第四章　递归和分治

4.1　基于归纳的递归算法

4.2　分治法

4.1 基于归纳的递归算法

4.1.1 基于归纳的递归算法的思想方法

4.1.2 递归算法的例子

4.1.5 整数划分问题的递归算法

4.1.1 归纳法的思想方法

- 什么是递归（Recurrence）？

  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。

- 递归的基本思想：把一个问题划分为一个或多个规模更小的子问题，然后用同样的方法解规模更小的子问题。

- 递归算法的基本设计步骤：
  - 找到问题的初始条件（递归出口），即当问题规模小到某个值时，该问题变得很简单，能够直接求解；
  - 设计一个策略，用于将一个问题划分为一个或多个一步步接近递归出口的、相似的、规模更小的子问题；
  - 将所解决的各个小问题的解组合起来，即可得到原问题的解。

- 设计递归算法时需注意以下几个问题：
  - 如何使定义的问题规模逐步缩小，而且始终保持同一问题类型？
  - 每个递归求解的问题其规模如何缩小？
  - 多大规模的问题可作为递归出口？
  - 随着问题规模的缩小，能到达递归出口吗？

对一个规模为 n 的问题 $P(n)$，归纳法的思想为：

1. 基础步：a_1是问题 $P(1)$ 的解；

2. 归纳步：对所有的 k ，$1 < k < n$，若a_k 是问题 $P(k)$ 的解，则 $P(a_k)$ 是问题 $P(k+1)$ 的解，其中, $P(a_k)$ 是对a_k的某种运算或处理。

4.1.2 递归算法的例子

- 计算阶乘函数 $n!$

  阶乘函数可归纳定义为 $n!=\begin{cases}1 & n=0\\ n(n-1)! & n>0\end{cases}$

  - 其中：
    - n=0时，$n!$=1为边界条件 ——基础步
    - n>0时，$n!=n(n-1)!$为递归方程 ——归纳步

说明：边界条件与递归方程是递归函数的两个要素，递归函数只有具备了这两个要素，才能在有限次计算后得出结果。

- 计算阶乘函数 $n!$

阶乘函数可归纳定义为 $n!=\begin{cases}1 & n=0\\ n(n-1)! & n>0\end{cases}$

算法 计算阶乘函数 $n!$

```
1. int factorial(int n)
2. {
3.     if (n==0)
4.         return 1;
5.     else
6.         return n * factorial(n-1);
7. }
```

1. 多项式求值的递归算法

n 阶多项式：

$$P_n(x)=a_0x^n+a_1x^{n-1}+\cdots+a_{n-1}x+a_n$$

$$=((\cdots(((a_0)x+a_1)x+a_2)x+a_3)x\cdots)x+a_{n-1})x+a_n$$

1）基础步：n=0，有 $p_0=a_0$；

2）归纳步：对任意的 k，$1\le k\le n$，如果前面 k-1 步已计算出 p_{k-1}，则有：$p_k=x\,p_{k-1}+a_k$

把 a_0 存放于 $A[0]$，a_1存放于 $A[1]$，如此等等。

算法4.1 多项式求值的递归算法

```
1. float horner_pol(float x, float A[ ], int n)
2. {
3.     float p;
4.     if (n==0)
5.         p = A[0];
6.     else
7.         p = horner_pol(x, A, n-1) * x + A[n];
8.     return p;
9. }
```

把第7行的乘法作为基本操作，算法的时间复杂性由如下的递归方程确定：

$$\begin{cases} f(0)=0 \\ f(n)=f(n-1)+1 \end{cases}$$

可得：$f(n)=\Theta(n)$

工作单元为$\Theta(n)$

2. 整数幂的计算

- 计算以 x 为底的 n 次幂，简单的方法是让 x 乘以自身 n 次。
- 但效率低，需要$\Theta(n)$ 个乘法。递归算法可以用$\Theta(\log n)$ 来实现它。

1）基础步：$n = 0$，则$x^n = 1$

2）归纳步：若$a = x^{\frac{n}{2}}$，

n 是偶数，即 $n\%2 = 0$, 则 $x^n = a^2 = (x^{n/2})^2$

n 是奇数，即 $n\%2 = 1$, 则 $x^n = x \cdot a^2 = x \cdot (x^{n/2})^2$

算法4.2 整数幂的递归算法

```
1. int power(int x,int n)
2. {
3.     int y;
4.     if (n==0) y = 1;
5.     else {
6.         y = power(x,n/2);
7.         y = y * y;
8.         if (n%2==1)
9.             y = y * x;
10.    }
11.    return y;
12. }
```

第 7 行的乘法作为算法的基本操作，时间复杂性估计如下

$$\begin{cases} f(1)=1 \\ f(n)=f(n/2)+1 \end{cases}$$

设 $n=2^k$，$g(k)=f(2^k)$，把上式改写成

$$\begin{cases} g(0)=1 \\ g(k)=g(k-1)+1 \end{cases}$$

有：$f(n)=g(k)=k+1=\log n+1=\Theta(\log n)$

工作单元为$\Theta(\log n)$

3. 基于递归的插入排序

对 n 个元素的数组进行排序：

1）基础步：当 n=1 时，数组只有一个元素，它已经是排序的；

2）归纳步：如果前面 k-1 个元素已经按递增顺序排序，只要对第 k 个元素逐一与前面 k-1 个元素比较，把它插入适当的位置，即可完成 k 个元素的排序。

算法4.3 基于递归的插入排序算法

```
1. template <class Type>
2. void insert_sort_rec(Type A[ ],int n)
3. {
4.     int k;
5.     Type a;
6.     n = n – 1;
7.     if (n>0) {
8.         insert_sort_rec(A,n);
9.         a = A[n];
10.        k = n – 1;
11.        while ((k>=0)&&(A[k]>a)) {
12.            A[k+1] = A[k];
13.            k = k - 1;
14.        }
15.        A[k+1] = a;
16.    }
17. }
```

算法的时间复杂性可由如下递归方程确定

$$\begin{cases} f(1) = 0 \\ f(n) = f(n-1) + (n-1) \end{cases}$$

容易得到：

$$f(n) = \sum_{i=1}^{n} (i-1) = \sum_{i=1}^{n-1} i = \frac{1}{2} n(n-1)$$

因此，算法的时间复杂性是$O(n^2)$

工作单元为$\Theta(n)$

4. 阿克曼函数

阿克曼函数的递归定义

$$A(m,n)=\begin{cases} n+1 & m=0 \\ A(m-1,1) & n=0 \\ A(m-1,A(m,n-1)) & \text{其他情况} \end{cases}$$

其中，m, n 是自然数

两个参数，其中一个参数又是阿克曼函数，所以，阿克曼函数是双递归函数。

算法4.4 阿克曼函数的递归算法

```
1. long acman(unsighed m,unsighed n)
2. {
3.     long acm;
4.     if (m==0)
5.         acm = n + 1;
6.     else if (n==0)
7.         acm = acman(m-1,1);
8.     else
9.         acm = acman(m-1,acman(m,n-1));
10.    return acm;
11. }
```

