算法设计与分析

第2章 递归与分治策略
(2)

主要内容

- 分治法的条件
- 分治法的算法框架
- 分治法的分析
- 案例
  - 二分搜索
  - Strassan矩阵乘法
  - 合并排序

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：
  - 可解性：该问题的规模缩小到一定的程度就可以**容易地解决**
  - **递归性**：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**
  - 合并性：利用该问题分解出的子问题的解可以**合并**为该问题的解
  - 独立性：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。

递归性：应用前提，此特征反映了递归思想的应用

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：
  - 可解性：该问题的规模缩小到一定的程度就可以**容易地解决**
  - **递归性**：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**
  - **合并性**：利用该问题分解出的子问题的解可以**合并**为该问题的解
  - 独立性：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

合并性：选择分治法必要条件,有1.2缺3条件则选贪心算法或动态规划。

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：
  - 可解性：该问题的规模缩小到一定的程度就可以**容易地解决**；
  - **递归性**：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**
  - 合并性：利用该问题分解出的子问题的解可以**合并**为该问题的解；
  - **独立性：**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。

这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。

分治法的基本步骤

- 输入： 原问题P；　　输出：问题解Y；

```
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
}
```

- 平衡(balancing)子问题思想
  - 实践中发现，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。
  - 几乎总是比子问题规模不等的做法要好。

分治法的时间复杂性分析

```
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
}
```

T(n)

O(1)

D(n)

f(n)

T(n/m)

C(n)

- 分析：
  - 设分解阈值n0=1，且adhoc解规模为1的问题耗费1个单位时间。
  - 设将规模为n的问题分成k个规模为n / m的子问题。
  - 设将原问题divide为k个子问题以及用merge将k个子问题的解合并为原问题的解需用**f(n)**个单位时间。

分治法的时间复杂性分析

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

T(n)

O(1)

D(n) → f(n)

T(n/m)

C(n) → f(n)

- 用T(n)表示解规模为|P|=n的问题所需的计算时间，则有：

  - $$\mathrm{T(n)} = \begin{cases} O(1) & n = 1 \\ \mathrm{k}T\left(\frac{n}{m}\right) + f(n) & n > 1 \end{cases}$$

分治法的时间复杂性分析

- 平衡子问题思想

$$T(n)=\begin{cases} O(1) & n=1 \\ kT\left(\frac{n}{m}\right)+f(n) & n>1 \end{cases}$$

- 通过迭代法求得方程的解：

$$T(n)=n^{\log_m k}+\sum_{j=0}^{\log_m n-1} k^j f(n/m^j)$$

影响因素：n, k, m, f

- 注意：
  - 递归方程及其解只给出n等于m的方幂时T(n)的值，但是如果认为T(n)足够平滑，那么由n等于m的方幂时T(n)的值可以估计T(n)的增长速度。
  - 通常假定T(n)是单调上升的，从而当$m_i \le n \le m_i+1$时，

    $T(m_i) \le T(n) \le T(m_i+1)$

分治法的时间复杂性分析

- 平衡子问题思想：

$$T(n)=\begin{cases} O(1) & n=1 \\ kT\left(\frac{n}{m}\right)+f(n) & n>1 \end{cases}$$

- 子问题规模减 i 思想：

$$T(n)=\sum_{i=1}^{k} a_i T(n-i)+f(n)$$

- 例子
  - Hanoi塔问题： T(n)=2T(n-1) +1 T(1)=1
  - 大整数乘法问题：T(n)=3T(n/2)+n T(1)=1
  - 二分搜索问题： T(n)=T(n/2)+1 T(1)=1

分治法-小结

- 分治算法的适用条件
- 分治算法描述
- 分治算法时间复杂度分析
  - 递归方程
- 分解的方法：关键缩小规模，平衡子问题和减i的方法

二分搜索技术

- 问题：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
- 分析：
  - 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
  - 分解出的子问题的解可以合并为原问题的解
  - 分解出的各个子问题是相互独立的

分析：如果n=1即只有一个元素，则只要比较这个元素和x就可以确定x是否在表中。因此这个问题满足分治法的第一个适用条件

二分搜索技术

- 问题：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
- 分析：
  - 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
  - 分解出的子问题的解可以合并为原问题的解
  - 分解出的各个子问题是相互独立的

- 分析：比较x和a的中间元素a[mid]

  若x=a[mid]，则x在 L中的位置就是mid；

  若x<a[mid]，由于a是递增排序的，只要在a[mid]的前面查找x；

  若x>a[mid]，同理只要在a[mid]的后面查找x即可。
- 无论是在前面还是后面查找x，其方法都和在a中查找x一样，只不过是查找的规模缩小了。**这说明此问题满足分治法的第二个和第三个适用条件**

二分搜索技术

- 问题：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
- 分析：
  - 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
  - 分解出的子问题的解可以合并为原问题的解
  - 分解出的各个子问题是相互独立的

- 分析：

　很显然此问题分解出的子问题相互独立，即在a[i]的前面或后面查找x是独立的子问题，因此满足分治法的第四个适用条件

二分搜索技术

- 问题：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
- 基本思想:
  - 将n个元素分成个数大致相同的两半
  - 取a[n/2]与x进行比较：
    - 如果x=a[n/2]，
      - 则找到,算法终止;
    - 如果x < a[n/2]
      - 则只要在数组a的左半部搜索x;
    - 如果x>a[n/2]
      - 则只要在数组a的右半部继续搜索x.

二分搜索技术

- 问题：给定已按升序排好序的n个元素a[1:n]，现要在这n个元素中找出一特定元素x。
- 设计：
  - 分：找中间位置，规模缩小一半
  - a[1,n/2-1], a[n/2+1, n]
  - 解：递归求解1个子问题a[1,n/2-1]或a[n/2+1, n]
  - 当a[n/2]与x相等，则找到,算法终止;
  - 如果x < a[n/2] 在数组a的左半部搜索x;
  - 如果x>a[n/2] 在在数组a的右半部继续搜索x.
  - 合：无需合

a[1,n] → a[1,n/2-1], a[n/2+1],n

何时终止

二分搜索技术

- 问题：给定已按升序排好序的n个元素a[1:n]，现要在这n个元素中找出一特定元素x。
- 分析：
  - 分：找中间位置，规模缩小一半
  - a[1,n/2-1], a[n/2+1, n]
  - 解：递归求解1个子问题a[1,n/2-1]或a[n/2+1, n]
  - 当a[n/2]与x相等，则找到,算法终止;
  - 如果x < a[n/2] 在数组a的左半部搜索x;
  - 如果x>a[n/2] 在在数组a的右半部继续搜索x.
  - 合：无需合

O(1)，
$T(n) \rightarrow T(n/2)$

1* T(n/2)

$T(n)= T(n/2)+1$

二分搜索技术

- 输入：非降序排列的n个元素数组a[0..n-1]和元素x;
- 输出： m, if x=a[m];

  -1, if x不存在;

```
Binarysearch(a, x, l, r){    //l,r分别表示搜索开始和终止位置
    if l>r then return -1;
    else
      m = (l+r)/2;
      if (x = a[m])     return m;
      if (x < a[m])     return Binarysearch(a, x, l, m-1);
      else        return Binarysearch(a, x, m+1, r);
}
```

二分搜索技术

- 输入：非降序排列的n个元素数组a[0..n-1]和元素x;
- 输出： m, if x=a[m];

  -1, if x不存在;

```
Binarysearch(a, x, l, r){    //l,r分别表示搜索开始和终止位置        T(n)
    if l>r then return -1;
    else
      m = (l+r)/2;                                                   O(1)
      if (x = a[m])    return m;                                     O(1)
      if (x < a[m])    return Binarysearch(a, x, l, m-1);            T(n/2)
      else       return Binarysearch(a, x, m+1, r);                  T(n/2)
}
```

算法复杂度分析：

$T(n)=T(n/2)+1$

二分搜索技术

- 算法的分析

$$T(n)=\begin{cases} 1 & n=1 \\ T\left(\frac{n}{2}\right)+1 & n>1 \end{cases}$$

解：$T(n)=T(n/2)+1$

$=T(n/4)+2$

$=T(n/8)+3$

$=\ldots$

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

当$n/2^i=1$ 即$i=\log_2 n$时终止分解

所以：$T(n)=T(1)+\log_2 n=\log_2 n$

二分搜索技术

分治法算法框架

```
divide-and-conquer(P){
   if ( |P| <= n0) adhoc(P);
   divide P into 小的子问题P1,P2,...,Pk ;

   for (i=1,i<=k,i++)
        yi=divide-and-conquer(Pi);
   return merge(y1,...,yk);
 }
```

二分法

```
Binarysearch(a,x,l, r){
  if l>r then return -1;
   else{
      m = (l+r)/2;
      if (x = a[m])  return m;
      if (x<a[m])
          return Binarysearch(a,x,l,m-1);
      else
           return Binarysearch(a,x,m+1, r);
   }
}
```

二分搜索技术

- 输入：n个元素的非降序数组a[1...n]和元素x.
- 输出：如果x=a[m]，1<= m <= n，则输出m，否则输出-1。

非递归算法：BINARYSEARCH (a[1..n], x, l, r)

1、 **while** (r >= l){
2、 int m = $\lfloor (l + r)/2 \rfloor$
3、 if (x == a[m]) return m;
4、 if (x < a[m]) r = m-1;
5、 else l = m+1;
6、 }
7、 return -1;

通过修改数组的边界范围，达到缩小问题规模，实现分治的目的

二分搜索技术

- 输入：n个元素的非降序数组a[1...n]和元素x.
- 输出：如果x=a[m]，1<= m <= n，则输出m，否则输出-1。

非递归算法：BINARYSEARCH (a[1..n], x, l, r)

1、 **while (r >= l){**
2、 **int m = $\lfloor (l + r)/2 \rfloor$**
3、 **if (x == a[m]) return m;**
4、 **if (x < a[m]) r = m-1;**
5、 **else l = m+1;**
6、 **}**
7、 **return -1;**

