“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

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。

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 1 章　算法概述 …… 1

算法分析题 1 …… 1

1-1　函数的渐近表达式 …… 1

1-2　$O(1)$和 $O(2)$的区别 …… 1

1-3　按渐近阶排列表达式 …… 1

1-4　算法效率 …… 1

1-5　硬件效率 …… 1

1-6　函数渐近阶 …… 2

1-7　$n!$的阶 …… 2

1-8　$3n+1$ 问题 …… 2

1-9　平均情况下的计算时间复杂性 …… 2

算法实现题 1 …… 3

1-1　统计数字问题 …… 3

1-2　字典序问题 …… 4

1-3　最多约数问题 …… 4

1-4　金币阵列问题 …… 6

1-5　最大间隙问题 …… 8

第 2 章　递归与分治策略 …… 11

算法分析题 2 …… 11

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 …… 11

2-2　判断这 7 个算法的正确性 …… 12

2-3　改写二分搜索算法 …… 15

2-4　大整数乘法的 $O(nm^{\log(3/2)})$算法 …… 16

2-5　5 次 $n/3$ 位整数的乘法 …… 16

2-6　矩阵乘法 …… 18

2-7　多项式乘积 …… 18

2-8　$O(1)$空间子数组换位算法 …… 19

2-9　$O(1)$空间合并算法 …… 21

2-10　$\sqrt{n}$ 段合并排序算法 …… 27

2-11　自然合并排序算法 …… 28

2-12　第 k 小元素问题的计算时间下界 …… 29

2-13　非增序快速排序算法 …… 31

2-14 构造 Gray 码的分治算法 31
2-15 网球循环赛日程表 32
2-16 二叉树 T 的前序、中序和后序序列 35
算法实现题 2 36
2-1 众数问题 36
2-2 马的 Hamilton 周游路线问题 37
2-3 半数集问题 44
2-4 半数单集问题 46
2-5 有重复元素的排列问题 46
2-6 排列的字典序问题 47
2-7 集合划分问题 49
2-8 集合划分问题 50
2-9 双色 Hanoi 塔问题 51
2-10 标准二维表问题 52
2-11 整数因子分解问题 53
第 3 章 动态规划 54
算法分析题 3 54
3-1 最长单调递增子序列 54
3-2 最长单调递增子序列的 $O(n\log n)$算法 54
3-3 整数线性规划问题 55
3-4 二维 0-1 背包问题 56
3-5 Ackermann 函数 57
算法实现题 3 59
3-1 独立任务最优调度问题 59
3-2 最优批处理问题 61
3-3 石子合并问题 67
3-4 数字三角形问题 68
3-5 乘法表问题 69
3-6 租用游艇问题 70
3-7 汽车加油行驶问题 70
3-8 最小 m 段和问题 71
3-9 圈乘运算问题 72
3-10 最大长方体问题 78
3-11 正则表达式匹配问题 79
3-12 双调旅行售货员问题 83
3-13 最大 k 乘积问题 84
3-14 最少费用购物问题 86
3-15 收集样本问题 87

3-16 最优时间表问题 89
3-17 字符串比较问题 89
3-18 有向树 k 中值问题 90
3-19 有向树独立 k 中值问题 94
3-20 有向直线 m 中值问题 98
3-21 有向直线 2 中值问题 101
3-22 树的最大连通分支问题 103
3-23 直线 k 中值问题 105
3-24 直线 k 覆盖问题 109
3-25 m 处理器问题 113

第 4 章 贪心算法 116

算法分析题 4 116

4-1 程序最优存储问题 116
4-2 最优装载问题的贪心算法 116
4-3 Fibonacci 序列的哈夫曼编码 116
4-4 最优前缀码的编码序列 117

算法实现题 4 117

4-1 会场安排问题 117
4-2 最优合并问题 118
4-3 磁带最优存储问题 118
4-4 磁盘文件最优存储问题 119
4-5 程序存储问题 120
4-6 最优服务次序问题 120
4-7 多处最优服务次序问题 121
4-8 d 森林问题 122
4-9 虚拟汽车加油问题 123
4-10 区间覆盖问题 124
4-11 删数问题 124
4-12 磁带最大利用率问题 125
4-13 非单位时间任务安排问题 126
4-14 多元 Huffman 编码问题 127
4-15 最优分解问题 128

第 5 章 回溯法 130

算法分析题 5 130

5-1 装载问题改进回溯法 1 130
5-2 装载问题改进回溯法 2 131
5-3 0-1 背包问题的最优解 132
5-4 最大团问题的迭代回溯法 134

5-5 旅行售货员问题的费用上界 135
5-6 旅行售货员问题的上界函数 136
算法实现题 5 137
5-1 子集和问题 137
5-2 最小长度电路板排列问题 138
5-3 最小重量机器设计问题 140
5-4 运动员最佳配对问题 141
5-5 无分隔符字典问题 142
5-6 无和集问题 144
5-7 n 色方柱问题 145
5-8 整数变换问题 150
5-9 拉丁矩阵问题 151
5-10 排列宝石问题 152
5-11 重复拉丁矩阵问题 154
5-12 罗密欧与朱丽叶的迷宫问题 156
5-13 工作分配问题 158
5-14 布线问题 159
5-15 最佳调度问题 160
5-16 无优先级运算问题 161
5-17 世界名画陈列馆问题 163
5-18 世界名画陈列馆问题（不重复监视） 166
5-19 算 m 点问题 169
5-20 部落卫队问题 171
5-21 子集树问题 173
5-22 0-1 背包问题 174
5-23 排列树问题 176
5-24 一般解空间搜索问题 177
5-25 最短加法链问题 179
第 6 章 分支限界法 185
算法分析题 6 185
6-1 0-1 背包问题的栈式分支限界法 185
6-2 释放结点空间的队列式分支限界法 187
6-3 及时删除不用的结点 188
6-4 用最大堆存储活结点的优先队列式分支限界法 189
6-5 释放结点空间的优先队列式分支限界法 192
6-6 团顶点数的上界 194
6-7 团顶点数改进的上界 194
6-8 修改解旅行售货员问题的分支限界法 195
6-9 试修改解旅行售货员问题的分支限界法，使得算法保存已产生的排列树 197

6-10 电路板排列问题的队列式分支限界法 199
算法实现题 6 201
6-1 最小长度电路板排列问题 201
6-2 最小权顶点覆盖问题 203
6-3 无向图的最大割问题 206
6-4 最小重量机器设计问题 209
6-5 运动员最佳配对问题 212
6-6 n 后问题 214
6-7 布线问题 216
6-8 最佳调度问题 218
6-9 无优先级运算问题 220
6-10 世界名画陈列馆问题 223
6-11 子集空间树问题 226
6-12 排列空间树问题 229
6-13 一般解空间的队列式分支限界法 232
6-14 子集空间树问题 236
6-15 排列空间树问题 241
6-16 一般解空间的优先队列式分支限界法 246
6-17 推箱子问题 250
第 7 章 概率算法 256
算法分析题 7 256
7-1 模拟正态分布随机变量 256
7-2 随机抽样算法 256
7-3 随机产生 m 个整数 257
7-4 集合大小的概率算法 258
7-5 生日问题 258
7-6 易验证问题的拉斯维加斯算法 259
7-7 用数组模拟有序链表 260
7-8 $O(n^{3/2})$舍伍德型排序算法 260
7-9 n 后问题解的存在性 260
7-10 整数因子分解算法 262
7-11 非蒙特卡罗算法的例子 262
7-12 重复 3 次的蒙特卡罗算法 263
7-13 集合随机元素算法 263
7-14 由蒙特卡罗算法构造拉斯维加斯算法 265
7-15 产生素数算法 265
7-16 矩阵方程问题 265
算法实现题 7 266

7-1 模平方根问题 266
7-2 素数测试问题 268
7-3 集合相等问题 269
7-4 逆矩阵问题 269
7-5 多项式乘积问题 270
7-6 皇后控制问题 270
7-7 3-SAT 问题 274
7-8 战车问题 275
第 8 章 线性规划与网络流 278
算法分析题 8 278
8-1 线性规划可行区域无界的例子 278
8-2 单源最短路与线性规划 278
8-3 网络最大流与线性规划 279
8-4 最小费用流与线性规划 279
8-5 运输计划问题 279
8-6 单纯形算法 280
8-7 边连通度问题 281
8-8 有向无环网络的最大流 281
8-9 无向网络的最大流 281
8-10 最大流更新算法 282
8-11 混合图欧拉回路问题 282
8-12 单源最短路与最小费用流 282
8-13 中国邮路问题 282
算法实现题 8 283
8-1 飞行员配对方案问题 283
8-2 太空飞行计划问题 284
8-3 最小路径覆盖问题 285
8-4 魔术球问题 286
8-5 圆桌问题 287
8-6 最长递增子序列问题 287
8-7 试题库问题 290
8-8 机器人路径规划问题 291
8-9 方格取数问题 294
8-10 餐巾计划问题 298
8-11 航空路线问题 299
8-12 软件补丁问题 300
8-13 星际转移问题 301
8-14 孤岛营救问题 302
8-15 汽车加油行驶问题 304

8-16 数字梯形问题 307
8-17 运输问题 311
8-18 分配工作问题 314
8-19 负载平衡问题 315
8-20 最长 k 可重区间集问题 317
8-21 最长 k 可重线段集问题 319
第 9 章 串与序列的算法 323
算法分析题 9 323
9-1 简单子串搜索算法最坏情况复杂性 323
9-2 后缀重叠问题 323
9-3 改进前缀函数 323
9-4 确定所有匹配位置的 KMP 算法 324
9-5 特殊情况下简单子串搜索算法的改进 325
9-6 简单子串搜索算法的平均性能 325
9-7 带间隙字符的模式串搜索 326
9-8 串接的前缀函数 326
9-9 串的循环旋转 327
9-10 失败函数性质 327
9-11 输出函数性质 328
9-12 后缀数组类 328
9-13 最长公共扩展查询 329
9-14 最长公共扩展性质 332
9-15 后缀数组性质 333
9-16 后缀数组搜索 334
9-17 后缀数组快速搜索 335
算法实现题 9 338
9-1 安全基因序列问题 338
9-2 最长重复子串问题 342
9-3 最长回文子串问题 343
9-4 相似基因序列性问题 344
9-5 计算机病毒问题 345
9-6 带有子串包含约束的最长公共子序列问题 347
9-7 多子串排斥约束的最长公共子序列问题 349
参考文献 351

第5章 回溯法

算法分析题5

5-1 装载问题改进回溯法1。

用主教材中的改进策略1重写装载问题回溯法，使改进后算法的计算时间复杂性为$O(2^n)$。

分析与解答：先运行只计算最优值的算法maxLoading1，计算出最优装载量bestw。由于该算法不记录最优解，故所需的计算时间为$O(2^n)$。

```
template<class T>
void Loading<T>::maxLoading1(int i) {
   if(i > n) {
      bestw = cw;
      return;
   }
   r -= w[i];
   if(cw+w[i] <= c) {
      cw += w[i];
      maxLoading1(i+1);
      cw -= w[i];
   }
   if(cw+r > bestw)
      maxLoading1(i+1);
   r += w[i];
}
```

然后运行改进后的算法maxLoading，在首次到达的叶结点处，即首次遇到$i>n$时终止算法。由此返回的bestx即为最优解。

```
template<class T>
void Loading<T>::maxLoading(int i) {
   if(found)
      return;
   if(i > n) {
      for (int j=1; j <= n; j++)
         bestx[j] = x[j];
      found = true;
      return;
   }
   r -= w[i];
   if(cw+w[i] <= c) {
      x[i] = 1;
      cw += w[i];
```

```
        maxLoading(i+1);
        cw -= w[i];
    }
    if(cw+r >= bestw) {
        x[i] = 0;
        maxLoading(i+1);}
    r += w[i];
}
```

5-2 装载问题改进回溯法 2。

用主教材中的改进策略 2 重写装载问题回溯法，使改进后算法的计算时间复杂性为 $O(2^n)$。

分析与解答：在算法中动态地更新 bestx。在第 i 层的当前结点处，当前最优解由 $x[j]$（$1 \leqslant j < i$）和 bestx[j]（$i \leqslant j \leqslant n$）组成。每当算法回溯一层，将 $x[i]$存入 bestx[i]。这样在每个结点处更新 bestx 只需 $O(1)$时间，从而整个算法中更新 bestx 所需的时间为 $O(2^n)$。

```
template<class T>
void Loading<T>::maxLoading(int i) {
    if(i > n) {
        ii = n;
        bestw = cw;
        return;
    }
    r -= w[i];
    if(cw+w[i] <= c) {
        x[i] = 1;
        cw += w[i];
        maxLoading(i+1);
        if(ii == i) {
            bestx[i] = 1;
            ii--;
        }
        cw -= w[i];
    }
    if(cw+r > bestw) {
        x[i] = 0;
        maxLoading(i+1);
        if(ii == i) {
            bestx[i] = 0;
            ii--;
        }
    }
    r += w[i];
}
template<class T>
T MaxLoading(T w[], T c, int n, int bestx[]) {
    Loading<T> X;
    X.x = new int [n+1];
    X.w = w;
```

```
    X.c = c;
    X.n = n;
    X.bestx = bestx;
    X.bestw = 0;
    X.cw = 0;
    X.ii = 0;
    X.r = 0;
    for(int i=1;i <= n; i++)
       X.r += w[i];
    X.maxLoading(1);
    delete[] X.x;
    return X.bestw;
}
```

5-3 0-1 背包问题的最优解。

重写 0-1 背包问题的回溯法，使算法能输出最优解。

分析与解答：为了构造最优解，必须在算法中记录与当前最优值相应的当前最优解。为此，在类 Knap 中增加 2 个私有数据成员 x 和 bestx。x 用于记录从根至当前结点的路径；bestx 记录当前最优解。算法搜索到达叶结点处，就修正 bestx 的值。

