到 E
    }
    else 使用插入排序算法对 E 进行排序
}
```

算法复杂度

- 设 $t(n)$ 为分治排序算法，则有以下递推公式：

$$t(n) = \begin{cases} d & n < k \\ t(n/k) + t(n - n/k) + cn & n \geq k \end{cases}$$

- 当 $n/k \approx n - (n/k)$ 时 $t(n)$ 的值最小(balance 原理)
- 因此,当 $k=2$ 时,分治法通常具有最佳性能:当 $k>2$ 时递归展开的深度 $\log_a n$, $a=k/(k-1)$,超过 $\log_2 n$.



算法复杂度（续1）

当 $k=2$ 时,可得到如下递推公式:

$$t(n) = \begin{cases} d & n \leq 1 \\ t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + cn & n > 1 \end{cases}$$



14.2.3 快速排序

- 分治法还可以用于实现另一种完全不同的排序方法:快速排序(quick sort).
- 在这种方法中, n 个元素被分成三段:左段left, 右段right和中段middle.
- 中段仅包含一个元素;左段中各元素都小于等于中段元素;右段中各元素都大于等于中段元素.因此left和right中的元素可以独立排序,并且不必对left和right的排序结果进行合并, middle中的元素被称为支点(pivot).



图14-9 快速排序的伪代码

//使用快速排序方法对 $a[0:n-1]$ 排序

- 从 $a[0:n-1]$ 中选择一个元素作为 $middle$ ，该元素为支点
- 把余下的元素分割为两段 $left$ 和 $right$ ，使得 $left$ 中的元素都小于等于支点，而 $right$ 中的元素都大于等于支点
- 递归地使用快速排序方法对 $left$ 进行排序
- 递归地使用快速排序方法对 $right$ 进行排序
- 所得结果为 $left + middle + right$



程序14-6 快速排序

```
template<class T>
void QuickSort(T*a, int n)
{
    // 对 a[0:n-1] 进行快速排序
    // 要求 a[n] 必需有最大关键值
    quickSort(a, 0, n-1);
}

template<class T>
void quickSort(T a[], int l, int r)
{
    // 排序 a[l:r], a[r+1] 有大值
    if (l >= r) return;
    int i = l,    // 从左至右的游标
        j = r + 1; // 从右到左的游标
    T pivot = a[l];
```



程序14-6 快速排序（续1）

```
// 把左侧  $\geq$  pivot 的元素与右侧  $\leq$  pivot 的元素进行交换
while (true) {
    do { // 在左侧寻找  $\geq$  pivot 的元素
        i = i + 1;
    } while (a[i] < pivot);
    do { // 在右侧寻找  $\leq$  pivot 的元素
        j = j - 1;
    } while (a[j] > pivot);
    if (i  $\geq$  j) break; // 未发现交换对象
    Swap(a[i], a[j]);
}
```




程序14-6 快速排序（续2）