算法复杂度分析：

- 每执行一次算法的while循环， 待搜索数组的大小减少一半。
- 因此，在最坏情况下，while循环被执行了O(logn) 次。
- 循环体内运算需要O(1) 时间
- 因此整个算法在最坏情况下的计算时间复杂性为O(logn) 。

二分搜索技术

- O(logn) ???
- 计算第二次迭代时A[1...n]中剩余元素的数目，要根据n是奇数还是偶数分两种情况考虑：
  - 偶数：A[mid+1...n]中的数为n/2;
  - 奇数: (n-1)/2.
- 两种情况下都有：A[mid+1...n]中的元素个数为 $\lfloor \boldsymbol{n}/\mathbf{2} \rfloor$
- 类似的，第三次迭代时，要搜索的剩余元素个数是

$$\lfloor \lfloor n/2 \rfloor /2 \rfloor = \lfloor n/4 \rfloor$$

二分搜索技术

- 在while循环中第j次循环时剩余元素个数是 $\lfloor n/2^{j-1} \rfloor$
- 循环停止的条件
  - 或者找到x，
  - 或者要搜索的子序列长度为1　$\lfloor n/2^{j-1} \rfloor=1$
- 因此，搜索x的最大循环次数就是满足条件 $1 \le n/2^{j-1} < 2$时的j值
- 根据底函数的定义，这种情况发生在当

$2^{j-1} \le n < 2^j$ 或 $j-1 \le \log n < j$

时。因为j是整数，可以得出结论

$$j = \lfloor \log n \rfloor + 1$$

Strassen矩阵乘法

- 问题：求矩阵A和矩阵B的乘积。
  - 输入：矩阵A，B
  - 输出：矩阵A与B的乘积
- 分析：A和B的乘积矩阵C中的元素C[i,j]定义为:

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

$$\begin{pmatrix} \cdots\cdots\ .\ .. \\ \cdots\cdots c_{ij}\ .. \\ \cdots\cdots\ .\ .. \\ \cdots\cdots\ .\ .. \end{pmatrix} = \begin{pmatrix} \cdots\cdots \\ ***** \\ \cdots\cdots \\ \cdots\cdots \end{pmatrix} \begin{pmatrix} \cdots * .. \\ \cdots * .. \\ \cdots * .. \\ \cdots * .. \end{pmatrix}$$

◆传统方法：O(n^3)

若依此定义来计算A和B的乘积矩阵C，则每计算C的一个元素C[i][j]，需要做**n**次乘法和**n-1**次加法。因此，算出矩阵C的 n^2个元素所需的计算时间为**O(n^3)**

Strassen矩阵乘法

- 传统方法
  - 两个 n×n 矩阵 A and B, 复杂度(C=A×B) = ?

$$C_{ij} = \sum_{k=1}^{n} A_{ik}B_{kj}$$

$$\begin{pmatrix} \cdots\cdots\ .\ .. \\ \cdots\cdots c_{ij}\,.. \\ \cdots\cdots\ .\ .. \\ \cdots\cdots\ .\ .. \end{pmatrix} = \begin{pmatrix} \cdots\cdots\cdots \\ *\,*\,*\,*\,* \\ \cdots\cdots\cdots \\ \cdots\cdots\cdots \end{pmatrix} \begin{pmatrix} \cdots\cdots *\,.. \\ \cdots\cdots *\,.. \\ \cdots\cdots *\,.. \\ \cdots\cdots *\,.. \end{pmatrix}$$

MATRIX-MULTIPLY(A, B)
for $i \leftarrow 1$ **to** n
    for $j \leftarrow 1$ **to** n
        $C[i, j] \leftarrow 0$
        for $k \leftarrow 1$ **to** n
            $C[i, j] \leftarrow C[i, j] + A[i, k] \cdot B[k, j]$
return C

复杂度分析y:
$O(n^3)$ 乘法和加法.
$T(n) = O(n^3)$.

Strassen矩阵乘法

- 传统方法：$O(n^3)$
- 分治法:
  - 一个 n×n 矩阵分解为4个n/2×n/2 的大小相等的子矩阵
    - $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$, $B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$, $C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$
  - 由此可将方程C=AB重写为：

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

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

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

复杂度分析:

8 次乘法 (子问题), and 4 次加法 ($n/2 \times n/2 \times 4$).

$T(2)=1$, $T(n) = 8T(\lceil n/2 \rceil)+n^2$.

应用主定理法，可得：**$T(n)= O(n^3)$.**

两个$n/2\times n/2$矩阵之间的加法

Strassen矩阵乘法

- 传统方法：$O(n^3)$
- 分治法:
  - 分：n x n 矩阵分为4个 (n/2)x (n/2) 子矩阵
  - 解：进行8个子矩阵之间的乘法
  - n=2, 2阶方阵时直接计算
  - 合： 8 次乘法 (子问题), and 4个 (n/2)x (n/2) 子矩阵的加法

复杂度

$$T(n)=\begin{cases} O(1) & n=2 \\ 8T(n/2)+O(n^2) & n>2 \end{cases}$$

$T(n)=O(n^3)$

???

Strassen矩阵乘法

- 传统方法：$O(n^3)$
- 分治法:
  - 为了降低时间复杂度，必须减少乘法的次数

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

14个子矩阵

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

$$\begin{aligned}
C_{11} &= M_5 + M_4 - M_2 + M_6 \\
C_{12} &= M_1 + M_2 \\
C_{21} &= M_3 + M_4 \\
C_{22} &= M_5 + M_1 - M_3 - M_7
\end{aligned}$$

Strassen矩阵乘法

- Strassen方法四个步骤
  - 分：(1)把输入矩阵A和B划分为$n/2 \times n/2$的子矩阵
  - 解：(2)运用$O(n^2)$次标量加法和减法运算,计算出14个$n/2 \times n/2$的矩阵$A_1,B_1,A_2,B_2,A_3,B_3,A_4,B_4,A_5,B_5,A_6,B_6,A_7,B_7$
  - (3)递归计算出7个矩阵乘积 $M_i=A_iB_i$
  - 合：(4)仅使用$O(n^2)$次标量加法与减法运算,对M_i矩阵的各种组合进行求和或求差运算,获得结果矩阵的四个子矩阵

Strassen矩阵乘法✦

- 传统方法：$O(n^3)$
- 分治法:
  - 为了降低时间复杂度，必须减少乘法的次数
  - Strassen方法

复杂度分析

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

$T(n)=O(n^{\log 7})=O(n^{2.81})$ ✓ **较大的改进**

Strassen矩阵乘法

- 分治法:
  - Strassen方法
    - 输入：A, B矩阵
    - 输出：C=A*B

```
StrassenMultiply(A, B){
    subA← divide(A)    // A11,A12,A21,A22)
    subB← divide(B)    // (B11,B12,B21,B22)
    P7Matrix=calculate7Matr(subA, subB)
    subC=calculateP7(P7Matrix)
    return  compose(subC)
}
```

复杂度分析:
总共7 次乘法和 18 次加法.

$T(1)=1$,

$T(n) = 7T(\lceil n/2 \rceil)+ cn^2$.

应用主定理得：

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

Define

$P_1 = (A_{11}+A_{22})(B_{11}+B_{22})$

$P_2 = (A_{11}+A_{22})B_{11}$

$P_3 = A_{11} (B_{11}-B_{22})$

$P_4 = A_{22} (-B_{11}+B_{22})$

$P_5 = (A_{11}+A_{12})B_{22}$

$P_6 = (-A_{11}+A_{21})(B_{11}+B_{12})$

$P_7 = (A_{12}- A_{22})(B_{21}+B_{22})$

Then

$C_{11}=P_1+P_4-P_5+P_7$, $C_{12}=P_3+P_5$

$C_{21}=P_2+P_4$, $C_{22}=P_1+P_3-P_2+P_6$

```
calculate7Matr(subA, subB){
    StrassenMultiply((subA.A11+ subA. A22 ),(subB.B11+ subB. B22))
    StrassenMultiply((subA. A11+ subA. A22 , subB. B11)
    StrassenMultiply(subA. A11 ,(subB. B11–subB. B22))
    StrassenMultiply(subA. subA. A22 ,(–subB. B11+ subB. B22))
    StrassenMultiply((subA. A11+ subA. A12), subB. B22)
    StrassenMultiply(((– subA. A11+ subA. A21),(subB. B11+ subB. B12))
    StrassenMultiply((subA. A12– subA. A22),(subB. B21+ subB.B22))
}
```

Strassen矩阵乘法

- 传统方法：$O(n^3)$
- 分治法: $O(n^{2.81})$
- 更快的方法??

➢Hopcroft和Kerr已经证明，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。目前最好的计算时间上界是 **$O(n^{2.376})$**

➢是否能找到$O(n^2)$的算法？

Strassen矩阵乘法

- 最新相关研究
  - 赵玉文等, 大整数乘法Sch(o)nhage-Strassen算法的多核并行化研究（Research on Large Integer Multiplication Sch(o)nhage-Strassen Algorithm ’s Multi-Core Parallelization），软件学报，2018,29(12):3604-3013.

