“十二五”普通高等教育本科国家级规划教材

国家精品课程教材

计算机算法设计与分析习题解答

（第5版）

◎ 王晓东　编著

非外借

中国工信出版集团

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

“十二五”普通高等教育本科国家级规划教材
国家精品课程教材

计算机算法设计与分析习题解答

（第5版）

王晓东 编著

電子工業出版社
Publishing House of Electronics Industry
北京 · BEIJING

内 容 简 介

本书是与“十二五”普通高等教育本科国家级规划教材《计算机算法设计与分析（第5版）》配套的辅助教材和国家精品课程教材，分别对主教材中的算法分析题和算法实现题给出了解答或解题思路提示。为了提高学生灵活运用算法设计策略解决实际问题的能力，本书还将主教材中的许多习题改造成算法实现题，要求学生设计出求解算法并上机实现。本书教学资料包含各章算法实现题、测试数据和答案，可在华信教育资源网免费注册下载。

本书内容丰富，理论联系实际，可作为高等学校计算机科学与技术、软件工程、信息安全、信息与计算科学等专业本科生和研究生学习计算机算法设计的辅助教材，也是工程技术人员和自学者的参考书。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

计算机算法设计与分析习题解答/王晓东编著. —5版. —北京：电子工业出版社，2018.10

ISBN 978-7-121-34438-1

Ⅰ. ① 计… Ⅱ. ① 王… Ⅲ. ① 电子计算机－算法设计－高等学校－题解 ② 电子计算机－算法分析－高等学校－题解 Ⅳ. ① TP301.6-44

中国版本图书馆CIP数据核字（2018）第120711号

策划编辑：章海涛
责任编辑：章海涛
印　　刷：三河市良远印务有限公司
装　　订：三河市良远印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　邮编　100036
开　　本：787×1092　1/16　　印张：22.75　　字数：580千字
版　　次：2005年8月第1版
　　2018年10月第5版
印　　次：2018年10月第1次印刷
定　　价：56.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

本书咨询联系方式：192910558（QQ群）。

前　言

一些著名的计算机科学家在有关计算机科学教育的论述中认为，计算机科学是一种创造性思维活动，其教育必须面向设计。“计算机算法设计与分析”正是一门面向设计，且处于计算机学科核心地位的教育课程。通过对计算机算法系统的学习与研究，理解掌握算法设计的主要方法，培养对算法的计算复杂性正确分析的能力，为独立设计算法和对算法进行复杂性分析奠定坚实的理论基础，对每一位从事计算机系统结构、系统软件和应用软件研究与开发的科技工作者都是非常重要和必不可少的。课程结合我国高等学校教育工作的现状，追踪国际计算机科学技术的发展水平，更新了教学内容和教学方法，以算法设计策略为知识单元，在内容选材、深度把握、系统性和可用性方面进行了精心设计，力图适合高校本科生教学对学时数和知识结构的要求。

本书是“十二五”普通高等教育本科国家级规划教材《计算机算法设计与分析（第5版）》（ISBN 978-7-121-34439-8）配套的辅助教材，对《计算机算法设计与分析（第5版）》一书中的全部习题做了详尽的解答，旨在让使用该书的教师更容易教，学生更容易学。为了便于对照阅读，本书的章序与《计算机算法设计与分析（第5版）》一书的章序保持一致，且一一对应。

本书的内容是对《计算机算法设计与分析（第5版）》的较深入的扩展，许多教材中无法讲述的较深入的主题通过习题的形式展现出来。为了加强学生灵活运用算法设计策略解决实际问题的能力，本书将主教材中的许多习题改造成算法实现题，要求学生不仅设计出解决具体问题的算法，而且能上机实现。作者的教学实践反映出这类算法实现题的教学效果非常好。作者还结合国家精品课程建设，建立了“算法设计与分析”教学网站。国家精品资源共享课地址：http://www.icourses.cn/sCourse/course_2535.html。欢迎广大读者访问作者的教学网站并提出宝贵意见。

在本书编写过程中，福州大学“211工程”计算机与信息工程重点学科实验室为本书的写作提供了优良的设备与工作环境。电子工业出版社负责本书编辑出版工作的全体同仁为本书的出版付出了大量辛勤劳动，他们认真细致、一丝不苟的工作精神保证了本书的出版质量。在此，谨向每位曾经关心和支持本书编写工作的各方面人士表示衷心的谢意！

作　者

目　录

第 2 章　递归与分治策略

算法分析题 2

2-1　证明 Hanoi 塔问题的递归算法与非递归算法实际上是一回事。

分析与解答：Hanoi 塔问题的递归算法：

```
void hanoi(int n, int A, int B, int C) {
  if(n > 0) {
    hanoi(n-1, A, C, B);
    move(n, A, B);
    hanoi(n-1, C, B, A);
  }
}
```

主教材中所述非递归算法的目的塔座不确定。当 n 为奇数时，目的塔座是 B，按顺时针方向移动；而当 n 为偶数时，目的塔座为 C，按反时针方向移动。为讨论方便，规定目的塔座为 B。Hanoi 塔问题的非递归算法可描述如下。

```
void hanoi(int n) {
  int  top[3]={0, 0, 0};
  int  **tower;
  Make2DArray(tower, n+1, 3);
  int  b, bb, x, y, min = 0;
  for(int i=0; i <= n; i++) {
    tower[i][0] = n-i+1;
    tower[i][1] = n+1;
    tower[i][2] = n+1;
  }
  top[0] = n;
  b = odd(n);
  bb = 1;
  while(top[1] < n) {
    if(bb) {
      x = min;
      if(b)
        y = (x+1) % 3;
      else
        y = (x+2) % 3;
      min=y;
      bb=0;
    }
    else {
      x = (min+1) % 3;
```

```
        y = (min+2) % 3;
        bb = 1;
        if(tower[top[x]][x] > tower[top[y]][y])
          Swap(x, y);
      }
      move(tower[top[x]][x], x+1, y+1);
      tower[top[y]+1][y] = tower[top[x]][x];
      top[x]--;
      top[y]++;
    }
}
```

下面用数学归纳法证明递归算法和非递归算法产生相同的移动序列。

当 n=1 和 n=2 时容易直接验证。设当 $k \leqslant n-1$ 时，递归算法和非递归算法产生完全相同的移动序列。考察 $k=n$ 的情形。

将移动分为顺时针移动（CW）、逆时针移动（CC）和非最小圆盘塔座间的移动（O）三种情况。

当 n 为奇数时，顺时针非递归算法产生的移动序列为 CW, O, CW, O, …, CW；逆时针非递归算法产生的移动序列为 CC, O, CC, O, …, CC。

当 n 为偶数时，顺时针非递归算法产生的移动序列为 CC, O, CC, O, …, CC；逆时针非递归算法产生的移动序列为 CW, O, C, O, …, CW。

（1）当 n 为奇数时，顺时针递归算法 hanoi(n, A, B, C)产生的移动序列为：hanoi($n-1$, A, C, B)产生的移动序列，O，hanoi($n-1$, C, B, A)产生的移动序列

其中，hanoi($n-1$, A, C, B)和 hanoi($n-1$, C, B, A)均为偶数圆盘逆时针移动问题。由数学归纳法知，它们产生的移动序列均为 CW, O, C, O, …, CW。因此，hanoi(n, A, B, C)产生的移动序列为 CW, O, CW, O, …, CW。

（2）当 n 为偶数时，顺时针递归算法 hanoi(n, A, B, C)产生的移动序列为

hanoi($n-1$, A, C, B)产生的移动序列，O，hanoi($n-1$, C, B, A)产生的移动序列

其中，hanoi($n-1$, A, C, B)和 hanoi($n-1$, C, B, A)均为奇数圆盘逆时针移动问题。由数学归纳法知，它们产生的移动序列均为 CC, O, CC, O, …, CC。因此，hanoi(n, A, B, C)产生的移动序列为 CC, O, CC, O, …, CC。

当 n 为奇数和偶数时的逆时针递归算法也类似。

由数学归纳法可知，递归算法和非递归算法产生相同的移动序列。

2-2　判断这 7 个算法的正确性。

下面的 7 个算法与本章中的二分搜索算法 BinarySearch 略有不同。如果算法不正确，请说明产生错误的原因。如果算法正确，请给出算法的正确性证明。

（1）

```
template<class Type>
int BinarySearch1(Type a[], const Type& x, int n) {
  int  left = 0;
  int  right = n-1;
  while (left <= right) {
    int  middle = (left+right)/2;
```

```
    if (x == a[middle])
      return middle;
    if (x > a[middle])
      left = middle;
    else
      right = middle;
  }
  return -1;
}
```

(2)

```
template<class Type>
int BinarySearch2(Type a[], const Type& x, int n) {
  int  left = 0;
  int  right = n-1;
  while (left < right-1) {
    int  middle = (left+right)/2;
    if (x<a[middle])
      right = middle;
    else
      left = middle;
  }
  if (x == a[left])
    return left;
  else
    return -1;
}
```

(3)

```
template<class Type>
int BinarySearch3(Type a[], const Type& x, int n) {
  int  left = 0;
  int  right = n-1;
  while (left+1 != right) {
    int  middle = (left+right)/2;
    if (x >= a[middle])
      left = middle;
    else
      right = middle;
  }
  if (x == a[left])
    return left;
  else
    return -1;
}
```

(4)

```
template<class Type>
int BinarySearch4(Type a[], const Type& x, int n) {
```

```
  if (n > 0 && x >= a[0]) {
    int  left = 0;
    int  right = n-1;
    while (left < right) {
      int  middle = (left+right)/2;
      if (x < a[middle])
        right = middle-1;
      else
        left = middle;
    }
    if (x == a[left])
      return left;
  }
  return -1;
}
```

(5)

```
template<class Type>
int BinarySearch5(Type a[], const Type& x, int n) {
  if (n > 0 && x >= a[0]) {
    int  left = 0;
    int  right = n-1;
    while (left < right) {
      int  middle = (left+right+1)/2;
      if (x < a[middle])
        right = middle-1;
      else
        left = middle ;
    }
    if (x == a[left])
      return left;
  }
  return -1;
}
```

(6)

```
template<class Type>
int BinarySearch6(Type a[], const Type& x, int n) {
  if (n > 0 && x >= a[0]) {
    int  left = 0;
    int  right = n-1;
    while (left < right) {
      int  middle = (left+right+1)/2;
      if (x < a[middle])
        right = middle-1;
      else
        left = middle+1 ;
    }
    if (x == a[left])
```

```
        return left;
    }
    return -1;
}
```

（7）

```
template<class Type>
int BinarySearch7(Type a[], const Type& x, int n) {
    if (n > 0 && x >= a[0]) {
        int  left = 0;
        int  right = n-1;
        while (left < right) {
            int  middle = (left+right+1)/2;
            if (x < a[middle])
                right = middle;
            else
                left = middle ;
        }
        if (x == a[left])
            return left;
    }
    return -1;
}
```

分析与解答：

（1）与主教材中的算法 BinarySearch 相比，数组段左、右游标 left 和 right 的调整不正确，导致陷入死循环。

（2）与主教材中的算法 BinarySearch 相比，数组段左、右游标 left 和 right 的调整不正确，导致当 x=a[n−1]时返回错误。

（3）与正确算法 BinarySearch5 相比，数组段左、右游标 left 和 right 的调整不正确，导致当 x=a[n−1]时返回错误。

（4）与正确算法 BinarySearch5 相比，数组段左、右游标 left 和 right 的调整不正确，导致陷入死循环。

（5）算法正确，且当数组中有重复元素时，返回满足条件的最右元素。

（6）与正确算法 BinarySearch5 相比，数组段左、右游标 left 和 right 的调整不正确，导致当 x=a[n−1]时返回错误。

（7）与正确算法 BinarySearch5 相比，数组段左、右游标 left 和 right 的调整不正确，导致当 x=a[0]时陷入死循环。

2-3 改写二分搜索算法。

请改写二分搜索算法，使得当搜索元素 x 不在数组中时，返回小于 x 的最大元素位置 i 和大于 x 的最小元素位置 j。设 a[0:n−1]是已排好序的数组。当搜索元素在数组中时，i 和 j 相同，均为 x 在数组中的位置。

分析与解答：改写二分搜索算法如下。

```
template<class T>
int binarySearch(T a[], const T&x, int left, int right, int &i, int &j) {
    int  middle;
```

```
    while (left <= right) {
      middle = (left +right)/2;
      if (x ==a[middle]) {
        i = j = middle;
        return 1;
      }
      if (x > a[middle])
        left = middle+1;
      else
        right = middle-1;
    }
    i = right;
    j = left;
    return 0;
}
```