4.1.5 整数划分问题的递归算法

1. 整数划分问题

1）用一系列正整数之和的表达式来表示一个正整数，称为整数的划分。

例：7 可划分为：

7

6 + 1

5 + 2，5 + 1 + 1

4 + 3，4 + 2 + 1，4 + 1 + 1 + 1

3 + 3 + 1，3 + 2 + 2，3 + 2 + 1 + 1，3 + 1 + 1 + 1 + 1

2 + 2 + 2 + 1，2 + 2 + 1 + 1 + 1，2 + 1 + 1 + 1+ 1 + 1

1 + 1 + 1+ 1 + 1 + 1 + 1

上述任何一个表达式都称为整数 7 的一个划分。

2）正整数 n 的不同的划分个数称为正整数 n 的划分数，记为 $p(n)$；

3）求正整数 n 的划分数称为整数划分问题。

2. 递归式的推导

1）定义两个函数：

r(n,m)：正整数 n 的划分中加数含 m 而不含大于 m 的所有划分数；

q(n,m)：正整数 n 的划分中加数小于或等于 m 的所有划分数。

例：在 7 的划分中：

含 6 而不含大于 6 的划分有：

6 + 1，因此，r(7,6)=1；

含 5 而不含大于 5 的划分有：

5 + 2，5 + 1 + 1，因此，r(7,5)=2；

含 4 而不含大于 4 的划分有：

4 + 3，4 + 2 + 1，4 + 1 + 1 + 1，因此，r(7,4)=3；

含 3 而不含大于 3 的划分有：

3 + 3 + 1，3 + 2 + 2，3 + 2 + 1 + 1，3 + 1 + 1 + 1 + 1

因此，r(7,3)=4

2）$q(n,m)$ 和 $r(n,m)$ 的关系：

⑴ 加数小于或等于 6 的划分数：

$$q(7,6)=r(7,6)+r(7,5)+r(7,4)+r(7,3)+r(7,2)+r(7,1)=14$$

得：

$$q(n,m)=\sum_{i=1}^{m}r(n,i)=\sum_{i=1}^{m-1}r(n,i)+r(n,m) \tag{4.1.1}$$

$$=q(n,m-1)+r(n,m)$$

3） $r(n,m)$ 是整数 $n-m$ 的不含大于 m 的划分数：

例，$r(7,3)$ 是整数 7 含有 3 而不含大于 3 的所 有划分的个数：

3 + 3 + 1，3 + 2 + 2, 3 + 2 + 1 + 1，3 + 1 + 1 + 1 + 1

同时也是整数 7-3=4 的不含大于 3 的划分数：

3 + 1，2 + 2，2 + 1 + 1，1 + 1 + 1 + 1

因此，有：

$$r(n,m)=q(n-m,m) \qquad (4.1.2)$$

4）递归式：

⑴ 由式 (4.1.1) 和式 (4.1.2) 可得下面递归关系：

$$q(n,m)=q(n,m-1)+q(n-m,m)$$

⑵ 对所有的正整数 n，$r(n,n)=1$ 有：

$$q(n,n)=q(n,n-1)+1$$

⑶ n 的划分不可能包含大于 n 的加数

$$q(n,m)=q(n,n) \qquad m>n$$

⑷ 整数 1 只有一个划分，而不管 m 有多大

$$q(1,m)=1$$

⑸ 对所有整数 n，含 1 而不含大于 1 的划分只有一个，

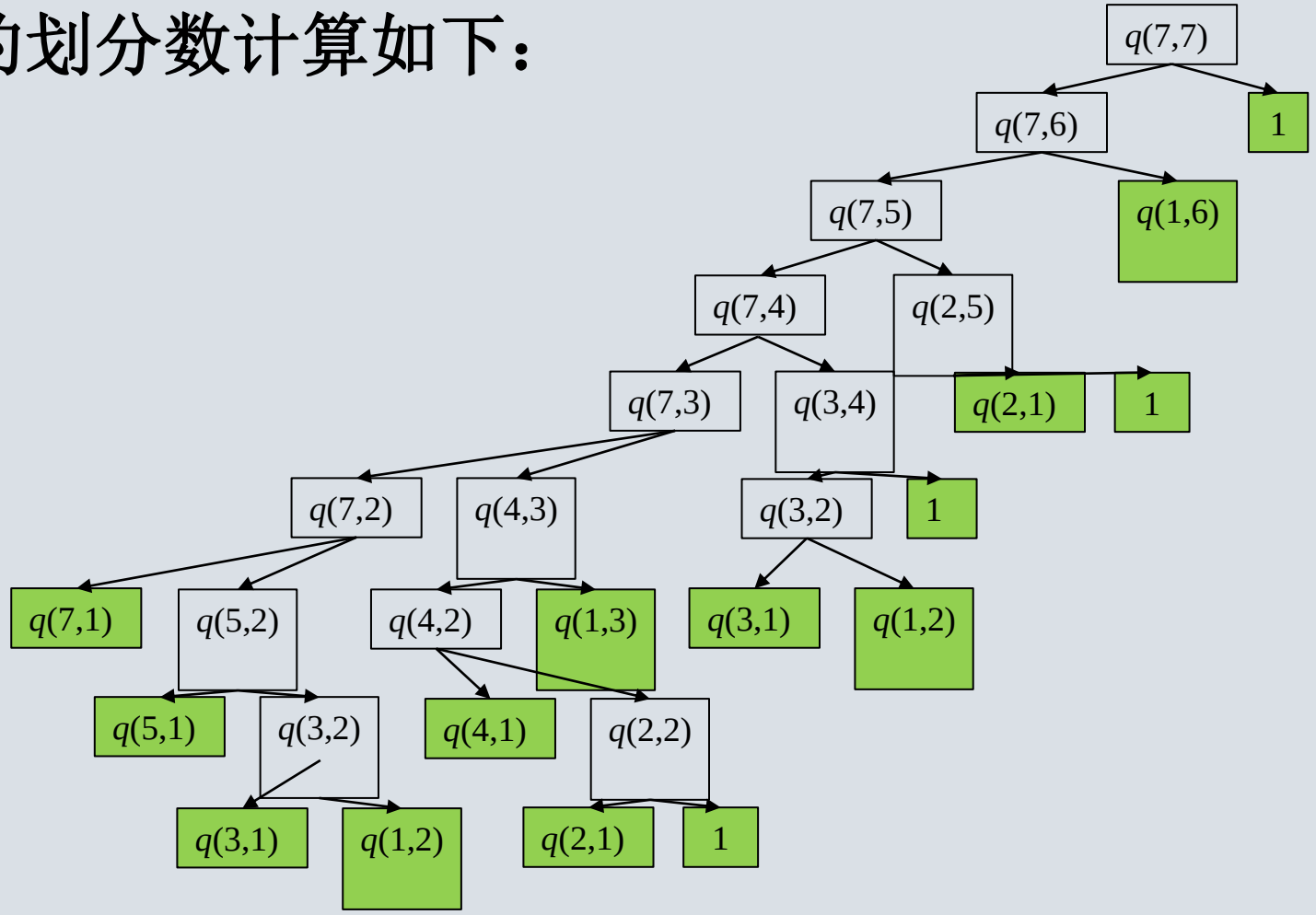
$$q(n,1)=1$$

3. 算法描述

```
1. int q(unsigned n, unsigned m)
2. {
3.     unsigned p;
4.     if (n<1 || m<1) p = 0;
5.     else if (n==1 || m==1)   p = 1;
6.     else if (n<=m)   p = q(n,n-1) + 1;
7.     else p = q(n, m-1) + q(n-m, m);
8.     return p;
9. }
```