修改后的算法描述如下。

```
template<class Typew, class Typep>
class Knap {
   friend Typep Knapsack(Typep*, Typew*, Typew, int, int[]);
private:
   Typep Bound(int i);
   void Backtrack(int i);
   Typew  c;                                    // 背包容量
   int  n;                                      // 物品数
        *x,                                     // 当前解
        *bestx;                                 // 当前最优解
   Typew  *w;                                   // 物品重量数组
   Typep  *p;                                   // 物品价值数组
   Typew  cw;                                   // 当前重量
   Typep  cp;                                   // 当前价值
   Typep  bestp;                                // 当前最优价值
};
```

在回溯过程中记录从根至当前结点的路径。

```
template<class Typew, class Typep>
void Knap<Typew, Typep>::Knapsack(int i) {
   if(i > n) {
      for(int j=1; j <= n; j++)
         bestx[j] = x[j];
      bestp = cp;
      return;
   }
   if(cw+w[i] <= c) {
       x[i] = 1;
```

```
        cw += w[i];
        cp += p[i];
        Knapsack(i+1);
        cw -= w[i];
        cp -= p[i];
        x[i] = 0;
    }
    if(Bound(i+1) > bestp)
        Knapsack(i+1);
}
```

Knapsack()函数进行初始化，并用回溯法求解。

```
template<class Typew, class Typep>
Typep Knapsack(Typep p[], Typew w[], Typew c, int n, int bestx[]) {
    Typew  W = 0;
    Typep  P = 0;
    Object  *Q = new Object[n];
    for(int i=1; i <= n; i++) {
        Q[i-1].ID = i;
        Q[i-1].d = 10*p[i]/w[i];
        P += p[i];
        W += w[i];
    }
    if(W <= c)
        return P;
    MergeSort(Q, n);
    Knap<Typew, Typep> K;
    K.p = new Typep[n+1];
    K.w = new Typew[n+1];
    K.x = new int[n+1];
    for(i=1; i <= n; i++) {
        K.p[i] = p[Q[i-1].ID];
        K.w[i] = w[Q[i-1].ID];
        K.x[i]  =0;
    }
    K.cp = 0;
    K.cw = 0;
    K.c = c;
    K.n = n;
    K.bestp = 0;
    K.bestx = bestx;
    K.Knapsack(1);
    for(i=1; i <= n; i++)
        K.x[i] = K.bestx[i];
    for(i=1; i <= n; i++)
        K.bestx[Q[i-1].ID] = K.x[i];
    delete []Q;
    delete []K.w;
    delete []K.p;
```

```
    delete []K.x;
    return K.bestp;
}
```

5-4 最大团问题的迭代回溯法。

试设计一个解最大团问题的迭代回溯算法。

分析与解答：与主教材中装载问题的迭代回溯法类似，最大团问题的迭代回溯法描述如下。

```
void Clique::iterClique() {
    for(int i=0; i <= n; i++) {
        x[i] = 0;
        i = 1;
        while(true) {
            while(i <= n && ok(i)) {
                x[i++] = 1;
                cn++;
            }
            if(i >= n) {
                for(int j=1; j <= n; j++)
                    bestx[j] = x[j];
                bestn = cn;
            }
            else
                x[i++] = 0;
            while(cn+n-i <= bestn) {
                i--;
                while(i && !x[i])
                    i--;
                if(i == 0)
                    return;
                x[i++] = 0;
                cn--;
            }
        }
    }
}
```

其中，ok()函数用于判断当前顶点是否可加入当前团。

```
bool Clique::ok(int i) {
    for(int j=1; j < i; j++)
        if(x[j] && a[i][j] == NoEdge)
            return false;
    return true;
}
```

IterClique()函数用于初始化并调用迭代回溯法求解。

```
int Clique::IterClique(int v[]) {
    x = new int[n+1];
    cn = 0;
```

```
    bestn = 0;
    bestx = v;
    IterClique();
    delete []x;
    return bestn;
}
```

5-5 旅行售货员问题的费用上界。

设 G 是一个有 n 个顶点的有向图，从顶点 i 发出的边的最大费用记为 $\max(i)$。

（1）证明旅行售货员回路的费用不超过 $\sum_{i=1}^{n}\max(i)+1$。

（2）在旅行售货员问题的回溯法中，用上面的界作为 bestc 的初始值，重写该算法，并尽可能地简化代码。

分析与解答：

（1）任一旅行售货员回路可表示为 n 个顶点的一个排列 $(\pi(1), \pi(2), \cdots, \pi(n))$。这个回路的费用为

$$h(\pi)=\sum_{i=1}^{n}a(\pi(i)),\pi(i \bmod n+1))$$

由此可知

$$h(\pi)=\sum_{i=1}^{n}a(\pi(i),\pi(i \bmod n+1)) \leqslant \sum_{i=1}^{n}\max(\pi(i))$$

$$=\sum_{i=1}^{n}\max(i)<\sum_{i=1}^{n}\max(i)+1$$

（2）对图 G 的简单遍历即可计算出 $\sum_{i=1}^{n}\max(i)+1$ 的值。

```
template<class T>
T Traveling<T>::TSP1(int v[]) {
    bestc = 1;
    for(int i=1, MaxCost=0; i <= n; i++) {
        for(int j=1; j <= n; j++)
            if(a[i][j] != NoEdge && a[i][j] > MaxCost)
                MaxCost = a[i][j];
        if(MaxCost == NoEdge)
            return NoEdge;
        bestc += MaxCost;
    }
    x = new int[n+1];
    for(i=1; i <= n; i++)
        x[i] = i;
    bestx = v;
    cc = 0;
    tSP1(2);
    delete []x;
    return bestc;
}
```

主教材的 TSP 回溯法中的语句“bestc==NoEdge;”可以删去，修改如下。

```
template<class T>
void Traveling<T>::tSP1(int i) {
   if(i == n) {
      if(a[x[n-1]][x[n]] != NoEdge && a[x[n]][1] != NoEdge && (cc+a[x[n-1]][x[n]]+a[x[n]][1]<bestc)) {
         for(int j=1; j <= n; j++)
            bestx[j] = x[j];
         bestc = cc+a[x[n-1]][x[n]] + a[x[n]][1];
      }
   }
   else {
      for(int j=i; j <= n; j++) {
         if(a[x[i-1]][x[j]] != NoEdge && (cc+a[x[i-1]][x[j]] < bestc)) {
            Swap(x[i], x[j]);
            cc += a[x[i-1]][x[i]];
            tSP1(i+1);
            cc -= a[x[i-1]][x[i]];
            Swap(x[i], x[j]);
         }
      }
   }
}
```

5-6　旅行售货员问题的上界函数。

设 G 是一个有 n 个顶点的有向图，从顶点 i 发出的边的最小费用记为 $\min(i)$。

（1）证明图 G 的所有前缀为 $x[1:i]$的旅行售货员回路的费用至少为：

$$\sum_{j=2}^{i} a(x_{j-1},x_j)+\sum_{j=i}^{n}\min(x_j)$$

式中，$a(u, v)$是边(u, v)的费用。

（2）利用上述结论设计一个高效的上界函数，重写旅行售货员问题的回溯法，并与主教材中的算法进行比较。

分析与解答：

（1）前缀为 x[1:i]的旅行售货员回路任一旅行售货员回路可表示为 n 个顶点的一个排列 $(x[1], x[2], \cdots, x[i], \pi(i+1), \pi(i+2), \cdots, \pi(n))$。这个回路的费用为

$$h(\pi)=\sum_{j=2}^{i} a(x_{j-1}, x_j)+a(x_i, \pi(i+1))+\sum_{j=i+1}^{n} a(\pi(j), \pi(j \bmod n+1))$$

由此可知

$$\begin{aligned} h(\pi) &\geqslant \sum_{j=2}^{i} a(x_{j-1}, x_j)+\min(x_i)+\sum_{j=i+1}^{n}\min(\pi(j)) \\ &=\sum_{j=2}^{i} a(x_{j-1}, \pi_j)+\sum_{j=i}^{n}\min(x_i) \end{aligned}$$

（2）先对图 G 简单遍历，计算出 $\sum_{i=1}^{n}\min(i)$ 的值。

算法实现题 5

5-1　子集和问题。

问题描述：子集和问题的一个实例为<S, c>。其中，$S=\{x_1, x_2, \cdots, x_n\}$是一个正整数的集合，$c$ 是一个正整数。子集和问题判定是否存在 S 的一个子集 $S1$，使得 $\sum_{x \in S1} x = c$。试设计一个解子集和问题的回溯法。

算法设计：对于给定的正整数的集合 $S=\{x_1, x_2, \cdots, x_n\}$和正整数 c，计算 S 的一个子集 S_1，使得 $\sum_{x \in S_1} x = c$。

数据输入：由文件 input.txt 提供输入数据。文件第 1 行有 2 个正整数 n 和 c，n 表示 S 的大小，c 是子集和的目标值。接下来的 1 行中，有 n 个正整数，表示集合 S 中的元素。

结果输出：将子集和问题的解输出到文件 output.txt。当问题无解时，输出“No Solution!”。

输入文件示例	输出文件示例
input.txt	output.txt
5 10	2 2 6
2 2 6 5 4	

分析与解答：与装载问题类似，可设计解子集和问题的回溯法如下。

```
template<class T>
bool Subsum<T>::backtrack(int i) {
  if (i > n) {
    for (int j = 1; j <= n; j++)
      bestx[j] = x[j];
    bestw = cw;
    if(bestw == c)
      return true;
    else
      return false;
    }
    r-= w[i];
    if (cw + w[i] <= c) {
      x[i] = 1;
      cw += w[i];
      if (backtrack(i+1))
        return true;
      cw-= w[i];
    }
    if (cw + r > bestw) {
      x[i] = 0;
      if(backtrack(i+1))
        return true;
    }
  }
  r += w[i];
  return false;
```

}

5-2 最小长度电路板排列问题。

问题描述：最小长度电路板排列问题是大规模电子系统设计中提出的实际问题。该问题的提法是，将 n 块电路板以最佳排列方案插入带有 n 个插槽的机箱中。n 块电路板的不同的排列方式对应于不同的电路板插入方案。

设 $B=\{1, 2, \cdots, n\}$ 是 n 块电路板的集合。集合 $L=\{N_1, N_2, \cdots, N_m\}$ 是 n 块电路板的 m 个连接块。其中每个连接块 N_i 是 B 的一个子集，且 N_i 中的电路板用同一根导线连接在一起。在最小长度电路板排列问题中，连接块的长度是指该连接块中第 1 块电路板到最后 1 块电路板之间的距离。例如，设 n=8，m=5，给定 n 块电路板及其 m 个连接块如下：

$$B=\{1, 2, 3, 4, 5, 6, 7, 8\}；\ L=\{N_1, N_2, N_3, N_4, N_5\}$$

$$N_1=\{4, 5, 6\}；\ N_2=\{2, 3\}；\ N_3=\{1, 3\}；\ N_4=\{3, 6\}；\ N_5=\{7, 8\}$$

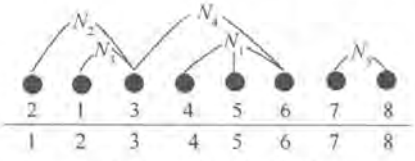

图 5-1 电路板排列

这 8 块电路板的一个可能的排列如图 5-1 所示。

在最小长度电路板排列问题中，连接块的长度是指该连接块中第 1 块电路板到最后 1 块电路板之间的距离。例如，在图 5-1 所示的电路板排列中，连接块 N_4 的第 1 块电路板在插槽 3 中，它的最后 1 块电路板在插槽 6 中，因此 N_4 的长度为 3。同理 N_2 的长度为 2。图 5-1 中的连接块最大长度为 3。

试设计一个回溯法找出所给 n 块电路板的最佳排列，使得 m 个连接块中的最大长度达到最小。

算法设计：对于给定的电路板连接块，设计一个算法，找出所给 n 个电路板的最佳排列，使得 m 个连接块中最大长度达到最小。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m（$1 \leqslant m, n \leqslant 20$）。接下来的 n 行中，每行有 m 个数。第 k 行的第 j 个数为 0 表示电路板 k 不在连接块 j 中，为 1 表示电路板 k 在连接块 j 中。

结果输出：将计算的电路板排列最小长度及其最佳排列输出到文件 output.txt。文件的第一行是最小长度；接下来的 1 行是最佳排列。

输入文件示例	输出文件示例
input.txt	output.txt
8 5	4
1 1 1 1 1	5 4 3 1 6 2 8 7
0 1 0 1 0	
0 1 1 1 0	
1 0 1 1 0 1 0 1 0 0	
1 1 0 1 0	
0 0 0 0 1	
0 1 0 0 1	

分析与解答：与主教材中电路板排列问题类似，可设计解最小长度电路板排列问题的回溯法如下。主要区别是计算连接块的长度，由算法 len 完成。

```
class Board {
   friend ArrangeBoards(int **, int, int, int []);
private:
```

```
  void Backtrack(int i);
  int len(int ii);
  int  *x, *bestx, *low, *high, bestd, n, m, **B;
};
int Board::len(int ii) {
  for (int i=1; i <= m; i++) {
    high[i] = 0;
    low[i] = n+1;
  }
  for (i=1; i <= ii; i++) {
   for (int k=1; k <= m; k++) {
    if(B[x[i]][k]) {
      if(i < low[k])
        low[k] = i;
      if(i > high[k])
        high[k] = i;
    }
  }
  int  tmp = 0;
  for (int k = 1; k <= m; k++)
    if(low[k] <= n && high[k] > 0 && tmp < high[k]-low[k])
      tmp = high[k]-low[k];
  return tmp;
}
```