2-4 大整数乘法的 $O(nm^{\log(3/2)})$算法。

给定两个大整数 u 和 v，它们分别有 m 和 n 位数字，且 $m \leqslant n$。用通常的乘法求 uv 的值需要 $O(mn)$时间。可以将 u 和 v 均看作有 n 位数字的大整数，用本章介绍的分治法，在 $O(n^{\log 3})$时间内计算 uv 的值。当 m 比 n 小得多时，用这种方法就显得效率不够高。试设计一个算法，在上述情况下用 $O(nm^{\log(3/2)})$时间求出 uv 的值。

分析与解答： 当 m 比 n 小得多时，将 v 分成 n/m 段，每段 m 位。计算 uv 需要 n/m 次 m 位乘法运算。每次 m 位乘法可以用主教材中的分治法计算，耗时 $O(m^{\log 3})$。因此，算法所需的计算时间为 $O((n/m)m^{\log 3})=O(nm^{\log(3/2)})$。

2-5 5 次 $n/3$ 位整数的乘法。

在用分治法求两个 n 位大整数 u 和 v 的乘积时，将 u 和 v 都分割为长度为 $n/3$ 位的 3 段。证明可以用 5 次 $n/3$ 位整数的乘法求得 uv 的值。按此思想设计一个求两个大整数乘积的分治算法，并分析算法的计算复杂性（提示：n 位的大整数除以一个常数 k 可以在 $\theta(n)$时间内完成。符号 θ 所隐含的常数可能依赖于 k）。

分析与解答： 这个问题有更一般的解。将两个 n 位大整数 u 和 v 都分割为长度为 n/m 位的 m 段，可以用 $2m-1$ 次 n/m 位整数的乘法求得 uv 的值。

事实上，设 $x=2^{n/m}$，可以将 u 和 v 及其乘积 $w=uv$ 表示为

$$u=u_0+u_1x+\cdots+u_{m-1}x^{m-1}$$

$$v=v_0+v_1x+\cdots+v_{m-1}x^{m-1}$$

$$w=uv=w_0+w_1x+w_2x^2+\cdots+w_{2m-2}x^{2m-2}$$

将 u、v 和 w 都看作关于变量 x 的多项式，并取 $2m-1$ 个不同的数 $x_1, x_2, \cdots, x_{2m-1}$ 代入多项式，可得

$$u(x_i)=u_0+u_1x_i+\cdots+u_{m-1}x_i^{m-1}$$

$$v(x_i)=v_0+v_1x_i+\cdots+v_{m-1}x_i^{m-1}$$

$$w(x_i)=u(x_i)v(x_i)=w_0+w_1x_i+w_2x_i^2+\cdots+w_{2m-2}x_i^{2m-2}$$

用矩阵形式表示为

$$\begin{bmatrix} w(x_1) \\ w(x_2) \\ \vdots \\ w(x_{2m-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{2m-2} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{2m-2} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{2m-1} & x_{2m-1}^2 & \cdots & x_{2m-1}^{2m-2} \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{2m-2} \end{bmatrix}$$

设

$$\boldsymbol{B} = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{2m-2} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{2m-2} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{2m-1} & x_{2m-1}^2 & \cdots & x_{2m-1}^{2m-2} \end{bmatrix}$$

则

$$\begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{2m-2} \end{bmatrix} = \boldsymbol{B}^{-1} \begin{bmatrix} w(x_1) \\ w(x_2) \\ \vdots \\ w(x_{2m-1}) \end{bmatrix}$$

式中，$w(x_i)=u(x_i)v(x_i)$是 2 个 n/m 位数的乘法运算，共有 $2m-1$ 个乘法。其他均为加减法或数乘运算。

下面用 $m=3$ 的具体例子来说明。设 $x=2^{n/3}$，可以将 u 和 v 及其乘积 $w=uv$ 表示为

$$u=u_0+u_1x+u_2x^2$$

$$v=v_0+v_1x+v_2x^2$$

$$w=uv=w_0+w_1x+w_2x^2+w_3x^3+w_4x^4$$

取 5 个数 $x_1, x_2, \cdots, x_5$ 如下：$x_1=0$，$x_2=-2$，$x_3=2$，$x_4=-1$，$x_5=1$，代入多项式得

$$a=w(x_1)=u_0v_0$$

$$b=w(x_2)=(u_0-2u_1+4u_2)(v_0-2v_1+4v_2)$$

$$c=w(x_3)=(u_0+2u_1+4u_2)(v_0+2v_1+4v_2)$$

$$d=w(x_4)=(u_0-u_1+u_2)(v_0-v_1+v_2)$$

$$e=w(x_5)=(u_0+u_1+u_2)(v_0+v_1+v_2)$$

$$\begin{bmatrix} a \\ b \\ c \\ d \\ e \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & -2 & 4 & -8 & 16 \\ 1 & 2 & 4 & 8 & 16 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix}$$

$$\begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & -2 & 4 & -8 & 16 \\ 1 & 2 & 4 & 8 & 16 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}^{-1} \begin{bmatrix} a \\ b \\ c \\ d \\ e \end{bmatrix}$$

解得

$$w_0 = a$$

$$w_1 = \frac{b-c-8(d-e)}{12}$$

$$w_2 = \frac{-b-c+16(d+e)-30a}{24}$$

$$w_3 = \frac{-b+c+2(d-e)}{12}$$

$$w_4 = \frac{b+c-4(d+e)+6a}{24}$$

由 $x_1, x_2, \cdots, x_5$ 的不同取法，可得到不同的分解方法。

按此分解设计的求两个 n 位大整数乘积的分治算法需要 5 次 $n/3$ 位整数乘法。分割及合并步所需的加减法和数乘运算时间为 $O(n)$。设 $T(n)$是算法所需的计算时间，则

$$T(n) = \begin{cases} O(1) & n=1 \\ 5T(n/3)+O(n) & n>1 \end{cases}$$

由此可得 $T(n)=O(n\log_3 5)$。

在一般情况下，将两个 n 位大整数 u 和 v 都分割为长度为 n/m 位的 m 段，可以用 $2m-1$ 次 n/m 位整数的乘法求得 uv 的值。由此设计出的求两个 n 位大整数乘积的分治算法需要 $2m-1$ 次 n/m 位整数乘法。分割及合并步所需的加减法和数乘运算时间为 $O(n)$。因此其计算时间 $T(n)$满足：

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

解此递归式可得 $T(n)=O(n\log_m(2m-1))$。

2-6 矩阵乘法。

对任何非零偶数 n，总可以找到奇数 m 和正整数 k，使得 $n=m2^k$。为了求出两个 n 阶矩阵的乘积，可以把一个 n 阶矩阵分成 $m\times m$ 个子矩阵，每个子矩阵有 $2^k\times 2^k$ 个元素。当需要求 $2^k\times 2^k$ 的子矩阵的积时，使用 Strassen 算法。设计一个传统方法与 Strassen 算法相结合的矩阵相乘算法，对任何偶数 n，都可以求出两个 n 阶矩阵的乘积。并分析算法的计算时间复杂性。

分析与解答：将 n 阶矩阵分块为 $m\times m$ 的矩阵。用传统方法求两个 m 阶矩阵的乘积需要计算 $O(m^3)$次 2 个 $2^k\times 2^k$ 矩阵的乘积。用 Strassen 矩阵乘法计算两个 $2^k\times 2^k$ 矩阵的乘积需要的计算时间为 $O(7^k)$，因此算法的计算时间为 $O(7^k m^3)$。

2-7 多项式乘积。

设 $P(x)=a_0+a_1x+\cdots+a_dx^d$ 是一个 d 次多项式。假设已有一算法能在 $O(i)$时间内计算一个 i 次多项式与一个一次多项式的乘积，以及一个算法能在 $O(i\log i)$时间内计算两个 i 次多项式的乘积。对于任意给定的 d 个整数 $n_1, n_2, \cdots, n_d$，用分治法设计一个有效算法，计算出满足 $P(n_1)=P(n_2)=\cdots=P(n_d)=0$ 且最高次项系数为 1 的 d 次多项式 $P(x)$，并分析算法的效率。

分析与解答：

$$P(x)=\prod_{i=1}^{d}(x-n_i)=\prod_{i=1}^{\lfloor d/2\rfloor}(x-n_i)\prod_{i=\lfloor d/2\rfloor+1}^{d}(x-n_i)=P_1(x)P_2(x)$$

用分治法将 d 次多项式转化为两个 $d/2$ 次多项式的乘积。

设用此算法计算 d 次多项式所需计算时间为 $T(d)$，则 $T(d)$满足如下递归式：

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

解此递归式可得 $T(d)=O(d\log^2 d)$。

2-8 $O(1)$空间子数组换位算法。

设 a[0:n−1]是有 n 个元素的数组，k（0≤k≤n−1）是一个非负整数。试设计一个算法将子数组 a[0:k−1]与 a[k:n−1]换位。要求：算法在最坏情况下耗时 $O(n)$，且只用到 $O(1)$的辅助空间。

分析与解答：

算法 1：循环换位算法

（1）向前循环换位

```
template<class T>
void forward(T a[], int n, int k) {
   for(int i=0; i < k; i++) {
      T  tmp = a[0];
      for(int j=1; j < n; j++)
         a[j-1] = a[j];
      a[n-1] = tmp;
   }
}
```

（2）向后循环换位

```
template<class T>
void backward(T a[], int n, int k) {
   for(int i=k; i < n; i++) {
      T tmp = a[n-1];
      for(int j=n-1; j > 0; j--)
         a[j] = a[j-1];
      a[0] = tmp;
   }
}
```

（3）选择较小的数组块进行循环

```
template<class T>
void exch0(T a[], int n, int k)
{   if(k>n-k) backward(a,n,k);
    else forward(a,n,k);
}
```

在最坏情况下，算法所需的元素移动次数为 $\min\{k, n-k\}\times(n+1)$。算法仅用到一个辅助单元 tmp，因此算法只用到 $O(1)$的辅助空间。当 $k=n/2$ 时，计算时间非线性。

算法 2：3 次反转算法

将数组块 a[i:j]反转的算法如下。

```
template<class T>
void reverse(T a[], int i, int j) {
```

```
  while(i < j) {
    Swap(a[i], a[j]);
    i++;
    j--;
  }
}
```

设 a[0:k−1]为 U，a[k:n−1]为 V，换位算法要求将 UV 变换为 VU。3 次反转算法先将 U 反转为 U^{-1}，再将 V 反转为 V^{-1}，最后将 $U^{-1}V^{-1}$ 反转为 VU。

```
template<class T>
void exch1(T a[], int n, int k) {
  reverse(a, 0, k-1);
  reverse(a, k, n-1);
  reverse(a, 0, n-1);
}
```

3 次反转算法用了$\lfloor k/2 \rfloor+\lfloor (n-k)/2 \rfloor+\lfloor n/2 \rfloor \leqslant n$次数组单元交换运算。每个数组单元交换运算需要 3 次元素移动。因此在最坏情况下，3 次反转算法用了 $3n$ 次元素移动。算法显然只用到 $O(1)$的辅助空间。

算法 3：排列循环算法

向后循环换位算法实际上执行了数组元素的一个重新排列，因此向后循环换位对应 n 个元素的一个置换。这类置换具有如下特殊性质。

【循环置换分解定理】 对于给定数组 a[0:n−1]向后循环换位 n−k 位运算，可分解为恰好 gcd(k, n−k)个循环置换，且 0，…，gcd(k, n−k)−1 中的每个数恰属于一个循环置换。其中，gcd(x, y)表示整数 x 和 y 的最大公因数。

基于循环置换分解定理可设计下面的数组块换位算法。

```
template<class T>
void exch(T a[], int n, int k) {
  for(int i=0, cyc=gcd(k, nk); i < cyc; i++) {
    T  tmp = a[i];
    int  p=i, j = (k+i) % n;
    while(j != i) {
      a[p] = a[j];
      p = j;
      j = (k+p) % n;
    }
    a[p] = tmp;
  }
}
```

上述算法总共移动元素 n+gcd(k, n−k)次，算法显然只用到 $O(1)$的辅助空间。

当换位数组块的长度相等时，算法更简单。例如，将数组块 a[b:b+l−1]和 a[c:c+l−1]换位的算法可表述如下。

```
template<class T>
void eqexch(T a[], int b, int c, int l) {
  // equal size block exchange a[b:b+l-1] and a[c:c+l-1].
```

```
    for(int i=0; i < l; i++)
      Swap(a[b+i], a[c+i]);
}
```

上述算法总共移动元素 3×1 次。

2-9 $O(1)$空间合并算法。

设子数组 a[0:k−1]和 $a[k:n-1]$已排好序（$0 \leqslant k \leqslant n-1$）。试设计一个合并这两个子数组为排好序的数组 a[0:n−1]的算法。要求算法在最坏情况下所用的计算时间为 $O(n)$，且只用到 $O(1)$的辅助空间。