整数7的划分数的递归算法*q*(7,7)的工作过程

整数7 的划分数计算如下：



4.2 分治法

4.2.1 分治法的例子

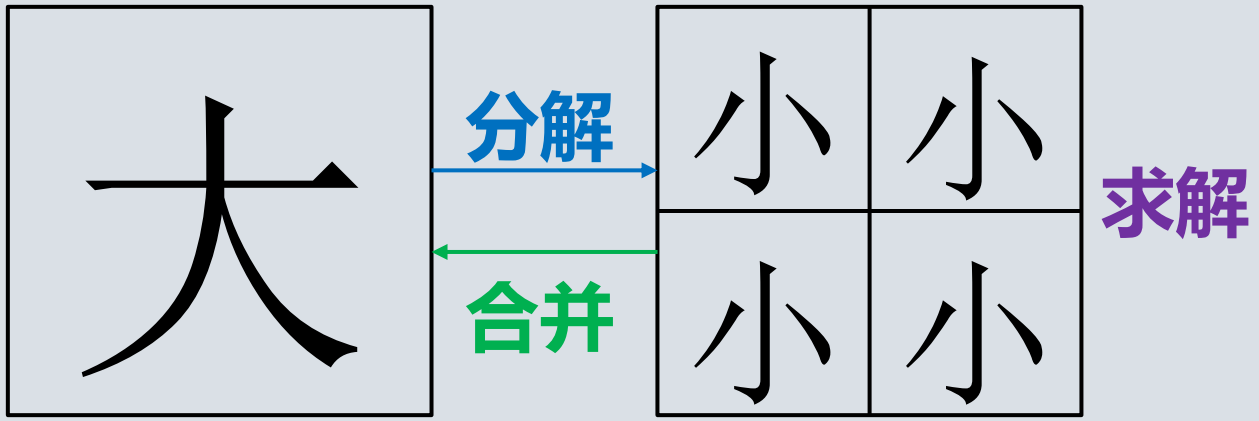
4.2.2 分治法的设计原理

4.2.3 快速排序

4.2.4 多项式乘积和大整数乘法

分治算法总体思想

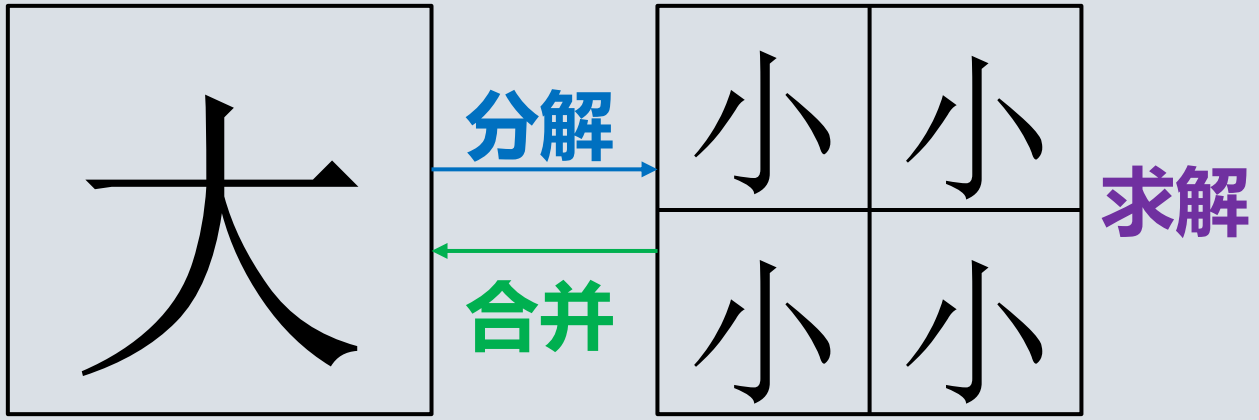
分治策略



- 将要求解的较大规模问题分割成若干个更小规模的子问题；
- 对这些子问题分别求解，如果子问题的规模仍不够小，则再划分为更小的子问题，如此递归的进行下去，直到问题规模足够小、很容易求出其解为止。
- 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。

分治算法总体思想

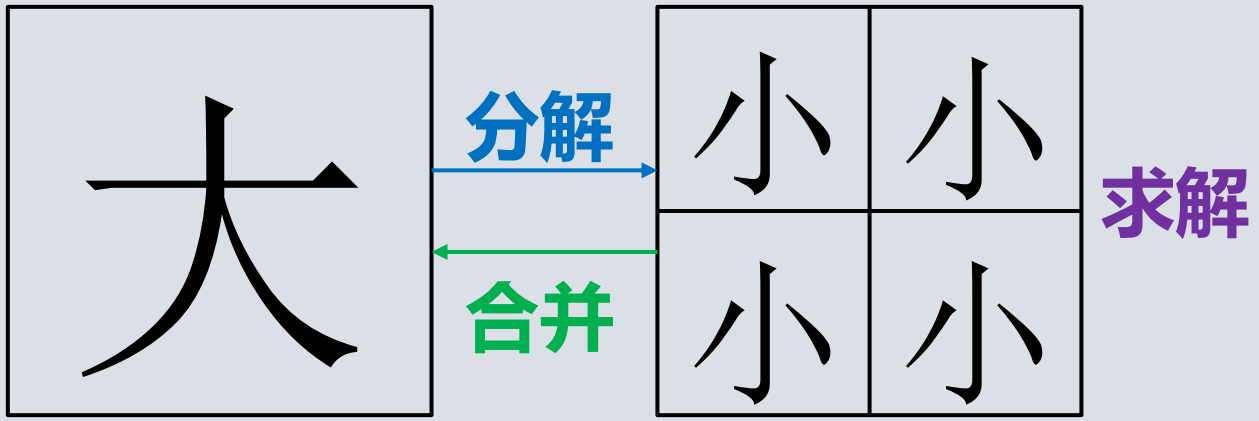
分治策略



一个难以直接解决的大问题 → 一些规模较小的相同问题

各个击破
分而治之

分治策略



P(n) —> {P(1), P(2),…, P(k)}

for i=1 —> k

{

yi<— func(P(i))

}

yn <— merge(y1,y2,…,yk)

通常是将这k个子问题划分成结构和模式相同的方式，利于后续递归方法的调用和边界条件的计算。

将k个子问题的解进行组合，使之合并后的解为所求大规模问题的解。

4.2.1 分治法的例子

1. 最大最小问题的分治算法
2. 合并排序的分治算法

1. 最大最小问题的分治算法

```
1. void max_min(int A[ ], int &e_max, int &e_min, int n)
2. {
3.       int i;
4.       e_max = A[0];  e_min = A[0];
5.        for (i=0; i<n; i++) {
6.              if (A[i]< e_min )    e_min = A[i];
7.              if (A[i]> e_max)    e_max = A[i];
8.        }
9. }
```