回溯法实体是 Backtrack()函数。

```
void Board::Backtrack(int i) {
  if(i == n) {
    int  tmp = len(i);
    if(tmp < bestd) {
      bestd = tmp;
      for(int j=1; j <= n; j++)
        bestx[j] = x[j];
    }
  }
  else {
    for (int j = i; j <= n; j++) {
      Swap(x[i], x[j]);
      int  ld = len(i);
      if (ld < bestd)
        Backtrack(i+1);
      Swap(x[i], x[j]);
    }
  }
}
```

最后由 ArrangeBoards()函数完成计算。

```
int ArrangeBoards(int **B, int n, int m, int bestx[]) {
  Board X;
```

```
    X.x = new int[n+1];
    X.low = new int[m+1];
    X.high = new int[m+1];
    X.B  = B;
    X.n= n;
    X.m = m;
     X.bestx = bestx;
    X.bestd = n+1;
    for(int i=1; i <= n; i++)
       X.x[i] = i;
    X.Backtrack(1);
    delete []X.x;
    delete []X.low;
    delete []X.high;
    return X.bestd;
}
```

实现算法的主函数如下。

```
int main() {
    int  n, m, *p;
    fin >> n >> m;
    p = new int[n+1];
    int  **B;
    Make2DArray(B, n+1, m+1);
    for (int i =1; i <= n; i++)
       for (int j = 1; j <= m; j++)
          fin >> B[i][j];
    cout << ArrangeBoards(B, n, m, p) << endl;
    for (i=1; i <= n; i++)
       cout << p[i] <<' ';
    cout << endl;
    return 0;
}
```

5-3　最小重量机器设计问题。

问题描述：设某一机器由 n 个部件组成，每种部件都可以从 m 个不同的供应商处购得。设 w_{ij} 是从供应商 j 处购得的部件 i 的重量，c_{ij} 是相应的价格。试设计一个算法，给出总价格不超过 c 的最小重量机器设计。

算法设计：对于给定的机器部件重量和机器部件价格，计算总价格不超过 d 的最小重量机器设计。

数据输入：由文件 input.txt 给出输入数据。第一行有 3 个正整数 n、m 和 d。接下来的 $2n$ 行，每行 n 个数。前 n 行是 c，后 n 行是 w。

结果输出：将计算的最小重量及每个部件的供应商输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3 3 4	4
1 2 3	1 3 1

3 2 1
2 2 2
1 2 3
3 2 1
2 2 2

分析与解答：与背包问题类似，可设计解最小重量机器设计问题的回溯法如下。

```
template<class Typew, class Typep>
bool Machine<Typew, Typep>::backtrack(int i) {
   if (i > n) {
      bestw = cw;
      for(int j=1; j <= n; j++)
         bestx[j] = x[j];
      return true;
   }
   bool  found= false;
   if(bestw <= cc)
      found = true;
   for(int j=1; j <= m; j++) {
      x[i] = j;
      cw += w[i][j];
      cp += c[i][j];
      if (cp <= cc && cw < bestw)
         if(backtrack(i+1))
            found = true;
      cw- = w[i][j];
      cp- = c[i][j];
   }
   return found;
}
```

5-4　运动员最佳配对问题。

问题描述：羽毛球队有男女运动员各 n 人。给定 2 个 $n \times n$ 矩阵 P 和 Q。$P[i][j]$是男运动员 i 和女运动员 j 配对组成混合双打的男运动员竞赛优势；$Q[i][j]$是女运动员 i 和男运动员 j 配合的女运动员竞赛优势。由于技术配合和心理状态等各种因素影响，$P[i][j]$不一定等于 $Q[j][i]$。男运动员 i 和女运动员 j 配对组成混合双打的男女双方竞赛优势为 $P[i][j] \times Q[j][i]$。设计一个算法，计算男女运动员最佳配对法，使各组男女双方竞赛优势的总和达到最大。

算法设计：设计一个算法，对于给定的男女运动员竞赛优势，计算男女运动员最佳配对法，使各组男女双方竞赛优势的总和达到最大。

数据输入：由文件 input.txt 给出输入数据。第一行有 1 个正整数 n（$1 \leqslant n \leqslant 20$）。接下来的 $2n$ 行，每行 n 个数。前 n 行是 p，后 n 行是 q。

结果输出：将计算的男女双方竞赛优势的总和的最大值输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3	52
10 2 3	
2 3 4	

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

分析与解答：此题的解空间显然是一棵排列树，可以套用搜索排列树的回溯法框架。

```
void pref::Backtrack(int t) {
   if(t > n)
      Compute();
   else {
      for(int j = t; j <= n; j++) {
         swap(r[t], r[j]);
         Backtrack(t+1);
         swap(r[t], r[j]);
      }
   }
}
```

其中，Compute()函数计算当前配对的竞赛优势的总和。

```
void pref::Compute(void) {
   for(int i=1, temp=0; i <= n; i++)
      temp += p[i][r[i]]*q[r[i]][i];
   if(temp > best) {
      best = temp;
      for(int i=1; i <= n; i++)
         bestr[i] = r[i];
   }
}
```

5-5　无分隔符字典问题。

问题描述：设$\Sigma=(a_1,a_2,\cdots,a_n)$是 n 个互不相同的符号组成的符号集。$L_k=\{\beta_1\beta_2\cdots\beta_k \mid \beta_i\in\Sigma,\ 1\leqslant i\leqslant k\}$是$\Sigma$中字符组成的长度为 k 的字符串全体。$S\subseteq L_k$是 L_k的 1 个无分隔符字典是指对任意 $a_1a_2\cdots a_k\in S$ 和 $b_1b_2\cdots b_k\in S$，$\{a_2a_3\cdots a_kb_1, a_3a_4\cdots b_1b_2, \cdots, a_kb_1b_2\cdots b_{k-1}\}\cap S=\varnothing$。

无分隔符字典问题要求对给定的 n 和Σ及正整数 k，计算 L_k的最大无分隔符字典。

算法设计：设计一个算法，对于给定的正整数 n 和 k，计算 L_k的最大无分隔符字典。

数据输入：由文件 input.txt 给出输入数据。文件第 1 行有 2 个正整数 n 和 k。

结果输出：将计算的 L_k的最大无分隔符字典的元素个数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
2 2	2

分析与解答：用逐步加深的回溯法搜索解空间。

```
void search(int dep) {
   if(dep > lk) {
      if(s.size() > best) {
         best = s.size();
         out();
      }
```

```
        return;
    }
    if(oka(dep)) {
        s.insert(dep);
        search(dep+1);
        s.erase(dep);
    }
    search(dep+1);
}
```

其中，oka(dep)函数判断当前字符串 dep 是否可加入字典。本题中将字符串 $a_1a_2\cdots a_k$ 看作 k 位 n 进制数。当前字典中的字符串存储在集合 S 中。

```
bool oka(int b) {
    for(it=s.begin(); it != s.end(); it++) {
        int  a = *it;
        if(pref(a, b))
            return false;
    }
    return true;
}
```

pref(a, b)函数用于判断字符串 a 和 b 是否互不为前缀。

```
bool pref(int a, int b) {
    int  x= a, y = b/n;
    for(int i=0; i < k1; i++) {
        ak[k-i-2] = x%n;
        x /= n;
        ak[2*k-i-3] = y%n;
        y /= n;
    }
    for(i=1; i<  k; i++)
        if(s.find(digi(i)) !=s.end())
            return true;
    x = b, y = a/n;
    for(i=0; i < k-1; i++) {
        ak[k-i-2] = x%n;
        x /= n;
        ak[2*k-i-3] = y%n;
        y /= n;
    }
    for(i=1; i < k; i++)
        if(s.find(digi(i)) != s.end())
            return true;
    return false;
}
```

digi()函数将相应字符串转换为 n 进制数。

```
int digi(int i) {
    int  ii = k+i-2;
```

```
    int  x = ak[ii--];
    for(int j=0; j < k-1; j++) {
        x *= n;
        x += ak[ii];
        ii--;
    }
    return x;
}
```

readin()函数读入数据。

```
void readin(){
    fin >> n >> k;
    ak = new int[2*k];
    lk = n;
    for(int i=1;i<k;i++)
        lk *= n;
    lk--;
    best = 0;
}
```

实现算法的主函数如下。

```
int main() {
    readin();
    if(k < 3) {
        cout << n << endl;
        return 0;
    }
    search(0);
    cout << best << endl;
    return 0;
}
```

5-6 无和集问题。

问题描述：设 S 是正整数集合。S 是一个无和集，当且仅当 $x, y \in S$ 蕴含 $x+y \notin S$。对于任意正整数 k，如果可将 $\{1, 2, \cdots, k\}$ 划分为 n 个无和子集 $S_1, S_2, \cdots, S_n$，则称正整数 k 是 n 可分的。记 $F(n)=\max\{k \mid k$ 是 n 可分的$\}$。试设计一个算法，对任意给定的 n，计算 $F(n)$的值。

算法设计：对任意给定的 n，计算 $F(n)$的值。

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：将计算的 $F(n)$的值以及$\{1, 2, \cdots, F(n)\}$的一个 n 划分输出到文件 output.txt。文件的第 1 行是 $F(n)$的值。接下来的 n 行，每行是一个无和子集 S_i。

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

分析与解答：本题是子集选取问题，其解空间显然是一棵子集树，可以套用搜索子集树的回溯法框架。

由于搜索空间很大，用搜索时间控制搜索深度。

```
bool search(int dep) {
   t1 = clock();
   elapsed += (t1-t0) / ((double) CLOCKS_PER_SEC);
   t0 = t1;
   if(elapsed > 15.0)
      return false;
   if(dep > k) {
      out();
      return true;
   }
   for(int i=1; i <= n; i++){
      if(sum[i][dep] == 0) {
         t[dep] = i;
         s[i][dep] = true;
         for(int j=1; j < dep; j++)
            if(s[i][j])
               sum[i][dep+j]++;
         if (search(dep+1))
            return true;
         s[i][dep] = false;
         t[dep] = 0;
         for(j=1; j < dep; j++)
            if(s[i][j])
               sum[i][dep+j]--;
      }
   }
   return false;
}
```

5-7　n 色方柱问题。

问题描述：设有 n 个立方体，每个立方体的每面用红、黄、蓝、绿等 n 种颜色之一染色。要把这 n 个立方体叠成一个方形柱体，使得柱体的 4 个侧面的每侧均有 n 种不同的颜色。试设计一个回溯算法，计算出 n 个立方体的一种满足要求的叠置方案。

算法设计：对于给定的 n 个立方体以及每个立方体各面的颜色，计算出 n 个立方体的一种叠置方案，使得柱体的 4 个侧面的每一侧均有 n 种不同的颜色。

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（$0<n<27$），表示给定的立方体个数和颜色数均为 n。第 2 行是 n 个大写英文字母组成的字符串。该字符串的第 k（$0\leqslant k<n$）个字符代表第 k 种颜色。接下来的 n 行中，每行有 6 个数，表示立方体各面的颜色。立方体各面的编号如图 5-2 所示。

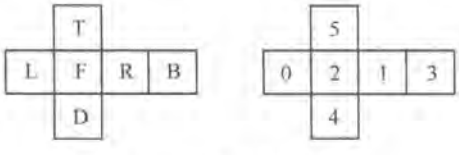

图 5-2　立方体各面的编号

图 5-2 中 F 表示前面，B 表示背面，L 表示左面，R 表示右面，T 表示顶面，D 表示底面。相应地，2 表示前面，3 表示背面，0 表示左面，1 表示右面，5 表示顶面，4 表示底面。

例如，在示例输出文件中，第 3 行的 6 个数 0、2、1、3、0、0 分别表示第 1 个立方体

的左面的颜色为 R，右面的颜色为 B，前面的颜色为 G，背面的颜色为 Y，底面的颜色为 R，顶面的颜色为 R。

结果输出：将计算的 n 个立方体的一种可行的叠置方案输出到文件 output.txt。每行 6 个字符，表示立方体各面的颜色。如果不存在所要求的叠置方案，输出“No Solution!”。

输入文件示例	输出文件示例
input.txt	output.txt
4	RBGYRR
RGBY	YRBGRG
0 2 1 3 0 0	BGRBGY
3 0 2 1 0 1	GYYRBB
2 1 0 2 1 3	
1 3 3 0 2 2	

分析与解答：

（1）算法思想

每个立方体可以按 3 个方向旋转，每个方向有 4 个不同的面，因此每个立方体可有 64 种不同状态。用回溯法对 n 个立方体的每种状态进行搜索，可以找到满足要求的叠置方案。然而，这样做的计算量较大。下面讨论用图论的方法进行简化。

在本问题中，立方体的每对相对的面的颜色是要考察的关键因素。将每个立方体表示为有 n 个顶点的图。图中每个顶点表示一种颜色。在立方体每对相对面的顶点间连一条边。例如，图 5-3(b)是图 5-3(a)所示的 4 个立方体所相应的子图。

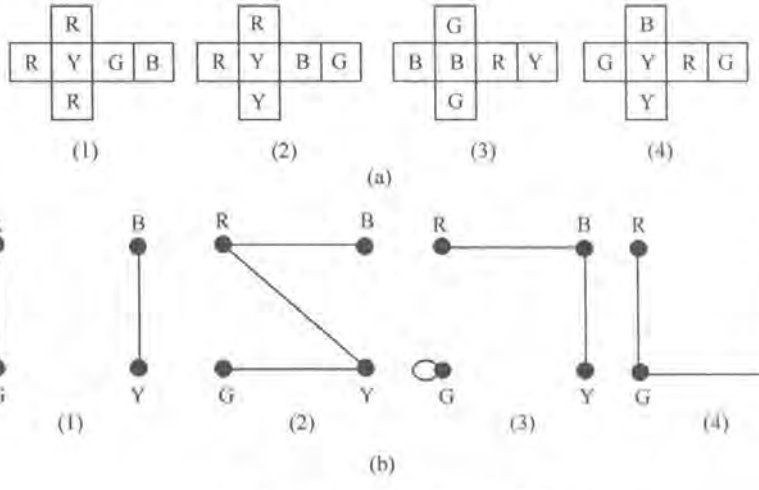

图 5-3　立方体及其相应的子图

将上述子图合并，并标明每一条边来自哪一个立方体，如图 5-4 所示。

下一步在构成的图中，找出 2 个特殊子图。一个子图表示叠置的 n 个立方体的前侧面与背侧面，另一子图表示叠置的 n 个立方体的左侧面与右侧面。这两个子图应满足下述性质：① 每个子图有 n 条边，且每个立方体恰好一条边；② 2 个子图没有公共边；③ 子图中每个顶点的度均为 2。对于图 5-4 中的图，找出满足要求的两个子图如图 5-5 所示。