分析与解答：

算法 1：循环换位合并算法

（1）向右循环换位合并

向右循环换位合并算法首先用二分搜索算法在数组段 a[k:n−1]中搜索 a[0]的插入位置，即找到位置 p 使得 a[p]<a[0]≤a[p+1]。数组段 a[0:p]向右循环换位 $p-k+1$ 个位置，使 a[k−1]移动到 a[p]的位置。此时，原数组元素 a[0]及其左边的所有元素均已排好序。对剩余的数组元素重复上述过程，直至只剩下一个数组段，此时整个数组已排好序。

向右循环换位合并算法可描述如下。

```
template<class T>
void mergefor(T a[], int k, int n) {
  // 合并a[0:k-1]和c[k:n-1].
  int  i = 0, j = k;
  while(i<j && j<n){
    int  p = binarySearch(a, a[i], j, n-1);
    shiftright(a, i, p, p-j+1);
    j = p+1;
    i += p-j+2;
  }
}
```

其中，算法 binarySearch(a, x, left, right)用于在数组段 a[left:right]中搜索元素 x 的插入位置。

```
template<class T>
int binarySearch(T a[], const T& x, int left,int right) {
  int  middle;
  while (left <= right) {
    middle = (left +right)/2;
    if (x == a[middle])
      return middle;
    if (x >a[middle])
      left = middle+1;
    else
      right = middle-1;
  }
  if (x >a[middle])
    return middle;
```

```
  else
    return middle-1;
}
```

算法 shiftright(a, s, t, k)用于将数组段 a[s:t]中元素循环右移位 k 个位置。

```
template<class T>
void shiftright(T a[], int s, int t, int k) {
  for(int i=0; i < k; i++) {
    T  tmp = a[t];
    for(int j=t; j > s; j--)
      a[j] = a[j-1];
    a[s] = tmp;
  }
}
```

上述算法中，数组段 a[0:k−1]中元素的移动次数不超过 k 次，数组段 a[k:n−1]中的元素最多移动 1 次。因此，算法的元素移动总次数不超过 $k^2+(n-k)$次。算法的元素比较次数不超过 klog(n−k)次。当 $k<\sqrt{n}$ 时，算法的计算时间为 $O(n)$。而当 $k=O(n)$时，算法的计算时间为 $O(n^2)$。由于数组段循环右移位算法只用了 $O(1)$的辅助空间，所以整个算法所用的辅助空间为 $O(1)$。

（2）类似地，可以设计向左循环换位合并算法 mergeback(T a[], int k, int n)。

算法 2：内部缓存算法

（1）算法思想概述

为便于叙述，假定 n 是一个完全平方数，稍后讨论一般情况。图 2-1(a)是当 n=25 和 k=12 时的一个示例，用大写字母表示数组元素的键值，其下标表示相同键值出现的次序。

首先，将待合并数组划分为 $\sqrt{n}$ 个数组块，每块的大小为 $\sqrt{n}$。将数组中最大的 $\sqrt{n}$ 个元素置于数组的最左端，作为算法的内部缓存，如图 2-1(b)所示。接下来用选择排序算法对除了内部缓存外的 $\sqrt{n}-1$ 个数组块排序，使数组块的最右端元素按非减序排列，每个数组块内部元素的相对次序保持不变。这个排序过程需要用 $O(n)$次元素比较和 $O(n)$次元素移动。用算法分析题 2-8 中的算法 eqexch 可保证上述排序过程只用 $O(1)$的辅助空间。数组块排序后的结果如图 2-1(c)所示。

$B_1B_2B_3D_1D_2E_1E_2F_1G_1H_1H_2J_1$	$A_1A_2A_3B_4B_5C_1C_2E_3G_2G_3H_3I_1I_2$
0 $k-1$	k $n-1$

（a）给定的待合并数组

$H_2H_3I_1I_2J_1$	$B_1B_2B_3D_1D_2$	$E_1E_2F_1G_1H_1$	$A_1A_2A_3B_4B_5$	$C_1C_2E_3G_2G_3$
内部缓存	数组段 1		数组段 2	

（b）抽取内部缓存

$H_2H_3I_1I_2J_1$	$A_1A_2A_3B_4B_5$	$B_1B_2B_3D_1D_2$	$C_1C_2E_3G_2G_3$	$E_1E_2F_1G_1H_1$
内部缓存	数组块 2	数组块 3	数组块 4	数组块 5

（c）数组块排序

图 2-1　数组块重排结果

接下来，确定待合并的数组块序列。第 1 个序列是从数组块 2 开始的最长的非减数组块序列。第 2 个序列是接着的那个数组块，如图 2-2(a)所示。

现在可以用内部缓存对两个序列进行合并。每次比较两个序列的最小元素，将较小者与内部缓存的最左端元素交换位置。合并过程中，内部缓存可能被分为两个不连续的段，如图 2-2(b)所示。第 1 个序列的最后一个元素进入正确位置后，完成这一轮序列合并动作，此时内部缓存形成一个连续的块，且这两个序列中至少还有 1 个元素，如图 2-2(c)所示。

$H_2H_3I_1I_2J_1$	$A_1A_2A_3B_4B_5$	$B_1B_2B_3D_1D_2$	$C_1C_2E_3G_2G_3$	$E_1E_2F_1G_1H_1$
内部缓存	待合并序列 1		待合并序列 2	数组块 5

(a) 确定待合并序列

$A_1A_2A_3B_4B_5B_1B_2B_3C_1C_2$	$H_2H_3I_1$	D_1D_2	I_2J_1	$E_3G_2G_3$	$E_1E_2F_1G_1H_1$
已合并序列	缓存		缓存		数组块 5

(b) 内部缓存合并过程

$A_1A_2A_3B_4B_5B_1B_2B_3C_1C_2D_1D_2$	$I_1H_2H_3I_2J_1$	$E_3G_2G_3$	$E_1E_2F_1G_1H_1$
已合并序列	内部缓存		数组块 5

(c) 完成序列合并

图 2-2　合并两个序列

下一轮的序列合并是类似的。第 1 个序列是从内部缓存右端的下一个元素开始的最长的非减数组块序列。第 2 个序列是接着的那个数组块，如图 2-3(a)所示。继续用内部缓存合并这两个序列，直至第 1 个序列完成合并，如图 2-3(b)所示。上述过程一直进行到只剩下一个序列时为止。此时只要将剩下的这个序列左移，内部缓存成为最后的一个数组块，如图 2-3(c)所示。此时内部缓存左边的所有元素均已排好序，且内部缓存中的元素是整个数组中最大的 $\sqrt{n}$ 个元素，用选择排序算法对内部缓存中元素排序，完成整个合并过程，如图 2-3(d)所示。

$A_1A_2A_3B_4B_5B_1B_2B_3C_1C_2D_1D_2$	$I_1H_2H_3I_2J_1$	E_3 G_2 G_3	$E_1E_2F_1C_1H_1$
已合并序列	内部缓存	待合并序列 1	待合并序列 2

(a) 确定下一对待合并序列

$A_1A_2A_3B_4B_5B_1B_2B_3C_1C_2D_1D_2E_3E_1E_2F_1G_2G_3$	$J_1I_1H_2H_3I_2$	G_1 H_1
	内部缓存	

(b) 完成序列合并

$A_1A_2A_3B_4B_5B_1B_2B_3C_1C_2D_1D_2E_3E_1E_2F_1G_2G_3G_1$	$H_1H_2H_3I_2J_1I_1$
	内部缓存

(c) 移位完成单序列合并

$A_1A_2A_3B_4B_5B_1B_2B_3C_1C_2D_1D_2E_3E_1E_2F_1G_2G_3G_1H_1H_2H_3I_2I_1J_1$

(d) 对内部缓存排序，完成整个数组合并

图 2-3　整个数组的合并过程

由于合并过程只用到内部缓存，上述整个合并过程只用 $O(1)$的辅助空间。数组块排序、

序列合并以及内部缓存排序显然只需要 $O(n)$计算时间。因此，内部缓存算法需要 $O(n)$计算时间和 $O(1)$的辅助空间。

（2）一般情况

在一般情况下，可以用 $O(\sqrt{n})$时间将问题转换为前面讨论的特殊情况。

当待合并的两个数组段中有一个数组段的长度小于 $\sqrt{n}$ 时，可以用前面讨论过的循环换位合并算法在 $O(n)$计算时间内，用 $O(1)$的辅助空间完成合并。

接下来的讨论中，设 $s=\lfloor\sqrt{n}\rfloor$，且 $\min\{k, n-k\}>\sqrt{n}$。数组中大小为 s 的块记为 s-block。由于$(s+1)^2>n$，数组中 s-block 的个数不超过 $s+2$。

首先，找出数组中组成内部缓存的 s 个最大元素。一般情况下，这 s 个元素由数组中大小分别为 s_1 和 s_2 的 2 部分 A 和 B 组成，$s_1+s_2=s$。紧挨着 A 的左边的 s_2 个元素记作 C。D 是紧挨着 B 的左边的数组块，其大小是使数组块 2 剩余元素为 s 的倍数的最小值。按此划分，可以将数组看作由 s-block 组成的数组块。除了第 1 块的大小 t_1 和最后一块的大小 t_2 外，每个数组块 s-block 的大小均为 s，而且 $0<t_1\leq s$，$0<t_2<2s$，如图 2-4(a)所示。

（a）确定数组段 A、B、C、D

（b）处理最右端数组块

（c）确定数组块 F 和 G

（d）合并数组块 F 和 G

（e）处理最左端数组块

（f）进入主算法

图 2-4　对一般情况的处理

然后，交换数组块 C 和 B，使数组块 B 和 A 构成内部缓存。然后用内部缓存合并数组块 D 和 C，构成排好序的数组块 E，如图 2-4(b)所示。

当 $t_1<s$ 时，需要对最左端的 t_1-block 进行特殊处理。设 F 是这个特殊的数组块，G 是紧挨着内部缓存的 s-block，如图 2-4(c)所示。用内部缓存的最右 t_1 个单元将 F 和 G 合并得到 H 和 I，如图 2-4(d)所示。将 H 与最左端的内部缓存 t_1-block 交换，使 H 已在最终位置，且内部缓存连成一体，如图 2-4(e)所示。最后，将内部缓存与第 1 个 s-block 换位，转化为前面讨

论过的规则情形，如图 2-4(f)所示。

以上这些处理只需要 $O(s)=O(\sqrt{n})$的计算时间和 $O(1)$的辅助空间。

（3）算法实现

上面讨论的算法可以分为 2 个阶段和 4 个子任务分别实现如下。

当 $k<\sqrt{n}$ 或 $n-k<\sqrt{n}$ 时，可以用前面讨论过的循环换位合并算法 mergefor 和 mergeback 在 $O(n)$计算时间内，用 $O(1)$的辅助空间完成合并。

```
template<class T>
void merge(T a[], int k, int n) {
   // Merge a[0:k-1] and c[k:n-1]
   int  s = (int) sqrt(n);
   if(k<s) {
      mergefor(a, k, n);
      return;
   }
   if(n-k < s) {
      mergeback(a, k, n);
      return;
   }
   int  j = task1(a, k, n, s);
   eqexch(a, k-s, j, n-j);
   int  bfs = k-s, bft = k-1;
   int  ds = j-(j-k)%s, dt = j-1;
   task2(a, n, ds, dt);
   task3(a, k, n, s, bfs, bft);
   maintask(a, k, n, s, bfs, ds);
}
```

算法的第 1 阶段是初始化阶段，分别由 task1，task2 和 task3 子任务来完成。第 2 阶段进入主算法，由子任务 maintask 来完成。下面分别讨论。

算法 task1 完成抽取内部缓存的任务，对应图 2-4（a），其返回值是数组块 B 的最左元素的下标。

```
template<class T>
int task1(T a[], int k, int n, int s) {
   int i = k-1, j = n-1;
   for(int t=0; t < s; t++) {
      if(a[i] < a[j])
         j--;
      else
         i--;
   }
   return j+1;
}
```

算法 task2 完成对最右数组块 E 的排序，对应图 2-4(b)。

```
template<class T>
void task2(T a[], int n, int ds, int dt) {
   if(ds > dt)
```

```
    return;
  selectionSort(a, ds, n-1);
}
```

算法 task3 完成对最左数组块的排序，对应图 2-4(c)～(f)。

```
template<class T>
void task3(T a[], int k, int n, int s, int &bfs, int &bft) {
  int  t1 = k%s;
  bfs = bft-t1+1;
  bfmerge(a ,bfs, bft, 0, t1-1, bft+1, bft+s);
  eqexch(a, 0, bft-t1+1, t1);
  eqexch(a, t1, bft-s+1, s);
  bfs = t1;
  bft = bfs+s-1;
}
```