- 算法时间复杂度

  算法执行的元素比较次数是$2n-2$

- 问题描述

  用分治法查找数组元素的最大值和最小值

- 方法概述

  1. 将数据集A分为A_1和A_2；
  2. 递归处理A_1和A_2，求解A_1和A_2中的最大和最小值；
  3. 最终的最大和最小值可以计算得到：

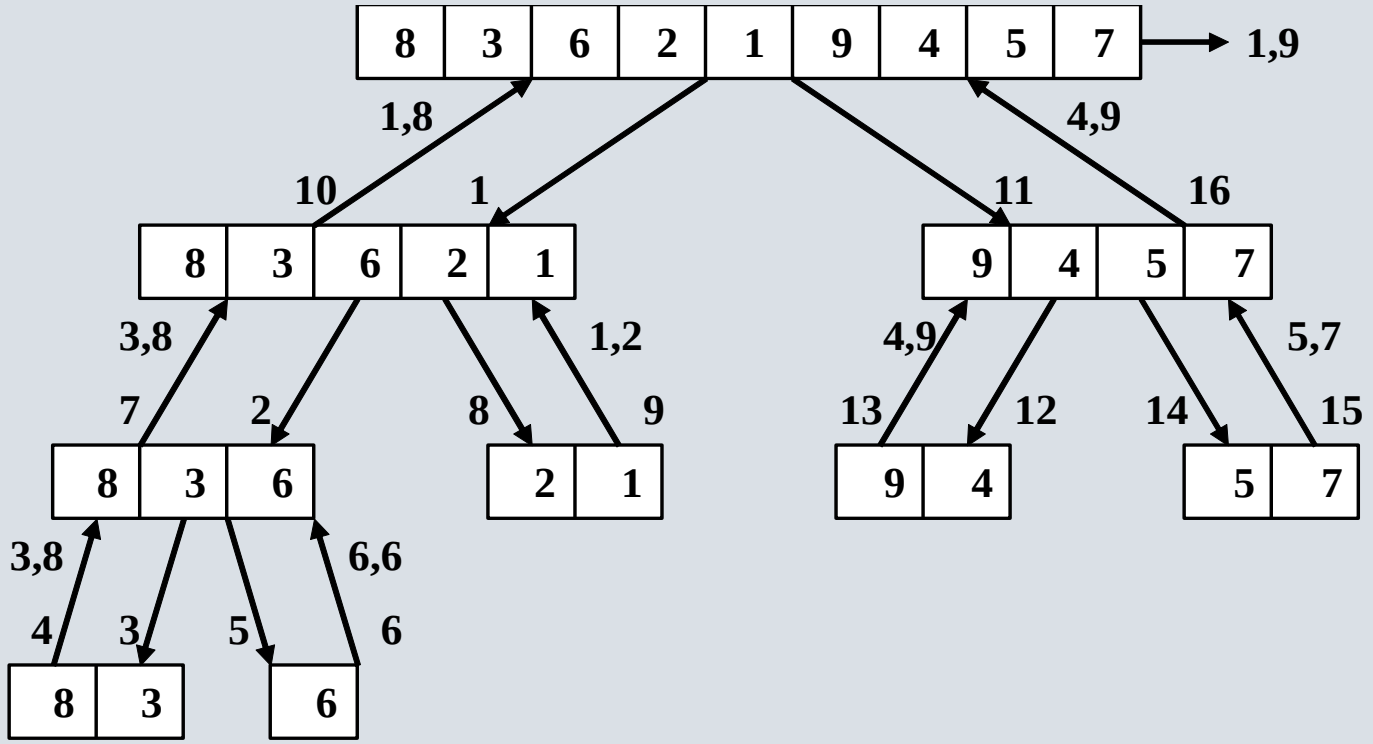
     $\min(A_1,A_2)$,$\max(A_1,A_2)$。

算法描述

```
1. void maxmin(int A[ ], int &e_max, int &e_min, int low, int high)
2. {  int mid, x1, y1, x2, y2;
3.     if ((high-low <= 1)) {   e_max = max(A[low], A[high]);
4.                                e_min = min(A[low], A[high]);
5.     }
6.     else {  mid = (low + high) / 2;    //数组划分为两个子数组
7.             maxmin(A, &x1, &y1, low, mid);    //求取前一个子数组的最大最小
8.             maxmin(A, &x2, &y2, mid+1, high);  //求取后一个子数组最大最小
9.              e_max = max(x1, x2);
10.             e_min  = min(y1, y2);    //合并得到原数组的最大最小
11.    }
12. }
```