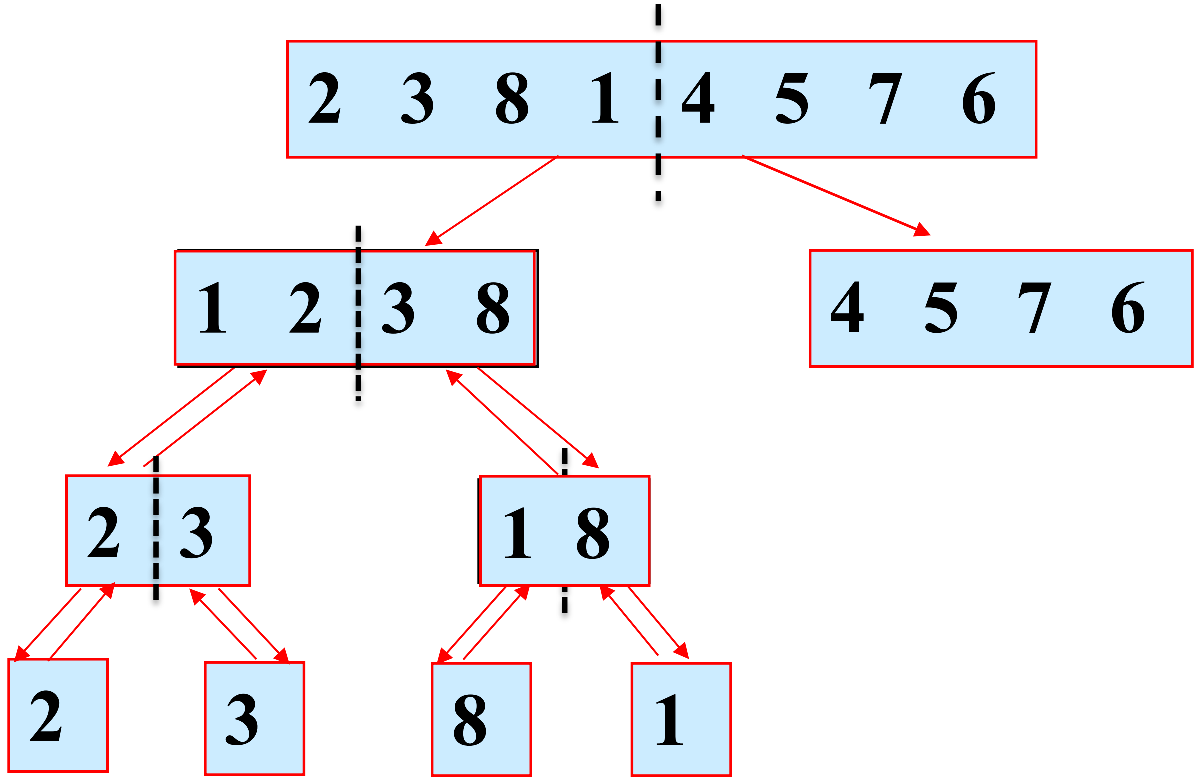
合并排序

- 问题：对n个元素进行排序
  - 输入：待排序元素数组
  - 输出：排好序元素数组

- 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
    - 将子集合继续划分为规模为n/4的4个子问题，继续划分…
    - 当子集合元素个数为1时终止划分
  - 从规模为1到n/2， 陆续合并已排好序的子集合，每合并一次，数组规模扩大一倍，直到得到所要求的的排好序的集合。

合并排序

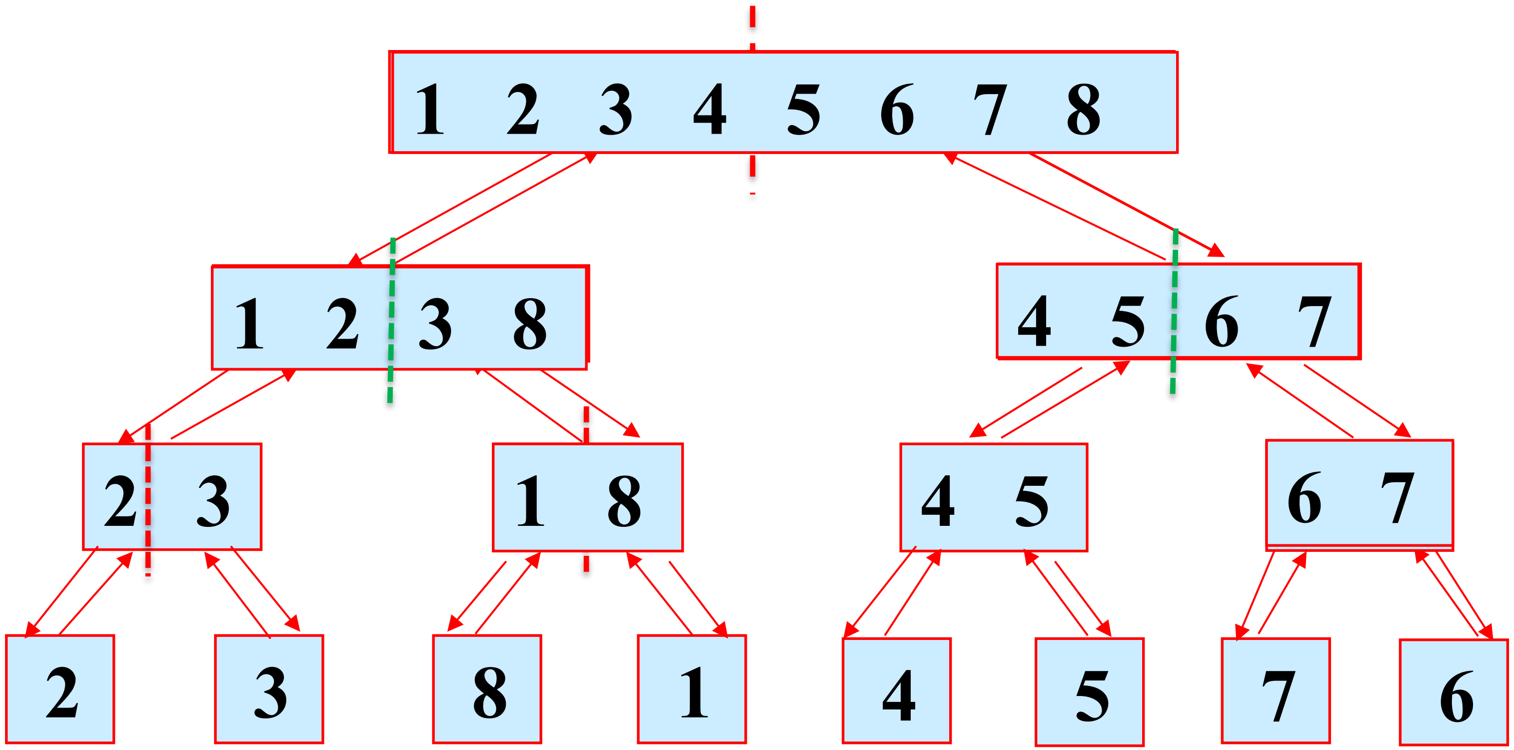
- 例子



分解：将n个元素序列分成各含n/2个元素的子序列；
解决：用合并排序法对两个子序列递归的排序；
只含1个元素的序列即表示已排好序；
合并：合并两个已排序的子序列以得到排序结果

合并排序

- 例子



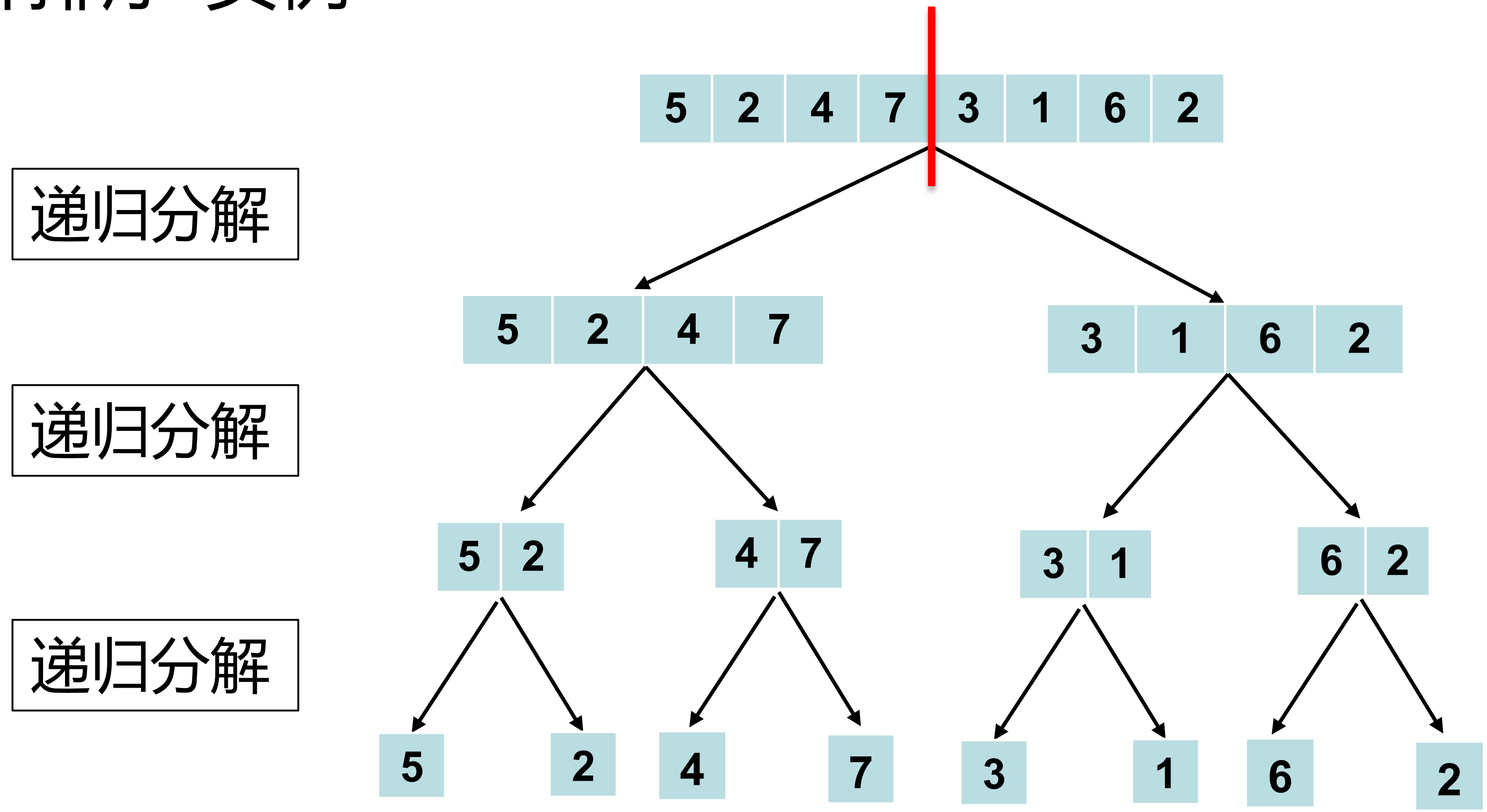
分解：将n个元素序列分成各含n/2个元素的子序列；
解决：用合并排序法对两个子序列递归的排序；
只含1个元素的序列即表示已排好序；
合并：合并两个已排序的子序列以得到排序结果