```
// 设置 pivot
```

```
a[l] = a[j];
```

```
a[j] = pivot;
```

```
quickSort(a, l, j-1); // 对左段排序
```

```
quickSort(a, j+1, r); // 对右段排序
```

```
}
```



分划用的比较次数

- 当关键字彼此不同时,例如, $1, 2, \dots, n$,永远有:
 $j=i-1$.这是因为:分划结束时 j 指向左边区间的右端点($a[j] < \text{pivot}$);而 i 指向右边区间的左端点($a[i] > \text{pivot}$).
 - 每次进行 $i-1 + r-j+1 = r-1+2$ 次比较.
 - 第一遍分划进行 $n+1$ 次比较.
 - 等于 pivot 的关键字被均匀的分在两边.



算法的时间复杂度

程序14-6需要的递归栈空间为 $O(n)$.

- 如使用堆栈将递归化为迭代并每次分划后将长度较大的段压入栈中则栈空间长度为 $O(\log n)$.
- 在最坏情况下`left`总是为空,快速排序所需的计算时间为 $\Theta(n^2)$.
- 在最好情况下,`left`和`right`中的元素数目大致相同,快速排序的复杂性为 $\Theta(n \log_2 n)$.
- 快速排序的平均复杂性也是 $\Theta(n \log_2 n)$.



Worst case analysis

■ Worst case:

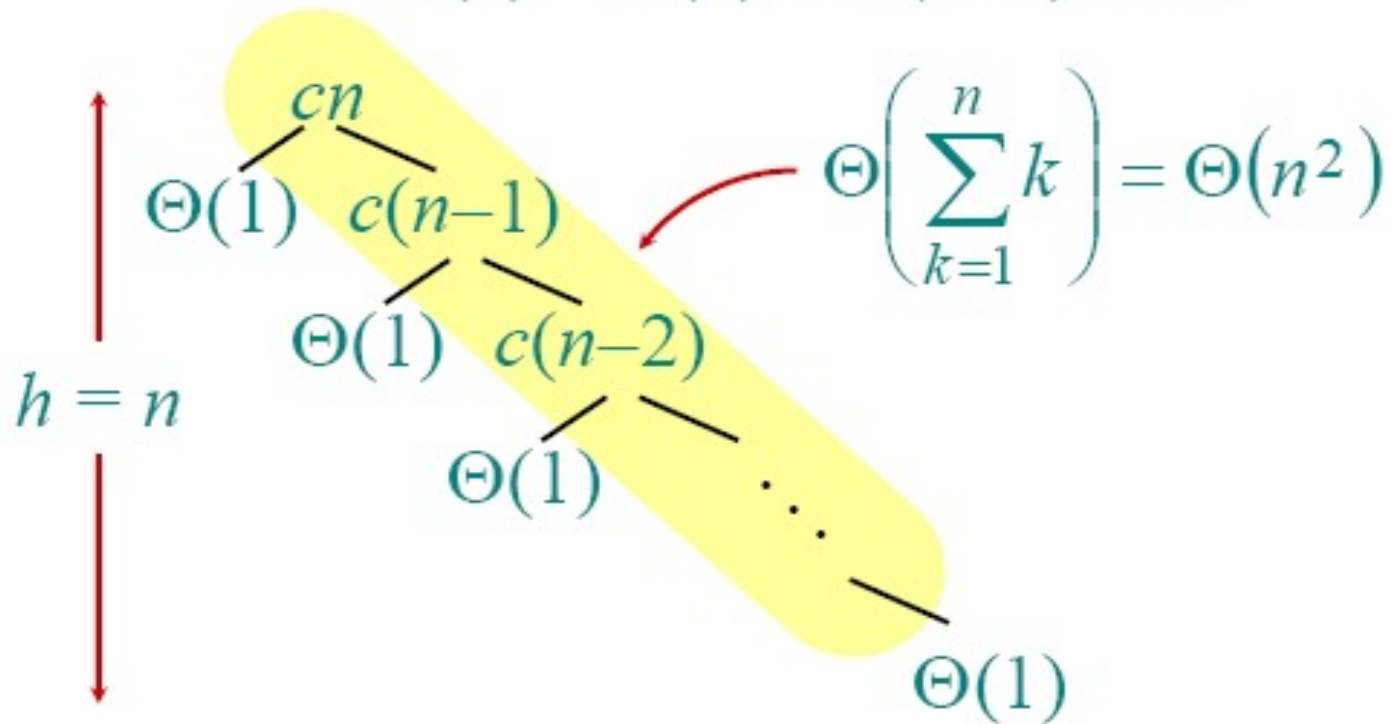
- Input sorted or reverse sorted.
- Partition around min or max element.
- One side of partition always has no elements.

$$\begin{aligned}T(n) &= T(0) + T(n-1) + \Theta(n) \\&= \Theta(1) + T(n-1) + \Theta(n) \\&= T(n-1) + \Theta(n) \\&= \Theta(n^2) \quad (\textit{arithmetic series})\end{aligned}$$

Worst case analysis

- 迭代展开:

$$T(n) = T(0) + T(n-1) + cn$$





Best case analysis

- 如果分划点总是在中间:

$$\begin{aligned}T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n)\end{aligned}$$

- 然而分点落在 $(1/10n, 9/10n)$, 我们也有 $\Theta(n \lg n)$:

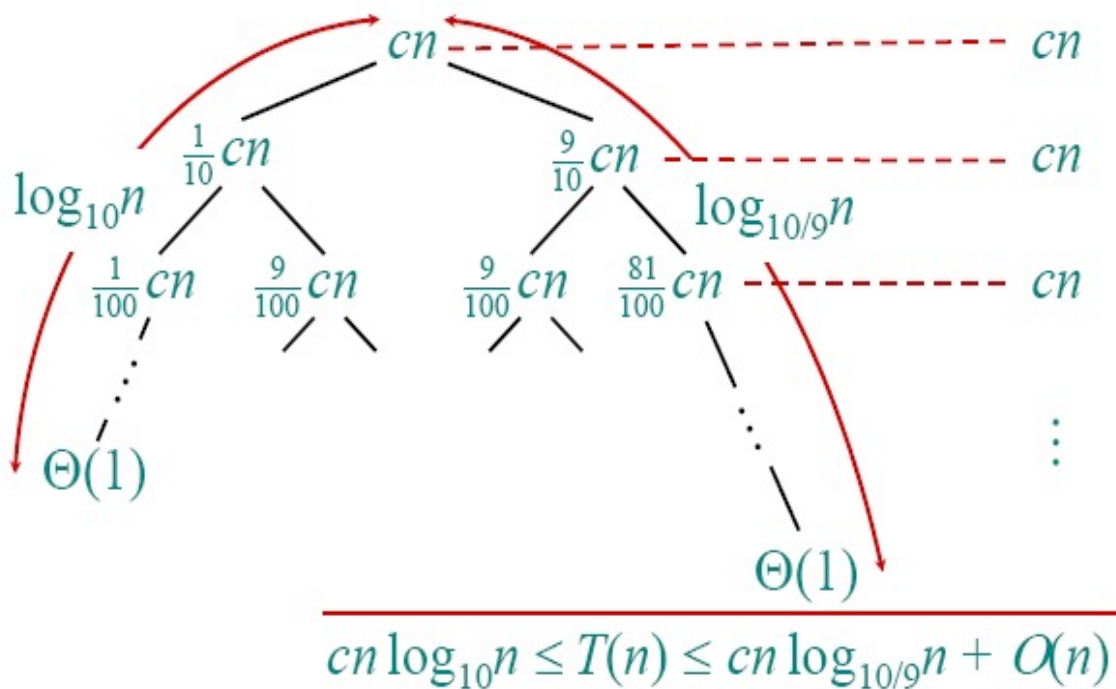
$$T(n) = T(n/10) + T(9n/10) + \Theta(n)$$

运气也很好! (上式中应取整!)

- 分划点碰上好运气的概率很大. 所以快速排序平均情形性能很好!

Best case

递归树



- 归纳证明上述不等式



平均情形分析

设 $t(n) \leq d \quad n \leq 1$

- 当 $n > 1$ 时

$$t(n) \leq cn + \frac{1}{n} \sum_{s=0}^{n-1} [t(s) + t(n-s-1)]$$

- 假定关键字彼此不同则平均关键字比较次数

$$t(n) = n + 1 + \frac{1}{n} \sum_{s=0}^{n-1} \{t(s) + t(n-s-1)\}$$



平均情形分析

■ 迭代计算如下:

$$nt(n) = n(n+1) + 2 \sum_{s=0}^{n-1} t(s)$$

$$(n+1)t(n+1) = (n+1)(n+2) + 2 \sum_{s=0}^n t(s)$$

$$(n+1)t(n+1) - nt(n) = 2(n+1) + 2t(n) \Rightarrow$$

$$(n+1)t(n+1) = (n+2)t(n) + 2(n+1)$$

$$\frac{t(n+1)}{n+2} = \frac{t(n)}{n+1} + \frac{2}{n+2}$$



平均情形分析

■ 令 $f(n)=t(n)/(n+1)$, 有:

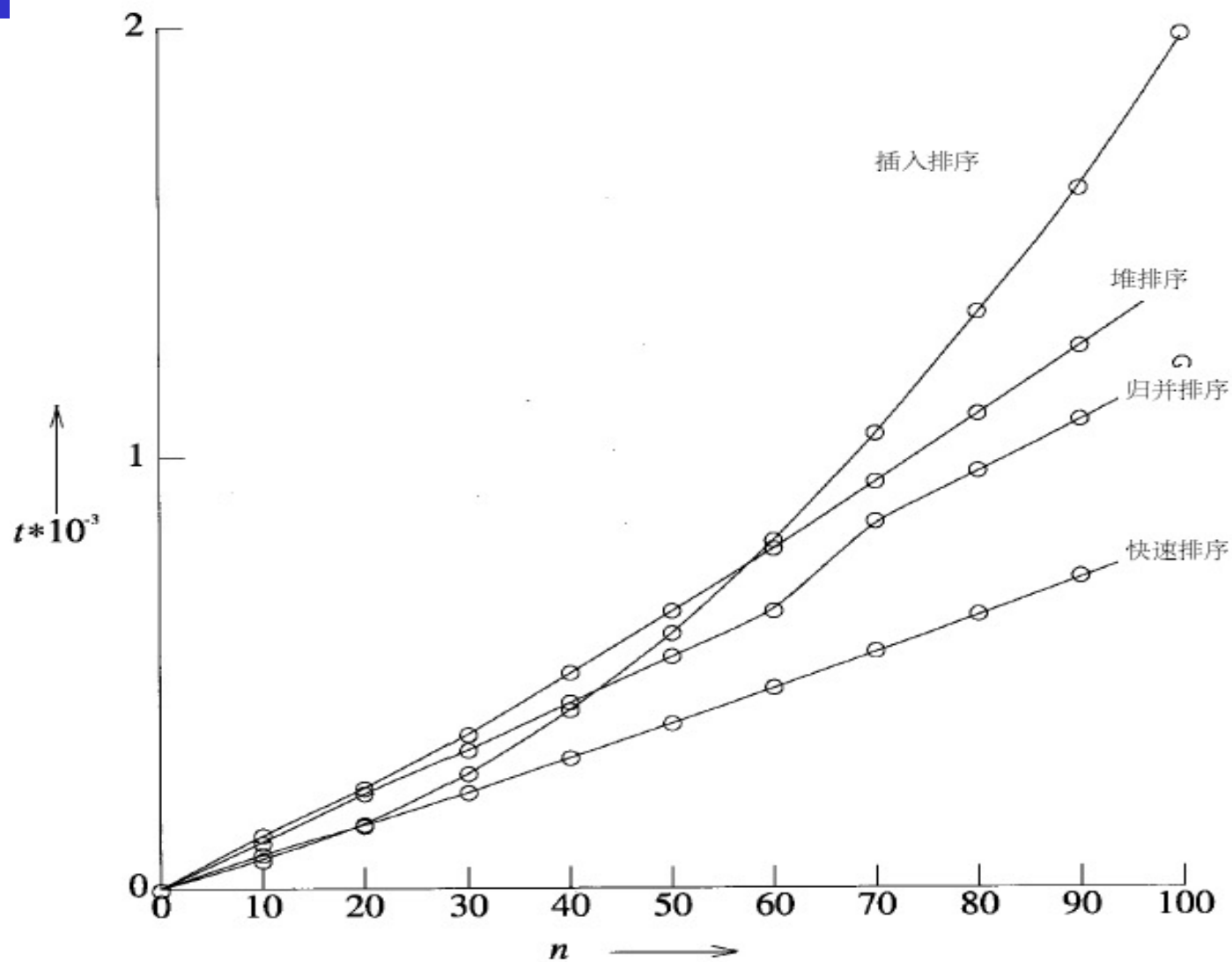
- $f(n)=f(n-1)+1/(n)$
- $f(n)=f(1)+\sum(1/n)$ 求和从 $n=3$ 开始.
- 调和级数收敛到 $\log_e n$, 所以
$$t(n)=(n+1)f(n)=\Theta(n\log_2 n)$$



图14-10 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	$n \log n$	$n \log n$
归并排序	$n \log n$	$n \log n$
快速排序	n^2	$n \log n$

图14-12 各排序算法平均时间的曲线图





14.2.4 选择问题

定义：对于给定的 n 个元素的数组 $a[0:n-1]$ ，要求从中找出第 k 小的元素。

- 选择问题可在 $O(n \log n)$ 时间内解决，方法是首先对这 n 个元素进行排序(如使用堆排序或归并排序)，然后取出 $a[k-1]$ 中的元素。
- 若使用快速排序（如图14-12所示），可以获得更好的平均性能。尽管该算法在最坏情形下有一个比较差的渐近复杂性 $O(n^2)$ 。

程序14-7 寻找第k 个元素