分治法解最大最小问题的过程

分治法解最大最小问题的过程例子：



分治法解最大最小问题的分析

$$T(1) = 0$$

$$T(2) = 1$$

$$\cdots$$

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

$$= 2^2 T\left(\frac{n}{2^2}\right) + 2^2 + 2$$

$$= \cdots$$

$n = 2^k$

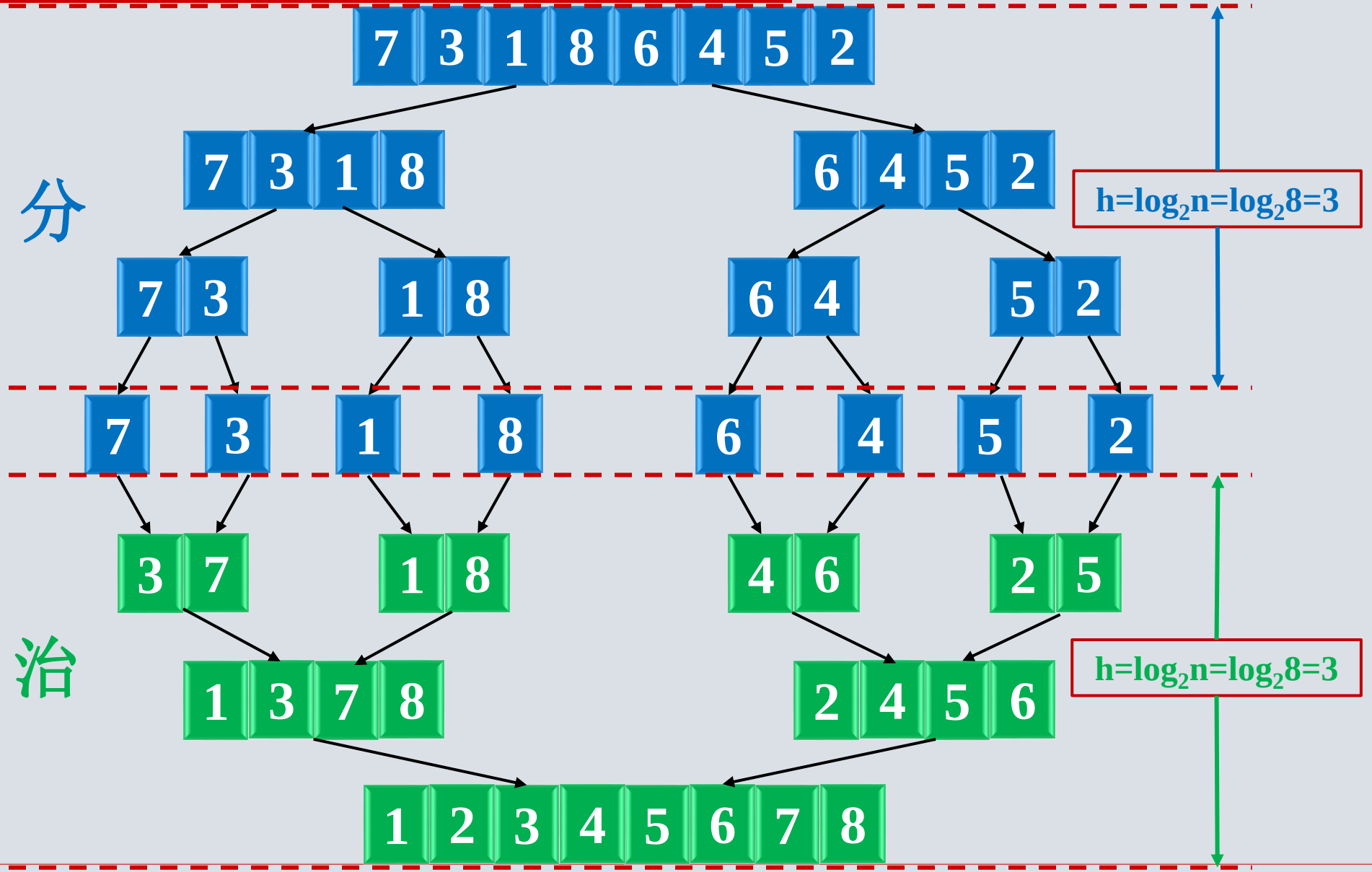
$h(k) = 2h(k\text{-}1)+2$

$$= 2^{k-1}T(2) + 2^{k-1} + 2^{k-2} + \cdots + 2^2 + 2$$

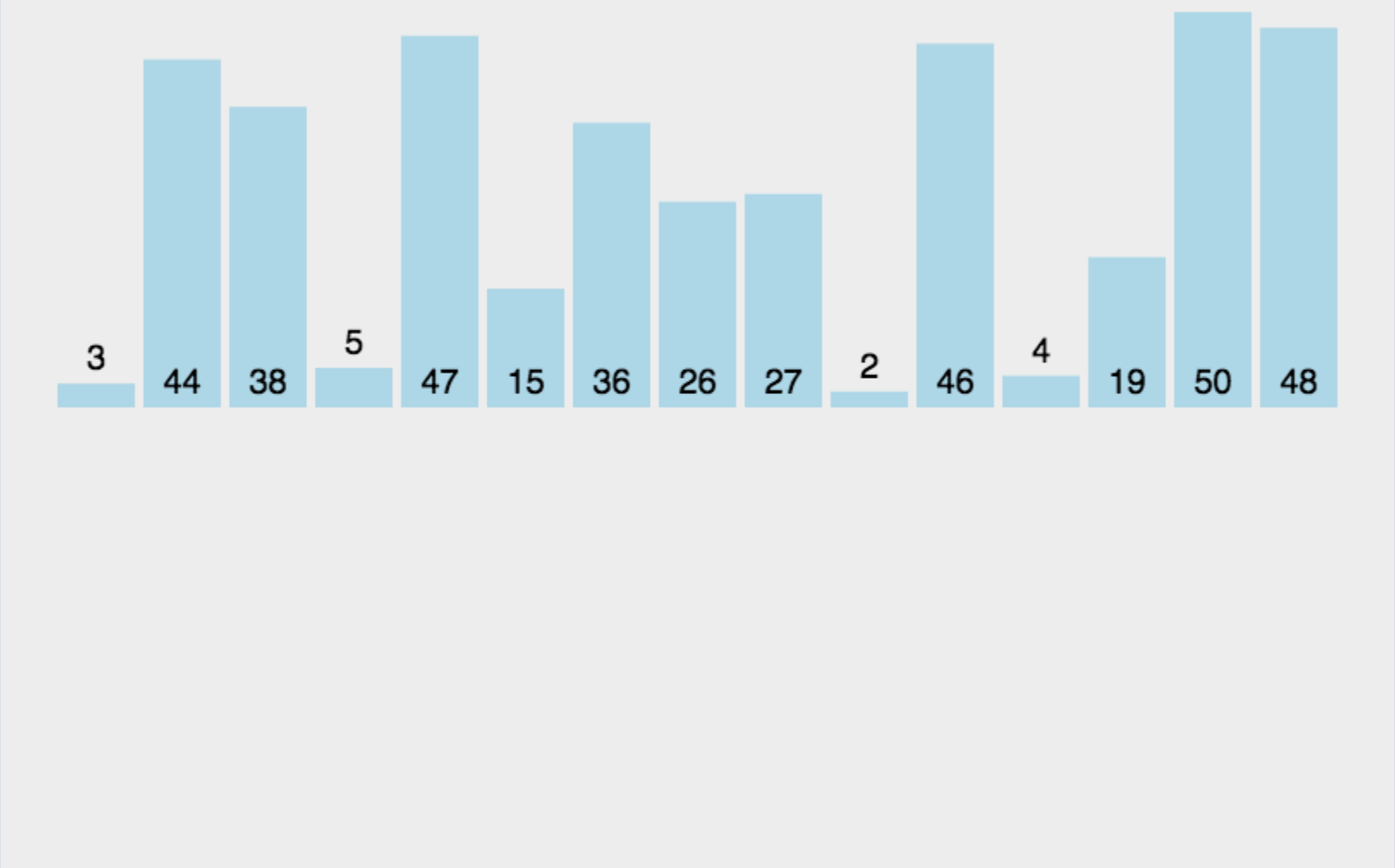
$$= 2^{k-1} + 2^k - 2$$

$$= \frac{n}{2} + n - 2 = \frac{3n}{2} - 2$$

2. 合并排序的分治算法


7 3 1 8 6 4 5 2
7 3 1 8
6 4 5 2
7 3
1 8
6 4
5 2
7
3
1
8
6
4
5
2
3 7
1 8
4 6
2 5
1 3 7 8
2 4 5 6
1 2 3 4 5 6 7 8
分
治
h=log2n=log28=3
h=log2n=log28=3

排序：3,44,38,5,47,15,36,26,27,2,46,4,19,50,48



算法的实现

```
mergesort(Type A[ ], int low, int high)
{    int mid;
     if (high > low)
     {    mid = (high + low) / 2;
          mergesort(A, low, mid);
          mergesort(A, mid + 1, high);
          merge(A, low, mid, high);
     }
}
```