合并排序

- 问题：将包含n个元素的序列通过合并排序算法进行排序
  - 输入：待排序元素数组
  - 输出：排好序元素数组
- 基本思想：
  - 分解：将n个元素序列分成各含n/2个元素的子序列； a[l,r]→a[l,m], a[m+1,r]
  - 解决：用合并排序法对两个子序列递归的排序；
    只含1个元素的序列即表示已排好序；
  - 合并：合并两个已排序的子序列以得到排序结果

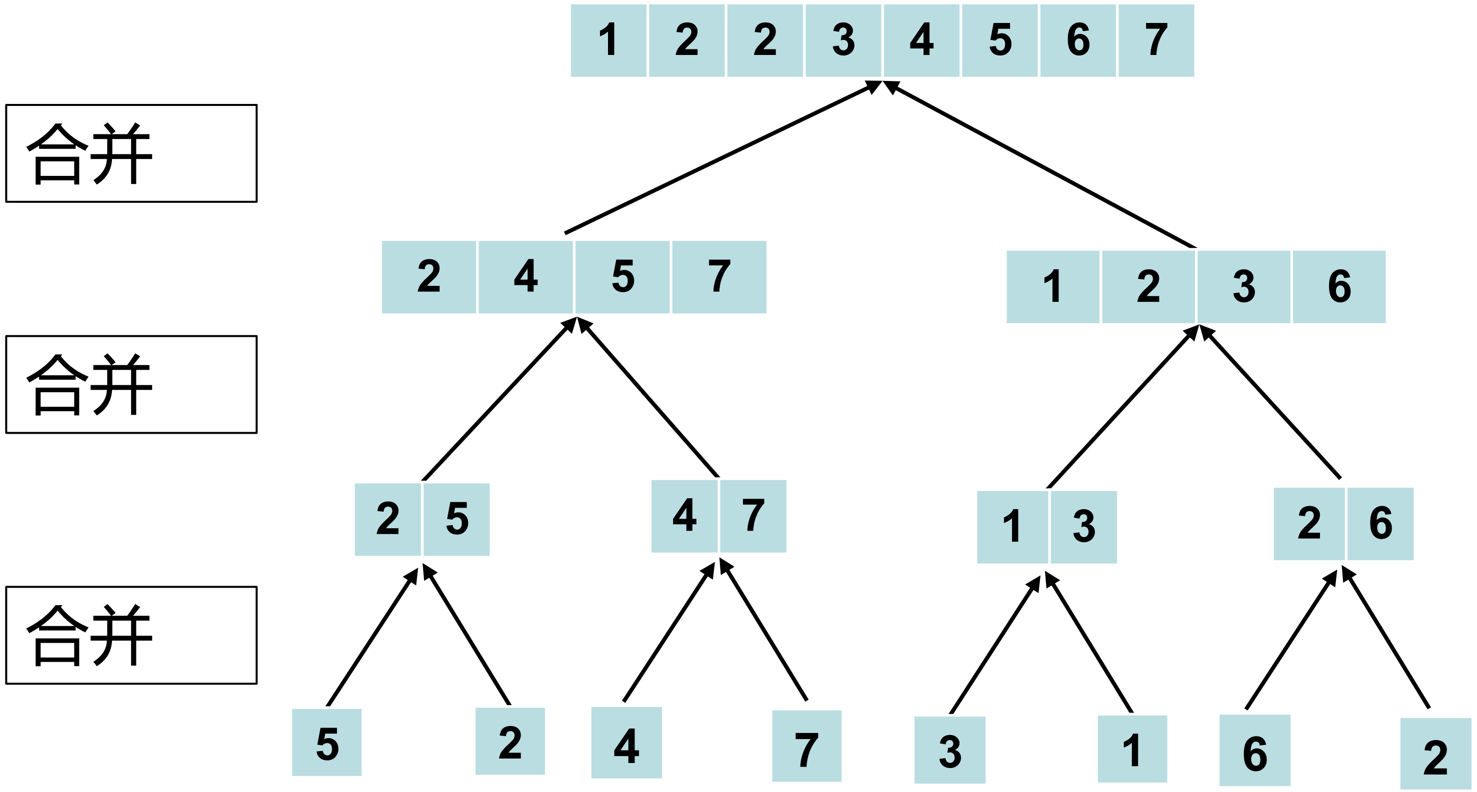
合并排序-实例

- 合并排序-实例



合并排序-实例

- 合并排序-实例



合并排序

- 问题：对n个元素排成非递减顺序。
  - 输入：待排序元素数组a[1..n]；
  - 输出：排好序元素数组a[1..n]；

- 设计
  - 分: 取中心m，a[1..n]分为左右两部分a[1,m], a[m+1,n]
  - 解:
    - 递归解n/2规模的子问题，即分别对两个子集合进行排序
    - 终止条件：长度为1的序列无需排序
  - 合：将左右已排好序的两部分合并

合并排序----设计

- 算法MergeSort()
  - 输入: n个元素的数组A[1..n], 左边界left, 右边界right;
  - 输出: 排好序的A;

```
MergeSort(Type a[], int left, int right) {
    if (left<right) {//   /至少有2个元素
        int q=(left+right)/2;  //取中点
        mergeSort(a, left, q);
        mergeSort(a, q+1, right);
        merge(a, left, q, right);  //合并到数组a
    }
}
```

问题足够小条件

分

解

合

合并排序----分析

- 算法MergeSort()
  - 输入: n个元素的数组A[1..n], 左边界left, 右边界right;
  - 输出: 排好序的A;

```
MergeSort(Type a[], int left, int right) {
    if (left<right) {//   /至少有2个元素
        int q=(left+right)/2;  //取中点
        mergeSort(a, left, q);
        mergeSort(a, q+1, right);
        merge(a, left, q, right);  //合并到数组a
    }
}
```

MergeSort(Type a[], int left, int right)	T(n)
int q=(left+right)/2;	O(1)
mergeSort(a, left, q); mergeSort(a, q+1, right);	2T(n/2)
merge(a, left, q, right);	O(n)

分解: 计算出子数组的中间位置,需要常量时间, 为O(1)

解决: 递归地解两个规模为n/2的子问题,时间为2T(n/2)

合并:在一个含有n个元素的子数组上，merge过程运行时间为O(n)

合并排序----分析

```
MergeSort(Type a[], int left, int right) {
    if (left<right) {//   /至少有2个元素
        int q=(left+right)/2;  //取中点
        mergeSort(a, left, q);
        mergeSort(a, q+1, right);
        merge(a, left, q, right);  //合并到数组a
    }
}
```

T(n)

O(1)

2T(n/2)

O(n)

复杂度分析

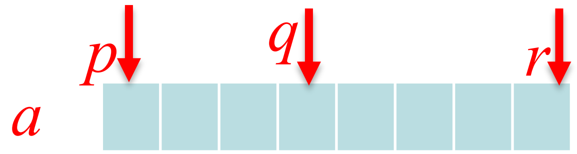
$$T(n)=\begin{cases} O(1) & n\le 1 \\ 2T(n/2)+O(n) & n>1 \end{cases}$$

T(n)=O(nlgn) 渐进意义下的最优算法

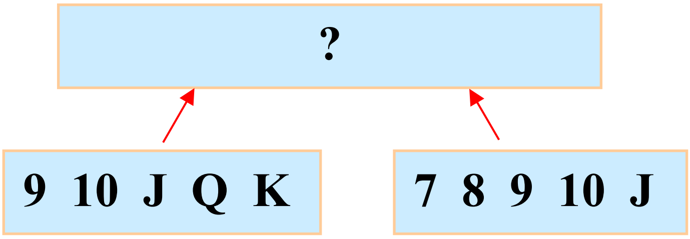
合并排序

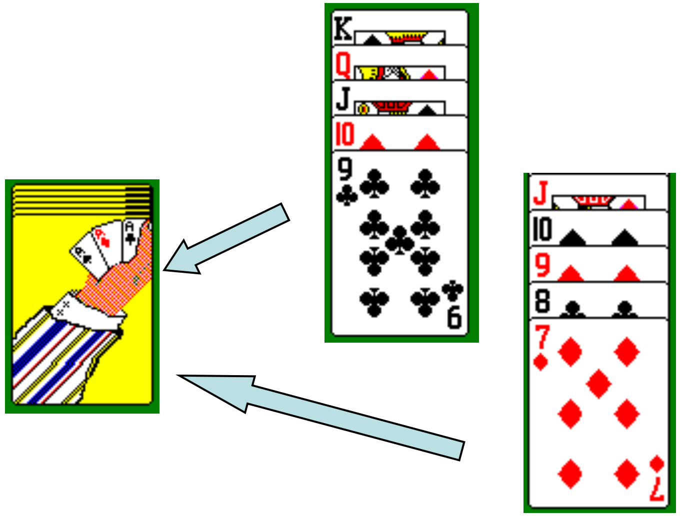
重点在合

- Merge算法
  - merge(A,p,q,r): 合并排序算法中的“合并”关键步骤



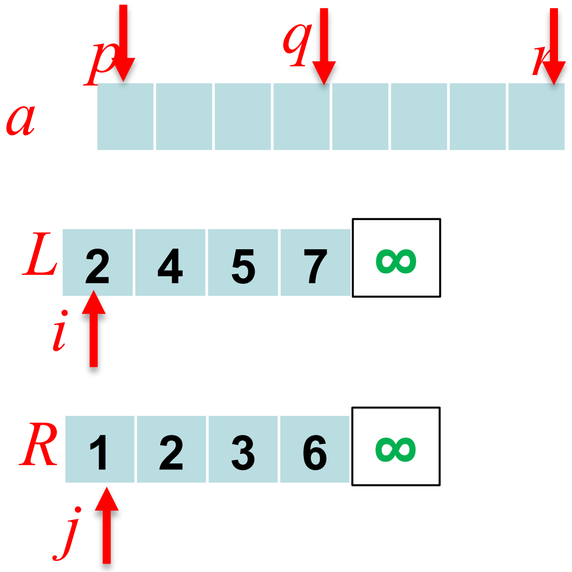
  - 输入：A, p,q,r
    - p, q, r 是数组的索引位置，表示处理的开始位置、中间位置和终止位置， p≤q<r
    - 子数组 A[p .. q] and A[q+1 .. r] 已经排好序.
  - 输出：A[p .. r]
    - 合并子数组A[p .. q] 和A[q+1 .. r]得到单一的已排序数组，替换原来的 A[p .. r].
  - 步骤：





