**北京邮电大学软件学院**

**2022-2023学年第一学期实验报告**

**课程名称：** 算法分析与设计

**项目名称：** 实验项目二——回溯法

**项目完成人：**

**姓名：** 王宇涵 **学号：** 2020211730

**指导教师：** 李朝晖

**日 期： 2023年 3 月 30 日**

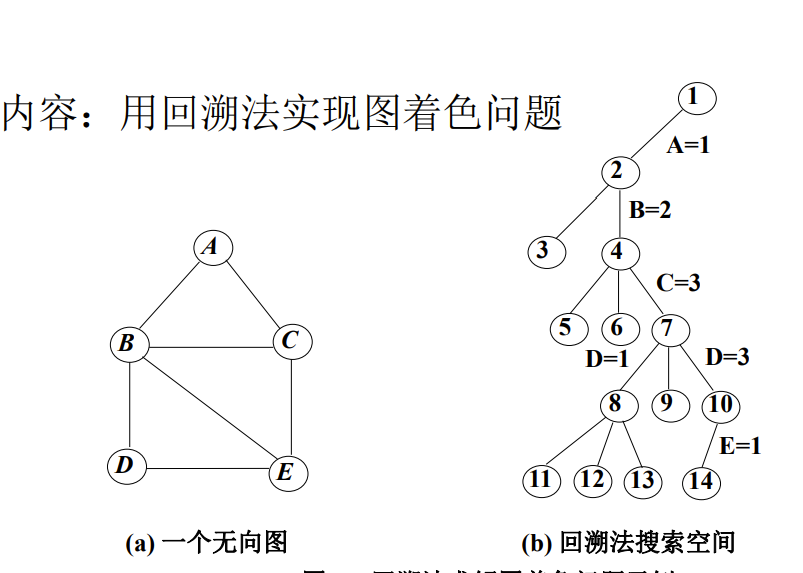
1. **实验目的**
   1. 深刻理解并掌握回溯法的设计思想；
   2. 提高应用回溯法设计算法的技能；
2. **实验内容**
3. (必做)应用回溯法图着色问题；√
4. (必做)对所设计的算法进行时间复杂性分析；√
5. (选作)实现警卫问题; √
6. (选作)PTA-1,PTA-2,PTA-3; √
7. **实验环境**