```
template<class T>
T Select(T a[], int n, int k)
{
    // 返回a[0:n-1]中第k小的元素
    // 假定 a[n] 是一个伪最大元素
    if (k < 1 || k > n) throw OutOfBounds();
    return select(a, 0, n-1, k);
}

template<class T>
T select(T a[], int l, int r, int k)
{
    // 在 a[l:r]中选择第k小的元素
    if (l >= r) return a[l];
    int i = l,    // 从左至右的游标
        j = r + 1; // 从右到左的游标
    T pivot = a[l];
```

程序14-7 寻找第k 个元素（续1）

// 把左侧 \geq pivot 的元素与右侧 \leq pivot 的元素进行交换

```
while (true) {  
    do { // 在左侧寻找  $\geq$  pivot 的元素  
        i = i + 1;  
    } while (a[i] < pivot);  
    do { // 在右侧寻找  $\leq$  pivot 的元素  
        j = j - 1;  
    } while (a[j] > pivot);  
    if (i  $\geq$  j) break; // 未发现交换对象  
    Swap(a[i], a[j]);  
}
```

程序14-7 寻找第k 个元素（续2）

```
    if (j - l + 1 == k) return pivot;

    // 设置pivot
    a[l] = a[j];
    a[j] = pivot;

    // 对一个段进行递归调用
    if (j - l + 1 < k)
        return select(a, j+1, r, k-j+l-1);
    else return select(a, l, j-1, k);
}
```


程序14-7复杂度分析

程序14-7在最坏情况下的复杂性是 $\Theta(n^2)$

- 如果left 和right总是同样大小或者相差不超过一个元素，那么可以得到以下递归表式：

$$t(n) \leq \begin{cases} d & n \leq 1 \\ t(\lfloor n/2 \rfloor) + cn & n > 1 \end{cases} \quad (14-10)$$

- 如果n 是2的幂，则通过使用迭代方法，可以得到 $t(n) = \Theta(n)$ 。
- 提示：选择较好的分点可得到较好的性能



中间的中规则

- 若仔细地选择支点元素，则最坏情况下的时间开销也可以变成 $\Theta(n)$ 。
- 一种选择支点元素的方法是使用“中间的中 (median-of-median)”规则：首先将数组 a 中的 n 个元素分成 n/r 组， r 为某一整常数，除了最后一组外，每组都有 r 个元素。然后通过在这组中对 r 个元素进行排序来寻找每组中位于中间位置的元素。最后对所得到的 n/r 个中间元素，递归使用选择算法，求得“中间之中间”作为支点元素。



例14-6 [中间的中间]

- 考察如下情形： $r=5$, $n=27$, 并且 $a=[2, 6, 8, 1, 4, 10, 20, 6, 22, 11, 9, 8, 4, 3, 7, 8, 16, 11, 10, 8, 2, 14, 15, 1, 12, 5, 4]$ 。
- 说明：这27个元素可以被分为6组，每组的中间元素分别为4, 11, 7, 10, 12和4，这些中间元素的中间元素为7，将其取为支点元素。

定理14-2

当按“中间的中间”规则选取支点元素时，以下结论为真：

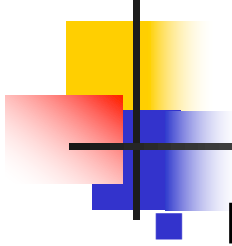
若 $r=5$ ，且 a 中所有元素都不同，那么当 $n \geq 24$ 时，有 $\max\{|left|, |right|\} \leq 3n/4$ 。

有 $\left\lceil \frac{1}{2} \left\lfloor \frac{n}{5} \right\rfloor \right\rceil$ 多个中间元素不大于 mm ；

大于 mm 元素的数目有上界

$$n - \frac{1}{2} \left\lfloor \frac{n}{5} \right\rfloor \leq n - 1.5(n/5 - 4/5) = 0.7n + 1.2$$