给子图的每条边一个方向，使每个顶点有一条出边和一条入边。有向边的始点对应于前面和左面；有向边的终点对应于背面和右面。图 5-5 给出的满足要求的解如表 5-1 所示。

上述算法的关键是用回溯法找出满足性质①、②和③的子图。

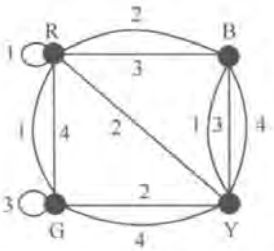

图 5-4　n 个立方体及其相应的子图

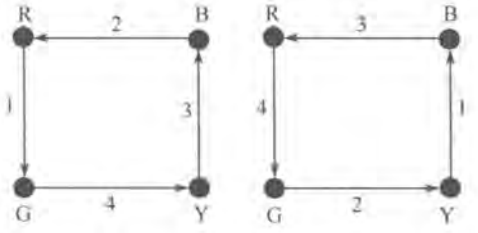

图 5-5　表示 n 个立方体 4 个侧面的子图

表 5-1　满足要求的解

	0（L）	1（R）	2（F）	3（B）
Cube1	Y	B	G	R
Cube2	G	Y	R	B
Cube3	B	R	B	Y
Cube4	R	G	Y	G

（2）算法实现

二维数组 board[n][6]存储 n 个立方体各面的颜色，solu[n][6]存储解。找满足性质①、②和③的子图的回溯法如下。

```
void search() {
   int  i, t, cube, newg, over, ok;
   int  *vert = new int[n];
   int  *edge = new int[n*2];
   for(i=0; i < n; i++)
      vert[i] = 0;
   t = -1;
   newg = 1;
   while(t > -2) {
      t++;
      cube = t%n;                                    // 每个立方体找 2 次
      if(newg)
         edge[t] = -1;
      over = 0;
      ok = 0;
      while(!ok && !over) {
         edge[t]++;
         if(edge[t]>  2)
            over = 1;                                // 每个立方体只有 3 条边
         else
            ok = (t<n || edge[t] != edge[cube]);     // 是否已用过
      }
      if(!over) {
         if(++vert[board[cube][edge[t]*2]]>2+t/n*2)
            ok = 0;
         if(++vert[board[cube][edge[t]*2+1]]>2+t/n*2)
            ok = 0;
         if(t%n == n-1 && ok)                        // check that each vertex is order 2
```

```
            for(i=0; i<n; i++)
               if(vert[i]>2+t/n*2)
                  ok = 0;
            if(ok) {
               if(t == n*2-1) {                    // 找到解
                  ans++;
                  out(edge);
                  return;
               }
               else
                  newg = 1;
            }                                      // ok
            else {                                 // 取下一条边
               --vert[board[cube][edge[t]*2]];
               --vert[board[cube][edge[t]*2+1]];
               t--;
               newg = 0;
            }
         }                                         // over
         else {                                    // 回溯
            t--;
            if(t>-1) {
               cube = t%n;
               --vert[board[cube][edge[t]*2]];
               --vert[board[cube][edge[t]*2+1]];
            }
            t--;
            newg = 0;
         }
      }
   }
```

找到一个解后由 out()函数输出。

```
void out(int edge[]){
   int  k, a, b, c, d;
   for(int i=0; i < 2; i++) {
      for(int j=0; j < n; j++)
         used[j] = 0;
      do {
         j = 0;                                    // 找下一条未用边
         d = c = -1;
         while(j < n && used[j])
            j++;
         if(j < n) {
            do {
               a = board[j][edge[i*n+j]*2];
               b = board[j][edge[i*n+j]*2+1];
               if(b == d) {
                  k = a;
```

```
                a = b;
                b = k;
            }
            solu[j][i*2] = a;
            solu[j][i*2+1] = b;
            used[j] = 1;
            if(c < 0)                          // 开始顶点
               c = a;
            d = b;
            for(k=0; k < n; k++)               // 找下一个立方体
               if(!used[k] && (board[k][edge[i*n+k]*2] == b || board[k][edge[i*n+k]*2+1] == b))
                  j = k;
         } while(b != c);
      } while(j < n);
   }
   for(int j=0; j < n; j++) {
      k = 3-edge[j]-edge[j+n];
      a = board[j][k*2];
      b = board[j][k*2+1];
      solu[j][4] = a;
      solu[j][5] = b;
   }
   for (i=0; i < n; i++){
      for (j=0; j < 6; j++)
         cout << color[solu[i][j]];
      cout<<endl;
   }
  }
}
```

执行算法的主函数如下。

```
int main() {
   readin();
   search();
   if(ans == 0)
      cout << "No Solution!" << endl;
   return 0;
}
```

初始数据由 readin()函数读入。

```
void readin() {
   fin >> n;
   Make2DArray(board, n, 6);
   Make2DArray(solu, n, 6);
   color = new char[n];
   used = new int[n];
   for(int j=0; j < n; j++)
      fin >> color[j];
   for(int i=0; i < n; i++)
```

```
    for(int j=0; j < 6; j++)
      fin >> board[i][j];
}
```

5-8　整数变换问题。

问题描述：关于整数 i 的变换 f 和 g 定义如下：$f(i)=3i$，$g(i)=\lfloor i/2 \rfloor$。

试设计一个算法，对于给定的 2 个整数 n 和 m，用最少的 f 和 g 变换次数将 n 变换为 m。例如，可以将整数 15 用 4 次变换将它变换为整数 4：$4=gfgg(15)$。当整数 n 不可能变换为整数 m 时，算法应如何处理？

算法设计：对任意给定的整数 n 和 m，计算将整数 n 变换为整数 m 所需要的最少变换次数。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m。

结果输出：将计算的最少变换次数以及相应的变换序列输出到文件 output.txt。文件的第 1 行是最少变换次数。文件的第 2 行是相应的变换序列。

输入文件示例	输出文件示例
input.txt	output.txt
15 4	4
	gfgg

分析与解答：此题是 $3n+1$ 问题的变形。为了找最短变换序列，用逐步加深的回溯法搜索。

```
void compute() {
  k = 1;
  while(!search(1, n)) {
    k++;
    if(k > maxdep)
      break;
    init();
  }
  if(found)
    output();
  else
    cout << "No Solution!" << endl;
}
```

search()函数实现回溯搜索。

```
bool search(int dep, int n) {
  if(dep>k)
    return false;
  for(int i=0; i < 2; i++) {
    int  n1 = f(n, i);
    t[dep] = i;
    if(n1==m || search(dep+1, n1)) {
      found = true;
      out();
      return true;
    }
  }
  return false;
```

```
}
```

5-9 拉丁矩阵问题。

问题描述：现有 n 种不同形状的宝石，每种宝石有足够多颗。欲将这些宝石排列成 m 行 n 列的一个矩阵，$m \leq n$，使矩阵中每行和每列的宝石都没有相同形状。试设计一个算法，计算出对于给定的 m 和 n，有多少种不同的宝石排列方案。

算法设计：对于给定的 m 和 n，计算出不同的宝石排列方案数。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 m 和 n（$0<m \leq n<9$）。

结果输出：将计算的宝石排列方案数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3 3	12

分析与解答：

（1）算法思想

设 n 种宝石编号为 1, 2, …, n，宝石矩阵的第 1 行从左到右排列为 1, 2, …, n，且第 1 列从上到下排列为 1, 2, …, m 的阵列为标准拉丁矩阵，m 行 n 列的标准拉丁矩阵个数为 $L(m, n)$。一般情况下，m 行 n 列的拉丁矩阵个数为 $R(m, n)$。本题要求 $R(m, n)$。

容易证明，$R(m, n) = n!(n-1)!L(m, n)/(n-m)!$，于是问题可转化为求标准拉丁矩阵个数 $L(m, n)$。问题显然与排列有关，可用主教材中的排列树回溯法框架求解。

（2）算法实现

二维数组 board[m][n]存储宝石矩阵。每行初始化为单位排列，第 1 列从上到下排列为 1, 2, …, m。

```
void init() {
   fin >> m >> n;
   cout<<m<<" "<<n<<endl;
   for(int i=1; i <= n; ++i)
      for(int j=1; j <= n; ++j)
         board[i][j] = j;
   for(i=2; i <= n; ++i)
      Swap(board[i][1], board[i][i]);
   if(m == n)
      m--;
}
```

backtrack()函数对数组 board 从上到下、从左到右递归搜索。

```
void backtrack(int r, int c) {
   for(int i=c; i <= n; i++)
      if(ok(r, c, board[r][i])) {
         Swap(board[r][c], board[r][i]);
         if(c == n) {
            if(r == m)
               count += 1.0;
            else
               backtrack(r+1, 2);
         }
```

```
        else
            backtrack(r, c+1);
        Swap(board[r][c], board[r][i]);
      }
   }
}
```

其中，ok()函数用于判断在当前列中宝石是否重复。

```
int ok(int r, int c, int k) {
   for(int i=1; i < r; i++)
      if(board[i][c] == k)
         return 0;
   return 1;
}
```

执行算法的主函数如下。

```
int main() {
   init();
   backtrack(2, 2);
   outlong(count);
   return 0;
}
```

其中，outlong()函数按公式 $R(m, n)$计算输出 $R(m, n)=n!(n-1)!L(m, n)/(n-m)!$的值。由于输出的值较大，这一步需要高精度计算。

注意，当 $m=n$ 时，第 $n-1$ 行排定后，第 n 行就已确定，无须回溯。这就是 init()函数中的语句“if(m == n)　　m—;”的含义。当然，还有其他优化方法。

5-10　排列宝石问题。

问题描述：现有 n 种不同形状的宝石，每种 n 颗，共 n^2 颗。同一种形状的 n 颗宝石分别具有 n 种不同的颜色 $c_1, c_2, \cdots, c_n$ 中的一种颜色。欲将这 n^2 颗宝石排列成 n 行 n 列的一个方阵，使方阵中每行和每列的宝石都有 n 种不同形状和 n 种不同颜色。试设计一个算法，计算出对于给定的 n，有多少种不同的宝石排列方案。

算法设计：对于给定的 n，计算出不同的宝石排列方案数。

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（$0<n<9$）。

结果输出：将计算的宝石排列方案数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
1	1

分析与解答：

（1）算法思想

本题与算法实现题 5-9 类似，用回溯法时，需对形状和颜色两种因素循环考察。

（2）算法实现

二维数组 a[n][n]、b[n][n]分别存储宝石形状矩阵和颜色矩阵。每行初始化为单位排列。二维数组 cc[n][n]的单元 cc[i][j]用于搜索时记录形状为 i，颜色为 j 的宝石是否已用过，初始化为 0。

```
void init() {
   fin >> n;
   for(int i=1; i <= n; ++i) {
      for(int j=1; j <= n; ++j) {
         a[i][j] = j;
         b[i][j] = j;
         cc[i][j] = 0;
      }
   }
}
```

backtrack()函数对数组 a 和 b 从上到下、从左到右递归搜索。

```
void backtrack(int r, int c) {
   for(int i=c; i <= n; i++) {
      if(ok(r, c, i, 0)) {
         Swap(a[r][c], a[r][i]);
         for(int j=c; j <= n; j++) {
            if(ok(r, c, j, 1)) {
               Swap(b[r][c], b[r][j]);
               cc[a[r][c]][b[r][c]] = 1;
               if(c == n) {
                  if(r == n)
                     count += 1.0;
                  else
                     backtrack(r+1, 1);
               }
               else
                  backtrack(r, c+1);
               cc[a[r][c]][b[r][c]] = 0;
               Swap(b[r][c], b[r][j]);
            }
            Swap(a[r][c], a[r][i]);
         }
      }
   }
}
```

其中，ok()函数用于判断在当前列中宝石的形状和颜色是否重复。

```
int ok(int r, int c, int k, int fla) {
   if(fla) {
      if(cc[a[r][c]][b[r][k]])
         return 0;
      for(int i=1; i < r; i++)
         if(b[i][c] == b[r][k])
         return 0;
   }
   else {
      for(int i=1; i < r; i++)
         if(a[i][c] == a[r][k])
```

```
            return 0;
        }
    return 1;
}
```

执行算法的主函数如下。

```
int main() {
    init();
    backtrack(1, 1);
    cout << (int)count << endl;
    return 0;
}
```

与算法实现题 5-9 一样，注意到第 $n-1$ 行排定后，第 n 行就已确定，无须回溯，但必须判断是否矛盾。这个任务可由 last()函数完成。

```
int last() {
    for(int j=1; j <= n; j++) {
        for(int i=1; i <= n; i++) {
            dd[i][0] = 0;
            dd[i][1] = 0;
        }
        for(i=1; i < n; i++) {
            dd[a[i][j]][0] = 1;
            dd[b[i][j]][1] = 1;
        }
        for(i=1; i <= n; i++) {
            if(dd[i][0] == 0)
                ee[j][0] = i;
            if(dd[i][1] == 0)
                ee[j][1] = i;
        }
    }
    for(int i=1; i <= n; i++)
        if(cc[ee[i][0]][ee[i][1]])
            return 0;
    return 1;
}
```