合并排序

- Merge算法




```
MERGE(a[], p  q, r)
1  n₁ ← q-p+1
2  n₂ ← r-q
3  create arrays L[1 .. n₁+1] and R[1 .. n₂+1]
4  for i←1 to n₁
5          do L[i]←a[l+i-1]
6  for j←1 to n₂
7          do R[j]←A[q+j]
8  L[n₁+1]←∞
9  R[n₂+1]←∞
10 i←1
11 j←1
12 for k←p to r
13      do if L[i]≤R[j]
14            then a[k]←L[i]
15                 i←i+1
16            else a[k]←R[j]
17                 j←j+1
```

1-2 计算左右子数组长度

3 创建L和R数组

4-7 将两个子数组分别**复制到L和R中**

8-9 设置哨兵牌，避免每一步去检查是否为空

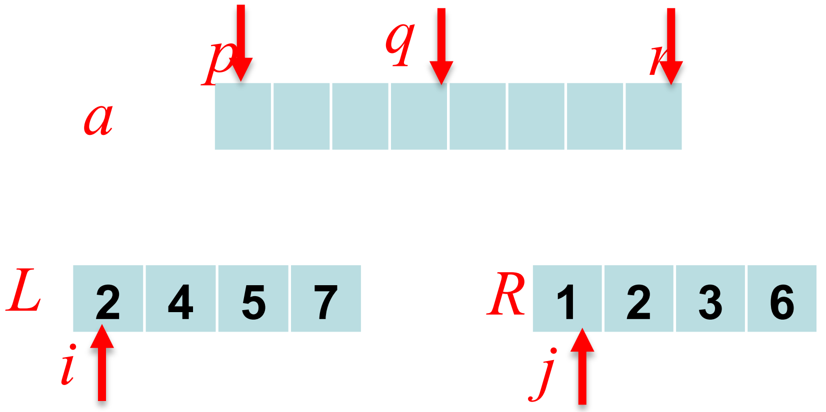
10-11 初始化两个指针

12-17 比较L[i]和R[j],每次选取更小的存入a数组

合并排序

- 实例　　a[] = {2,4,5,7,1,2,3,6}

  Merge(*a, p, q, r*)过程演示



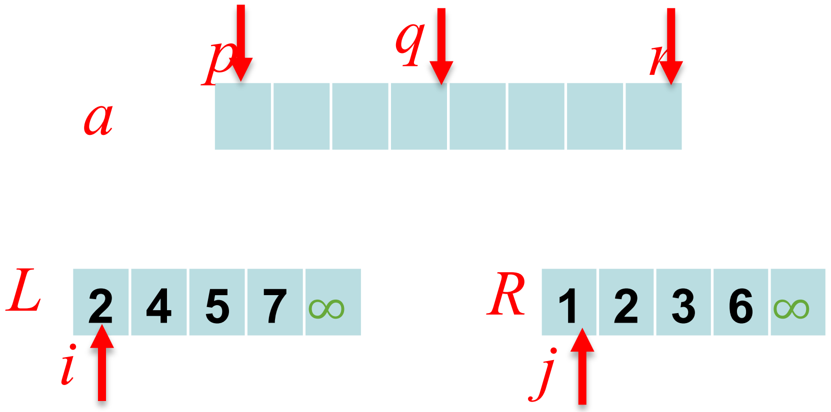
Step 3, 4-5, 6-7

```
MERGE(A, p, q, r)
1    n1 ← q-p+1;
2    n2 ← r-q;
3    create arrays L[1 .. n1+1] and R[1 .. n2+1];
4    for i←1 to n1
5            do L[i]←A[p+i-1];
6    for j←1 to n2
7            do R[j]←A[q+j];
8    L[n1+1]←∞;
9    R[n2+1]←∞;
10   i←1;
11   j←1;
12   for k←p to r
13       do if L[i]≤R[j]
14              then A[k]←L[i];
15                   i←i+1;
16              else A[k]←R[j];
17                   j←j+1;
```

合并排序

- 实例 a[] = {2,4,5,7,1,2,3,6}

Merge(*a, p, q, r*)过程演示



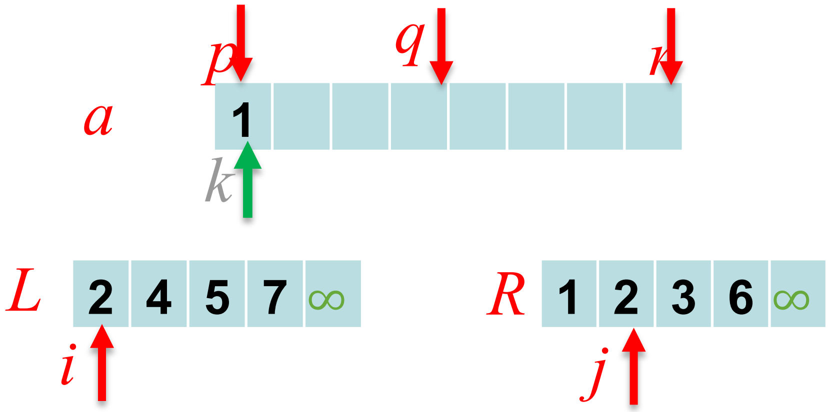
Step 8, 9,10,11

```
MERGE(A, p, q, r)
1     n1 ← q-p+1;
2     n2 ← r-q;
3     create arrays L[1 .. n1+1] and R[1 .. n2+1];
4     for i←1 to n1
5             do L[i]←A[p+i-1];
6     for j←1 to n2
7             do R[j]←A[q+j];
8     L[n1+1]←∞;
9     R[n2+1]←∞;
10    i←1;
11    j←1;
12    for k←p to r
13        do if L[i]≤R[j]
14               then A[k]←L[i];
15                    i←i+1;
16               else A[k]←R[j];
17                    j←j+1;
```

合并排序

- 实例　　a[] = {2,4,5,7,1,2,3,6}

  Merge(*a, p, q, r*)过程演示



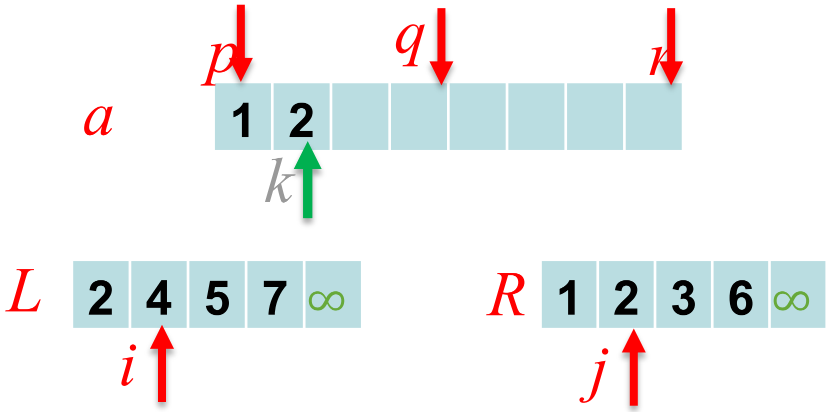
Step 12-17: 1<2

MERGE(A, p, q, r)
1　$n_1 \leftarrow q\text{-}p+1$;
2　$n_2 \leftarrow r\text{-}q$;
3　create arrays $L[1 .. n_1+1]$ and $R[1 .. n_2+1]$;
4　**for** $i \leftarrow 1$ **to** n_1
5　　do $L[i] \leftarrow A[p+i-1]$;
6　**for** $j \leftarrow 1$ **to** n_2
7　　**do** $R[j] \leftarrow A[q+j]$;
8　$L[n_1+1] \leftarrow \infty$;
9　$R[n_2+1] \leftarrow \infty$;
10　$i \leftarrow 1$;
11　$j \leftarrow 1$;
12　**for** $k \leftarrow p$ **to** r
13　　**do if** $L[i] \leq R[j]$
14　　　**then** $A[k] \leftarrow L[i]$;
15　　　　$i \leftarrow i+1$;
16　　　**else** $A[k] \leftarrow R[j]$;
17　　　　$j \leftarrow j+1$;

合并排序

- 实例 a[] = {2,4,5,7,1,2,3,6}

Merge(*a, p, q, r*)过程演示



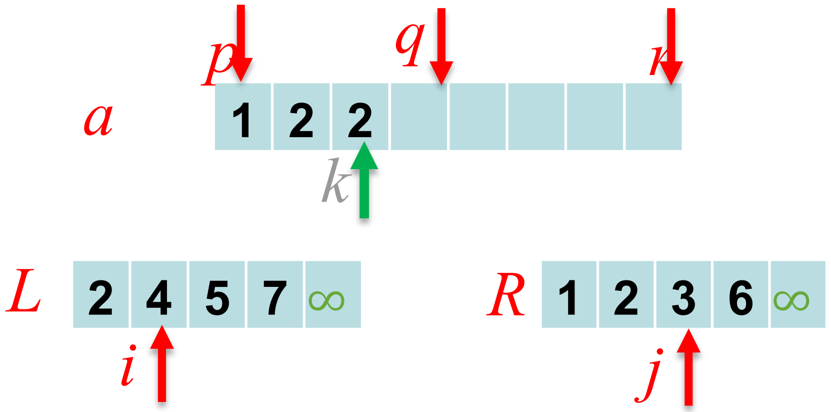
Step 12-17:

```
MERGE(A, p, q, r)
1     n1 ← q-p+1;
2     n2 ← r-q;
3     create arrays L[1 .. n1+1] and R[1 .. n2+1];
4     for i←1 to n1
5             do L[i]←A[p+i-1];
6     for j←1 to n2
7             do R[j]←A[q+j];
8     L[n1+1]←∞;
9     R[n2+1]←∞;
10    i←1;
11    j←1;
12    for k←p to r
13        do if L[i]≤R[j]
14               then A[k]←L[i];
15                    i←i+1;
16               else A[k]←R[j];
17                    j←j+1;
```