其中，用到的数组块交换算法 eqexch 在算法分析题 2-8 中已有描述。用内部缓存进行合并的算法 bfmerge 表述如下。

```
template<class T>
void bfmerge(T a[], int &bs, int bt, int s1, int t1, int s2, int t2) {
  // buffer Merge
  int  ps1 = s1;
  while(s1 <= t1) {
    if(s2 <= t2 && a[s1] > a[s2])
      Swap(a[s2++], a[bs++]);
    else
      Swap(a[s1++], a[bs++]);
    if(bs == s2)
      bs = ps1;
  }
}
```

maintask 是主算法，完成最后的合并任务。

```
template<class T>
void maintask(T a[], int k, int n, int s, int bfs, int ds) {
  sortblock(a, bfs+s, ds-1, s);
  while(bfs < n-s){
    int  s1 = bfs+s, t1 = s1+(ds-bfs-1)%s;
    while(t1 < ds && a[t1] <= a[t1+1])
      t1 += s;
    if(t1 > n-1)
      t1 = n-1;
    int  s2 = t1+1, t2 = t1+s;
    if(s2 > ds)
      t2 = n-1;
    bfmerge(a, bfs, bfs+s-1, s1, t1, s2, t2);
    if(s1>t1 && s2 <= t2) {
      eqexch(a, bfs, s2, t2-s2-1);
      break;
```

```
    }
  }
  selectionSort(a, bfs, n-1);
}
```

sortblock 是对数组块进行选择排序的算法。

```
template<class T>
int maxblock(T a[], int left, int r, int s) {
  int  pos = 1;
  for (int i =2; i <= r; i++)
    if (a[left+pos*s-1] < a[left+i*s-1])
      pos = i;
   return pos;
}
template<class T>
void sortblock(T a[], int left, int right, int s) {
  int m=(right-left+1)/s;
  for (int r=m; r > 1; r--) {
    int  j = maxblock(a, left, r, s);
    if(j < r)
      eqexch(a, left+(j-1)*s, left+(r-1)*s, s);
  }
}
```

2-10 $\sqrt{n}$段合并排序算法。

如果在合并排序算法的分割步骤中，将数组 $a[0:n-1]$划分为$\lfloor\sqrt{n}\rfloor$个子数组，每个子数组中有$O(\sqrt{n})$个元素，然后递归地对分割后的子数组进行排序，最后将所得到的$\lfloor\sqrt{n}\rfloor$个排好序的子数组合并成所要求的排好序的数组 a[0:$n-1$]。设计一个实现上述策略的合并排序算法，并分析算法的计算复杂性。

分析与解答：实现上述策略的合并排序算法如下。

```
template<class T>
void mergesort(T *a, int left, int right) {
  if(left < right) {
    int  j = (int) sqrt(right-left+1);
    if(j > 1){
      for(int i=0; i < j; i++)
        mergesort(a, left+i*j, left+(i+1)*j-1);
      mergesort(a, left+j*j, right);
    }
    mergeall(a, left, right);
  }
}
```

其中，算法 mergeall 合并$\sqrt{n}$个排好序的数组段。在最坏情况下，算法 mergeall 需要$O(n\log n)$时间。因此上述算法所需的计算时间 $T(n)$满足：

$$T(n)=\begin{cases}O(1) & n\leqslant 1\\ \sqrt{n}T(\sqrt{n}) & n>1\end{cases}$$

此递归式的解为 $T(n)=O(n\log n)$。

2-11 自然合并排序算法。

对所给元素存储于数组中和存储于链表中两种情形，写出自然合并排序算法。

分析与解答：对于所给元素存储于数组中的情形，自然合并排序算法如下。

```
template<class T>
void sort(T *a0, int m) {
   a = a0;
   n = m;
   b = new int[n];
   naturalmergesort();
}
void naturalmergesort(){
   while (!mergeruns(a, b) &!mergeruns(b, a))   ;
}
```

由 mergeruns 实际完成自然合并排序算法。

```
template<class T>
bool mergeruns(T *a, T *b) {
   int  i = 0, k = 0;
   bool  asc = true;
   T  x;
   while (i<n) {
      k = i;
      do x=a[i++];
      while (i<n && x<=a[i])  ;
      while (i<n && x>=a[i])
         x=a[i++];
      merge (a, b, k, i-1, asc);
      asc = !asc;
   }
   return k == 0;
}
template<class T>
void merge(T *a, T *b, int lo, int hi, bool asc) {
   int  k=asc ? lo : hi;
   int  c=asc ? 1 :-1;
   int  i = lo, j = hi;
   while(i <= j) {
      if (a[i] <= a[j])
         b[k] = a[i++];
      else
         b[k] = a[j--];
      k += c;
   }
}
```

对于所给元素存储于链表中的情形，自然合并排序算法如下。

链表结构如下：

```
typedef struct node *link;
struct node {
   Item item;
   link next;
};
link mergesort(link t) {
   link  a, b;
   QUEUE<link> Q;
   if (t == 0 || t->next == 0)
      return t;
   for (link u=t, v; t != 0; t = u) {
      while(u && u->next && u->item <= u->next->item)
         u = u->next;
      v = u;
      u = u->next;
      v->next = 0;
      Q.ENQUEUE(t);
   }
   Q.DEQUEUE(t);
   while (!Q.EMPTY()){
      Q.ENQUEUE(t);
      Q.DEQUEUE(a);
      Q.DEQUEUE(b);
      t =merge(a, b);
   }
   return t;
}
```

算法 merge 实现已排序链表的合并。

```
link merge(link a, link b) {
   link  c, head;
   c = head = new node;
   while ((a != 0) && (b != 0)) {
      if (a->item < b->item) {
         c->next = a;
         c = a;
         a = a->next;
      }
      else {
         c->next = b;
         c = b;
         b = b->next;
      }
   }
   c->next = (a == 0) ? b : a;
   return head->next;
}
```

2-12　第 k 小元素问题的计算时间下界。

试证明，在最坏情况下，求 n 个元素组成的集合 S 中的第 k 小元素至少需要 $n+\min(k, n-k+1)-2$ 次比较。

分析与解答：由于要建立下界，不失一般性，可设 S 中的 n 个元素互不相同。首先注意到，要确定第 k 小元素 z，就要确定 S 中其他元素与 z 的关系。对于 S 中每个元素 x，必须确定 $x>z$ 或 $x<z$。换句话说，要建立 S 中元素与 z 的序关系树，如图 2-5 所示。

图 2-5 中的每个顶点表示一个元素，每条边表示一次比较。较高的元素的值也较大。如果元素 y 与第 k 小元素 z 的大小关系不确定，则对手可以改变 y 的值，使其从 z 的一侧移向另一侧，而不改变已做过比较的序关系，从而改变了 z 的第 k 小的地位，产生矛盾，如图 2-6 所示。

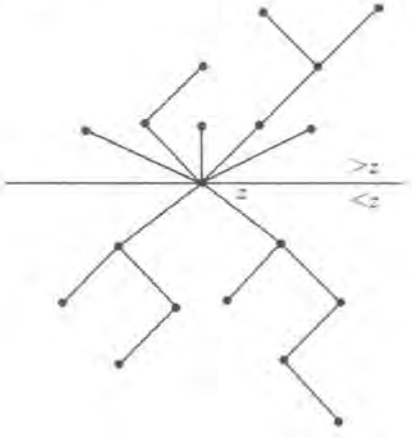

图 2-5 序关系树

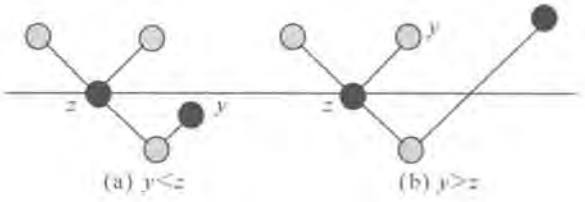

图 2-6 对手策略

由于关系树中有 n 个顶点，因此有 $n-1$ 条边，从而至少需要 $n-1$ 次比较。

下面进一步证明，采用对手论证方法，对手能迫使算法在得到所需的关系树中的 $n-1$ 次比较之前，做其他“无用”的比较。

当 $y\geq z$ 时，两个元素 x 和 y 之间的第 1 次比较 $x>y$ 对应于关系树中的一条边。同理，当 $y\leq z$ 时，两元素 x 和 y 之间的第 1 次比较 $x<y$ 也对应关系树中的一条边。这类比较称为关键比较。反之，当 $x>z$ 且 $y<z$ 时，两个元素 x 和 y 之间的比较是非关键比较。

表 2-1 对手策略

x	y	对手策略
N	N	x 取值大于 z，y 取值小于 z
N	L	x 取值小于 z
L	N	y 取值小于 z
N	S	x 取值大于 z
S	N	y 取值大于 z

下面的对手策略，将迫使算法做尽可能多的非关键比较。对手首先选定第 k 小元素 z 的值，集合中其他元素的值未定，仅在算法做与该元素有关的比较时确定该元素的值。在算法执行的任何阶段，集合中的元素有以下 3 种状态：L—该元素的值已确定，且大于 z；S—该元素的值已确定，且小于 z；N—该元素的值未确定。算法做两个元素 x 和 y 的比较时，对手根据元素 x 和 y 的状态，按照表 2-1 中的对手策略确定状态为 N 的元素的值。

对手策略中的每个比较均为非关键比较。每个这类比较最多产生一个元素状态 L，且最多产生一个元素状态 S。算法最终产生 $k-1$ 个状态为 S 的元素和 $n-k$ 个状态为 L 的元素。因此，上述对手策略至少迫使算法做了 $\min\{k-1, n-k\}$ 次非关键比较。因此，任何算法至少做了 $n-1+\min\{k-1, n-k-1\}$ 次比较。由此可见，在最坏情况下，第 k 小元素问题至少需要 $n+\min(k, n-k+1)-2$ 次比较。

当 n 是奇数时，集合 S 的中位数是第$(n+1)/2$ 小元素。由上述结论可知，$3n/2-3/2$ 是中位数问题的一个计算时间下界。

2-13 非增序快速排序算法。

如何修改 QuickSort 算法才能使其将输入元素按非增序排序？

分析与解答： 将算法 Partition 中的不等号反向即可。

```
template<class T>
int Partition (T a[], int p, int r) {
  int  i = p, j = r+1;
  T  x = a[p];
  while (true) {
    while(a[++i] > x && i < r)  ;                    // 将大于 x 的元素交换到左边区域
    while(a--[j] < x)  ;                             // 将小于 x 的元素交换到右边区域
    if(i >= j)
      break;
    Swap(a[i], a[j]);
  }
  Swap(a[j], a[p]);
  return j;
}
```

2-14 构造 Gray 码的分治算法。

Gray 码是一个长度为 2^n 的序列。序列中无相同元素，每个元素都是长度为 n 位的$(0, 1)$串，相邻元素恰好只有一位不同。用分治策略设计一个算法对任意的 n 构造相应的 Gray 码。

分析与解答： 考察 n=1, 2, 3 的简单情形，如表 2-2 所示。

表 2-2 Gray 码

长度(位)	(0, 1)串			
n=1	0	1		
n=2	00	01		
	11	10		
n=3	000	001	011	010
	110	111	101	100

设 n 位 Gray 码序列为 $G(n)$，$G(n)$以相反顺序排列的序列为 $G^{-1}(n)$。从上面的简单情形可以看出 $G(n)$的构造规律：$G(n+1)=0G(n)1G^{-1}(n)$。

注意到 $G(n)$的最后一个 n 位串与 $G^{-1}(n)$的第一个 n 位串相同，可用数学归纳法证明 $G(n)$的上述构造规律。由此规律容易设计构造 $G(n)$的分治法如下。

```
void gray(int n) {
  if (n == 1) {
    a[1] = 0;
    a[2] = 1;
    return;
  }
  gray(n-1);
  for(int k=1<<(n-1), i=k; i > 0; i--)
    a[2*k-i+1] = a[i]+k;
}
```

上述算法中将 n 位$(0, 1)$串看成二进制整数，第 i 个串存储在 a[i]中。

完成计算后，由 out 输出 Gray 码序列。

```
void out(int n) {
```

```
    char  str[32];
    int  m = 1<<n;
    for (int i=1; i <= m; i++) {
       _itoa(a[i], str, 2);
       int s = strlen(str);
       for(int j=0; j < n-s; j++)
          cout << "0";
       cout << str <<" ";
    }
    cout << endl;
}
```

2-15　网球循环赛日程表。

设有 n 个运动员要进行网球循环赛。设计一个满足以下要求的比赛日程表：

（1）每个选手必须与其他 $n-1$ 个选手各赛一次；

（2）每个选手一天只能赛一次；

（3）当 n 是偶数时，循环赛进行 $n-1$ 天。当 n 是奇数时，循环赛进行 n 天。

分析与解答：

（1）分治法

主教材中的分治法应描述如下：

```
void tourna(int n) {
   if (n==1) {
      a[1][1] = 1;
      return;
   }
   tourna(n/2);
   copy(n);
}
```

其中，算法 copy 将左上角递归计算出的小块中的所有数字按其相对位置抄到右下角，将右上角小块中的所有数字加 $n/2$ 后按其相对位置抄到左下角，将左下角小块中的所有数字按其相对位置抄到右上角，这样就完成了比赛日程表。

```
void copy(int n) {
   int  m=n/2;
   for(int i=1; i <= m; i++) {
      for(int j=1; j <= m; j++) {
         a[i][j+m] = a[i][j]+m;
         a[i+m][j] = a[i][j+m];
         a[i+m][j+m] = a[i][j];
      }
   }
}
```