最后，将回溯法中的语句“if(r == n)　　count += 1.0;”换成“if(r == n-1) {　if(last()) count += 1.0; }”。

5-11　重复拉丁矩阵问题。

问题描述：现有 k 种不同价值的宝石，每种宝石都有足够多颗。欲将这些宝石排列成一个 m 行 n 列的矩阵，$m \leqslant n$，使矩阵中每行和每列的同一种宝石数都不超过规定的数量。另规定，宝石阵列的第 1 行从左到右和第 1 列从上到下的宝石按宝石的价值最小字典序从小到大排列。试设计一个算法，对于给定的 k、m 和 n 以及每种宝石的规定数量，计算出有多少种不同的宝石排列方案。

算法设计：对于给定的 m、n 和 k，以及每种宝石的规定数量，计算出不同的宝石排列

方案数。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 3 个正整数 m、n 和 k（$0<m\leq n<9$）。第 2 行有 k 个数，第 j 个数表示第 j 种宝石在矩阵的每行和每列出现的最多次数。这 k 个数按照宝石的价值从小到大排列。设这 k 个数为 $1\leq v_1\leq v_2\leq\cdots\leq v_k$，则 $v_1+v_2+\cdots+v_k=n$。

结果输出：将计算的宝石排列方案数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
4 7 3	
2 2 3	84309

分析与解答：

（1）算法思想

本题与前两题类似，用回溯法时，需考察相同价值的情况。

（2）算法实现

二维数组 board[m][n]存储宝石矩阵。每行初始化为单位排列，第 1 列从上到下排列为 1, 2, …, m。用数组 mv 记录规定的每种宝石的重复数，mu 记录 n 个宝石中，每个宝石的价值序号。由 init()函数初始化各数组。

```
void init() {
   fin >> m>> n >> mm;
   for(int k=1, j=1, t=0; k <= mm; k++) {
      fin >> t;
      mv[k] = t;
      while(t) {
         mu[j++] = k;
         t--;
      }
   }
   for(int i=1; i <= n; ++i)
      for(int j=1; j <= n; ++j)
         board[i][j] = j;
   for(i=2; i <= n; ++i)
      Swap(board[i][1], board[i][i]);
}
```

backtrack()函数对数组 board 从上到下、从左到右递归搜索。

```
void backtrack(int r,int c) {
   for(int i=c; i <= n; i++)
      if(ok(r, c, i)) {
         Swap(board[r][c], board[r][i]);
         if( c== n) {
            if(r == m)
               count += 1.0;
            else
               backtrack(r+1, 2);
         }
         else
            backtrack(r,c+1);
```

```
            Swap(board[r][c], board[r][i]);
        }
    }
}
```

其中，ok()函数用于判断在当前列中宝石是否超过规定数。

```
int ok(int r, int c, int s) {
    int k = board[r][s];
    if(s > c)
        for(int t=c; t < s; t++)
            if(mu[board[r][t]] == mu[k])
                return 0;
    for (int i=1, j=0; i < r; i++)
        if(mu[board[i][c]] == mu[k])
            j++;
    if(j > mv[mu[k]]-1)
        return 0;
    else
        return 1;
}
```

执行算法的主函数如下。

```
int main() {
    init();
    backtrack(2, 2);
    cout << (int)count << endl;
    return 0;
}
```

5-12 罗密欧与朱丽叶的迷宫问题。

问题描述：罗密欧与朱丽叶身处一个 $m \times n$ 的方格迷宫中，如图 5-6 所示。每个方格表示迷宫中的一个房间。这 $m \times n$ 个房间中有一些房间是封闭的，不允许任何人进入。在迷宫中任何位置均可沿 8 个方向进入未封闭的房间。罗密欧位于迷宫的(p, q)方格中，他必须找出一条通向朱丽叶所在的(r, s)方格的路。在抵达朱丽叶之前，他必须对所有未封闭的房间各走一次，而且要使到达朱丽叶的转弯次数为最少。每改变一次前进方向算作转弯一次。请设计一个算法，帮助罗密欧找出这样一条道路。

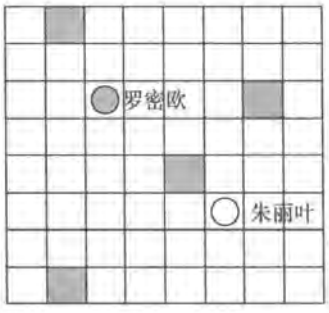

图 5-6 罗密欧与朱丽叶的迷宫

算法设计：对于给定的罗密欧与朱丽叶的迷宫，计算罗密欧通向朱丽叶的所有最少转弯道路。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 3 个正整数 n、m、k，分别表示迷宫的行数、列数和封闭的房间数。接下来的 k 行中，每行 2 个正整数，表示被封闭的房间所在的行号和列号。最后的 2 行，每行也有 2 个正整数，分别表示罗密欧所处的方格(p, q)和朱丽叶所处的方格(r, s)。

结果输出：将计算的罗密欧通向朱丽叶的最少转弯次数和有多少条不同的最少转弯道路输出到文件 output.txt。文件的第 1 行是最少转弯次数。第 2 行是不同的最少转弯道路数。接

下来的 n 行每行 m 个数，表示迷宫的一条最少转弯道路。A[i][j]=k 表示第 k 步到达方格(i, j)；A[i][j]=−1 表示方格(i, j)是封闭的。

如果罗密欧无法通向朱丽叶，则输出“No Solution!”。

输入文件示例	输出文件示例
input.txt	output.txt
3 4 2	6
1 2	7
3 4	1 −1 9 8
1 1	2 10 6 7
2 2	3 4 5 −1

分析与解答：在当前位置按照 8 个方向搜索。

```
void search (int dep, int x, int y, int di) {
  if(dep == m*n-k && x == x1 && y == y1 && dirs <= best) {
    if(dirs. < best) {
      best = dirs;
      count = 1;
      save();
    }
    else
      count++;
    return;
  }
  if(dep == m*n-k || x == x1 && y == y1 || dirs. > best)
    return;
  else {
    for(int i = 1; i <= 8; i++) {
      if(stepok(x+dx[i], y+dy[i])) {
        board[x+dx[i]][y+dy[i]] = dep+1;
        if(di != i)
          dirs++;
        search(dep+1, x+dx[i], y+dy[i], i);
        if(di!= i)
          dirs--;
        board[x+dx[i]][y+dy[i]] = 0;
      }
    }
  }
}
```

stepok()函数用于判断是否越界。

```
bool stepok(int x,int y) {
  return (x>0 && x<=n && y>0 && y<=m && board[x][y]==0);
}
```

save()函数保存找到的解。

```
void save() {
  for(int i = 1; i <= n; i++)
    for(int j = 1; j <= m; j++)
```

```
        bestb[i][j] = board[i][j];
}
```

对于当前位置还可加入剪枝函数 live()提早判断无解，进行剪枝。

```
bool live(int x, int y, int dep) {
   int  nm = n*m;
   if(d[x][y] > 1 && endpoint > 1)
      return false;
   for(int j=1; j <= 8; j++){
      int  p = x+dx[j], q = y+dy[j];
      if(stepok(p, q) && d[p][q] < 2 && dep < nm-k-2)
         return false;
   }
   return true;
}
```

加入剪枝函数后的回溯法如下。

```
void search(int dep, int x, int y, int di) {
   if(dep == m*n-k && x == x1 && y == y1 && dirs. <= best) {
      if(dirs. < best) {
         best = dirs;
         count = 1;
         save();
      }
      else
         count++;
      return;
   }
   if(dep == m*n-k || x == x1 && y == y1 || dirs> best)
      return;
   else {
      for(int i = 1; i <= 8; i++) {
         int  p = x+dx[i], q = y+dy[i];
         if(stepok(p, q) && live(p, q, dep)) {
            save(p, q, dep);
            if(di != i)
               dirs++;
            search(dep+1, p, q, i);
            if(di!= i)
               dirs--;
            restore(p, q);
         }
      }
   }
}
```

5-13　工作分配问题。

问题描述：设有 n 件工作分配给 n 个人。将工作 i 分配给第 j 个人所需的费用为 c_{ij}。试设计一个算法，为每个人都分配 1 件不同的工作，并使总费用达到最小。

算法设计：设计一个算法，对于给定的工作费用，计算最佳工作分配方案，使总费用达到最小。

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（$1 \leqslant n \leqslant 20$）。接下来的 n 行，每行 n 个数，表示工作费用。

结果输出：将计算的最小总费用输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3	9
10 2 3	
2 3 4	
3 4 5	

分析与解答：此题的解空间显然是一棵排列树，可以套用搜索排列树的回溯法框架。

```
void job::Backtrack(int t) {
  if(t > n)
    Compute();
  else {
    for(int j = t; j <= n; j++) {
      swap(r[t], r[j]);
      Backtrack(t+1);
      swap(r[t], r[j]);
    }
  }
}
```

其中，Compute()函数计算当前方案的费用。

```
void job::Compute(void) {
  for(int i=1, temp=0; i <= n; i++)
    temp += p[i][r[i]];
  if(temp < best) {
    best = temp;
    for(int i=1; i <= n; i++)
      bestr[i] = r[i];
  }
}
```

5-14 布线问题。

问题描述：假设要将一组元件安装在一块线路板上，为此需要设计一个线路板布线方案。各元件的连线数由连线矩阵 conn 给出。元件 i 和元件 j 之间的连线数为 conn(i,j)。如果将元件 i 安装在线路板上位置 r 处，而将元件 j 安装在线路板上位置 s 处，则元件 i 和元件 j 之间的距离为 dist(r,s)。确定了所给的 n 个元件的安装位置，就确定了一个布线方案。与此布线方案相应的布线成本为 $\text{dist}(r,s)\times\sum_{1\leqslant i\leqslant j\leqslant n}\text{conn}(i,j)$。试设计一个算法，找出所给 n 个元件的布线成本最小的布线方案。

算法设计：设计一个算法，对于给定的 n 个元件，计算最佳布线方案，使布线费用达到最小。

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（$1 \leqslant n \leqslant 20$）。接下来的 $n-1$ 行，每行 $n-i$ 个数，表示元件 i 和元件 j 之间连线数（$1 \leqslant i < j \leqslant 20$）。

结果输出：将计算的最小布线费用以及相应的最佳布线方案输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3	10
2 3	1 3 2
3	

分析与解答：与主教材中的电路板排列问题类似。回溯法如下。

```
void Board::Backtrack(int i) {
   if(i == n) {
      int  tmp = len(i);
      if(tmp < bestd) {
         bestd = tmp;
         for (int j=1; j <= n; j++)
            bestx[j] = x[j];
      }
   }
   else {
      for(int j = i; j <= n; j++) {
         Swap(x[i], x[j]);
         int  ld = len(i);
         if(ld < bestd)
            Backtrack(i+1);
         Swap(x[i], x[j]);
      }
   }
}
```

len()函数计算布线费用。

```
int Board::len(int ii) {
   for(int i=1, sum=0; i <= ii; i++) {
      for(int j=i+1; j <= ii; j++) {
         int  dist = x[i]>x[j] ? x[i]-x[j] : x[j]-x[i];
         sum += conn[i][j]*dist;
      }
   }
   return sum;
}
```

5-15　最佳调度问题。

问题描述：假设有 n 个任务由 k 个可并行工作的机器来完成，完成任务 i 需要的时间为 t_i。试设计一个算法，找出完成这 n 个任务的最佳调度，使得完成全部任务的时间最早。

算法设计：对任意给定的整数 n 和 k，以及完成任务 i 需要的时间为 t_i（i=1～n）。计算完成这 n 个任务的最佳调度。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 k。第 2 行的 n 个正整数是完成 n 个任务需要的时间。

结果输出：将计算的完成全部任务的最早时间输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
7 3	17
2 14 4 16 6 5 3	

分析与解答：简单回溯搜索。

```
void search(int dep) {
  if(dep == n) {
    int  tmp = comp();
    if(tmp < best)
      best = tmp;
    return;
  }
  for(int i=0; i < k; i++) {
    len[i] += t[dep];
    if(len[i] < best)
      search(dep+1);
    len[i] -= t[dep];
  }
}
```

comp()函数计算完成任务的时间。

```
int comp() {
  int  tmp = 0;
  for(int i=0; i < k; i++)
    if(len[i] > tmp)
      tmp = len[i];
  return tmp;
}
```

5-16　无优先级运算问题。

问题描述：给定 n 个正整数和 4 个运算符+、−、*、/，且运算符无优先级，如 2+3×5=25。对于任意给定的整数 m，试设计一个算法，用以上给出的 n 个数和 4 个运算符，产生整数 m，且用的运算次数最少。给出的 n 个数中每个数最多只能用 1 次，但每种运算符可以任意使用。

算法设计：对于给定的 n 个正整数，设计一个算法，用最少的无优先级运算次数产生整数 m。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m。第 2 行是给定的用于运算的 n 个正整数。

结果输出：将计算的产生整数 m 的最少无优先级运算次数以及最优无优先级运算表达式输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
5 25	2
5 2 3 6 7	2+3*5

分析与解答：readin()函数读入初始数据。

```
void readin() {
   fin >> n >> m;
   a = new int[n];
   num = new int[n];
   oper = new int[n];
   flag = new int[n];
   for(int i=0; i < n; i++) {
      fin >> a[i];
      flag[i] = 0;
   }
}
```

采用迭代加深的回溯法如下。

```
bool search(int dep) {
   if(dep>k) {
      if(found())
         return true;
      else
         return false;
   }
   for(int i=0; i < n; i++) {
      if(flag[i] == 0) {
         num[dep] = a[i];
         flag[i] = 1;
         for(int j=0; j < 4; j++) {
            oper[dep]= j;
            if(search(dep+1))
               return true;
         }
         flag[i]=0;
      }
   }
   return false;
}
```

found()函数判断是否找到解。

```
bool found(){
   int  x = num[0];
   for(int i=0; i < k; i++) {
      switch (oper[i]) {
         case 0:  x+=num[i+1];  break;
         case 1:  x=num[i+1];  break;
         case 2:  x*=num[i+1];  break;
         case 3:  x/=num[i+1];  break;
      }
   }
   return (x == m);
}
```