算法的分析

（1）merge 的时间复杂性：

最好：$n / 2$　　最坏：$n - 1$

（2）最坏情况下的算法分析

$$\begin{cases} f(n) = 2f(n/2) + n - 1 \\ f(1) = 0 \end{cases}$$

令 $n = 2^k$

$$\begin{cases} h(k) = 2h(k\text{-}1) + 2^k - 1 \\ h(0) = 0 \end{cases}$$

$$\begin{cases} h(0) = 0 \\ h(k) = 2h(k-1) + 2^k - 1 \end{cases}$$

$$\begin{aligned} h(k) &= 2(2h(k-2) + 2^{k-1} - 1) + 2^k - 1 \\ &= 2^2 h(k-2) + 2 \times 2^k - 2^1 - 2^0 \\ &= 2^2 (2h(k-3) + 2^{k-2} - 1) + 2^k - 2^1 - 2^0 \\ &= 2^3 h(k-3) + 3 \times 2^k - 2^2 - 2^1 - 2^0 \\ &= \cdots\cdots \\ &= 2^k h(0) + k \times 2^k - 2^{k-1} - \cdots - 2^1 - 2^0 \\ &= k \times 2^k - (2^k - 1) \\ &= n\log n - n + 1 \end{aligned}$$

4.2.2 分治法的设计原理

- 分治法的适用条件
- 分治法的一般描述
- 分治法的设计步骤
- 分治法的时间复杂性

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；

因为问题的计算复杂性一般是随着问题规模的增加而增加，因此大部分问题满足这个特征。

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，即该问题具有最优子结构性质；

这条特征是应用分治法的前提，它也是大多数问题可以满足的，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：
  - 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，即该问题具有最优子结构性质；
  - 利用该问题分解出的子问题的解可以合并为该问题的解；

能否利用分治法，完全取决于问题是否具有这条特征，如果具备了前两条特征，而不具备第三条特征，则可以考虑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：
  - 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，即该问题具有最优子结构性质；
  - 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。

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，此时虽然也可以用分治法，但一般用动态规划较好。

例：找伪币问题

- 一个装有16个硬币的袋子。16个硬币中有一个是伪造的，并且那个伪造的硬币比真的硬币要轻一些。任务是找出这个伪造的硬币。为了完成这项任务，将提供一台可用来比较两组硬币重量的仪器，利用这台仪器，可以知道两组硬币的重量是否相同。

- 算法描述
  - 1）将16个硬币分成A、B两半；
  - 2）将A放仪器的一边，B放另一边。如果A轻，则表明伪币在A，解子问题A即可，否则解子问题B。
- 称几次？

16 —— 8

8 —— 4

4 —— 2

2 —— 1

伪币问题的推广：二分搜索

- 给定已升序排好的n个元素，现在要在这n个元素中找出一特定元素。
- 分析：
  - ✓ 该问题的规模缩小到一定的程度就可以轻易地解决；
  - ✓ 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；
  - ✓ 分解出的各子问题是相互独立的。

伪币问题的推广：二分搜索

- 给定已升序排好的n个元素，现在要在这n个元素中找出一特定元素。

```
template<class Type>
 int binary_search(int A[], const Type &x, int l, int r)
  {
      while (r>=l) {
          int m=(r+l)/2;
          if (x==A[m])  return m;
          if (x<A[m])  r = m –1; else l=m+1;
      }
     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小一半。因此在最坏情况下，while循环被执行了$O(\log n)$次。循环体内运行需要$O(1)$时间，因此整个算法在最坏情况下的计算时间复杂性为$O(\log n)$。

分治法的一般描述