对于一般的正整数 n，当 n 是奇数时，增设一个虚拟选手 $n+1$，将问题转换为 n 是偶数的情形。当选手与虚拟选手比赛时，表示轮空。因此只要关注 n 为偶数的情形即可。

当 $n/2$ 为偶数时，与 $n=2^k$ 的情形类似，可用分治法求解。

当 $n/2$ 为奇数时，递归返回的轮空的比赛要做进一步处理。其中一种处理方法是在前 $n/2$ 轮比赛中让轮空选手与下一个未参赛选手进行比赛。

一般情况下的分治法 tournament 可描述如下。

```
void tournament(int n) {
   if(n == 1) {
      a[1][1] = 1;
      return;
   }
   if(odd(n)){
      tournament(n+1);
      return;
   }
   tournament(n/2);
   makecopy(n);
}
bool odd(int n) {
   return n & 1;
}
```

其中，算法 makecopy 与算法 copy 类似，但要区分 $n/2$ 为奇数或偶数的情形。

```
void makecopy(int n) {
   if(n/2>1 && odd(n/2))
      copyodd(n);
   else
      copy(n);
}
```

算法 copyodd 实现 $n/2$ 为奇数时的复制。

```
void copyodd(int n) {
   int  m = n/2;
   for(int i=1; i <= m; i++) {
      b[i] = m+i;
      b[m+i] = b[i];
   }
   for(i=1; i <= m; i++) {
      for(int j=1; j <= m+1; j++) {
         if(a[i][j] > m) {
            a[i][j] = b[i];
            a[m+i][j] = (b[i]+m) % n;
         }
         else
            a[m+i][j] = a[i][j]+m;
      }
      for(j=2; j <= m; j++) {
         a[i][m+j] = b[i+j-1];
         a[b[i+j-1]][m+j] = i;
      }
   }
```

}

用上述算法计算出的 n=10 的比赛日程表如表 2-3 所示。

表 2-3　分治法 n=10 的比赛日程表

1	2	3	4	5	6	7	8	9	10
2	1	5	3	7	4	8	9	10	6
3	8	1	2	4	5	9	10	6	7
4	5	9	1	3	2	10	6	7	8
5	4	2	10	1	3	6	7	8	9
6	7	8	9	10	1	5	4	3	2
7	6	10	8	2	9	1	5	4	3
8	3	6	7	9	10	2	1	5	4
9	10	4	6	8	7	3	2	1	5
10	9	7	5	6	8	4	3	2	1

（2）多边形方法

n 是偶数的情形：循环赛进行 n–1 天，每个选手与其他 n–1 个选手各赛一次。

用一个 n–1 边的正多边形表示一轮比赛。多边形的顶点和中心点表示参赛选手。n=8 时的循环赛多边形如图 2-7 所示。

用水平线连接循环赛多边形的 n–2 个顶点，并将剩下的那个顶点与中心点连接起来，如图 2-8 所示。每一条连线表示一场比赛。图 2-8 所表示的第 1 轮比赛的场次是(7, 6)，(1, 5)，(2, 4)和(3, 8)。

将多边形绕中心顺时针旋转 2π/(n–1)弧度得到新的循环赛多边形，如图 2-9 所示。

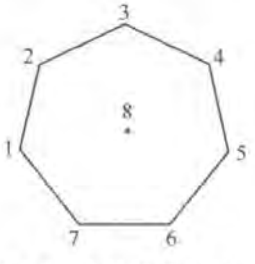

图 2-7　循环赛多边形

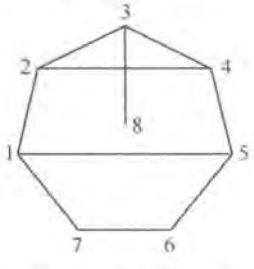

图 2-8　顶点间的连线表示比赛场次

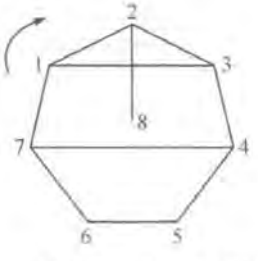

图 2-9　顺时针旋转

由此得到新一轮比赛的场次是(6, 5)，(7, 4)，(1, 3)和(2, 8)。

按此方式可旋转多边形 n–2 次。后继的旋转如图 2-10 所示。

n 是奇数的情形同样可转换为 n 是偶数的情形。

按照上述思想实现的构造法如下。

```
void construct(int n) {
   if (n == 1)
      return n;
   int  m = odd(n) ? n : n-1;
   a[n][1] = n;
   for(int i=1; i <= m; i++) {
      a[i][1] = i;
```

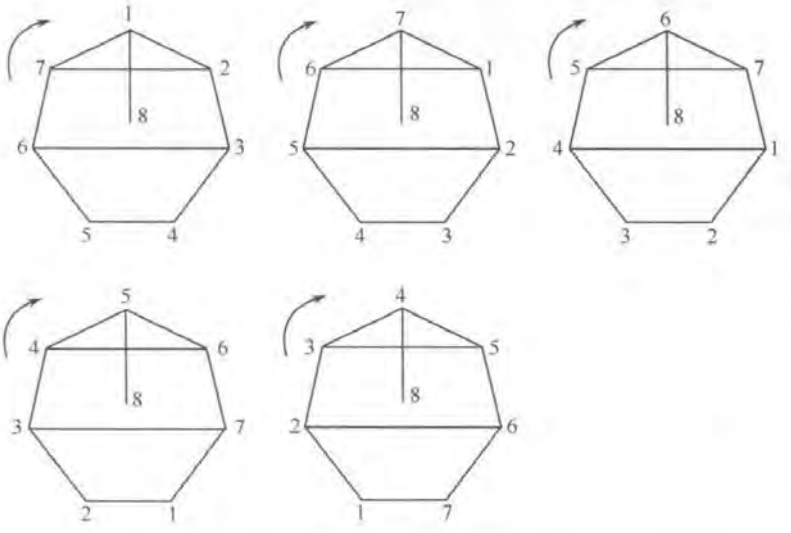

图 2-10　顺时针旋转多边形 $n-2$ 次

```
    b[i] = i+1;
    b[m+i] = i+1;
  }
  for(i=1; i <= m; i++) {
    a[1][i+1] = b[i];
    a[b[i]][i+1] = 1;
    for(int j=1; j <= m/2; j++) {
      int  k = b[i+j], r = b[i+m-j];
      a[k][i+1] = r;
      a[r][i+1] = k;
    }
  }
}
```

用上述算法计算出的 n=10 的比赛日程表如表 2-4 所示。

用上述算法计算出的 n=8 的比赛日程表完全图如图 2-11 所示。

表 2-4　多边形方法 n=10 的比赛日程表

1	2	3	4	5	6	7	8	9	10
2	1	4	6	8	10	3	5	7	9
3	10	1	5	7	9	2	4	6	8
4	9	2	1	6	8	10	3	5	7
5	8	10	3	1	7	9	2	4	6
6	7	9	2	4	1	8	10	3	5
7	6	8	10	3	5	1	9	2	4
8	5	7	9	2	4	6	1	10	3
9	4	6	8	10	3	5	7	1	2
10	3	5	7	9	2	4	6	8	1

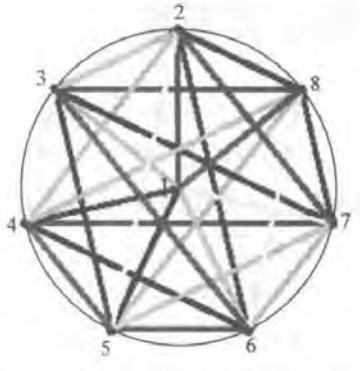

图 2-11　循环赛日程表 n=8 的解

2-16　二叉树 T 的前序、中序和后序序列。

设有 n 个运动员要进行网球循环赛。设计一个满足以下要求的比赛日程表：

（1）每个选手必须与其他 $n-1$ 个选手各赛一次；

（2）每个选手一天只能赛一次；

（3）当 n 是偶数时，循环赛进行 $n-1$ 天。当 n 是奇数时，循环赛进行 n 天。

分析与解答：（1）能；（2）能；（3）不能。

设置字符串 pre、post、inor，根据后序和中序序列确定前序序列。

```
void preorder(string a, string b) {
  if(b.length() == 1)
    pre += b;
  else {
    int  k = a.find(b.substr(b.length()-1, 1));
    pre += a[k];
    if(k>0)
      preorder(a.substr(0, k), b.substr(0, k));
    if(k<a.length()-1)
      preorder(a.substr(k+1, a.length()-k-1), b.substr (k, b.length()-k-1));
  }
}
```

根据前序和中序序列确定后序序列。

```
void postorder(string a,string b) {
  if(b.length() == 1)
    post += b;
  else {
    int k = a.find(b.substr(0, 1));
    if(k > 0)
      postorder(a.substr(0, k), b.substr(1, k));
    if(k<a.length()-1)
      postorder(a.substr(k+1, a.length()-k-1), b.substr(k+1, b.length()-k-1));
    post += a[k];
  }
}
```

算法实现题 2

2-1　众数问题。

问题描述：给定含有 n 个元素的多重集合 S，每个元素在 S 中出现的次数称为该元素的重数。多重集 S 中重数最大的元素称为众数。例如，$S=\{1, 2, 2, 2, 3, 5\}$。多重集 S 的众数是 2，其重数为 3。

算法设计：对于给定的由 n 个自然数组成的多重集 S，计算 S 的众数及其重数。

数据输入：输入数据由文件名为 input.txt 的文本文件提供。文件的第 1 行为多重集 S 中元素个数 n；在接下来的 n 行中，每行有一个自然数。

结果输出：将计算结果输出到文件 output.txt。输出文件有 2 行，第 1 行是众数，第 2 行是重数。

输入文件示例	输出文件示例
input.txt	output.txt
6	2
1	3
2	
2	
2	
3	
5	

分析与解答：算法具体实现如下。

```
void mode(int l, int r) {
   int  l1, r1;
   int  med = median(a, l, r);
   split(a, med, l, r, l1, r1);
   if(largest < r1-l1+1)
      largest = r1-l1+1, element = med;
   if (l1-l > largest)
      mode(l, l1-1);
   if (r-r1 > largest)
      mode(r1+1, r);
}
```

其中，median()函数用于找中位数，split()函数用中位数将数组分割为 2 段。

2-2　马的 Hamilton 周游路线问题。

问题描述：8×8 的国际象棋棋盘上的一只马，恰好走过除起点外的其他 63 个位置各一次，最后回到起点。这条路线称为马的一条 Hamilton 周游路线。对于给定的 $m \times n$ 的国际象棋棋盘，m 和 n 均为大于 5 的偶数，且$|m-n| \leqslant 2$，试设计一个分治算法找出马的一条 Hamilton 周游路线。

算法设计：对于给定的偶数 $m, n \geqslant 6$，且$|m-n| \leqslant 2$，计算 $m \times n$ 的国际象棋棋盘上马的一条 Hamilton 周游路线。

数据输入：由文件 input.txt 给出输入数据。第 1 行有两个正整数 m 和 n，表示给定的国际象棋棋盘由 m 行，每行 n 个格子组成。

结果输出：将计算出的马的 Hamilton 周游路线用下面的两种表达方式输出到文件 output.txt。

第 1 种表达方式按照马步的次序给出马的 Hamilton 周游路线。马的每一步用所在的方格坐标(x, y)来表示。x 表示行坐标，编号为 $0, 1, \cdots, m-1$；y 表示列坐标，编号为 $0, 1, \cdots, n-1$。起始方格为$(0, 0)$。

第 2 种表达方式在棋盘的方格中标明马到达该方格的步数。(0，0)方格为起跳步，并标明为第 1 步。

输入文件示例	输出文件示例
input.txt	output.txt
6 6	(0, 0)(2, 1)(4, 0)(5, 2)(4, 4)(2, 3)
	(0, 4)(2, 5)(1, 3)(0, 5)(2, 4)(4, 5)
	(5, 3)(3, 2)(5, 1)(3, 0)(1, 1)(0, 3)

(1, 5)(3, 4)(5, 5)(4, 3)(3, 1)(5, 0)
(4, 2)(5, 4)(3, 5)(1, 4)(0, 2)(1, 0)
(2, 2)(0, 1)(2, 0)(4, 1)(3, 3)(1, 2)
1 32 29 18 7 10
30 17 36 9 28 19
33 2 31 6 11 8
16 23 14 35 20 27
3 34 25 22 5 12
24 15 4 13 26 21