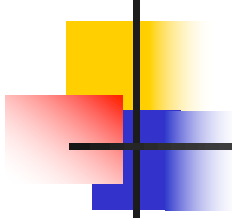
■ 因此，当 $n \geq 24$ 时， $0.7n + 1.2 \leq 3n/4$



■ $r=5$

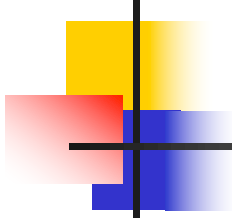
$$t(n) \leq t(n/5) + t(3n/4) + cn$$

- 用归纳法可证明当 $n \geq 24$ 时有 $t(n) \leq 20cn$
- 当 $r=9$, 当 $n \geq 90$ 时, 有 $\max\{|left|, |right|\} \leq 7n/8$ 。

- 
- 分析r=7时选择算法的时间复杂度。
 - 当r=7时，>mm的元素个数不超过

$$\begin{aligned}n - 4 \left\lceil \frac{1}{2} \left\lfloor \frac{n}{7} \right\rfloor \right\rceil &\leq n - 2 \left\lfloor \frac{n}{7} \right\rfloor \\&\leq n - 2 \left(\frac{n}{7} - \frac{6}{7} \right) = \frac{5n}{7} + \frac{12}{7} \\&\leq \frac{5n}{6} \quad n \geq 15\end{aligned}$$

- 所以, $T(n) \leq T(n/7) + T(5n/6) + cn$, 当 $n > 14$ 时成立。

- 
- 分析当 r 取3时是否能在 $O(n)$ 时间内求解选择问题?
 - 当 $r=3$ 时, 找不到满足 $\alpha + 1/3 < 1$ 的正数 α 使得 $n - 2 * \lceil (1/2)(n/3) \rceil \leq \alpha n$ ($\lceil \cdot \rceil$ 表示向上取整). 在 $r=7$ 时我们找到 $\alpha=5/6$, $r=5$ 时 $\alpha=3/4$.
 - 所以, 当 $r=3$ 时, 算法不能保证在 $O(n)$ 时间内求解.

中间的中间规则的复杂度

■ 用于寻找第 k 个元素的时间 $t(n)$ 可按如下递归公式来计算：

$$t(n) = \begin{cases} cn \log n & n < 90 \\ t(\lceil n/9 \rceil) + t(\lfloor 7n/8 \rfloor) + cn & n \geq 90 \end{cases} \quad (14-11)$$

- 在上述递归公式中，假设当 $n < 90$ 时使用任意的求解算法，当 $n \geq 90$ 时，采用“中间的中间”规则用分治法求解。利用归纳法可以证明，当 $n \geq 90$ 时有 $t(n) \leq 72cn$ 。



14.2.5 距离最近的点对

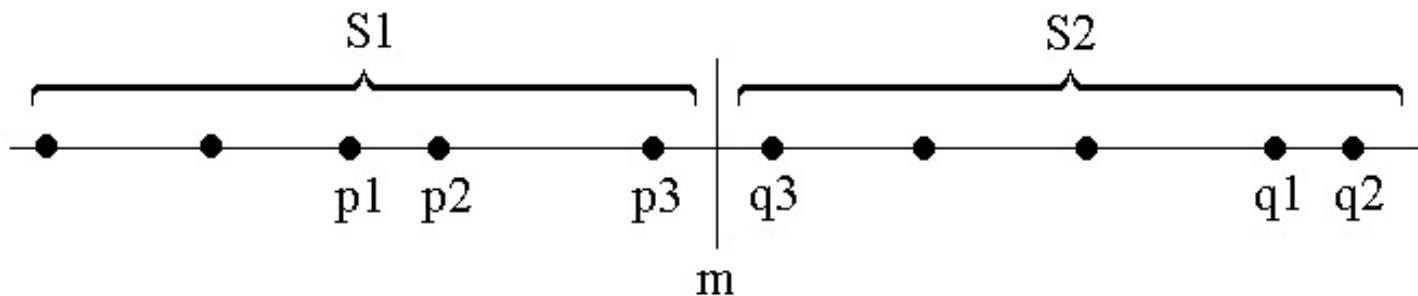
- 问题描述：给定平面上 n 个点，找其中的一对点，使得在 n 个点所组成的所有点对中，该点对间的距离最小。
- 说明：
 - 严格来讲，最接近点对可能多于一对，为简便起见，我们只找其中的一对作为问题的解。
 - 一个简单的做法是将每一个点与其他 $n-1$ 个点的距离算出，找出最小距离的点对即可。该方法的时间复杂性是 $T(n)=n(n-1)/2+n=O(n^2)$ ，效率较低。

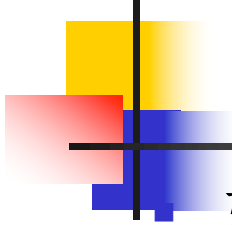


一维空间中的情形

- 为了使问题易于理解和分析，先来考虑一维的情形。此时， S 中的 n 个点退化为 x 轴上的 n 个实数 x_1, x_2, \dots, x_n 。最接近点对即为这 n 个实数中相差最小的2个实数。
- 一个简单的办法是先把 x_1, x_2, \dots, x_n 排好序，再进行一次线性扫描就可以找出最接近点对， $T(n)=O(n\log n)$ 。然而这种方法无法推广到二维情形。

- 假设我们用x轴上某个点 m 将 S 划分为2个子集 S_1 和 S_2 ，基于平衡子问题的思想，用 S 中各点坐标的中位数来作分割点。
- 递归地在 S_1 和 S_2 上找出其最接近点对 $\{p_1, p_2\}$ 和 $\{q_1, q_2\}$ ，并设 $d = \min\{|p_1 - p_2|, |q_1 - q_2|\}$ ， S 中的最接近点对或者是 $\{p_1, p_2\}$ ，或者是 $\{q_1, q_2\}$ ，或者是某个 $\{p_3, q_3\}$ ，其中 $p_3 \in S_1$ 且 $q_3 \in S_2$ 。