Windows 10 Dev-C++

1. **实验结果**

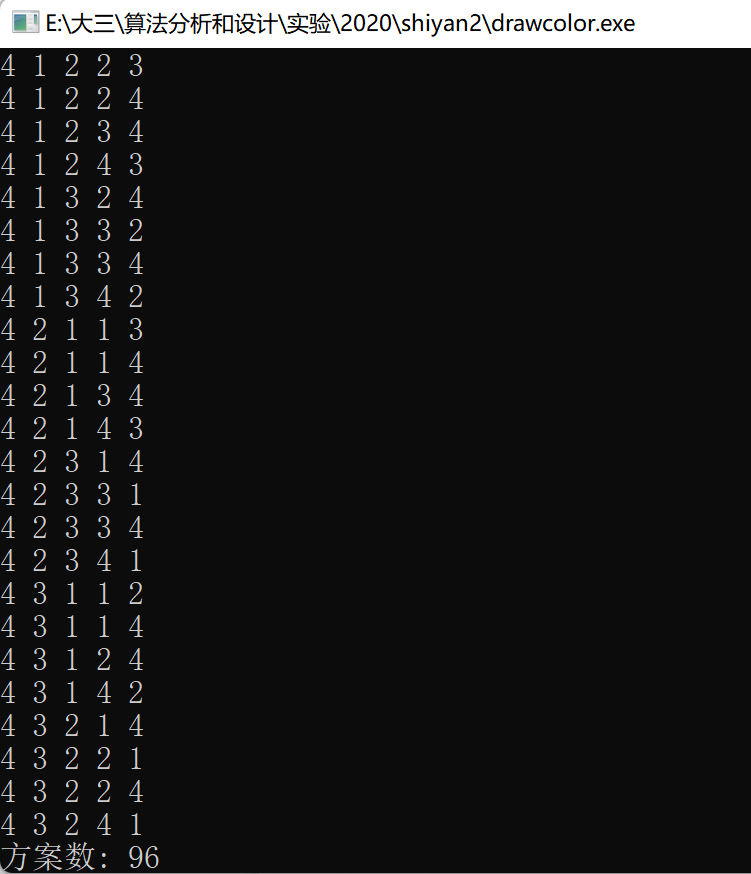
**第一题**

**测试1：**

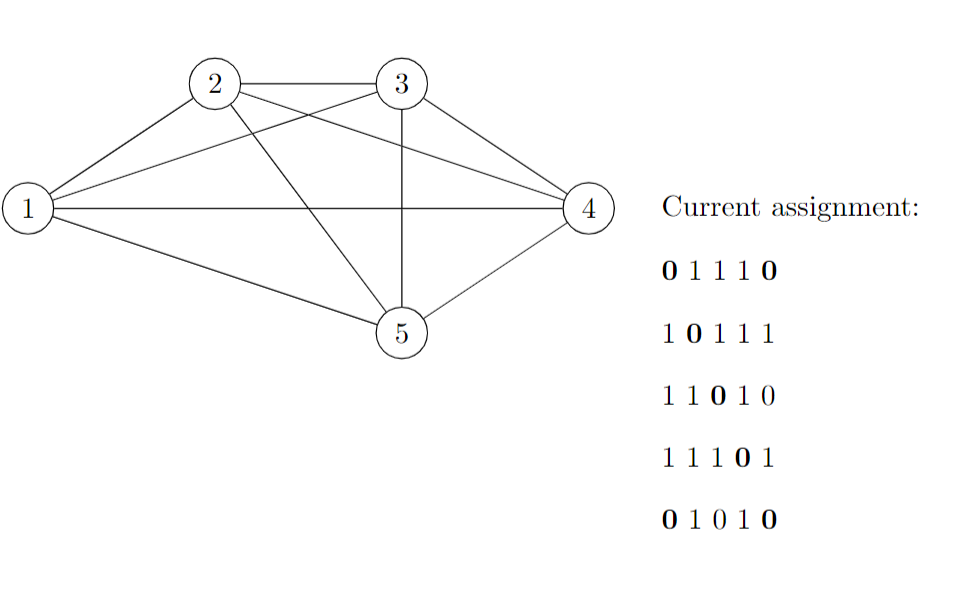