分析与解答：

（1）算法思想

在 $n\times n$ 的国际象棋棋盘上的一只马，可按 8 个不同方向移动。定义 $n\times n$ 的国际象棋棋盘上的马步图为 $G=(V,E)$。棋盘上的每个方格对应于图 G 中的一个顶点，$V=\{(i,j)|0\le i,j<n\}$。从一个顶点到另一个马步可跳达的顶点之间有一条边 $E=\{(u,v),(s,t)|\{|u-s|,|v-t|\}=\{1,2\}\}$。

图 G 有 n^2 个顶点和 $4n^2-12n+8$ 条边。马的 Hamilton 周游路线问题即是图 G 的 Hamilton 回路问题。容易看出，当 n 为奇数时，该问题无解。事实上，由于马在棋盘上移动的方格是黑白相间的，如果有解，则走到的黑白格子数相同，因此棋盘格子总数应为偶数，然而 n^2 为奇数，此为矛盾。下面给出的算法可以证明，当 $n\ge 6$ 是偶数时，问题有解，而且可以用分治法在线性时间内构造出一个解。

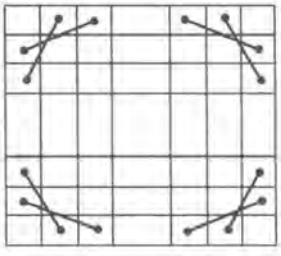

图 2-12　结构化的 Hamilton 回路

考察稍一般的情况，即给定的国际象棋棋盘有 m 行和 n 列，且 $|m-n|\le 2$ 的情况。因此可能有 $m\times m$、$m\times(m-2)$ 和 $m\times(m+2)$ 共 3 种不同规格的棋盘。为了采用分治策略，考察一类具有特殊结构的解，这类解在棋盘的 4 个角都包含 2 条特殊的边，如图 2-12 所示。我们称具有这类特殊结构的 Hamilton 回路为结构化的 Hamilton 回路。

用回溯法可在 $O(1)$时间内找出 6×6、6×8、8×8、8×10、10×10、10×12 棋盘上的结构化的 Hamilton 回路，如图 2-13 所示。其中，6×8，8×10 和 10×12 棋盘上的结构化 Hamilton 回路，旋转 90° 可以得到 8×6，10×8 和 12×10 棋盘上的结构化 Hamilton 回路。在棋盘上画出的结构化 Hamilton 回路，如图 2-14 所示。对于 $m,n\ge 12$ 的情形，采用分治策略。

分割步：将棋盘尽可能平均地分割成 4 块。当 $m,n=4k$ 时，分割为 2 个 $2k$；当 $m,n=4k+2$ 时，分割为 1 个 $2k$ 和 1 个 $2k+2$。

合并步：4 个子棋盘拼接后的结构如图 2-15 所示。

分别删除 4 个子棋盘中的结构化边 A、B、C、D，添入新边 E、F、G、H 构成整个棋盘的结构化 Hamilton 回路，如图 2-16 所示。

上述分治法得到的 Hamilton 回路显然仍是结构化 Hamilton 回路。

图 2-17 是用上述分治法计算出的 16×16 棋盘上的结构化 Hamilton 回路。

（2）算法复杂性

设用上述分治法计算 $n\times n$ 棋盘上的 Hamilton 回路所需计算时间为 $T(n)$，则 $T(n)$满足如下递归式

$$T(n)=\begin{cases}O(1) & n<12\\ 4T(n/2)+O(1) & n\ge 12\end{cases}$$

1	30	33	16	3	24
32	17	2	23	34	15
29	36	31	14	25	4
18	9	6	35	22	13
7	28	11	20	5	26
10	19	8	27	12	21

(a) 6×6 棋盘上的结构化 Hamilton 回路

1	10	31	40	21	14	29	38
32	41	2	11	30	39	22	13
9	48	33	20	15	12	37	28
42	3	44	47	6	25	18	23
45	8	5	34	19	16	27	36
4	43	46	7	26	35	24	17

(b) 6×8 棋盘上的结构化 Hamilton 回路

1	46	17	50	3	6	31	52
18	49	2	7	30	51	56	5
45	64	47	16	27	4	53	32
48	19	8	29	10	55	26	57
63	44	11	22	15	28	33	54
12	41	20	9	36	23	58	25
43	62	39	14	21	60	37	34
40	13	42	61	38	35	24	59

(c) 8×8 棋盘上的结构化 Hamilton 回路

1	46	37	66	3	48	35	68	5	8
38	65	2	47	36	67	4	7	34	69
45	80	39	24	49	18	31	52	9	6
64	23	44	21	30	15	50	19	70	33
79	40	25	14	17	20	53	32	51	10
26	63	22	43	54	29	16	73	58	71
41	78	61	28	13	76	59	56	11	74
62	27	42	77	60	55	12	75	72	57

(d) 8×10 棋盘上的结构化 Hamilton 回路

1	54	69	66	3	56	39	64	5	8
68	71	2	55	38	65	4	7	88	63
53	100	67	70	57	26	35	40	9	6
72	75	52	27	42	37	58	87	62	89
99	30	73	44	25	34	41	36	59	10
74	51	76	31	28	43	86	81	90	61
77	98	29	24	45	80	33	60	11	92
50	23	48	79	32	85	82	91	14	17
97	78	21	84	95	46	19	16	93	12
22	49	96	47	20	83	94	13	18	15

(e) 10×10 棋盘上的结构化 Hamilton 回路

1	4	119	100	65	6	69	102	71	8	75	104
118	99	2	5	68	101	42	7	28	103	72	9
3	120	97	64	41	66	25	70	39	74	105	76
98	117	48	67	62	43	40	27	60	29	10	73
93	96	63	44	47	26	61	24	33	38	77	106
116	51	94	49	20	23	46	37	30	59	34	11
95	92	115	52	45	54	21	32	35	80	107	78
114	89	50	19	22	85	36	55	58	31	12	81
91	18	87	112	53	16	57	110	83	14	79	108
88	113	90	17	86	111	84	15	56	109	82	13

(f) 10×12 棋盘上的结构化 Hamilton 回路

图 2-13　几种棋盘上的结构化 Hamilton 回路

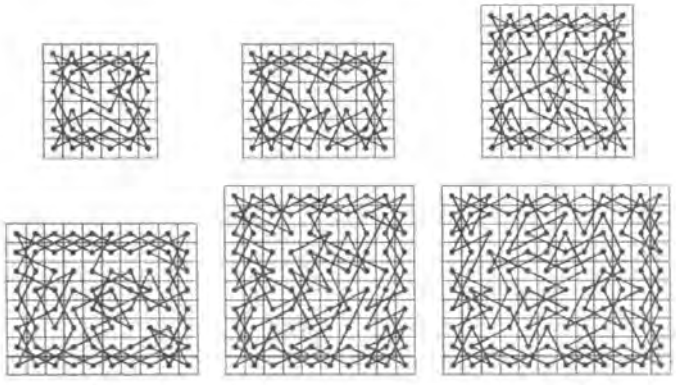

图 2-14　在棋盘上画出的结构化 Hamilton 回路

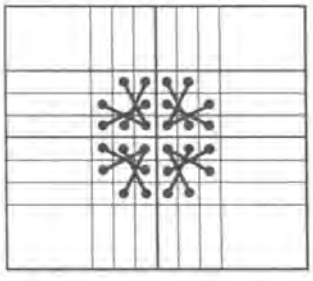

图 2-15　子棋盘的拼接

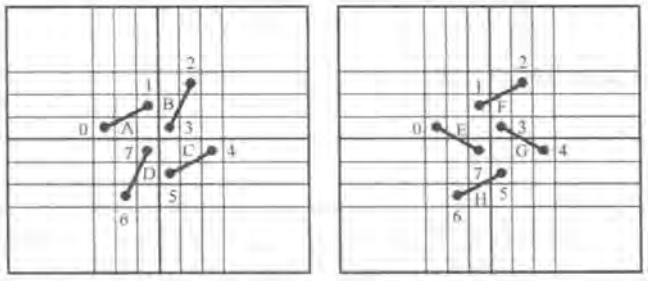

图 2-16　子棋盘中 Hamilton 回路的合并

1	238	17	242	3	6	31	244	59	40	75	44	61	64	89	46
18	241	2	7	30	243	248	5	76	43	60	65	88	45	50	63
237	256	239	16	27	4	245	32	39	58	41	74	85	62	47	90
240	19	8	29	10	247	26	249	42	77	66	87	68	49	84	51
255	236	11	22	15	28	33	246	57	38	69	80	73	86	91	48
12	233	20	9	228	23	250	25	70	35	78	67	94	81	52	83
235	254	231	14	21	252	229	34	37	56	97	72	79	54	95	92
232	13	234	253	230	227	24	251	98	71	36	55	96	93	82	53
175	220	191	224	177	180	205	226	115	134	99	130	113	110	149	128
192	223	176	181	204	225	166	179	162	131	114	109	150	129	124	111
219	174	221	190	201	178	163	206	135	116	133	100	153	112	127	148
222	193	182	203	184	165	200	167	132	161	108	151	106	125	154	123
173	218	185	196	189	202	207	164	117	136	105	158	101	152	147	126
186	215	194	183	210	197	168	199	104	139	160	107	144	157	122	155
217	172	213	188	195	170	211	208	137	118	141	102	159	120	143	146
214	187	216	171	212	209	198	169	140	103	138	119	142	145	156	121

图 2-17　用分治法计算出的 16×16 棋盘上的结构化 Hamilton 回路

解此递归式可得 $T(n)=O(n^2)$。

由此可见，上述算法是一个最优算法。

（3）算法实现

用一个类 Knight 实现算法。

```
class Knight{
public:
   Knight(int m, int n);
   ~Knight(){};
   void out();
private:
   int  m, n;
   grid  *b66, *b68, *b86, *b88,*b810, *b108, *b1010, *b1012, *b1210, **link;
   int pos(int x, int y, int col);
   void step(int m, int n, int **a, grid *b);
   void build(int m, int n, int offx, int offy, int col, grid *b);
   void base(int mm, int nn, int offx, int offy);
   bool comp(int mm, int nn, int offx, int offy);
};
```

其中，grid 是表示整数对的结构。

```
typedef struct {
   int  x;
   int  y;
} grid;
```

m 和 n 分别表示棋盘的行数和列数。二维数组 link 用来表示 Hamilton 回路。

b66，b68，b86，b88，b810，b108，b1010，b1012，b1210 分别表示 6×6，6×8，8×6，

8×8，8×10，10×8，10×10，10×12，12×10 棋盘上的结构化 Hamilton 回路。

构造函数读入基础数据，初始化各数组。

```
Knight::Knight(int mm, int nn) {
   int  i, j, **a;
   m = mm;
   n = nn;
   b66 = new grid[36];
   b68 = new grid[48];
   b86 = new grid[48];
   b88 = new grid[64];
   b810 = new grid[80];
   b108 = new grid[80];
   b1010 = new grid[100];
   b1012 = new grid[120];
   b1210 = new grid[120];
   Make2DArray(link, m, n);
   Make2DArray(a, 10, 12);
   for(i=0; i<6; i++)
      for(j=0; j<6; j++)
         fin0 >> a[i][j];
   step(6, 6, a, b66);
   for(i=0; i<6; i++)
      for(j=0; j<8; j++)
         fin0 >> a[i][j];
   step(6, 8, a, b68);
   step(8, 6, a, b86);
   for(i=0; i<8 ;i++)
      for(j=0; j<8; j++)
         fin0 >> a[i][j];
   step(8, 8, a, b88);
   for(i=0; i<8; i++)
      for(j=0; j<10; j++)
         fin0 >> a[i][j];
   step(8, 10, a, b810);
   step(10, 8, a, b108);
   for(i=0; i<10; i++)
      for(j=0; j<10; j++)
         fin0 >> a[i][j];
   step(10, 10, a, b1010);
   for(i=0; i<10; i++)
      for(j=0; j<12; j++)
         fin0 >> a[i][j];
   step(10, 12, a, b1012);
   step(12, 10, a, b1210);
}
```

其中，step()函数用于将读入的基础棋盘的 Hamilton 回路转化为网格数据。

```
void Knight::step(int m, int n, int **a, grid *b) {
```

```
  int  i, j, k = m*n;
  if(m < n) {
    for(i=0; i<m; i++) {
      for (j=0; j<n; j++) {
        int  p = a[i][j]-1;
        b[p].x = i;
        b[p].y = j;
      }
    }
  }
  else {
    for(i=0; i<m; i++) {
      for (j=0; j<n; j++) {
        int  p = a[j][i]-1;
        b[p].x = i;
        b[p].y = j;
      }
    }
  }
}
```