合并排序

- 实例　　a[] = {2,4,5,7,1,2,3,6}

Merge(*a, p, q, r*)过程演示



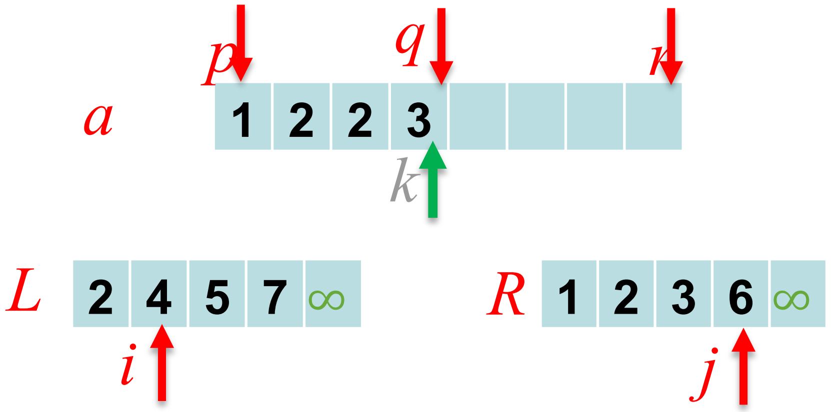
Step 12-17:

```
MERGE(A, p, q, r)
1    n1 ← q-p+1;
2    n2 ← r-q;
3    create arrays L[1 .. n1+1] and R[1 .. n2+1];
4    for i←1 to n1
5            do L[i]←A[p+i-1];
6    for j←1 to n2
7            do R[j]←A[q+j];
8    L[n1+1]←∞;
9    R[n2+1]←∞;
10   i←1;
11   j←1;
12   for k←p to r
13       do if L[i]≤R[j]
14               then A[k]←L[i];
15                    i←i+1;
16               else A[k]←R[j];
17                    j←j+1;
```

合并排序

- 实例　　a[] = {2,4,5,7,1,2,3,6}

Merge(*a, p, q, r*)过程演示



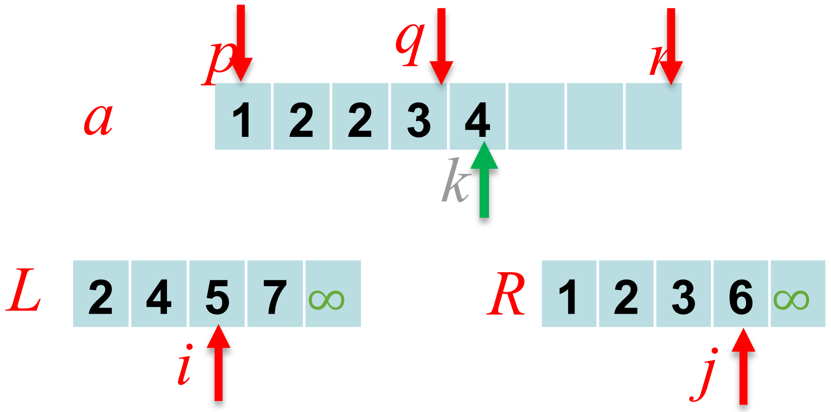
Step 12-17:

```
MERGE(A, p, q, r)
1    n1 ← q-p+1;
2    n2 ← r-q;
3    create arrays L[1 .. n1+1] and R[1 .. n2+1];
4    for i←1 to n1
5            do L[i]←A[p+i-1];
6    for j←1 to n2
7            do R[j]←A[q+j];
8    L[n1+1]←∞;
9    R[n2+1]←∞;
10   i←1;
11   j←1;
12   for k←p to r
13       do if L[i]≤R[j]
14              then A[k]←L[i];
15                   i←i+1;
16              else A[k]←R[j];
17                   j←j+1;
```

合并排序

- 实例　　a[] = {2,4,5,7,1,2,3,6}

Merge(*a, p, q, r*)过程演示



Step 12-17:

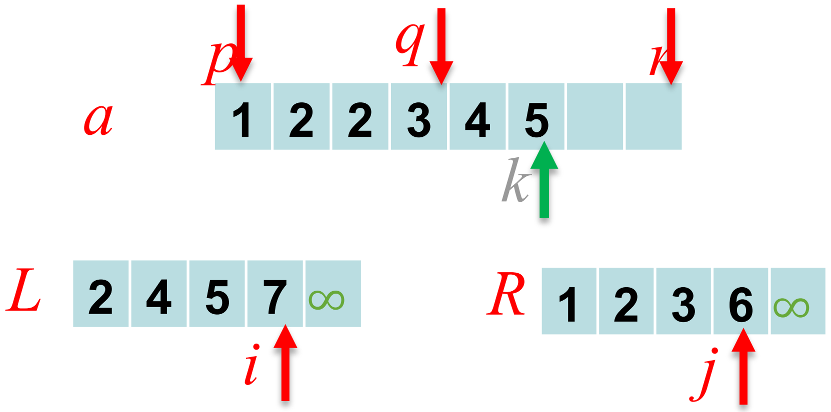
MERGE(A, p, q, r)

1 $n_1 \leftarrow q\text{-}p+1$;
2 $n_2 \leftarrow r\text{-}q$;
3 create arrays $L[1 .. n_1+1]$ and $R[1 .. n_2+1]$;
4 **for** $i \leftarrow 1$ **to** n_1
5 do $L[i] \leftarrow A[p+i-1]$;
6 **for** $j \leftarrow 1$ **to** n_2
7 **do** $R[j] \leftarrow A[q+j]$;
8 $L[n_1+1] \leftarrow \infty$;
9 $R[n_2+1] \leftarrow \infty$;
10 $i \leftarrow 1$;
11 $j \leftarrow 1$;
12 **for** $k \leftarrow p$ **to** r
13 **do if** $L[i] \le R[j]$
14 **then** $A[k] \leftarrow L[i]$;
15 $i \leftarrow i+1$;
16 **else** $A[k] \leftarrow R[j]$;
17 $j \leftarrow j+1$;

合并排序

- 实例　　a[] = {2,4,5,7,1,2,3,6}

Merge(*a, p, q, r*)过程演示



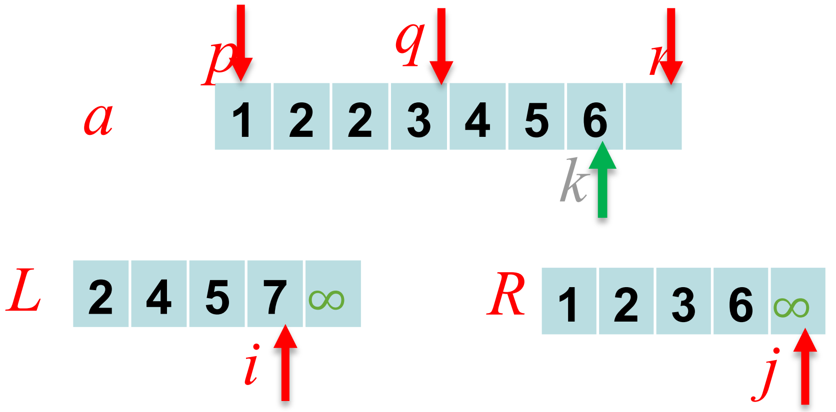
Step 12-17:

MERGE(A, p, q, r)
1　$n_1 \leftarrow q-p+1$;
2　$n_2 \leftarrow r-q$;
3　create arrays $L[1 .. n_1+1]$ and $R[1 .. n_2+1]$;
4　**for** $i \leftarrow 1$ **to** n_1
5　　do $L[i] \leftarrow A[p+i-1]$;
6　**for** $j \leftarrow 1$ **to** n_2
7　　**do** $R[j] \leftarrow A[q+j]$;
8　$L[n_1+1] \leftarrow \infty$;
9　$R[n_2+1] \leftarrow \infty$;
10　$i \leftarrow 1$;
11　$j \leftarrow 1$;
12　**for** $k \leftarrow p$ **to** r
13　　**do if** $L[i] \leq R[j]$
14　　　**then** $A[k] \leftarrow L[i]$;
15　　　　$i \leftarrow i+1$;
16　　　**else** $A[k] \leftarrow R[j]$;
17　　　　$j \leftarrow j+1$;

合并排序

- 实例　　a[] = {2,4,5,7,1,2,3,6}

Merge(*a, p, q, r*)过程演示



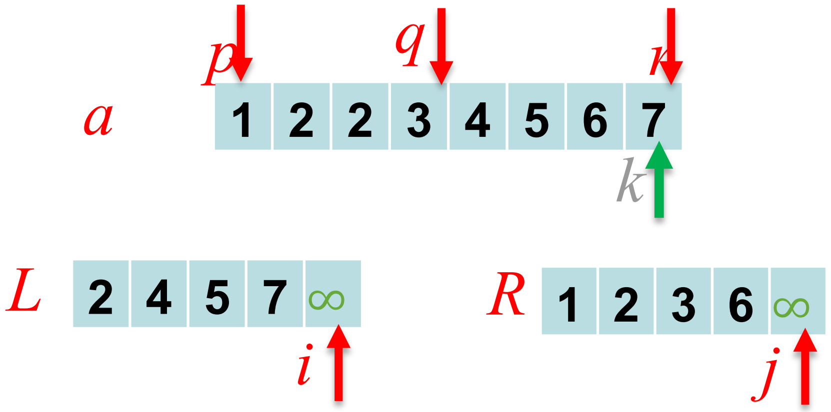
Step 12-17:

```
MERGE(A, p, q, r)
1    n1 ← q-p+1;
2    n2 ← r-q;
3    create arrays L[1 .. n1+1] and R[1 .. n2+1];
4    for i←1 to n1
5            do L[i]←A[p+i-1];
6    for j←1 to n2
7            do R[j]←A[q+j];
8    L[n1+1]←∞;
9    R[n2+1]←∞;
10   i←1;
11   j←1;
12   for k←p to r
13       do if L[i]≤R[j]
14               then A[k]←L[i];
15                    i←i+1;
16               else A[k]←R[j];
17                    j←j+1;
```