```
divide_and_conquer(P(n))
{
    if (n <=n0  )    return adhoc(P(n));//解决小规模问题
    else {
       divide  P into smaller subinstances  P1 ,…, 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
    }
}
```

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子问题的思想，它几乎总是比子问题规模不等的做法要好。

分治法的设计步骤

- ✓ 1）划分步：把输入的问题实例划分为 k 个子问题。尽量使 k 个子问题的规模大致相同，例如， $k = 2$，如最大最小问题取其中的一部分。 k =1时的划分，仍把问题分为两部分，但取其中一部分，而丢弃另一部分，如二叉检索问题用分治法处理的情况。
- ✓ 2）治理步：由 k 个递归调用组成；

  阈值n_0的选取：取决于算法中的 adhoc 对阈值的敏感程度，以及 merge 的处理情况。
- ✓ 3）组合步：把 k 个子问题的解组合起来。

分治法的时间复杂性

1）分治算法的运行时间：

- ✓ 令$n=k^m$，$n_0=1$；
- ✓ 把问题划分为k个子问题求解；
- ✓ merge 把 k 个子问题的解合并成原问题的解，花费 bn 个单位时间。
- ✓ 分治法的运行时间为：

$$\begin{cases} f(1) = 1 \\ f(n) = kf(n/k) + bn \end{cases}$$

因为 $n=k^m$，所以有：

$$f(k^m) = kf\left(k^{m-1}\right) + bk^m$$

令 $f(k^m) = h(m)$，则有：

$$\begin{cases} h(0) = 1 \\ h(m) = kh(m-1) + bk^m \end{cases}$$

解:

$$\begin{cases} h(0) = 1 \\ h(m) = kh(m-1) + bk^m \end{cases}$$

$$f(n) = h(m) = k\left(kh(m-2) + bk^{m-1}\right) + bk^m$$

$$= \cdots$$

$$= k^m h(0) + mbk^m$$

$$= n + bnlog_k n$$

- ✓ **adhoc 花费 n 个单位时间，组合步花费 $bnlog_k n$ 时间；**
- ✓ **当 b > 1， n 很大时，它确定了算法的实际性能。**

2）adhoc、merge及子问题个数与时间复杂性的关系：

- adhoc 的运行时间确定递归方程的初始项;
- merge 的运行时间确定递归方程的非齐次项;
- 子问题个数确定递归方程低阶项的系数。

定理4.1 令 b, d 是非负整数，n是2的幂，则递归方程

$$\begin{cases} f(1) = d \\ f(n) = 2f(n/2) + bn\log n \end{cases}$$

的解为：$f(n) = \Theta(n\log^2 n)$

引理4.1 令 a, c 是非负整数，b, d, x 是非负常数，对某个非负整数 k，有$n = c^k$，则递归方程:

$$\begin{cases} f(1)=d \\ f(n) = af(n/c) + bn^x \end{cases}$$

的解为:

$$f(n) = bn^x \log_c n + dn^x \qquad a = c^x$$

$$f(n) = \left(d + \frac{bc^x}{a - c^x}\right) n^{\log_c a} - \left(\frac{bc^x}{a - c^x}\right) n^x \qquad a \neq c^x$$

推论4.1 令 a, c 是非负整数，b, d, x 是非负常数，对某个非负整数 k，有$n = c^k$，则递归方程:

$$\begin{cases} f(1)=d \\ f(n) = af(n/c) + bn^x \end{cases}$$

的解满足：

$$f(n) = bn^x \log_c n + dn^x \qquad a = c^x$$

$$f(n) \le \begin{cases} dn^x & a < c^x, d \ge bc^x/(c^x - a) \\ \left(\dfrac{bc^x}{c^x - a}\right) n^x & a < c^x, d < bc^x/(c^x - a) \end{cases}$$

$$f(n) \le \left(d + \frac{bc^x}{a - c^x}\right) n^{\log_c a} \quad a > c^x$$

推论4.2 令 a, c 是非负整数，$b, d,$ 是非负常数， x=1对某个非负整数 k，有$n = c^k$，则递归方程:

$$\begin{cases} f(1)=d \\ f(n) = af(n/c) + bn \end{cases}$$

的解满足：

$$f(n) = bn\log_c n + dn \qquad a = c$$

$$f(n) = \left(d + \frac{bc}{a-c}\right)n^{\log_c a} - \left(\frac{bc}{a-c}\right)n \qquad a \neq c$$

定理4.2 令 a, c 是非负整数，b, d, x 是非负常数，对某个非负整数 k，有$n = c^k$，则递归方程:

$$\begin{cases} f(1)=d \\ f(n) = af(n/c) + bn^x \end{cases}$$

的解为:

$$f(n) = \begin{cases} \Theta(n^x) & n < c^x \\ \Theta(n^x logn) & n = c^x \\ \Theta\left(n^{log_c a}\right) & n > c^x \end{cases}$$

令 $x = 1$，有:

$$f(n) = \begin{cases} \Theta(n) & n < c \\ \Theta(nlogn) & n = c \\ \Theta\left(n^{log_c a}\right) & n > c \end{cases}$$

3）子问题规模对时间复杂性的影响

定理4.3 令 b 和c_1、c_2 是大于 0 的常数，则递归方程:

$$\begin{cases} f(1) = b \\ f(n) = f(\lfloor c_1 n \rfloor) + f(\lfloor c_2 n \rfloor) + bn \end{cases}$$

的解为：

$$f(n) = \begin{cases} \Theta(nlogn) & c_1 + c_2 = 1 \\ \Theta(n) & c_1 + c_2 < 1 \end{cases}$$

当 $c_1 + c_2 < 1$ 时，有：

$$f(n) \leq \frac{bn}{(1-c_1-c_2)} = O(n)$$

4.2.3 快速排序

- 快速排序的算法思想
- 快速排序算法的描述
- 序列的划分算法
- 最坏情况分析
- 平均情况分析

快速排序的算法思想

- 基本思想：先找一个基准元素（例如待排序数组的第一个元素），进行一趟快速排序，使得该基准元素左边的所有数据都比它小，而右边的所有数据都比它大；然后再按此方法，对左右两边的数据分别进行快速排序，整个排序过程可以递归进行，以此使整个数组变成有序序列。
- 优点：因为每趟可以确定不止一个元素的位置，而且呈指数增加，所以效率高。

- 该问题是将n个元素排成非递减顺序。
- 解决思路：快速排序基于分治思想
  - **Divide**：分割$A[p...r]$成为2个子集合（可以空），$A[p...q-1]$和$A[q+1...r]$，使得$A[p...q-1]$的每个元素都小于等于$A[q]$，$A[q+1...r]$中的每个元素都大于等于$A[q]$，索引q作为分割点；
  - **Conquer**：通过递归调用快速排序对$A[p...q-1]$和$A[q+1...r]$排序；
  - **Combine**：每个子集合排序完成之后，整个数组的排序也完成。

快速排序算法的描述

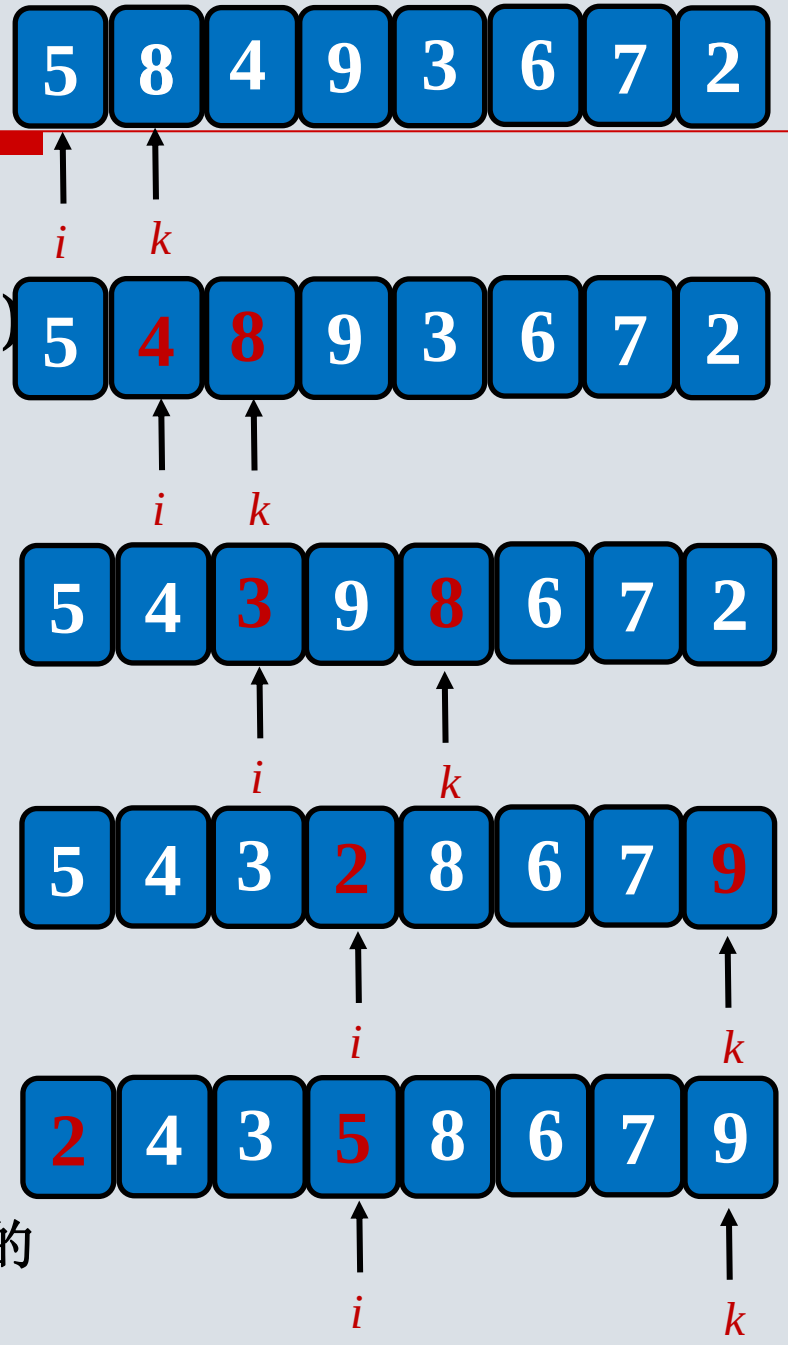
序列的划分算法

- 枢点元素：给定序列$a_1,a_2,\ldots,a_n$，如果存在元素a_k，使得对所有的$i, 1 \le i < k$，都有$a_i \le a_k$，对所有的$j, k < j \le n$，都有$a_k \le a_j$，就称a_k是这个序列的枢点（pivot）元素。
- 把序列的第一个元素作为枢点元素，重新排列这个序列。

算法：按枢点元素划分序列