分治法的主体由如下 comp 算法给出。

```
bool Knight::comp(int mm, int nn, int offx, int offy) {
  int  mm1 ,mm2, nn1, nn2;
  int  x[8], y[8], p[8];
  if(odd(mm) || odd(nn) || mm-nn>2 || nn-mm>2 || mm<6 || nn<6)
    return 1;
  if(mm<12 || nn<12) {
    base(mm, nn, offx, offy);                              // 基础解
    return 0;
  }
  mm1 = mm/2;
  if(mm%4 > 0)
    mm1--;
  mm2 = mm-mm1;
  nn1 = nn/2;
  if(nn%4 > 0)
    nn1--;
  nn2 = nn-nn1;
  // 分割步
  comp(mm1, nn1, offx, offy);
  comp(mm1, nn2, offx, offy+nn1);
  comp(mm2, nn1, offx+mm1, offy);
  comp(mm2, nn2, offx+mm1, offy+nn1);
  // 合并步
  x[0] = offx+mm1-1;   y[0] = offy+nn1-3;
  x[1] = x[0]-1;       y[1] = y[0]+2;
  x[2] = x[1]-1;       y[2] = y[1]+2;
  x[3] = x[2]+2;       y[3] = y[2]-1;
```

```
  x[4] = x[3]+1;        y[4] = y[3]+2;
  x[5] = x[4]+1;        y[5] = y[4]-2;
  x[6] = x[5]+1;        y[6] = y[5]-2;
  x[7] = x[6]-2;        y[7] = y[6]+1;
  for(int i=0; i<8; i++)
    p[i] = pos(x[i], y[i], n);
  for(i=1; i<8; i += 2) {
    int j1=(i+1)%8,j2=(i+2)%8;
    if(link[x[i]][y[i]].x == p[i-1])
      link[x[i]][y[i]].x = p[j1];
    else
      link[x[i]][y[i]].y = p[j1];
    if(link[x[j1]][y[j1]].x == p[j2])
      link[x[j1]][y[j1]].x = p[i];
    else
      link[x[j1]][y[j1]].y = p[i];
  }
  return 0;
}
```

其中，base()函数根据基础解构造子棋盘的结构化 Hamilton 回路。

```
void Knight::base(int mm, int nn, int offx, int offy) {
  if(mm==6 && nn==6)
    build(mm, nn, offx, offy, n, b66);
  if(mm==6 && nn==8)
    build(mm, nn, offx, offy, n, b68);
  if(mm==8 && nn==6)
    build(mm, nn, offx, offy, n, b86);
  if(mm==8 && nn==8)
    build(mm, nn, offx, offy, n, b88);
  if(mm==8 && nn==10)
    build(mm, nn, offx, offy, n, b810);
  if(mm==10 && nn==8)
    build(mm, nn, offx, offy, n, b108);
  if(mm==10 && nn==10)
    build(mm, nn, offx, offy, n, b1010);
  if(mm==10 && nn==12)
    build(mm, nn, offx, offy, n, b1012);
  if(mm==12 && nn==10)
    build(mm, nn, offx, offy, n, b1210);
}
```

其实质性的构造由算法 build 完成。

```
void Knight::build(int m, int n, int offx, int offy, int col, grid *b) {
  int  i, p, q, k = m*n;
  for(i=0; i<k; i++) {
    int  x1 = offx+b[i].x, y1 = offy+b[i].y, x2 = offx+b[(i+1)%k].x, y2 = offy+b[(i+1)%k].y;
    p = pos(x1, y1, col);
    q=pos(x2, y2, col);
```

```
      link[x1][y1].x = q;
      link[x2][y2].y = p;
   }
}
```

其中，pos 用于计算棋盘方格的编号。棋盘方格各行从上到下，各列从左到右依次编号为 0，1，…，$mn-1$。

```
int Knight::pos(int x, int y, int col) {
   return col*x+y;
}
```

最后，由 out()函数按照要求输出计算出的结构化 Hamilton 回路。

```
void Knight::out() {
   int  i, j, k, x, y, p, **a;
   Make2DArray(a, m, n);
   if(comp(m, n, 0, 0))
      return;
   for(i=0; i < m; i++)
      for(j=0; j < n; j++)
         a[i][j] = 0;
   i = 0;
   j = 0;
   k = 2;
   a[0][0] = 1;
   cout << "(0,0)" << " ";
   for(p=1; p < m*n; p++) {
      x = link[i][j].x;
      y = link[i][j].y;
      i = x/n;
      j = x%n;
      if(a[i][j] > 0) {
         i = y/n;
         j = y%n;
      }
      a[i][j] = k++;
      cout << "("<< i << "," << j << ")";
      if((k1)%n == 0)
         cout << endl;
   }
   cout << endl;
   for(i=0; i< m; i++) {
      for(j=0; j < n; j++)
         cout << a[i][j] << " ";
      cout << endl;
   }
}
```

2-3　半数集问题。

问题描述：给定一个自然数 n，由 n 开始可以依次产生半数集 set(n)中的数如下：

（1）$n \in \text{set}(n)$；

（2）在 n 的左边加上一个自然数，但该自然数不能超过最近添加的数的一半；

（3）按此规则进行处理，直到不能再添加自然数为止。

例如，set(6)={6, 16, 26, 126, 36, 136}。半数集 set(6)中有 6 个元素。注意，该半数集是多重集。

算法设计：对于给定的自然数 n，计算半数集 set(n)中的元素个数。

数据输入：输入数据由文件名为 input.txt 的文本文件提供。每个文件只有一行，给出整数 n（$0<n<1000$）。

结果输出：将计算结果输出到文件 output.txt。输出文件只有一行，给出半数集 set(n)中的元素个数。

输入文件示例	输出文件示例
input.txt	output.txt
6	6

分析与解答：设 set(n)中的元素个数为 $f(n)$，则

$$f(n)=1+\sum_{i=1}^{n/2} f(i)$$

据此可设计求 $f(n)$的递归算法如下。

```
long comp(int n) {
   long  ans = 1;
   if(n > 1)
      for (int i=1; i <= n/2; i++)
         ans += comp(i);
   return ans;
}
```

上述算法中显然有很多的重复子问题计算。用数组存储已计算过的结果，避免重复计算，可明显改进算法的效率。改进后的算法如下。

```
long comp(int n) {
   long  ans = 1;
   if (a[n] > 0)
      return a[n];
   for (int i=1; i <= n/2; i++)
      ans += comp(i);
   a[n] = ans;
   return ans;
}
int main() {
   while (cin >> n) {
      memset(a, sizeof(a), 0);
      a[1] = 1;
      cout << comp(n) << endl;
   }
   return 0;
}
```

2-4　半数单集问题。

问题描述：给定一个自然数 n，由 n 开始可以依次产生半数集 set(n)中的数如下：

（1）$n \in$ set(n)；

（2）在 n 的左边加上一个自然数，但该自然数不能超过最近添加的数的一半；

（3）按此规则进行处理，直到不能再添加自然数为止。

例如，set(6)={6, 16, 26, 126, 36, 136}。半数集 set(6)中有 6 个元素。注意，该半数集不是多重集。集合中已经有的元素不再添加到集合中。

算法设计：对于给定的自然数 n，计算半数集 set(n)中的元素个数。

数据输入：输入数据由文件名为 input.txt 的文本文件提供。每个文件只有一行，给出整数 n（$0<n<201$）。

结果输出：将计算结果输出到文件 output.txt。输出文件只有一行，给出半数集 set(n)中的元素个数。

输入文件示例	输出文件示例
input.txt	output.txt
6	6

分析与解答：此题与算法实现题 2-3 类似。主要区别在于此题的半数集为单集，不允许重复元素。因此在计算时应剔除重复元素。注意，题中条件 $0<n<201$，蕴含 $0<n/2\leqslant 100$。因此，在计算时，可能产生重复的元素是 2 位数。一个 2 位数 x 重复产生的条件是，在 1 位数 $y=x\%10$ 的半数集中已产生 x，因此 $x/10\leqslant y/2$，或等价地，$2(x/10)\leqslant x\%10$。在前面的算法中，加入剔除重复元素的语句即可。

```
long comp(int n) {
   long  ans = 1;
   if (a[n] > 0)
      return a[n];
   for (int i=1; i <= n/2; i++){
      ans += comp(i);
      if ((i > 10) && (2*(i/10) <= i%10))
         ans -= a[i/10];
   }
   a[n] = ans;
   return ans;
}
```

如果不利用题中对于 n 的范围限制，则应考虑一般情况，即有 $\log_{10}n$ 位数字的情况。

本题还可用散列表或数字检索树等数据结构方法来实现。

2-5　有重复元素的排列问题。

问题描述：设 $R=\{r_1, r_2, \cdots, r_n\}$ 是要进行排列的 n 个元素。其中元素 r_1、r_2、…、r_n 可能相同。试设计一个算法，列出 R 的所有不同排列。

算法设计：给定 n 及待排列的 n 个元素。计算出这 n 个元素的所有不同排列。

数据输入：由文件 input.txt 提供输入数据。文件的第 1 行是元素个数 n，$1\leqslant n\leqslant 500$。接下来的 1 行是待排列的 n 个元素。

结果输出：将计算出的 n 个元素的所有不同排列输出到文件 output.txt。文件最后 1 行中

的数是排列总数。

输入文件示例	输出文件示例
input.txt	output.txt
4	aacc
aacc	acac
	acca
	caac
	caca
	ccaa
	6

分析与解答：与主教材中的算法 Perm 类似。主要区别是对重复元素的处理。

```
template<class Type>
void Perm(Type list[], int k, int m) {
  if(k == m) {
    ans++;
    for(int i=0;i <= m;i++)
      cout << list[i];
    cout << endl;
  }
  else for (int i=k; i <= m; i++)
    if(ok(list, k, i)) {
      Swap(list[k], list[i]);
      Perm(list, k+1, m);
      Swap(list[k], list[i]);
  }
}
```

其中，函数 ok()用于判别重复元素。

```
template<class Type>
int ok(Type list[], int k, int i) {
  if(i>k)
    for(int t=k; t < i; t++)
      if(list[t] == list[i])
        return 0;
  return 1;
}
```

2-6 排列的字典序问题。

问题描述：n 个元素$\{1, 2, \cdots, n\}$有 $n!$个不同的排列。将这 $n!$个排列按字典序排列，并编号为 0, 1, …, $n!-1$。每个排列的编号为其字典序值。例如，当 $n=3$ 时，6 个不同排列的字典序值如下：

字典序值	0	1	2	3	4	5
排列	123	132	213	231	312	321

算法设计：给定 n 及 n 个元素$\{1, 2, \cdots, n\}$的一个排列，计算出这个排列的字典序值，以及按字典序排列的下一个排列。

数据输入：由文件 input.txt 提供输入数据。文件的第 1 行是元素个数 n。接下来的 1 行

是 n 个元素{1, 2, …, n}的一个排列。

结果输出：将计算出的排列的字典序值和按字典序排列的下一个排列输出到文件output.txt。文件的第 1 行是字典序值，第 2 行是按字典序排列的下一个排列。

输入文件示例	输出文件示例
input.txt	output.txt
8	8227
2 6 4 5 8 1 7 3	2 6 4 5 8 3 1 7

分析与解答：（1）由排列计算字典序值

设给定的{1, 2, …, n}的排列为 π，其字典序值为 rank(π, n)。按字典序的定义显然有

$$(\pi[1]-1)(n-1)!\leqslant \text{rank}(\pi, n)\leqslant \pi[1](n-1)!-1$$

设 r 是 π 在以 $\pi[1]$ 开头的所有排列中的序号，则 r 也是[$\pi[2]$, $\pi[3]$…, $\pi[n]$]作为集合{1, 2, …, n}/{$\pi[1]$}中排列的字典序值。如果将[$\pi[2]$, $\pi[3]$, …, $\pi[n]$]中每个大于 $\pi[1]$的元素都减 1，则得到集合{1, 2, …, n−1}的一个排列 π'，其字典序值也是 r。由此得到计算 rank(π, n)的递归式如下：

$$\text{rank}(\pi, n)=(\pi[1]-1)(n-1)!+\text{rank}(\pi', n-1)$$

式中，

$$\pi'[i]=\begin{cases}\pi[i+1]-1 & \pi[i+1]>\pi[1]\\ \pi[i+1] & \pi[i+1]<\pi[1]\end{cases}$$

初始条件为

$$\text{rank}([1], 1)=0$$

据此可设计计算 rank(π, n)的算法 permRank 如下。

```
int permRank(int n, int *pi) {
  for(int j=1, r=0; j <= n; j++)
    rho[j] = pi[j];
  for(j=1; j <= n; j++) {
    r += (rho[j]-1)*f[n-j];
    for(int i=j+1; i <= n; i++)
      if(rho[i] > rho[j])
        rho[i]--;
  }
  return r;
}
```

其中，f[j]存储预先计算出的 j!的值。

（2）由字典序值计算排列

对于每个整数 r，0≤r≤n!−1，都有唯一的阶层分解：$r=\sum_{i=1}^{n-1} d_i\cdot i!$, $0\leqslant d_i\leqslant i$。

设 r = rank(π, n)，则 $\pi[1]=d_{n-1+1}$。

进一步，由 $r'=r-d_{n-1}(n-1)!=\text{rank}(\pi', n-1)$可递归地找到排列 π'。最后令 $\pi[i]=\pi'[i+1]$可得到排列 π。