实现算法的主函数如下。

```
int main() {
   create();
   readin();
   for(k=0; k < n; k++) {
      if(search(0)) {
         cout << k << endl;
         out();
         return 0;
      }
   }
   cout << "No Solution!" << endl;
   return 0;
}
```

5-17 世界名画陈列馆问题。

问题描述：世界名画陈列馆由 $m \times n$ 个排列成矩形阵列的陈列室组成。为了防止名画被盗，需要在陈列室中设置警卫机器人哨位。除了监视所在的陈列室，每个警卫机器人还可以监视与它所在的陈列室相邻的上、下、左、右 4 个陈列室。试设计一个安排警卫机器人哨位的算法，使名画陈列馆中每个陈列室都在警卫机器人的监视下，且所用的警卫机器人数最少。

算法设计：设计一个算法，计算警卫机器人的最佳哨位安排方案，使名画陈列馆中每个陈列室都在警卫机器人的监视下，且所用的警卫机器人数最少。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 m 和 n（$1 \leqslant m, n \leqslant 20$）。

结果输出：将计算的警卫机器人数及其最佳哨位安排输出到文件 output.txt。文件的第 1 行是警卫机器人数；接下来的 m 行中每行 n 个数，0 表示无哨位，1 表示哨位。

输入文件示例	输出文件示例
input.txt	output.txt
4 4	4
	0 0 1 0
	1 0 0 0
	0 0 0 1
	0 1 0 0

分析与解答：

（1）状态空间树

本题的状态空间树是一棵子集树，可用对子集树进行搜索的回溯算法框架求解。依从上到下、从左到右的顺序依次考察每一个陈列室设置警卫机器人哨位的情况，以及该陈列室受警卫机器人监视的情况。用 $x[i,j]$ 表示陈列室 (i,j) 当前设置警卫机器人哨位的状态。当 $x[i,j]=1$ 时，表示已经在陈列室 (i,j) 设置了警卫机器人哨位。当 $x[i,j]=0$ 时，表示陈列室 (i,j) 尚未设置警卫机器人哨位。另，用 $y[i,j]$ 表示陈列室 (i,j) 当前受警卫机器人监视的状态。当 $y[i,j]=1$ 时，表示陈列室 (i,j) 已受警卫机器人监视。当 $y[i,j]=0$ 时，表示陈列室 (i,j) 尚未受警卫机器人监视。

（2）采用剪枝技术提高回溯法的效率

① 下界剪枝法。设当前已设置的警卫机器人哨位数为 k，已受警卫机器人监视的陈列室数为 t，当前最优警卫机器人哨位数为 best。在一般情况下，可根据 k 和 t 的值，估计出尚

需设置的警卫机器人哨位数下界$f(k,t)$。当$f(k,t)\geq$best时，可将状态空间树中以当前结点为根的子树剪去。

② 控制剪枝法。设p和q是状态空间树中2个不同的结点。如果按照结点p和q的某一相关关系，可以确定以结点q为根的子树中的解不优于以结点p为根的子树中的解，则称结点p控制了结点q。对于本题来说，可考虑以下的结点控制关系。

<a> 已受监视结点的控制关系。设在回溯搜索时当前所关注的是陈列室(i,j)。该陈列室已受监视，即$y[i,j]=1$。与其相邻的其他陈列室的受监视状态如图5-7所示。

此时在陈列室(i,j)处设置一个警卫机器人哨位，即取$x[i,j]=1$，相应于状态空间树中的一个结点q。在陈列室$(i+1,j+1)$处设置一个警卫机器人哨位，即取$x[i+1,j+1]=1$，相应于状态空间树中的另一个结点p。容易看出，此时以结点q为根的子树中的解不优于以结点p为根的子树中的解，即结点p控制了结点q。可将状态空间树中以结点q为根的子树剪去。由此总结出，在以从上到下、从左到右的顺序依次考察每一个陈列室时，已受监视的陈列室处不必设置警卫机器人哨位。这样可以避免许多无效搜索。

<b> 测试结点的控制关系。设陈列室(i,j)是当前以从上到下、从左到右的顺序搜索遇到的第一个未受监视的陈列室。为了使陈列室(i,j)受到监视，可在陈列室$(i+1,j)$、(i,j)、$(i,j+1)$处设置警卫机器人哨位。在这3处设置哨位所相应的状态空间树中的结点分别为p、q和r，如图5-8所示。

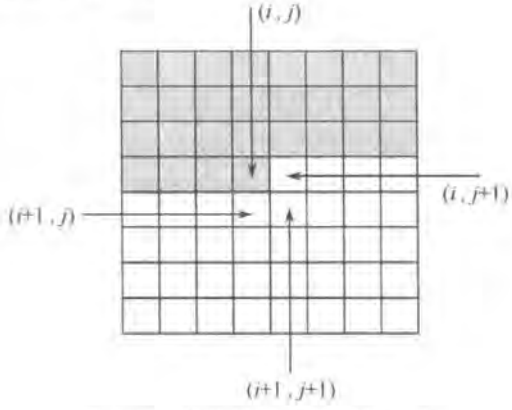

图5-7 受监视结点的控制关系

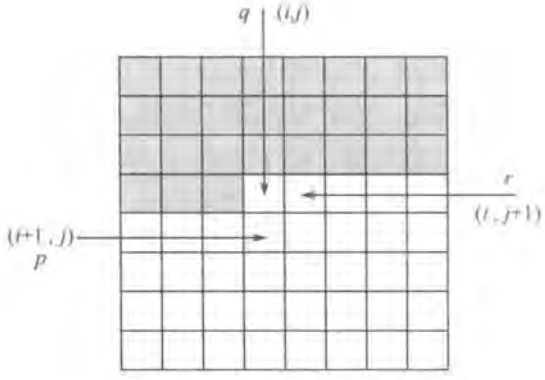

图5-8 测试结点的控制关系

显而易见，当$y(i,j+1)=1$时，结点p控制结点q；当$y(i,j+1)=1$且$y(i,j+2)=1$时，结点p控制结点r。因此，在搜索时应按p→q→r的顺序来扩展结点，并检测结点p对结点q和结点r的控制条件，及时剪去受控结点相应的子树。

（3）简化边界条件

在陈列室矩形阵列的外圈扩展一层，即增加0行和0列、$n+1$行和$m+1$列，使算法可以统一地处理边界条件。

根据前面的分析，可以设计安排警卫机器人哨位的回溯法如下。

算法中用到的变量类型说明如下。

```
#define MLEN 50
int  d[6][3] = {{0,0,0}, {0,0,0}, {0,0,-1}, {0,-1,0}, {0,0,1}, {0,1,0}};
int  x[MLEN+1][MLEN+1], y[MLEN+1][MLEN+1], bestx[MLEN+1][MLEN+1];
int  n, m, best, k=0, t=0, t1, t2, more;
```

```
bool p;
```

回溯法的主体由 search(i, j)来实现，参数 i 和 j 表示当前搜索位置。

```
void search(int i, int j) {
  do {
    j++;
    if(j > m) {
      i++;
      j = 1;
    }
  }while(!((y[i][j] == 0) || (i > n)));
  if(i > n) {
    if(k < best) {
      best=k;
      copy(bestx, x);
    }
    return;
  }
  if(k+(t1-t)/5 >= best)
    return;
  if((i < n-1) && (k+(t2-t)/5 >= best))
    return;
  if((i < n)) {
    change(i+1, j);
    search(i, j);
    restore(i+1, j);
  }
  if((j < m) && ((y[i][j+1] == 0) || (y[i][j+2] == 0))) {
    change(i, j+1);
    search(i, j);
    restore(i, j+1);
  }
  if(((y[i+1][j] == 0) && (y[i][j+1] == 0))) {
    change(i, j);
    search(i, j);
    restore(i, j);
  }
}
```

其中，change(i, j)用于在(i, j)处设置一个警卫机器人哨位，并相应地改变其相邻陈列室的受监视状况。

```
void change(int i, int j) {
  x[i][j] = 1;
  k++;
  for(int s=1; s <= 5; s++) {
    int  p = i+d[s][1];
    int  q = j+d[s][2];
    y[p][q]++;
    if((y[p][q] == 1))
```

```
            t++;
        }
    }
```

restore(*i*, *j*)则用于撤销在(*i*, *j*)处设置的警卫机器人哨位，并相应地改变其相邻陈列室的受监视状况。

```
void restore (int i, int j) {
    x[i][j] = 0;
    k--;
    for(int s=1; s <= 5; s++) {
        int  p = i+d[s][1];
        int  q = j+d[s][2];
        y[p][q]--;
        if((y[p][q] == 0))
            t--;
    }
}
```

最后由 compute()函数调用主体算法，搜索最优解。

```
void compute() {
    more = m/4+1;
    if(m%4 == 3)
        more++;
    else if(m%4 == 2)
        more += 2;
    t2 = m*n + mor e+ 4;
    t1 = m*n + 4;
    best = INT_MAX;
    memset(y, 0, sizeof(y));
    memset(x, 0, sizeof(x));
    if((n == 1) && (m == 1)) {
        cout << 1 << endl << 1 << endl;
        return;
    }
    for(int i=0; i <= m+1; i++) {
        y[0][i] = 1;
        y[n+1][i] = 1;
    }
    for(i=0; i <= n+1; i++) {
        y[i][0] = 1;
        y[i][m+1] = 1;
    }
    search(1, 0);
    output();
}
```

5-18　世界名画陈列馆问题（不重复监视）。

问题描述：世界名画陈列馆由 $m \times n$ 个排列成矩形阵列的陈列室组成。为了防止名画被盗，需要在陈列室中设置警卫机器人哨位。除了监视所在的陈列室，每个警卫机器人还可以

监视与它所在的陈列室相邻的上、下、左、右 4 个陈列室。试设计一个安排警卫机器人哨位的算法，使名画陈列馆中每个陈列室都在警卫机器人的监视下，并且要求每个陈列室仅受一个警卫机器人监视，且所用的警卫机器人数最少。

算法设计：设计一个算法，计算警卫机器人的最佳哨位安排方案，使名画陈列馆中每个陈列室都仅受一个警卫机器人监视。且所用的警卫机器人数最少。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 m 和 n（$1 \leqslant m, n \leqslant 20$）。

结果输出：将计算的警卫机器人数及其最佳哨位安排输出到文件 output.txt。文件的第 1 行是警卫机器人数；接下来的 m 行中每行 n 个数，0 表示无哨位，1 表示哨位。如果不存在满足要求的哨位安排方案，则输出“No Solution!”。

输入文件示例	输出文件示例
input.txt	output.txt
4 4	4
	0 0 1 0
	1 0 0 0
	0 0 0 1
	0 1 0 0

分析与解答：本题要求每个陈列室仅受一个警卫机器人监视，在许多情况下问题无解。下面分 3 种情形来讨论。

（1）$1=n \leqslant m$ 的情形

此时问题恒有解，且容易直接写出其最优解。

当 $m \bmod 3=1$ 时，将机器人哨位置于$(1, 3k+1)$ $k=0, 1, \cdots, \lfloor m/3 \rfloor$。

当 $m \bmod 3=0$ 或 2 时，将机器人哨位置于$(1, 3k+2)$ $k=0, 1, \cdots, \lfloor m/3 \rfloor$。

（2）$2=n \leqslant m$ 的情形

在这种情况下，如果问题有解，则易知必须在两端分别设置 2 个机器人哨位。这两个机器人哨位各监视 3 个陈列室。其余的 k 个机器人哨位均监视 4 个陈列室。由此可见，$2m=4k+6$，即 $m=2k+3$ 为奇数。当 m 为偶数时问题无解。

当 m 为奇数时，容易直接写出其最优解。将机器人哨位分别置于$(1, 4k+3)$和$(2, 4k+1)$，$k=0, 1, \cdots, \lfloor m/4 \rfloor$。

（3）$2<n \leqslant m$ 的情形

当 $n>2$ 时，用直接枚举法容易验证 $n=3$、$m=3$ 和 $n=3$、$m=4$ 时问题无解；$n=4$、$m=4$ 时，问题有解。当 $n \geqslant 3$ 且 $m \geqslant 5$ 时，问题无解。这一结论可证明如下。考虑左上角的 3×5 阵列。下面证明在不重复监视的前提下，无法使这 3×5 阵列中的每一个陈列室都受到监视。为此，考察本问题的一个变形问题 $P(n, m)$如下。在设置警卫机器人的哨位时，允许在第 $n+1$ 行和第 $m+1$ 列设置哨位，但不要求第 $n+1$ 行和第 $m+1$ 列的陈列室均受监视。当 $n \geqslant 3$ 且 $m \geqslant 5$ 时，在不重复监视的前提下原问题有解，则 $P(3, 5)$一定有解。换句话说，如果问题 $P(3, 5)$无解，则当 $n \geqslant 3$ 且 $m \geqslant 5$ 时，不重复监视问题无解。因此，问题转化为证明问题 $P(3, 5)$无解。对算法 search 做适当修改可用于解问题 $P(n, m)$。用修改过的算法 search 对问题 $P(3, 5)$求解得知该问题无解。这就证明了上述结论。