合并排序

- 实例 a[] = {2,4,5,7,1,2,3,6}

Merge(*a, p, q, r*)过程演示



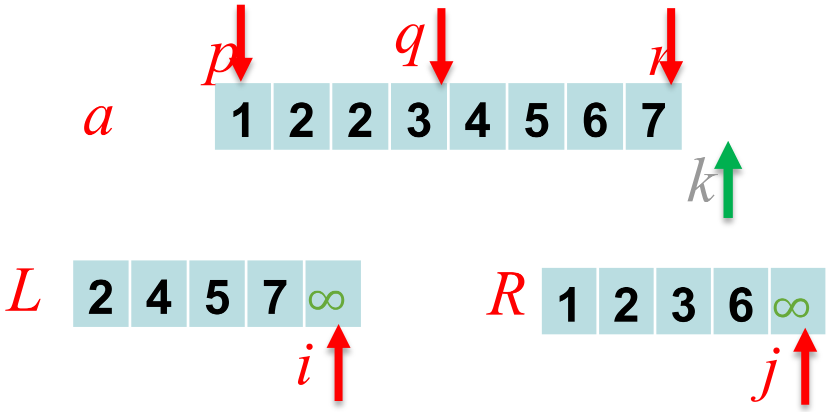
Step 12-17:

```
MERGE(A, p, q, r)
1    n1 ← q-p+1;
2    n2 ← r-q;
3    create arrays L[1 .. n1+1] and R[1 .. n2+1];
4    for i←1 to n1
5             do L[i]←A[p+i-1];
6    for j←1 to n2
7             do R[j]←A[q+j];
8    L[n1+1]←∞;
9    R[n2+1]←∞;
10   i←1;
11   j←1;
12   for k←p to r
13        do if L[i]≤R[j]
14                then A[k]←L[i];
15                         i←i+1;
16                else A[k]←R[j];
17                         j←j+1;
```

合并排序

- 实例　　a[] = {2,4,5,7,1,2,3,6}

Merge(*a, p, q, r*)过程演示



Step 12-17:

MERGE(A, p, q, r)

1 $n_1 \leftarrow q-p+1$;

2 $n_2 \leftarrow r-q$;

3 create arrays $L[1 .. n_1+1]$ and $R[1 .. n_2+1]$;

4 **for** $i \leftarrow 1$ **to** n_1

5 　　do $L[i] \leftarrow A[p+i-1]$;

6 **for** $j \leftarrow 1$ **to** n_2

7 　　**do** $R[j] \leftarrow A[q+j]$;

8 $L[n_1+1] \leftarrow \infty$;

9 $R[n_2+1] \leftarrow \infty$;

10 $i \leftarrow 1$;

11 $j \leftarrow 1$;

12 **for** $k \leftarrow p$ **to** r

13 　**do if** $L[i] \le R[j]$

14 　　**then** $A[k] \leftarrow L[i]$;

15 　　　$i \leftarrow i+1$;

16 　　**else** $A[k] \leftarrow R[j]$;

17 　　　$j \leftarrow j+1$;

合并排序

- Merge算法复杂度分析：

MERGE($a[]$, p, q, r)	cost	times
$n_1 \leftarrow q-l+1$;	c	1
$n_2 \leftarrow r-q$;	c	1
create arrays $L[1 .. n_1+1]$ and $R[1 .. n_2+1]$;	c	1
for $i \leftarrow 1$ **to** n_1	c	n_1+1
do $L[i] \leftarrow A[l+i-1]$;	c	n_1
for $j \leftarrow 1$ **to** n_2	c	n_2+1
do $R[j] \leftarrow A[q+j]$;	c	n_2
$L[n_1+1] \leftarrow \infty$;	c	1
$R[n_2+1] \leftarrow \infty$;	c	1
$i \leftarrow 1$;	c	1
$j \leftarrow 1$;	c	1
for $k \leftarrow p$ **to** r	c	$r-p+2$
do if $L[i] \le R[j]$	c	$r-p+1$
then $A[k] \leftarrow L[i]$;	c	x
$i \leftarrow i+1$;	c	x
else $A[k] \leftarrow R[j]$;	c	$r-p+1-x$
$j \leftarrow j+1$;	c	$r-p+1-x$

T(n)

O(n)　$r-p+1$
$= n_1 + n_2 = n$

O(n)　$1 =< x =< n_1$
$\Theta(n_1+n_2)=\Theta(n)$

有：T(n)=O(n)

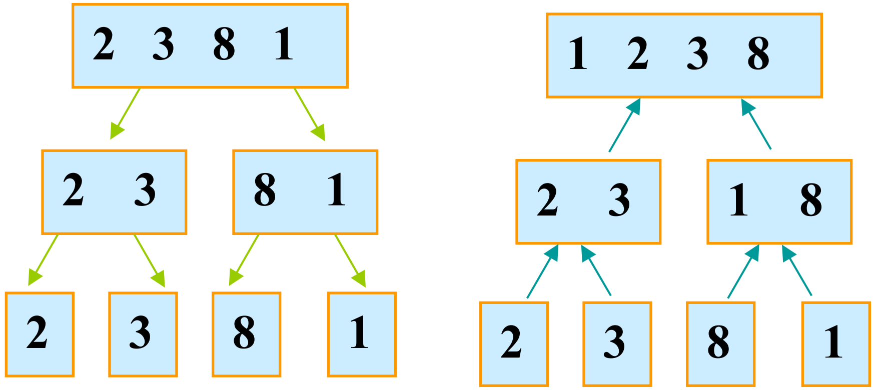
合并排序

- 合并排序算法MergeSort(A, p, r)
  - 分： q $\leftarrow \lfloor (p+r)/2 \rfloor$，分为两个子问题
  - 解：递归求解两个子问题，MergeSort(A, p, q), MergeSort(A, q+1,r)
  - 长度为1 无需排序 ， p>=r
  - 合：将左右已排好序的两部分合并 Merge(A, p, q,r)

MERGESORT(A, p, r)
1 **if** $p < r$
2 **Then** $\{q \leftarrow \lfloor (p+r)/2 \rfloor$;
3 MERGESORT(A, p, q);
4 MERGESORT(A, q+1, r);
5 Merge(A, p, q, r);}

合并排序 - 实例

```
MERGESORT(A, p, r)
1    if p < r
2        Then {q ← ⌊(p + r)/2⌋ ;
3              MERGESORT(A, p, q);
4              MERGESORT(A, q+1, r);
5              Merge(A, p, q, r);}
```




```
MERGE-SORT(A, 1, 4)
1  if 1 < 4
2      Then q ← ⌊(1 + 4)/2⌋ = 2
3          MERGE-SORT(A, 1, 2)
           1  if 1 < 2
           2  Then q ← ⌊(1 + 2)/2⌋ = 1
           3  MERGE-SORT(A, 1, 1)
                 1  if 1 < 1
           4  MERGE-SORT(A, 2, 2)
                 1  if 2 < 2
           5  MERGE(A, 1, 1, 2)
4          MERGE-SORT(A, 3, 4)
           1  if 3 < 4
           2  Then q ← ⌊(3 + 4)/2⌋ = 3
           3  MERGE-SORT(A, 3, 3)
                 1  if 3 < 3
           4  MERGE-SORT(A, 4, 4)
                 1  if 4 < 4
           5  MERGE(A, 3, 3, 4)
5          MERGE(A, 1, 2, 4)
```

合并排序-分析

- 当算法包含递归调用时，时间复杂度可以表示为一个递归表达式：

$$T(n)=\begin{cases}\Theta(1) & if\ n\le c \\ aT\left(\frac{n}{b}\right)+D(n)+C(n) & \text{otherwise}\end{cases}$$

问题规模足够小

- 分：D(n) 分解为子问题的时间
  - 得到数组中间位置，D(n)=Θ(1).
- 解：递归求解2个n/2规模的子问题
- 合：C(n) 合并子问题的解得到原问题解的时间
  - MERGE 过程C(n)=Θ(n)

```
MERGESORT(A, p, r)
    if p < r
        Then {q ← ⌊(p + r)/2⌋ ;
                MERGESORT(A, p, q);
                MERGESORT(A, q+1, r);
                Merge(A, p, q, r);}
```