据此可设计计算排列 π 使 r = rank(π, n)的算法 permUnrank 如下。

```
void permUnrank(int n, int r,int *pi) {
  pi[n] = 1;
  for(int j=1; j < n; j++) {
```

```
    int  d = (r%f[j+1])/f[j];
    r -= d*f[j];
   pi[n-j] = d+1;
   for(int i = n-j+1; i <= n; i++)
     if(pi[i] > d)
       pi[i]++;
  }
}
```

（3）由排列计算下一个排列

按字典序的定义可设计从一个排列计算下一个排列的算法。对于给定的排列 π，首先找到下标 i，使得 $\pi[i]<\pi[i+1]$，且 $\pi[i+1]>\pi[i+2]>\cdots>\pi[n]$；其次找到下标 j，使得 $\pi[i]<\pi[j]$且对所有 $j<k\leqslant n$ 有 $\pi[k]<\pi[i]$；然后交换 $\pi[i]$和 $\pi[j]$；最后将子排列[$\pi[i+1]$，$\pi[i+2]$，…，$\pi[n]$]反转。按此思想设计的算法 permSucc 如下。

```
void permSucc(int n ,int *pi, int &flag) {
  pi[0] = 0;
  int  i = n-1;
  while (pi[i+1] < pi[i])
    i--;
  if(i == 0)
    flag = 0;
  else {
    flag = 1;
    int  j = n;
    while(pi[j] < pi[i])
      j--;
    Swap(pi[i], pi[j]);
    for(int h=i+1; h <= n; h++)
      rho[h] = pi[h];
    for(h=i+1; h <= n; h++)
      pi[h] = rho[n+i+1-h];
  }
}
```

2-7　集合划分问题。

问题描述：n 个元素的集合{1, 2, …, n}可以划分为若干非空子集。例如，当 n=4 时，集合{1, 2, 3, 4}可以划分为 15 个不同的非空子集如下：

{{1}, {2}, {3}, {4}}　　{{1, 3}, {2, 4}}
{{1, 2}, {3}, {4}}　　{{1, 4}, {2, 3}}
{{1, 3}, {2}, {4}}　　{{1, 2, 3}, {4}}
{{1, 4}, {2}, {3}}　　{{1, 2, 4}, {3}}
{{2, 3}, {1}, {4}}　　{{1, 3, 4}, {2}}
{{2, 4}, {1}, {3}}　　{{2, 3, 4}, {1}}
{{3, 4}, {1}, {2}}　　{{1, 2, 3, 4}}
{{1, 2}, {3, 4}}

算法设计：给定正整数 n，计算出 n 个元素的集合$\{1, 2, \cdots, n\}$可以划分为多少个不同的非空子集。

数据输入：由文件 input.txt 提供输入数据。文件的第 1 行是元素个数 n。

结果输出：将计算出的不同的非空子集数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
5	52

分析与解答：所求的是 Bell 数，满足如下递归式：

$$B(n)=\sum_{i=0}^{n}\left|\begin{matrix}n-1\\ i\end{matrix}\right|B(i);\qquad B(0)=1$$

2-8　集合划分问题。

问题描述：n 个元素的集合$\{1, 2, \cdots, n\}$可以划分为若干非空子集。例如，当 n=4 时，集合$\{1, 2, 3, 4\}$可以划分为 15 个不同的非空子集如下：

{{1}, {2}, {3}, {4}}	{{1, 3}, {2, 4}}
{{1, 2}, {3}, {4}}	{{1, 4}, {2, 3}}
{{1, 3}, {2}, {4}}	{{1, 2, 3}, {4}}
{{1, 4}, {2}, {3}}	{{1, 2, 4}, {3}}
{{2, 3}, {1}, {4}}	{{1, 3, 4}, {2}}
{{2, 4}, {1}, {3}}	{{2, 3, 4}, {1}}
{{3, 4}, {1}, {2}}	{{1, 2, 3, 4}}
{{1, 2}, {3, 4}}	

其中，集合{{1, 2, 3, 4}}由 1 个子集组成；集合{{1, 2}, {3, 4}}，{{1, 3}, {2, 4}}，{{1, 4}, {2, 3}}，{{1, 2, 3}, {4}}，{{1, 2, 4}, {3}}，{{1, 3, 4}, {2}}，{{2, 3, 4}, {1}}由 2 个子集组成；集合{{1, 2}, {3}, {4}}，{{1, 3}, {2}, {4}}，{{1, 4}, {2}, {3}}，{{2, 3}, {1}, {4}}，{{2, 4}, {1}, {3}}，{{3, 4}, {1}, {2}}由 3 个子集组成；集合{{1}, {2}, {3}, {4}}由 4 个子集组成。

算法设计：给定正整数 n 和 m，计算出 n 个元素的集合$\{1, 2, \cdots, n\}$可以划分为多少个不同的由 m 个非空子集组成的集合。

数据输入：由文件 input.txt 提供输入数据。文件的第 1 行是元素个数 n 和非空子集数 m。

结果输出：将计算出的不同的由 m 个非空子集组成的集合数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
4 3	6

分析与解答：所求的是第 2 类 Stirling 数 $S(n, m)$，满足如下递归式：

$$S(n, n+1)=0$$

$$S(n, 0)=0$$

$$S(0, 0)=1$$

$$S(n, m)=mS(n-1, m)+S(n-1, m-1)$$

关于 Bell 数，显然有 $B(n)=\sum_{m=1}^{n}S(n,m)$

```
void StirlingNumbers2(int m, int n) {
  int  min;
  S[0][0] = 1;
  for(int i=1; i <= m; i++)
    S[i][0] = 0;
  for(i=0; i < m; i++)
    S[i][i+1] = 0;
  for(i=1; i <= m; i++) {
    if(i < n)
      min = i;
    else
      min = n;
    for(int j=1;j <= min; j++)
      S[i][j] = j*S[i-1][j] + S[i-1][j-1];
  }
}
int computeB(int m) {
  StirlingNumbers2(m, m);
  for(int i=0; i < m; i++)
    B[i] = 0;
  for(i=1; i <= m; i++){
    for(int j=0; j <= i; j++)
      B[i-1] += S[i][j];
  }
  return B[m-1];
}
```

2-9　双色 Hanoi 塔问题。

问题描述：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，由大到小地叠放在一起。各圆盘从小到大编号为 1, 2, …, n，奇数号圆盘着红色，偶数号圆盘着蓝色，如图 2-18 所示。现要求将塔座 A 上的这一叠圆盘移到塔座 B 上，并仍按同样顺序叠置。在移动圆盘时应遵守以下移动规则：

规则Ⅰ：每次只能移动 1 个圆盘；

规则Ⅱ：任何时刻都不允许将较大的圆盘压在较小的圆盘之上；

规则Ⅲ：任何时刻都不允许将同色圆盘叠放在一起；

规则Ⅳ：在满足移动规则Ⅰ～Ⅲ的前提下，可将圆盘移至 A、B、C 中任一塔座上。

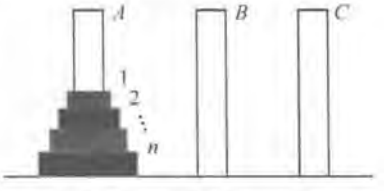

图 2-18　双色 Hanoi 塔

试设计一个算法，用最少的移动次数将塔座 A 上的 n 个圆盘移到塔座 B 上，并仍按同样顺序叠置。

算法设计：对于给定的正整数 n，计算最优移动方案。

数据输入：由文件 input.txt 给出输入数据。第 1 行是给定的正整数 n。

结果输出：将计算出的最优移动方案输出到文件 output.txt。文件的每行由一个正整数 k

和 2 个字符 c_1 和 c_2 组成，表示将第 k 个圆盘从塔座 c_1 移到塔座 c_2 上。

输入文件示例	输出文件示例
input.txt	output.txt
3	1 A B
	2 A C
	1 B C
	3 A B
	1 C A
	2 C B
	1 A B

分析与解答：可用主教材中的标准 Hanoi 塔算法。问题是要证明标准 Hanoi 塔算法不违反规则（3）。

用数学归纳法。设 hanoi(n, A, B, C)是将塔座 A 上的 n 个圆盘，以塔座 C 为辅助塔座，移到目的塔座 B 上的标准 Hanoi 塔算法。

归纳假设：当圆盘个数小于 n 时，hanoi(n, A, B, C)不违反规则（3），且在移动过程中，目的塔座 B 上最低圆盘的编号与 n 具有相同奇偶性，辅助塔座 C 上最低圆盘的编号与 n 具有不同奇偶性。

当圆盘个数为 n 时，标准 Hanoi 塔算法 hanoi(n, A, B, C)由以下 3 个步骤完成：

（1）hanoi(n−1, A, C, B)；

（2）move(A, B)；

（3）hanoi(n−1, C, B, A)。

按归纳假设，步骤（1）不违反规则（3），且在移动过程中，塔座 C 上最低圆盘的编号与 n−1 具有相同奇偶性，塔座 B 上最低圆盘的编号与 n−1 具有不同奇偶性，从而塔座 B 上最低圆盘的编号与 n 具有相同奇偶性，塔座 C 上最低圆盘的编号与 n 具有不同奇偶性。

步骤（2）也不违反规则（3），且塔座 B 上最低圆盘的编号与 n 相同。

按归纳假设，步骤（3）不违反规则（3），且在移动过程中，塔座 B 上倒数第 2 个圆盘的编号与 n−1 具有相同奇偶性，塔座 A 上最低圆盘的编号与 n−1 具有不同奇偶性，从而塔座 B 上倒数第 2 个圆盘的编号与 n 具有不同奇偶性，塔座 A 上最低圆盘的编号与 n 具有相同奇偶性。因此在移动过程中，塔座 B 上圆盘不违反规则（3），而且塔座 B 上最低圆盘的编号与 n 具有相同奇偶性，塔座 C 上最低圆盘的编号与 n 具有不同奇偶性。

由数学归纳法可知，hanoi(n, A, B, C)不违反规则（3）。

2-10　标准二维表问题。

问题描述：设 n 是一个正整数。2×n 的标准二维表是由正整数 1, 2, …, 2n 组成的 2×n 数组，该数组的每行从左到右递增，每列从上到下递增。2×n 的标准二维表全体记为 Tab(n)。例如，当 n=3 时，Tab(3)二维表如图 2-19 所示。

1	2	3	1	2	4	1	2	5	1	3	4	1	3	5
4	5	6	3	5	6	3	4	6	2	5	6	2	4	6

图 2-19　n=3 时 Tab(3)二维表

算法设计：给定正整数 n，计算 Tab(n)中 2×n 的标准二维表的个数。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n。

结果输出：将计算出的 Tab(n)中 $2\times n$ 的标准二维表的个数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3	5

分析与解答：Catalan 数。

2-11　整数因子分解问题。

问题描述：大于 1 的正整数 n 可以分解为 $n=x_1\times x_2\times\cdots\times x_m$。例如，当 n=12 时，有 8 种不同的分解式：

12=12	12=3×2×2
12=6×2	12=2×6
12=4×3	12=2×3×2
12=3×4	12=2×2×3

算法设计：对于给定的正整数 n，计算 n 共有多少种不同的分解式。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n（$1\leqslant n\leqslant 2000000000$）。

结果输出：将计算出的不同的分解式数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
12	8

分析与解答：对 n 的每个因子递归搜索。

```
void solve(int n) {
  if (n == 1)
    total++;
  else {
    for (int i=2; i <= n; i++)
      if (n%i == 0)
        solve(n/i);
  }
}
```

本题可用动态规划求解。

欢迎登录 免费 获取本书教学资源
http://www.hxedu.com.cn

计算机算法设计与分析习题解答（第5版）

本书是与“十二五”普通高等教育本科国家级规划教材《计算机算法设计与分析（第5版）》配套的辅助教材和国家精品课程教材，分别对主教材中的算法分析题和算法实现题给出了解答或解题思路提示。为了提高学生灵活运用算法设计策略解决实际问题的能力，本书还将主教材中的许多习题改造成算法实现题，要求学生设计出求解算法并上机实现。本书教学资料包含各章算法实现题、测试数据和答案，可在华信教育资源网免费注册下载。

本书内容丰富，理论联系实际，可作为高等学校计算机科学与技术、软件工程、信息安全、信息与计算科学等专业本科生和研究生学习计算机算法设计的辅助教材，也是工程技术人员和自学者的参考书。

提升学生“知识—能力—素质”	体现“基础—技术—应用”内容
把握教学“难度—深度—强度”	提供“教材—教辅—课件”支持

相关图书：《计算机算法设计与分析（第5版）》 ISBN 978-7-121-34439-8

策划编辑：章海涛
责任编辑：章海涛
封面设计：张 昱

ISBN 978-7-121-34438-1

定价：56.00 元