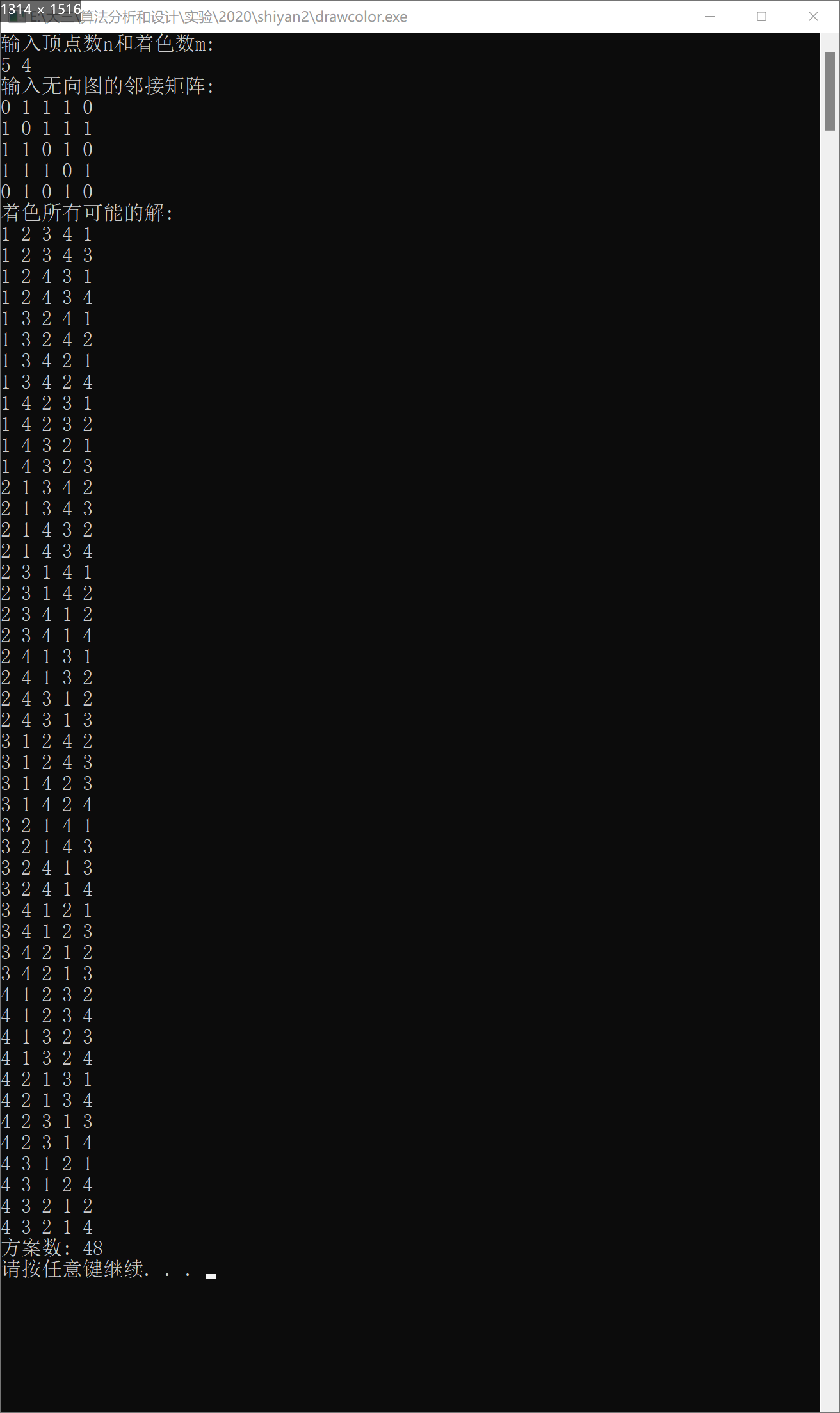
**测试1：（图为上图**





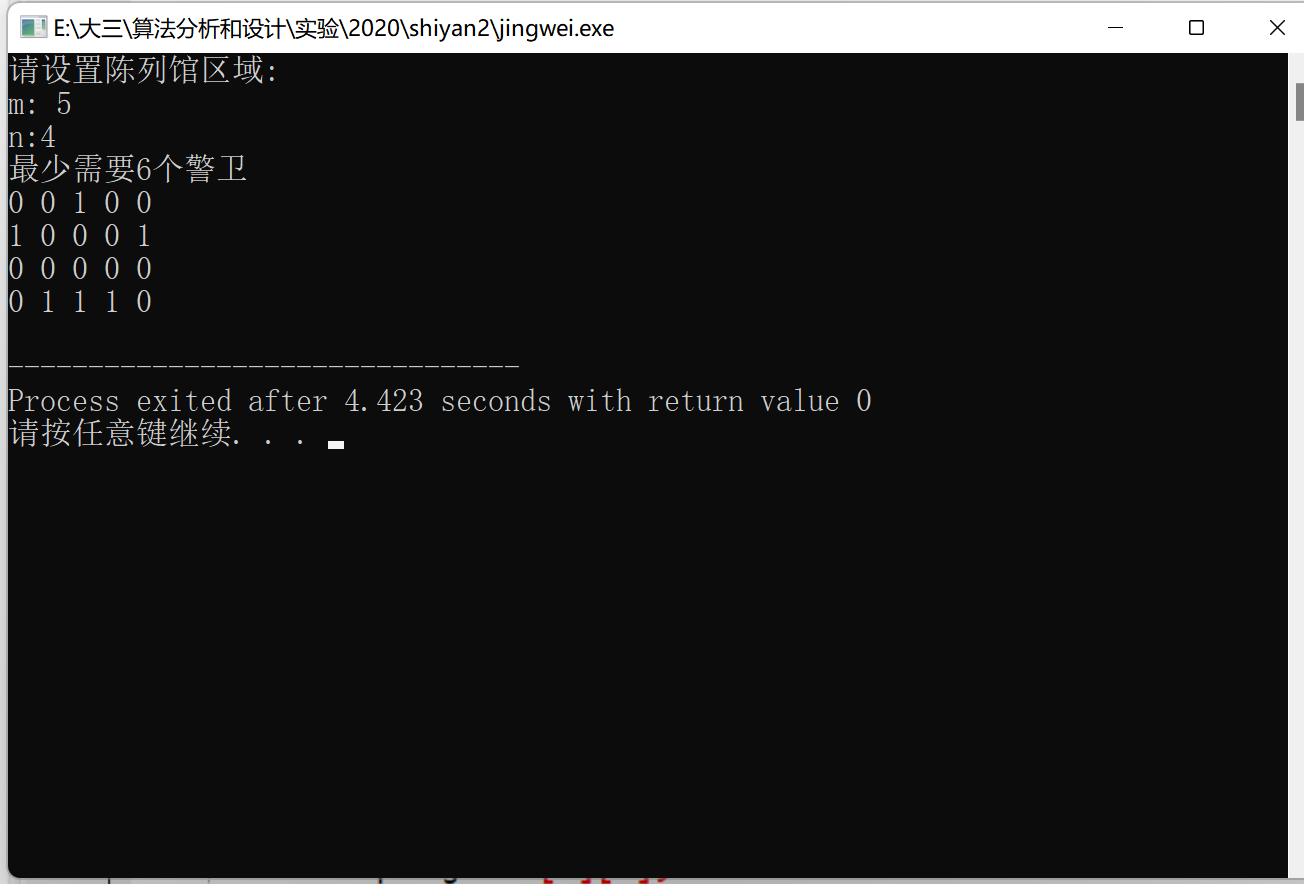