复杂度分析

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

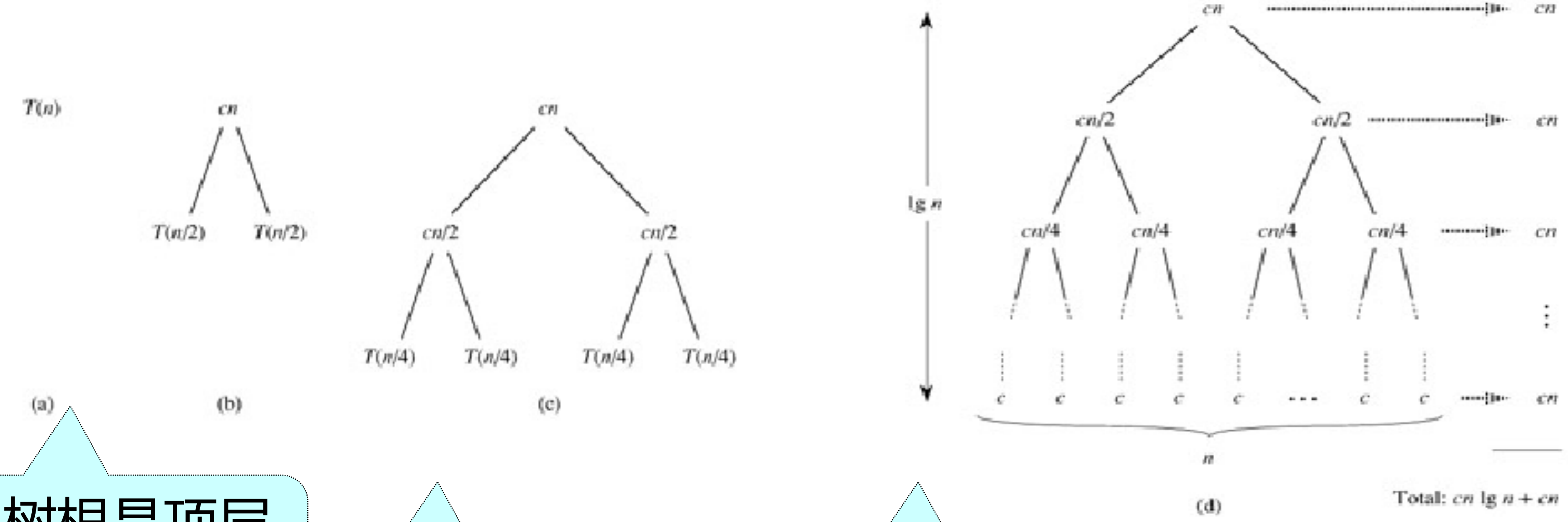
T(n)=O(nlgn) 渐进意义下的最优算法

合并排序 - 分治法分析

重写递归表达式:

$$T(n)=\begin{cases}\Theta(1) & \text{if } n=1\\ 2T(n/2)+\Theta(n) & \text{if } n>1\end{cases} \quad\Rightarrow\quad T(n)=\begin{cases}\Theta(1) & \text{if } n=1\\ 2T(n/2)+cn & \text{if } n>1\end{cases}$$

递归树法求解表达式:



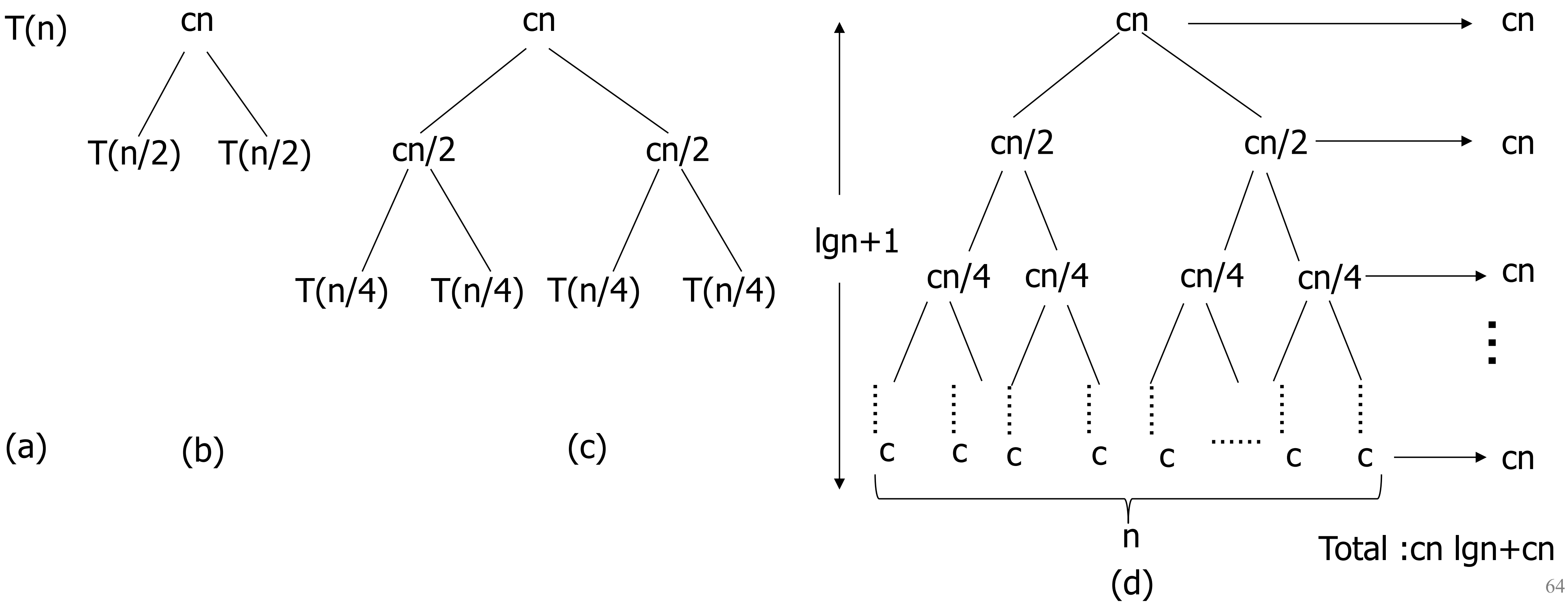
树根是顶层递归的代价

每层结点数

何时终止

合并排序 - 分治法分析

- 把每一层的代价加起来, 共有lgn+1层, 每层代价 cn, 总代价为:
  - $cn(lgn+1) = cnlgn + cn = \Theta(nlog)$
  - 优于插入排序,其时间复杂度为$\Theta(n^2)$

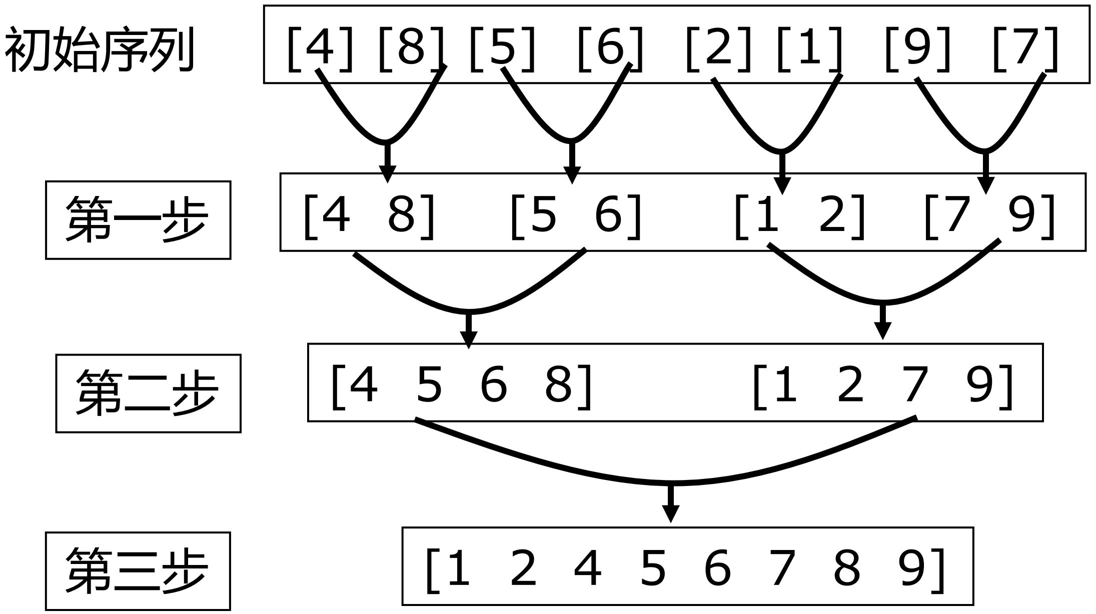


合并排序

- 非递归合并排序算法
  - 算法MERGESORT的递归过程可以消去
- 实现方法
  - 1. 可以把一个长度为n 的无序序列看成是 n 个长度为 1 的有序子序列
  - 2. 首先做两两归并，得到n/2个长度为2的有序子序列 ；再做两两归并，…，如此重复，直到最后得到一个长度为 n 的有序序列。

合并排序

- 例1:　　将初始序列看成是 n 个长度为 1 的有序子序列



整个归并排序仅需$\lceil \log n \rceil$趟

合并排序

- 自然合并排序:
  - 找出排好序的子数组段,再将相邻的子数组段两两合并

已排好序时,T(n)=O(n)

初始序列 [38] [49] [45] [70] [97] [13] [27]

第一步 [38 49] [45 70 97] [13 27]

第二步 [38 45 69 70 97] [13 27]

第三步 [13 27 38 45 49 70 97]

合并排序

- 最坏时间复杂度：O(nlogn)
- 平均时间复杂度：O(nlogn)
- 辅助空间：O(n)

复杂度分析

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

T(n)=O(nlogn) 渐进意义下的最优算法

小结

- 重点
  - 分治法使用条件
  - 分治法算法框架、分治法求解问题的思路
  - 分治法复杂性分析
  - 案例分析：
    - 二分搜索案例，Strasen矩阵乘法，合并排序等
    - 算法基本思想、步骤和时间复杂度
- 难点
  - 分治法算法三步曲
  - 分治法复杂性分析
    - 迭代法求解过程及结果

思考题

- 分析比较二分搜索法和合并排序法的异同点？

创新实践能力训练：统计逆序数

- 问题描述
  - 推荐类网站会根据你对一系列书籍的评价，从它的读者数据库中找出与你的评价非常类似的读者推荐给你，帮你找到品味相近的朋友
  - 假设你对五本书进行了评价，打分从低到高为[1,2,3,4,5]. 读者A对这五本书的打分是[2,4,1,3,5],读者B对这五本书的打分是[3,4,1,5,2].
  - 问题是：应该把读者A还是读者B推荐给你呢？
- 分析
  - 如何量化推荐的准则： 逆序量来度量相似度
  - 逆序：
    - 对于输入序列，若元素的索引i<j, 且$a_i > a_j$,则元素a_i和a_j是一对逆序