```
2. int split(Type A[ ], int low, int high)
3. {   int k, i = low;
5.     Type x = A[low];
6.     for (k=low+1; k<=high; k++) {
7.         if (A[k]<=x) {
8.             i = i + 1;
9.             if (i!=k)   swap(A[i], A[k]);
11.        }
12.    }
13.    swap(A[low], A[i]);
14.    return i;
15. }
```

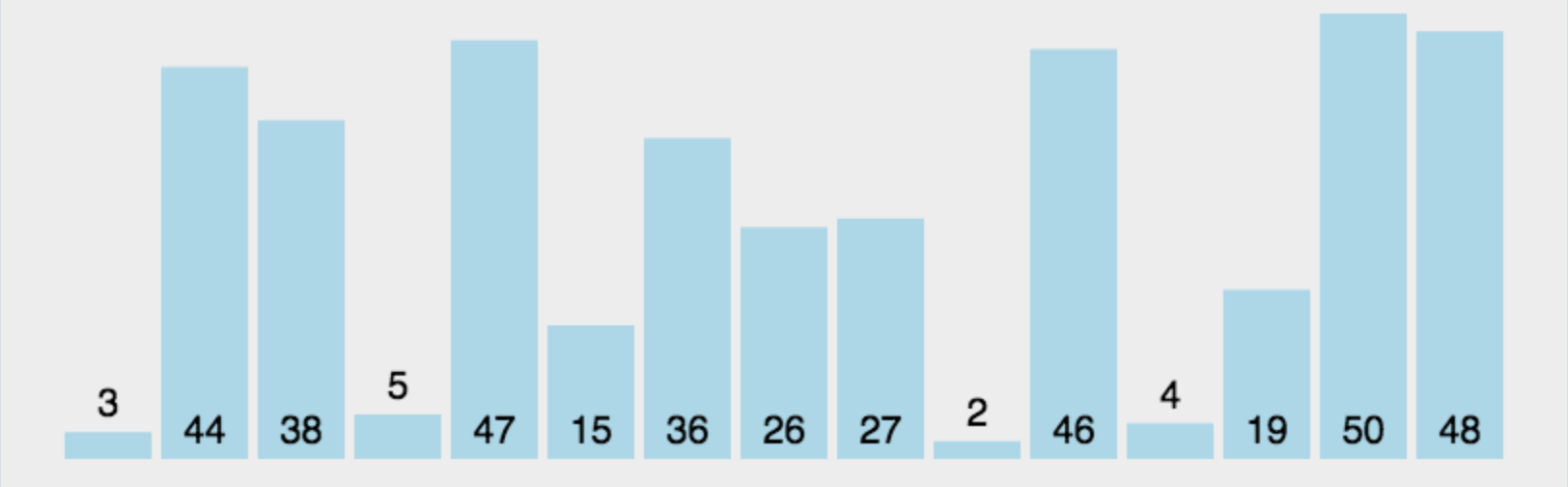
返回值：枢点元素的位置，n 个元素的序列，需执行 n – 1 次元素比较.



快速排序算法：

```
1. template <class Type>
2. void quick_sort(Type A[ ], int low, int high)
3. {
4.     int k;
5.     if (low < high) {
6.         k = split(A, low, high);
7.         quick_sort(A, low, k-1);   //对左半段排序
8.         quick_sort(A, k+1, high); //对右半段排序
9.     }
10. }
```

例　快速排序算法的执行过程


3
44
38
5
47
15
36
26
27
2
46
4
19
50
48

最坏情况分析

元素已按递增或递减顺序排列，处于最坏的情况。split 所得到的枢点元素位置，或者是子序列的开始位置，或者是子序列的结束位置。

第 i 次划分	子序列长度		所执行的元素比较次数
i = 1	0	n – 1	n – 1
i = 2	1	n – 2	n – 2
i = n	n-1	0	0

算法quick_sort所执行的元素比较的总次数是：

$$(n-1)+(n-2)+\cdots+1+0=\frac{1}{2}n(n-1)=\Theta(n^2)$$

随机划分：

```
int randomizedPartition (int a[],int p, int r)
  {
    i = random(p,r);
    swap(a[i], a[p]);
    return partition (a,p, r);
  }
```

平均情况分析

假定所有元素的关键字的值都不相同，被选取作为枢点元素的可能性都为 1/ n。枢点元素经 split 重新排列后，位于序列的第 k 位置，$1 \le k \le n$，则枢点元素左边的元素个数有 k–1 个，右边的元素个数有$n-k$个。有递归方程：

$$\begin{cases} f(0) = 0 \\ f(n) = (n-1) + \dfrac{1}{n}\sum_{k=1}^{n}(f(k-1) + f(n-k)) \end{cases}$$

因为：$$\sum_{k=1}^{n} f(k-1) = f(0) + f(1) + \cdots + f(n-1) = \sum_{k=1}^{n} f(n-k) = \sum_{k=0}^{n-1} f(k)$$

有：$$f(n) = (n-1) + \frac{2}{n}\sum_{k=0}^{n-1} f(k)$$

$$f(n) = (n-1) + \frac{2}{n}\sum_{k=0}^{n-1} f(k)$$

两边乘以 n：$nf(n) = n(n-1) + 2\sum_{k=0}^{n-1} f(k)$ (1)

用 $n-1$ 取代 n：$(n-1)f(n-1) = (n-1)(n-2) + 2\sum_{k=0}^{n-2} f(k)$ (2)

(1) – (2) 得：$nf(n) = (n+1)f(n-1) + 2(n-1)$

两边除以 n(n + 1)：$\frac{f(n)}{n+1} = \frac{f(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$

令 $h(n) = \frac{f(n)}{n+1}$ 得：$\begin{cases} h(0) = 0 \\ h(n) = h(n-1) + \frac{2(n-1)}{n(n+1)} \end{cases}$

$$\begin{cases} h(0) = 0 \\ h(n) = h(n-1) + \frac{2(n-1)}{n(n+1)} \end{cases}$$

$$h(n) = \sum_{k=1}^{n} \frac{2(k-1)}{k(k+1)} = 2\sum_{k=1}^{n} \frac{2}{k+1} - 2\sum_{k=1}^{n} \frac{1}{k}$$

$$= 4\sum_{k=2}^{n+1} \frac{1}{k} - 2\sum_{k=1}^{n} \frac{1}{k}$$

$$= 4\left[\sum_{k=1}^{n} \frac{1}{k} + \frac{1}{n+1} - 1\right] - 2\sum_{k=1}^{n} \frac{1}{k}$$

$$= 2\sum_{k=1}^{n} \frac{1}{k} - \frac{4n}{n+1}$$

$$= 2\sum_{k=1}^{n}\frac{1}{k} - \frac{4n}{n+1}$$

$$h(n) = 2lnn - \Theta(1)$$

$$= \frac{2logn}{loge} - \Theta(1)$$

$$= 1.44logn$$

$$f(n) = (n+1)h(n) = 1.44nlogn$$

思考？

- 递归和迭代
  - ✓ 例： 求$n!$
    - 递归：$f(n) = nf(n-1)$
    - 迭代：$x = x * i$

➢ 递归占用的存储空间随问题规模扩大；

➢ 迭代占用的存储空间与问题规模无关。

- 递归函数的内部执行过程：
  - 运行开始时，为递归调用建立一个工作栈，其结构包括实参、局部变量和返回地址；
  - 每次执行递归调用之前，把递归函数的实参和局部变量的当前值以及调用后的返回地址压栈；
  - 每次递归调用结束后，将栈顶元素出栈，使相应的实参和局部变量恢复为调用前的值，然后转向返回地址指定的位置继续进行执行。