- 能否在线性时间内找到 p_3, q_3 ？





如果S的最接近点对是 $\{p_3, q_3\}$ ，即 $|p_3 - q_3| < d$ ，则 p_3 和 q_3 两者与 m 的距离不超过 d ，即 $p_3 \in (m-d, m]$ ， $q_3 \in (m, m+d]$ 。

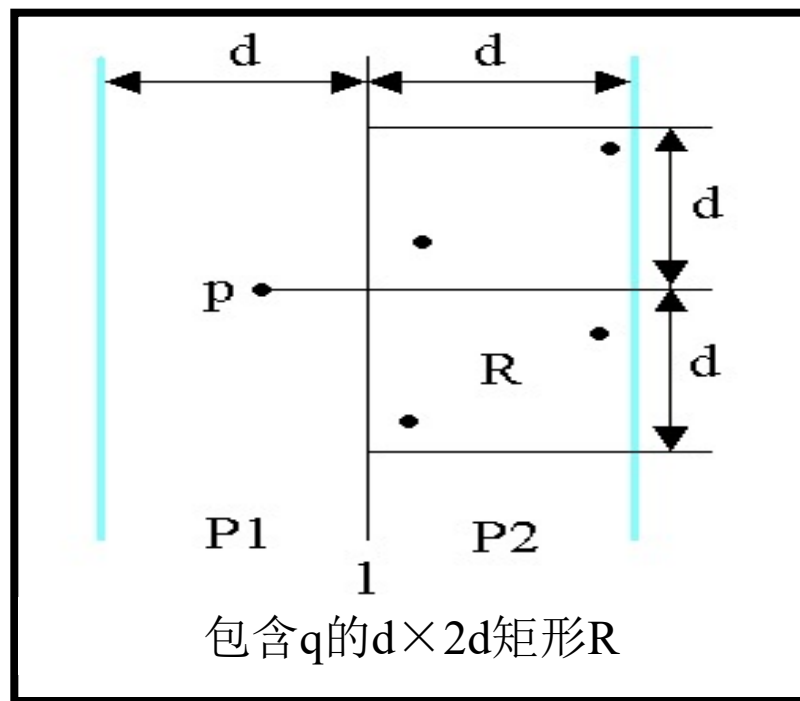
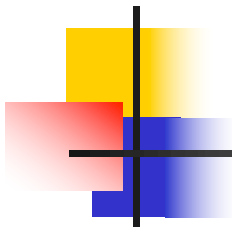
- 由于在 S_1 中，每个长度为 d 的半闭区间至多包含一个点（否则必有两点距离小于 d ），并且 m 是 S_1 和 S_2 的分割点，因此 $(m-d, m]$ 中至多包含 S 中的一个点。
- 因此，我们用线性时间就能找到区间 $(m-d, m]$ 和 $(m, m+d]$ 中所有点，即 p_3 和 q_3 。从而我们用线性时间就可以将 S_1 的解和 S_2 的解合并成为 S 的解。
- 分割点 m 的选取不当，会造成 $|S_i|=1$ ， $|S_j|=n-1$ 的情形，使得 $T(n) = T(n-1) + O(n) = O(n^2)$ 。这种情形可以通过“平衡子问题”方法加以解决：选取各点坐标的中位数作分割点。

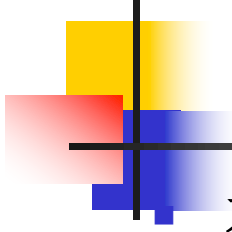


二维空间的最接近点对问题

下面来考虑二维的情形。

- 选取一垂直线 $l:x=m$ 来作为分割直线。其中 m 为 S 中各点 x 坐标的中位数。由此将 S 分割为 S_1 和 S_2 。
- 递归地在 S_1 和 S_2 上找出其最小距离 d_1 和 d_2 ，并设 $d=\min\{d_1, d_2\}$ ， S 中的最接近点对或者是 d ，或者是某个 $\{p, q\}$ ，其中 $p \in P_1$ 且 $q \in P_2$ 。
- 能否在线性时间内找到 p, q ？
 - 考虑 P_1 中任意一点 p ，它若与 P_2 中的点 q 构成最接近点对的候选者，则必有 $\text{distance}(p, q) < d$ 。满足这个条件的 P_2 中的点一定落在一个 $d \times 2d$ 的矩形 R 中，如图所示。
 - 由 d 的意义可知， P_2 中任何2个 S 中的点的距离都不小于 d 。
 - 结论：矩形 R 中最多只有6个 S 中的点（证明略）。





为了确切地知道要检查哪6个点，可以将 p 和 $P2$ 中所有 $S2$ 的点投影到垂直线 l 上。由于能与 p 点一起构成最接近点对候选者的 $S2$ 中点一定在矩形 R 中，所以它们在直线 l 上的投影点距 p 在 l 上投影点的距离小于 d 。由上面的分析可知，这种投影点最多只有6个。

- 因此，若将 $P1$ 和 $P2$ 中所有 S 中点按其 y 坐标排好序，则对 $P1$ 中所有点，对排好序的点列作一次扫描，就可以找出所有最接近点对的候选者。对 $P1$ 中每一点最多只要检查 $P2$ 中排好序的相继6个点。



复杂度分析:

- $$T(n) = \begin{cases} O(1) & n < 4 \\ 2T(n/2) + O(n) & n \geq 4 \end{cases}$$
- $T(n) = O(n \log n)$



14.4 复杂性的上限、下限

- 复杂性的上限
- 当且仅当某个问题至少有一个复杂性为 $O(f(n))$ 的求解算法时，存在一个复杂性的上限(upper bound) $f(n)$
- 证明一个问题复杂性上限为 $f(n)$ 的一种方法是设计一个复杂性为 $O(f(n))$ 的算法
- 例，在发现Strassen矩阵乘法之前，矩阵乘法的复杂性上限为 n^3 ，Strassen算法的发现使复杂性上限将为 $n^{2.81}$



复杂性的下限

- 当且仅当求解一个问题所有算法的复杂性均为 $\Omega(f(n))$ 时，则称 $f(n)$ 为该问题的复杂性的下限（lower bound）
- 为了确定一个问题的复杂性下限 $g(n)$ ，必须证明该问题的每一个求解算法的复杂性均为 $\Omega(g(n))$
- 要得到这样一个结论相当困难，因为不可能考察所有的求解算法
- 对于大多数问题，可以建立一个基于输入和/或输出数目的简单下限。
- 例，对 n 个元素进行排序的算法的复杂性为 $\Omega(n)$ ，因为所有的算法对每一个元素都必须检查至少一遍，否则未检查的元素可能会排列在错误的位置上