测试2：

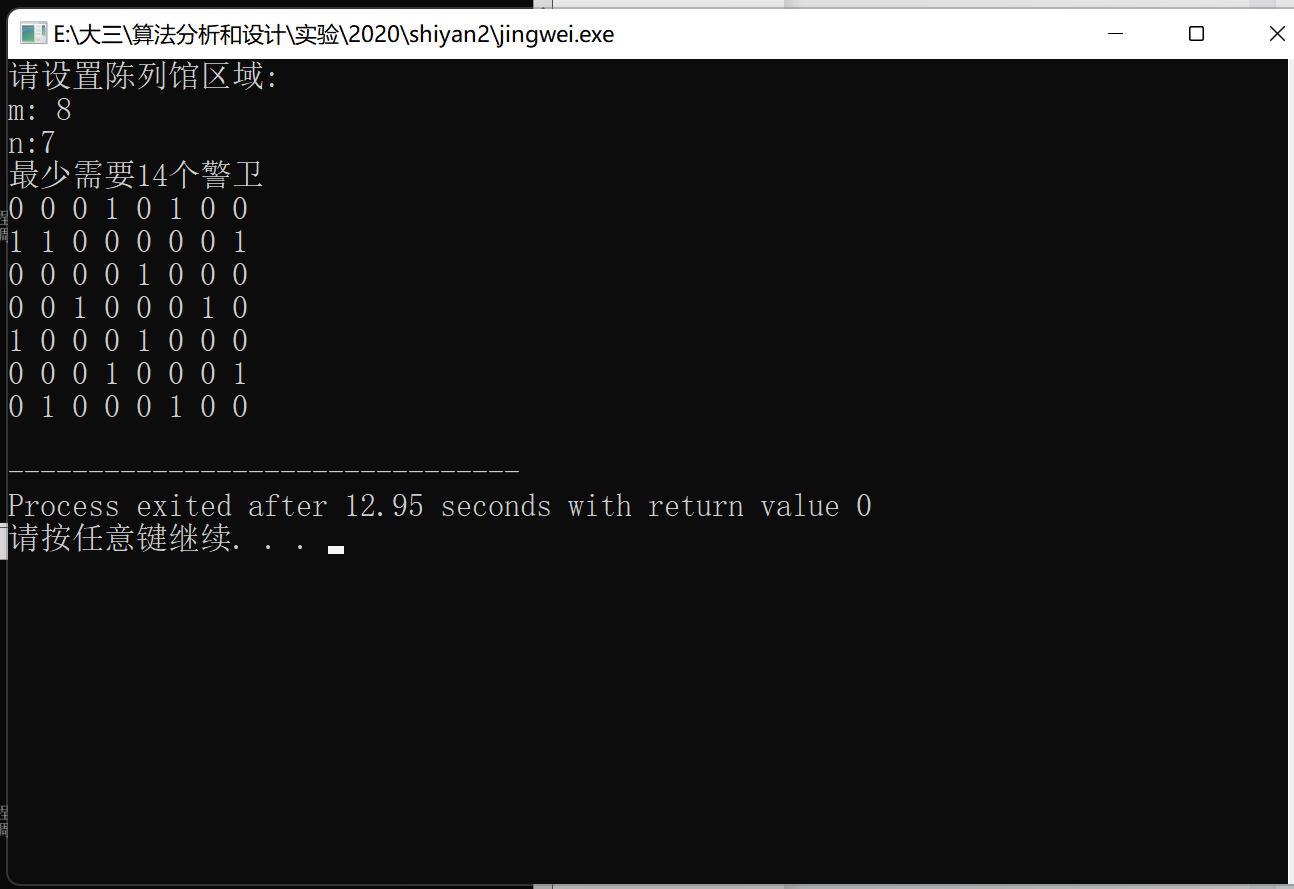




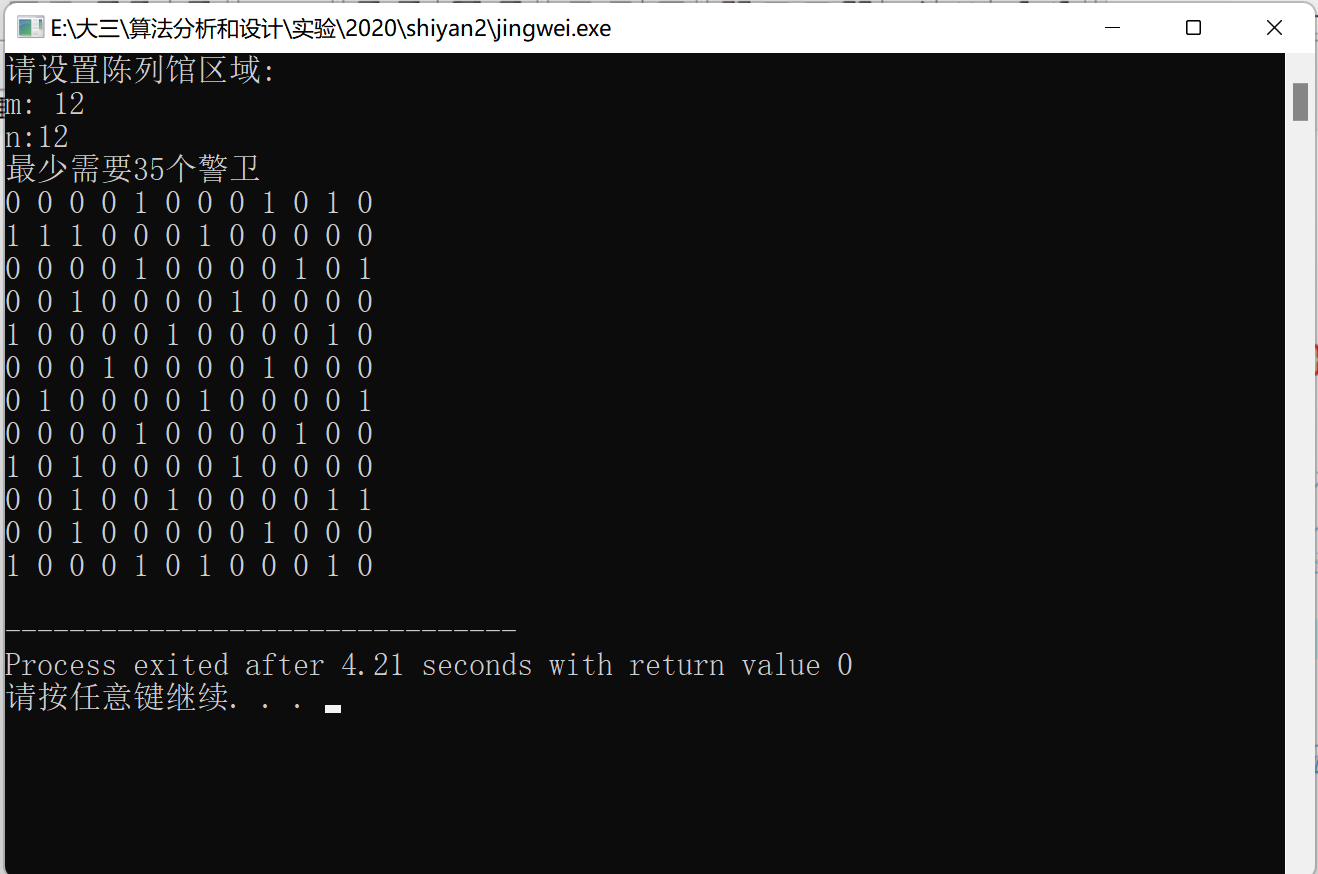