具体算法由 compute()函数实现如下。

```
void compute() {
```

```
memset(x, 0, sizeof(x));
bool  ok = false;
if(n == 1) {
   int  k = m/3;
   if(m%3 == 1)
      for(int j=0; j <= k; j++)
         x[1][3*j+1] = 1;
   else {
      if(m%3 == 0)
         k--;
      for(int j=0; j <= k; j++)
         x[1][3*j+2] = 1;
   }
   best = k+1;
   ok = true;
}
if(m == 1) {
   int  k = n/3;
   if(n%3 == 1)
      for(int j=0; j <= k; j++)
         x[3*j+1][1] = 1;
   else {
      if(n%3 == 0)
         k--;
      for(int j=0; j <= k; j++)
         x[3*j+2][1] = 1;
   }
   best = k+1;
   ok = true;
}
if(n == 2 && m%2 == 0) {
   int  k = m/4;
   if(m%4 == 0)
      k--;
   for(int j=0; j <= k; j++) {
      x[1][4*j+3] = 1;
      x[2][4*j+1] = 1;
   }
   best = 2*k+2;
   ok = true;
}
if(m == 2 && n%2 == 0) {
   int  k = n/4;
   if(n%4 == 0)
      k--;
   for(int j=0; j <= k; j++) {
      x[4*j+3][1] = 1;
      x[4*j+1][2] = 1;
   }
```

```
    best = 2*k+2;
    ok = true;
  }
  if(n == 4 && m == 4) {
    x[1][1] = 1;
    x[1][4] = 1;
    x[4][1] = 1;
    x[4][4] = 1;
    best = 4;
    ok = true;
  }
  if(ok)
    output();
  else
    cout << "No Solution!" << endl;
}
```

5-19 算 m 点问题。

问题描述：给定 k 个正整数，用算术运算符+、−、*、/将这 k 个正整数连接起来，使最终的得数恰为 m。

算法设计：对于给定的 k 个正整数，给出计算 m 的算术表达式。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 k 和 m，表示给定 k 个正整数，且最终的得数恰为 m。接下来的一行中有 k 个正整数。

结果输出：将计算 m 的算术表达式输出到文件 output.txt。如果有多个满足要求的表达式，只要输出一组，每步算式用分号隔开。如果无法得到 m，则输出“No Solution!”。

输入文件示例	输出文件示例
input.txt	output.txt
5 125 2 2 12 3	77*3=21; 21*12=252; 252−2=250; 250/2=125;

分析与解答：对输入的 k 个正整数与运算符+、−、*、/的所有组合用回溯法进行搜索。

```
void search(int k, vector<float> d) {
  vector<float> e(kk+1, 0);
  if(k == 1) {
    if(int((d[1]-mm)*10000) == 0) {
      outanswer();
      found = true;
    }
  }
  else {
    for(int i=1; i <= k-1; i++) {
      for(int j=i+1; j <= k; j++) {
        float  a = d[i], b = d[j];
        if(a<b)
          swap(a, b);
        for(int m=1, t=0; m <= k; m++)
          if((m != i) && (m != j))
            e[++t] = d[m];
```

```
            r[kk+1-k][1] = a;
            r[kk+1-k][3] = b;
            r[kk+1-k][4] = -1;
            for(m=1; m <= 6; m++) {
               switch(m) {
                  case 1:  r[kk+1-k][4] = a+b;
                           break;
                  case 2:  r[kk+1-k][4] = ab;
                           break;
                  case 3:  r[kk+1-k][4] = a*b;
                           break;
                  case 4:  if( b!=0)
                              r[kk+1-k][4] = a/b;
                           break;
                  case 5:  if(a != 0)
                              r[kk+1-k][4] = b/a;
                           break;
                  case 6:  r[kk+1-k][4] = b-a;
                           break;
               }
               r[kk+1-k][2] = m;
               if(r[kk+1-k][4] != -1) {
                  e[t+1] = r[kk+1-k][4];
                  search(k-1, e);
               }
            }
         }
      }
   }
}
```

其中，outanswer()函数输出找到的表达式。

```
void outanswer() {
   ostringstream  msg;
   for(int i=1; i < kk; i++) {
      switch(int(r[i][2])) {
         case 1:   msg<<((r[i][1]<0)?"(":"")<<r[i][1]<<((r[i][1]<0)?")":"")<<
                   "+"<<((r[i][3]<0)?"(":"")<<r[i][3]<<((r[i][3]<0)?")":"");
                   break;
         case 2:   msg<<((r[i][1]<0)?"(":"")<<r[i][1]<<((r[i][1]<0)?")":"")<<
                   "-"<<((r[i][3]<0)?"(":"")<<r[i][3]<<((r[i][3]<0)?")":"");
                   break;
         case 3:   msg<<((r[i][1]<0)?"(":"")<<r[i][1]<<((r[i][1]<0)?")":"")<<
                   "*"<<((r[i][3]<0)?"(":"")<<r[i][3]<<((r[i][3]<0)?")":"");
                   break;
         case 4:   msg<<((r[i][1]<0)?"(":")<<r[i][1]<<((r[i][1]<0)?")":"")<<
                   "/"<<((r[i][3]<0)?"(":"")<<r[i][3]<<((r[i][3]<0)?")":"");
                   break;
         case 5:   msg<<((r[i][3]<0)?"(":")<<r[i][3]<<((r[i][3]<0)?")":"")<<
```

```
                "/"<<((r[i][1]<0)?"(":"")<<r[i][1]<<((r[i][1]<0)?")":");
                break;
        case 6:  msg<<((r[i][3]<0)?"(":"")<<r[i][3]<<((r[i][3]<0)?")":"")<<
                "-"<<((r[i][1]<0)?"(":")<<r[i][1]<<((r[i][1]<0)?")":"");
                break;
      }
      msg << "=" << r[i][4] << ";";
    }
    string  ans=msg.str();
    answer.insert(ans);
}
```

readin()函数读入初始数据并进行初始化计算。

```
void readin(vector<float> &d) {
    fin >> kk >> mm;
    d.resize(kk+1);
    r.resize(kk, kk+1);
    for(int i = 1; i <= kk; i++)
      fin >> d[i];
    found = false;
}
```

实现算法的主函数如下。

```
void main() {
    vector<float> d;
    readin(d);
    search(kk, d);
    for(set<string>::iterator it=answer.begin(); it != answer.end(); it++)
      fout << *it << endl;
    if(!found)
      fout << "No Solution!" << endl;
}
```

5-20　部落卫队问题。

问题描述：原始部落 byteland 中的居民们为了争夺有限的资源，经常发生冲突。几乎每个居民都有他的仇敌。部落酋长为了组织一支保卫部落的队伍，希望从部落的居民中选出最多的居民入伍，并保证队伍中任何 2 个人都不是仇敌。

算法设计：给定 byteland 部落中居民间的仇敌关系，计算组成部落卫队的最佳方案。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m，表示 byteland 部落中有 n 个居民，居民间有 m 个仇敌关系。居民编号为 1, 2, …, n。接下来的 m 行中，每行有 2 个正整数 u 和 v，表示居民 u 与居民 v 是仇敌。

结果输出：将计算的部落卫队的最佳组建方案输出到文件 output.txt。文件的第 1 行是部落卫队的人数；第 2 行是卫队组成 x_i（$1 \leqslant i \leqslant n$）。$x_i$=0 表示居民 i 不在卫队中，x_i=1 表示居民 i 在卫队中。

输入文件示例	输出文件示例
input.txt	output.txt
7 10	3

```
1 2                    1010001
1 4
2 4
2 3
2 5
2 6
3 5
3 6
4 5
5 6
```

分析与解答：本题即设计解最大独立集问题的回溯法，与主教材中最大团问题的解法十分相似。

```
void AdjacencyGraph::maxInde(int i) {
   if(i > n) {
      for(int j=1; j <= n; j++)
         bestx[j] = x[j];
      bestn = cn;
      return;
   }
   int  OK = 1;
   for(int j=1; j < i; j++)
      if(x[j] &&a[i][j] != NoEdge) {
         OK = 0;
         break;
      }
   if(OK) {
      x[i] = 1;
      cn++;
      maxInde(i+1);
      x[i] = 0;
      cn--;
   }
   if(cn+n-i > bestn) {
      x[i] = 0;
      maxInde(i+1);
   }
}
```

MaxInde()函数进行初始化并用回溯法求解。

```
int AdjacencyGraph::MaxInde(int v[]) {
   x = new int[n+1];
   for(int i=0; i <= n; i++)
      x[i] = 0;
   cn = 0;
   bestn = 0;
   bestx = v;
   maxInde(1);
   delete []x;
```

```
    return bestn;
}
```

5-21　子集树问题。

问题描述：试设计一个用回溯法搜索子集空间树的函数。该函数的参数包括结点可行性判定函数和上界函数等必要的函数，并将此函数用于解装载问题。

装载问题描述如下：有一批共 n 个集装箱要装上艘载重量为 c 的轮船，其中集装箱 i 的重量为 w_i。找出一种最优装载方案，将轮船尽可能装满，即在装载体积不受限制的情况下，将尽可能重的集装箱装上轮船。

算法设计：对于给定的 n 个集装箱的重量和轮船的重量，计算最优装载方案。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 c。n 是集装箱数，c 是轮船的载重量。接下来的 1 行中有 n 个正整数，表示集装箱的重量。

结果输出：将计算的最大装载重量输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
5 10	10
7 2 6 5 4	

分析与解答：用主教材中搜索子集树的一般算法。

```
template<class T>
void Loading<T>::backtrack(int t) {
   if(t > n)
      Output();
   else {
      for(int i=0; i <= 1; i++) {
         x[t] = i;
      if(Constraint(t) && Bound(t)) {
         Change(t);
         backtrack(t+1);
         Restore(t);
      }
   }
}
```

其中，结点可行性判定函数 Constraint()和上界函数 Bound()等必要的函数通过类 Loading 的私有函数传递。

```
template<class T>
void Loading<T>::Output() {
   ii=n;
   bestw=cw;
   return;
}
template<class T>
bool Loading<T>::Constraint(int t) {
   if(x[t] == 0 || x[t] == 1 && cw+w[t] <= c)
      return true;
   else
```

```
    return false;
}
template<class T>
bool Loading<T>::Bound(int t) {
  if(x[t] == 1 || x[t] == 0 && cw+r-w[t] > bestw)
    return true;
  else
    return false;
}
template<class T>
void Loading<T>::Change(int t) {
  if(x[t] == 1)
    cw+=w[t];
  r-=w[t];
}
template<class T>
void Loading<T>::Restore(int t) {
  if(x[t] == 1)
    cw= w[t];
  r += w[t];
  if(ii == t) {
    bestx[t] = x[t];
    ii--;
  }
}
```

5-22 0-1 背包问题。

问题描述：设计一个用回溯法搜索子集空间树的函数，参数包括结点可行性判定函数和上界函数等必要的函数，并将此函数用于解 0-1 背包问题。

0-1 背包问题描述如下：给定 n 种物品和一个背包。物品 i 的重量是 w_i，其价值为 v_i，背包的容量为 C。应如何选择装入背包的物品，使装入背包中物品的总价值最大？

在选择装入背包的物品时，对每种物品 i 只有 2 种选择，即装入背包或不装入背包。不能将物品 i 装入背包多次，也不能只装入部分的物品 i。

0-1 背包问题形式化描述如下：给定 $C>0$，$w_i>0$，$v_i>0$（$1\leqslant i\leqslant n$），要求 n 元 0-1 向量$(x_1, x_2, \cdots, x_n)$，$x_i\in\{0, 1\}$（$1\leqslant i\leqslant n$），使得 $\sum_{i=1}^{n} w_i x_i \leqslant C$，而且 $\sum_{i=1}^{n} v_i x_i$ 达到最大。

算法设计：对于给定的 n 种物品的重量和价值，以及背包的容量，计算可装入背包的最大价值。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 c，n 是物品数，c 是背包的容量。接下来的 1 行中有 n 个正整数，表示物品的价值。第 3 行中有 n 个正整数，表示物品的重量。

结果输出：将计算的装入背包物品的最大价值和最优装入方案输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
5 10	15
6 3 5 4 6	1 1 0 0 1

22654

分析与解答：用主教材中搜索子集树的一般算法。

```
template<class Typew, class Typep>
void Knap<Typew, Typep>::backtrack(int t) {
  if(t > n)
    Output();
  else {
    for(int i=0; i <= 1; i++) {
      x[t] = i;
      if(Constraint(t) && Bound(t)) {
        Change(t);
        backtrack(t+1);
        Restore(t);
      }
    }
  }
}
```

其中，结点可行性判定函数 Constraint()和上界函数 Bound()等必要的函数通过类 Knap 的私有函数传递。

```
template<class Typew, class Typep>
void Knap<Typew, Typep>::Output() {
  bestp = cp;
  for(int j=1; j <= n; j++)
    bestx[j] = x[j];
}
template<class Typew, class Typep>
bool Knap<Typew, Typep>::Constraint(int t) {
  if(x[t] == 0 || x[t] == 1 && cw+w[t] <= c)
    return true;
  else
    return false;
}
template<class Typew, class Typep>
bool Knap<Typew, Typep>::Bound(int t) {
  if(x[t] == 1 || x[t] == 0 && UpBound(t+1) > bestp)
    return true;
  else
    return false;
}
template<class Typew, class Typep>
void Knap<Typew, Typep>::Change(int t) {
  if(x[t]==1) {
    cw += w[t];
    cp += p[t];
  }
}
template<class Typew, class Typep>
```

```
void Knap<Typew, Typep>::Restore(int t) {
  if(x[t] == 1) {
    cw -= w[t];
    cp -= p[t];
  }
}
```

5-23 排列树问题。

问题描述：试设计一个用回溯法搜索排列空间树的函数。该函数的参数包括结点可行性判定函数和上界函数等必要的函数，并将此函数用于解圆排列问题。

圆排列问题描述如下：给定 n 个大小不等的圆 $c_1, c_2, \cdots, c_n$，现要将这 n 个圆排进一个矩形框中，且要求各圆与矩形框的底边相切。圆排列问题要求从 n 个圆的所有排列中找出有最小长度的圆排列。例如，当 n=3，且所给的 3 个圆的半径分别为 1、1、2 时，这 3 个圆的最小长度的圆排列见图 5-9，其最小长度为 $2+4\sqrt{2}$。