- 每一个计算两个 $n \times n$ 矩阵乘法的算法都有复杂性 $\Omega(n^2)$ 。因为结果矩阵中有 n^2 个元素并且产生每个元素所需要的时间为 $\Omega(1)$ 。



比较算法

- 所谓比较算法是指算法的操作主要限于元素比较和元素移动。
- 本节将建立本章所介绍的两个分治法求解的问题的确切下限——寻找 n 个元素中的最小最大值问题以及排序问题。
对于这两个问题，仅考察基于关键字比较的算法。



14.4.1 最小最大问题的下限

- 程序14-1为按分治法设计的在 n 个元素中找最大值与最小值的算法，该算法需做 $\lceil 3n/2 \rceil - 2$ 次元素比较。
- 以下证明：解决该问题的任何一个基于比较的算法，至少需要做 $\lceil 3n/2 \rceil - 2$ 次。
- 问题下界指解决该问题的所有算法的最坏情形复杂度的最小值(min-max): **max**对所有实例; **min**对所有算法.



状态空间方法

- 状态空间方法(**state space method**)。
- 该方法要求首先定义算法的三种状态：起始状态、中间状态和完成状态，并需描述每次比较状态如何转换，然后确定从起始状态到完成状态**worst case**至少需要的状态转换数。
- 构造一个产生最坏情形的输入,我们称之为敌手(**adversary**)
- 假设 n 个元素互不相同。



Max-min问题的状态空间

- 算法状态可用元组(a, b, c, d)来描述,
 - a 表示算法需考察的候选的最大和最小元素的个数(尚未参加过比较的那些元素)
 - b 表示不再做为最小候选但仍作为最大候选的元素个数(只胜未败)
 - c 是不再做为最大候选但仍做为最小候选的元素个数(只败未胜)
 - d 是被确定为即非最大也非最小的元素个数。
- A, B, C, D 代表上述各种元素的集合。



证明（过程）

- 算法启动时：起始状态为 $(n, 0, 0, 0)$
- 算法结束时：完成状态为 $(0, 1, 1, n-2)$
- 在比较元素的过程中算法状态发生变化
- 当 A 中的两个元素比较时，较小的元素放入 C ，较大的元素放入 B (根据假设所有元素都不相同，因此不会出现相等的情形),引起下面状态转换：

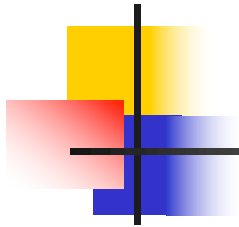
$$(a, b, c, d) \rightarrow (a-2, b+1, c+1, d)$$



证明（过程续1）

其他可能的状态转换如下：

- B 中元素比较之后，状态转换为：
 $(a, b, c, d) \rightarrow (a, b-1, c, d+1)$
- C 中元素比较之后，状态转换为：
 $(a, b, c, d) \rightarrow (a, b, c-1, d+1)$
- B 与 C 中元素比较：
 $(a, b, c, d) \rightarrow (a, b-1, c-1, d+2)$
 $*(a, b, c, d) \rightarrow (a, b, c, d)$
- *代表最坏情形状态变换



- A 中元素与 B 中元素进行比较, 状态转换为:
 - $(a, b, c, d) \rightarrow (a-1, b, c, d+1)$ (A 中元素 $>$ B 中元素)
 - * $(a, b, c, d) \rightarrow (a-1, b, c+1, d)$ (A 中元素 $<$ B 中元素)
- A 中元素与 C 中元素进行比较, 状态转换为:
 - $(a, b, c, d) \rightarrow (a-1, b, c, d+1)$ (A 中元素 $<$ C 中元素)
 - * $(a, b, c, d) \rightarrow (a-1, b+1, c, d)$ (A 中元素 $>$ C 中元素)

证明（过程续2）

A与D中元素比较状态转换：

$(a,b,c,d) \rightarrow (a-1,b+1,c,d)$

$(a,b,c,d) \rightarrow (a-1,b,c+1,d)$

- 因我们分析最坏情形复杂度的下界，所以A与B或C中元素比较时应考虑最坏情形：

$*(a,b,c,d) \rightarrow (a-1,b,c+1,d)$

$*(a,b,c,d) \rightarrow (a-1,b+1,c,d)$

$*(a,b,c,d) \rightarrow (a,b,c,d)$

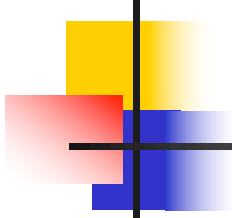
- 构造最坏情形输入使得算法在此输入下d的值增加最慢：

- A与B的元素比较令B的元素胜：

如实际上 $x > y$ ，因 y 从未输过，重新取 y 的值使 $y > x$ 。在新的输入下，以前比赛过程不会改变。

- A的元素与C的元素比较令C的元素败；

- B与C的元素比较令B的元素胜。

- 
- 其他比较无实质意义,例如**D**中元素 x 与**B**中元素 y 比较且 $x > y$,则改变 y 的值使其大于 x ,不会影响以前的比赛, d 值不变.**D**与**C**中元素比较同样处理.
 - 上面构造的输入使得从初始状态到结束状态至少要做的状态转数:
 - d 从0增至 $n-2$,至少要 $n-2$ 次比较,而且这些比较不会引起 a 减少(只能**B**中或**C**中元素自己比);
 - a 从 n 减至0,至少要做: 当 $n=2m$ (为偶数)时为 m 次; 当 $n=2m+1$ 时为 $m+1$ 次;
 - 所以至少要做 $\lceil 3n/2 \rceil - 2$ 次比较



证明（总结）

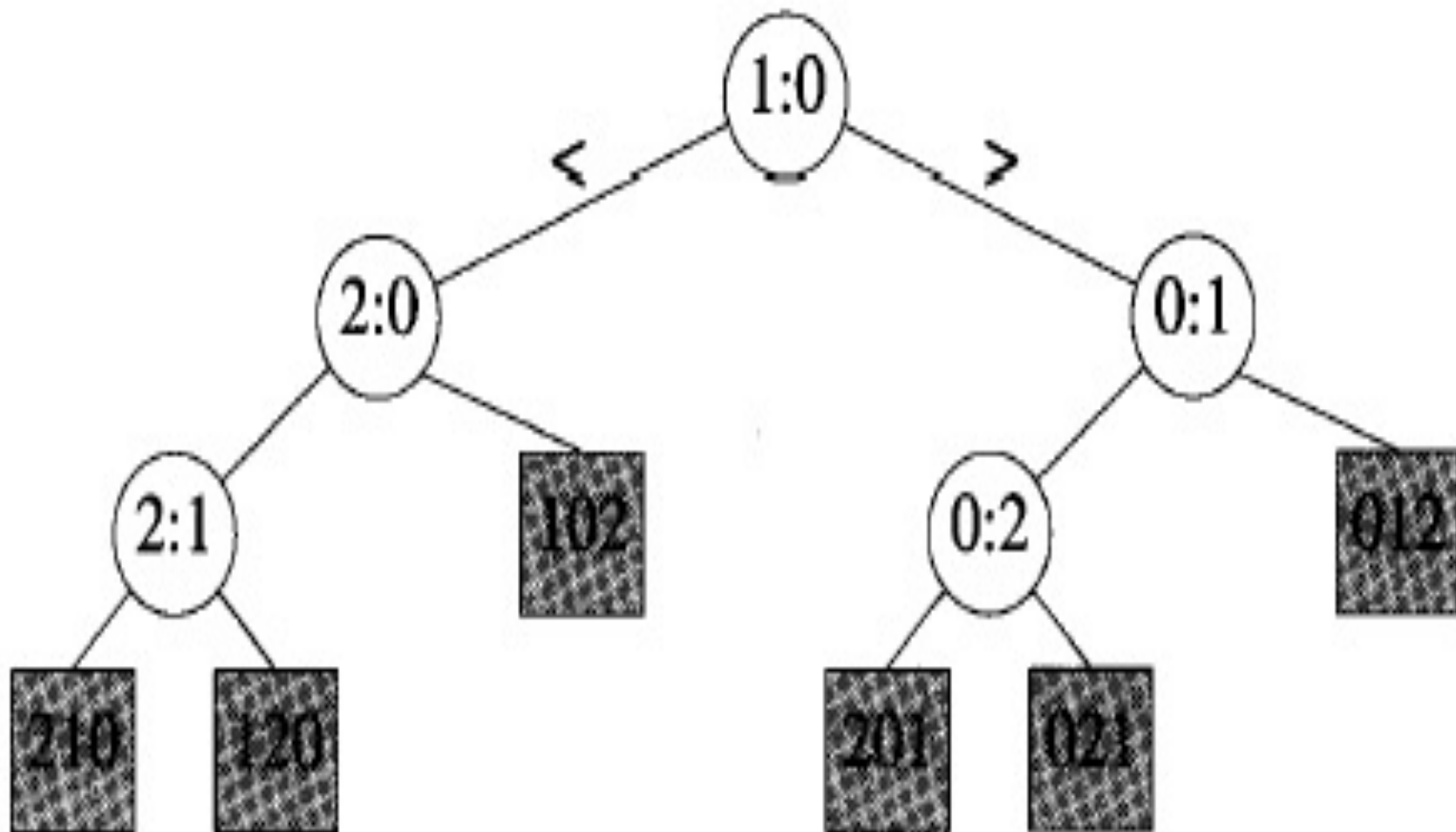
- 任何求最大最小的算法从起始状态到完成状态所用比较次数不可少于 $\lceil 3n/2 \rceil - 2$
- $\lceil 3n/2 \rceil - 2$ 是所有基于比较的求最大最小算法所需比较次数的下界。
- 程序14-1是解决最大最小问题的最优算法



14.4.2 排序算法的下限

- 可用决策树（decision-tree）来证明下限。
- 证明过程中用树来模拟算法的执行过程。
- 对于树的每个内部节点，算法执行一次比较并根据比较结果移向它的某一子节点。
- 算法在叶节点处终止。

图14-19 $n=3$ 时InsertionSort 的决策树





对图14-19证明过程 的说明

图14-19给出了对三个元素 $a[0:2]$ 使用InsertionSort（见程序2-15）排序时的决策树。每个内部节点有一个 $i:j$ 的标志，表示 $a[i]$ 与 $a[j]$ 进行比较。如果 $a[i] < a[j]$ ，算法移向左分枝；如果 $a[i] > a[j]$ ，移向右分枝。因为元素互不相同，所以 $a[i] = a[j]$ 不会发生。叶节点标出了所产生的排序。图14-19中最左路径代表： $a[1] < a[0]$ ， $a[2] < a[0]$ ， $a[2] < a[1]$ ，因此最左叶节点为 $(a[2], a[1], a[0])$ 。



决策树说明

假定输入是 $1, \dots, n$ 的一个排列,决策树中每个叶节点代表对输入排列的排序结果.例如,对输入(102)排序的结果为($a[1]a[0]a[2]$).

- 正确的排序算法对于 n 个输入排列必须产生 $n!$ 排序结果, 因此决策树有 $n!$ 个叶节点。
- 有 n 个叶节点的二叉树的深度至少为 $\log n$
- 因此决策树的高度至少为 $\log n! = \Theta(n \log n)$
- 每一个比较排序算法在最坏情况下至少要进行 $\Omega(n \log n)$ 次比较。
- 每个有 $n!$ 个叶节点的决策树的平均外路长度为 $\Omega(n \log n)$; 所以每个比较排序算法的平均复杂性也是 $\Omega(n \log n)$ 。



结论

- 由前面的证明可以看出，堆排序、归并排序在最坏情况下有较好的性能（针对渐进复杂性而言）
- 堆排序、归并排序、快速排序在平均情况下性能较优。



补充:

- 给定自然数 $1, \dots, n$ 的一个排列, 例如, $(1, 3, 4, 2, 5)$, 如果 $j > i$ 但 j 排在 i 的前面则称 (j, i) 为该排列的一个逆序。在上例中 $(3, 2)$, $(4, 2)$ 为该排列的逆序。该排列总共有2个逆序。
- 试用分治法设计一个计算给定排列的逆序总数的算法, 要求算法的时间复杂度为 $\Theta(n \log^2 n)$ 。



习题:

- 123页练习4, 使用迭代展开的方法.
- 153页练习9, 14(a).
- 证明程序14.1所用的元素比较次数为

$$t(n) = \lceil 3n/2 \rceil - 2$$

- 补充题: 给定自然数 $1, \dots, n$ 的一个排列, 例如, $(1, 3, 4, 2, 5)$, 如果 $j > i$ 但 j 排在 i 的前面则称 (j, i) 为该排列的一个逆序。在上例中 $(3, 2)$, $(4, 2)$ 为该排列的逆序。该排列总共有2个逆序。试用分治法设计一个计算给定排列的逆序总数的算法, 要求算法的时间复杂度为 $\Theta(n \log^2 n)$ 。