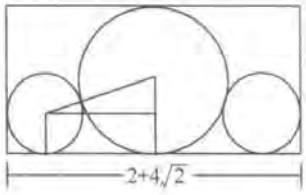

图 5-9 圆排列

算法设计：对于给定的 n 个圆，计算最小长度圆排列。

数据输入：由文件 input.txt 提供输入数据。文件的第 1 行是 1 个正整数 n，表示有 n 个圆。第 2 行有 n 个正数，分别表示 n 个圆的半径。

结果输出：将计算的最小长度输出到文件 output.txt。文件的第 1 行是最小长度，保留 5 位小数。

输入文件示例	输出文件示例
input.txt	output.txt
3	7.65685
1 1 2	

分析与解答：用主教材中搜索排列树的一般算法。

```
void Circle::Backtrack(int t) {
  if(t > n)
    Output();
  else {
    for(int i=t; i <= n; i++) {
      swap(x[t], x[i]);
      if(Constraint(t) && Bound(t)) {
        Change(t);
        Backtrack(t+1);
        Restore(t);
      }
      swap(x[t], x[i]);
    }
  }
}
```

其中，结点可行性判定函数 Constraint()和上界函数 Bound()等必要的函数通过类 Circle 的私有函数传递。

```
void Circle::Output() {
```

```
    float  low = 0, high = 0;
    for(int i=1; i <= n; i++) {
        if(xr[i]-x[i] < low)
            low = xr[i]-x[i];
        if(xr[i]+x[i] > high)
            high = xr[i]+x[i];
    }
    if(high-low < min)
        min = high-low;
}
bool Circle::Constraint(int t) {
    return true;
}
bool Circle::Bound(int t) {
    centerx = Center(t);
    if(centerx+x[t]+x[1] < min)
        return true;
    else
        return false;
}
void Circle::Change(int t) {
    xr[t] = centerx;
}
void Circle::Restore(int t) { }
```

Center()函数计算当前所选择圆的圆心横坐标。

```
float Circle::Center(int t) {
    float  temp = 0;
    for(int j=1; j < t; j++) {
        float  valuex = xr[j] + 2.0*sqrt(x[t]*x[j]);
        if(valuex > temp)
            temp = valuex;
    }
    return temp;
}
```

5-24　一般解空间搜索问题。

问题描述：设计一个用回溯法搜索一般解空间的函数，参数包括：生成解空间中下一扩展结点的函数、结点可行性判定函数和上界函数等必要的函数，并将此函数用于解图的 m 着色问题。

图的 m 着色问题描述如下：给定无向连通图 G 和 m 种不同的颜色。用这些颜色为图 G 的各顶点着色，每个顶点着一种颜色。如果有一种着色法，使 G 中每条边的 2 个顶点着不同颜色，则称这个图是 m 可着色的。图的 m 着色问题是对于给定图 G 和 m 种颜色，找出所有不同的着色法。

算法设计：对于给定的无向连通图 G 和 m 种不同的颜色，计算图的所有不同的着色法。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 3 个正整数 n，k 和 m，表示给定的图 G 有 n 个顶点和 k 条边，m 种颜色。顶点编号为 $1, 2, \cdots, n$。接下来的 k 行中，每行有 2

个正整数 u、v，表示图 G 的一条边(u, v)。

结果输出：将计算的不同的着色方案数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
5 8 4	48
1 2	
1 3	
1 4	
2 3	
2 4	
2 5	
3 4	
4 5	

分析与解答：用主教材中搜索一般解空间的回溯法框架。

```
void Color::Backtrack(int t) {
  if(t > n)
    Output();
  else {
    for(int i=f(n, t); i <= g(n, t); i++) {
      x[t] = h(i);
      Change(t);
    if(Constraint(t) && Bound(t))
      Backtrack(t+1);
    Restore(t);
  }
}
```

其中，生成解空间中下一扩展结点的函数、结点可行性判定函数 Constraint()和上界函数 Bound()等必要的函数通过类 Color 的私有函数传递。

```
void Color::Output() {
{  sum++;
}
bool Color::Constraint(int t) {
  for(int j=1; j <= n; j++)
    if((a[t][j] == 1) && (x[j] == x[t]))
      return false;
  return true;
}
bool Color::Bound(int t) {
  return true;
}
void Color::Change(int t) { }
void Color::Restore(int t) {
  x[t] = 0;
}
int Color::f(int n, int t) {
  return 1;
```

```
}
int Color::g(int n, int t) {
  return m;
}
int Color::h(int i) {
   return i;
}
```

5-25　最短加法链问题。

问题描述：最优求幂问题：给定一个正整数 n 和一个实数 x，如何用最少的乘法次数计算出 x^n。例如，可以用 6 次乘法逐步计算 x^{23} 如下：$x, x^2, x^3, x^5, x^{10}, x^{20}, x^{23}$。可以证明，计算 x^{23} 最少需要 6 次乘法。计算 x^{23} 的幂序列中各幂次 1、2、3、5、10、20、23 组成了一个关于整数 23 的加法链。一般情况下，计算 x^n 的幂序列中各幂次组成正整数 n 的一个加法链：

$$1=a_0<a_1<a_2<\cdots<a_r=n$$

$$a_i=a_j+a_k \qquad k\leqslant j<i;\ i=1, 2, \cdots, r$$

上述最优求幂问题相应于正整数 n 的最短加法链问题，即求 n 的一个加法链，使其长度 r 达到最小。正整数 n 的最短加法链长度记为 $l(n)$。

算法设计：对于给定的正整数 n，计算相应于正整数 n 的最短加法链。

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：将计算的最短加法链长度 $l(n)$和相应的最短加法链输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
23	6
	1 2 3 5 10 20 23

分析与解答：

（1）标准回溯法

对最短加法链问题的状态空间树进行深度优先搜索的回溯法可描述如下。

```
void backtrack(int step) {
   if(a[step] == n) {
      if(step < best) {
         best = step;
         for(int i=1; i <= best; i++)
            chain[i] = a[i];
      }
      return;
   }
   for(int i=step; i >= 1; i--) {
      if(2*a[i] > a[step]) {
         for(int j=i; j >= 1; j--) {
            int  k = a[i]+a[j];
            a[step+1] = k;
            if(k > a[step] && k <= n)
               backtrack(step+1);
         }
      }
```

```
    }
}
```

由于加法链问题的状态空间树的每个第 k 层结点至少有 $k+1$ 个儿子结点，因此从根结点到第 k 层的任一结点的路径数至少是 $k!$，所以状态空间树以指数方式增长。用标准回溯法只能对较小的 n 构造出最短加法链。

（2）迭代搜索法

用回溯法搜索加法链问题的状态空间树时，由于采用了深度优先的搜索方法，算法所搜索到的第一个加法链不一定是最短加法链。如果利用广度优先的方式搜索加法链问题的状态空间树，则算法找到的第一个加法链就是最短加法链，但这种方法的空间开销太大。逐步深化的迭代搜索算法既能保证算法找到的第一个加法链就是最短加法链，又不需要太大的空间开销。其基本思想是控制回溯法的搜索深度 d，从 $d=1$ 开始搜索，每次搜索后使 d 增 1，加深搜索深度，直到找到一条加法链为止。

控制搜索深度的回溯法如下。

```
void backtrack(int step) {
   if(!found) {
      if(a[step] == n) {
         best = step;
         for(int i=1; i <= best; i++)
            chain[i] = a[i];
         found = true;
         return;
      }
      else if(step < lb) {
         for(int i=step; i >= 1; i--) {
            if(2*a[i] > a[step]) {
               for(int j=i; j >= 1; j--) {
                  int  k = a[i]+a[j];
                  a[step+1] = k;
                  if(k > a[step] && k <= n)
                     backtrack(step+1);
               }
            }
         }
      }
   }
}
```

逐步深化的迭代搜索算法如下。

```
void iterativedeepening() {
   best = n+1;
   found = false;
   lb = 2;
   while(!found){
      a[1] = 1;
      backtrack(1);
      lb++;
```

```
    }
}
```

（3）算法优化

算法可进一步进行如下改进。

- 利用 $l(n)$的下界 lb(n)对迭代深度作精确估计。
- 采用剪枝函数对问题的状态空间树进行剪枝搜索，加速搜索进程。
- 用幂树构造 $l(n)$的精确上界 ub(n)。lb(n)=ub(n)时，幂树给出的加法链已是最短加法链。lb(n)<ub(n)时，用改进后的逐步深化迭代搜索算法，从深度 d=lb(n)开始搜索。

关于下界 lb(n)有：

剪枝 1：设在求正整数 n 的最短加法链的逐步深化迭代搜索算法中，当前搜索深度为 d，则在状态空间树的第 i 层结点 a_i 处的一个剪枝条件是

$$\begin{cases} \log\left(\dfrac{n}{3a_i}\right)+i+2>d & 0\leqslant i\leqslant d-2 \\ \log\left(\dfrac{n}{a_i}\right)+i>d & d-1\leqslant i\leqslant d \end{cases}$$

剪枝 2：设在求正整数 n 的最短加法链的逐步深化迭代搜索算法中，当前搜索深度为 d。且正整数 n 可表示为 $n=2^t(2k+1)$（$k\geqslant 1$），则在状态空间树的第 i 层结点 a_i 处的一个剪枝条件是

$$\begin{cases} \log\left(\dfrac{n}{3a_i}\right)+i+2>d & 0\leqslant i\leqslant d-t-2 \\ \log\left(\dfrac{n}{a_i}\right)+i>d & d-t-1\leqslant i\leqslant d \end{cases}$$

与加法链问题密切相关的幂树给出了 $l(n)$的精确上界，如图 5-10 所示。

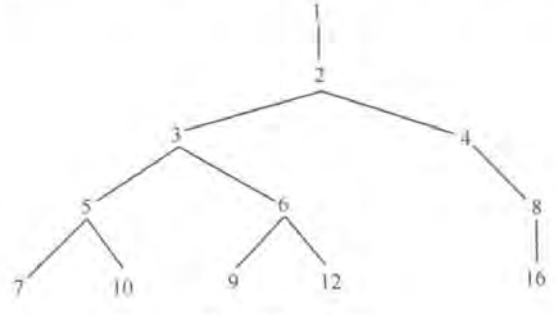

图 5-10　幂树

假设已定义了幂树 T 的第 k 层结点，则 T 的第 k+1 层结点可定义如下。依从左到右顺序取第 k 层结点 a_k，定义其按从左到右顺序排列的儿子结点为 a_k+a_j（$0\leqslant j\leqslant k$）。其中，a_0、a_1、…、a_k 是从 T 的根到结点 a_k 的路径，且 a_k+a_j 在 T 中未出现过。

含正整数 n 的部分幂树 T 容易在线性时间内构造如下。

find()函数递归构造幂树。

```
void find(int step) {
```

```
  int  i, k;
  if(!found) {
    if(a[step] == n) {
      best = step;
      for(i=1; i <= best; i++)
        chain[i] = a[i];
      found = true;
      return;
    }
    else if(step <= ub) {
      for(i=1; i <= step; i++) {
        k = a[step]+a[i];
        if(k <= n) {
          a[step+1] = k;
          if(parent[k] == 0)
            parent[k] = a[step];
          if(parent[k] == a[step])
            find(step+1);
        }
      }
    }
  }
}
```

powertree()函数以逐步深化的迭代搜索方式构造幂树。

```
int powertree(int n) {
  found = false;
  ub = 1;
  for(int i=1; i <= MAXN; i++)
    parent[i] = 0;
  while (!found) {
    a[1] = 1;
    find(1);
    ub++;
  }
  return best;
}
```

改进后的逐步深化迭代搜索算法描述如下。

```
void search() {
  lb = lowerb(n);
  ub = powertree(n);
  t = gett(n);
  if(lb < ub) {
    found = false;
    while(!found) {
      cout << "lb=" << lb << endl;
      a[1] = 1;
      backtrack(1);
```

```
        lb++;
        if(lb == ub)
          found = true;
      }
    }
}
```

其中，lowerb()和 gett()函数实现如下。

```
int lowerb(int m) {
   int  i=0, j=1;
   while(m > 1) {
      i++;
      if(odd(m))
         j++;
      m = m>>1;
   }
   i += log2(j)+1;
   return i;
}
int log2(int m) {
   int  i=0, j=1;
   while(m > 1) {
      i++;
      if(odd(m))
         j++;
      m = m>>1;
   }
   if(j > 1)
      i++;
   return i;
}
int gett(int num) {
   int  i = 0;
   while(!odd(num)) {
      num = num>>1;
      i++;
   }
   return i-1;
}
```

算法的主体是 backtrack()函数。

```
void backtrack(int step) {
   if(!found) {
      if(a[step] == n) {
         best = step;
         for(int i=1; i <= best; i++)
            chain[i] = a[i];
         found = true;
         return;
```

```
    }
    else if(step < lb) {
       for(int i=step; i >= 1; i--) {
          if(2*a[i] > a[step]) {
             for(int j=i; j >= 1; j--) {
                int  k = a[i]+a[j];
                a[step+1] = k;
                if(k > a[step] && k <= n)
                   if(!pruned(step+1))
                      backtrack(step+1);
             }
          }
       }
    }
  }
}
```

pruned()函数实现剪枝。

```
bool pruned(int step) {
  if(step < lb-t-1)
    return (h(3*a[step])+step+2>lb);
  else
    return (h(a[step])+step>lb);
}
int h(int num) {
  int  i = 0;
  while(num < n) {
    num = num<<1;
    i++;
  }
  return i;
}
```

实现算法的主函数如下。

```
int main() {
  cin >> n;
  if(n <= MAXN) {
    search();
    output();
  }
  else
    cout << "Done!" << endl;
  return 0;
}
```

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