**结果为: 返回所有的着色方案+总共的方案数量**

**第2题**

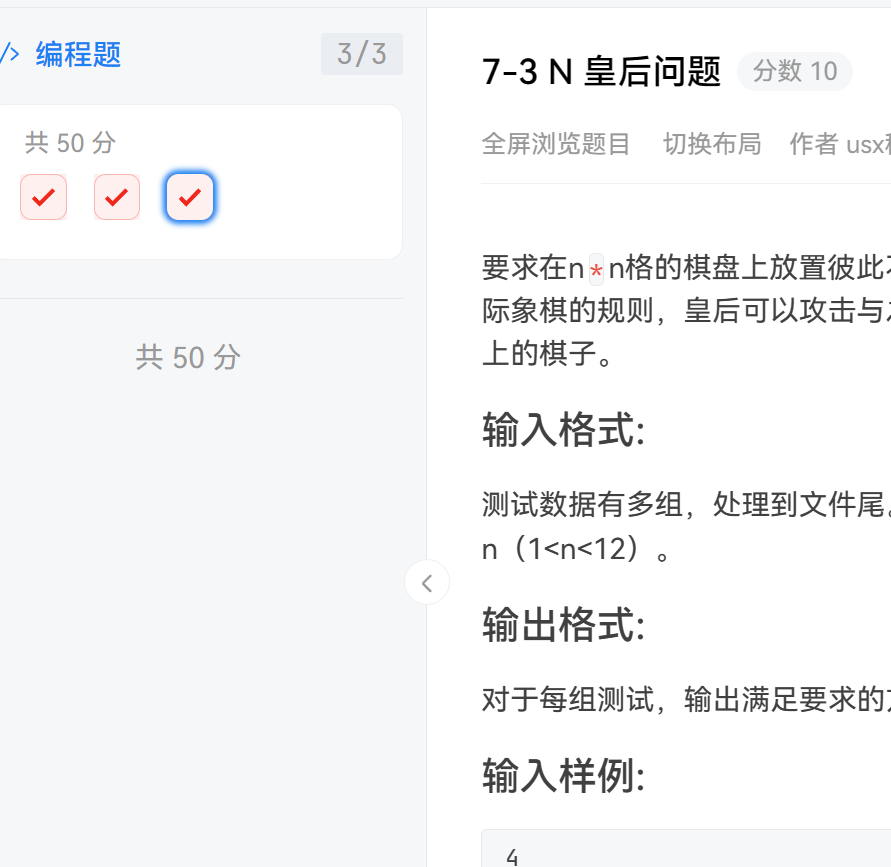








**PTA结果**









1. **附录**

**项目1：回溯法求解图着色问题**

【整体思路】

如果把每一个区域收缩为一个顶点，把相邻两个区域用一条边相连接，就可以把一个区域图抽象为一个平面图。

用m种颜色为图中的每个顶点着色，要求每个顶点着一种颜色，并使相邻两顶点之间有着不同的颜色

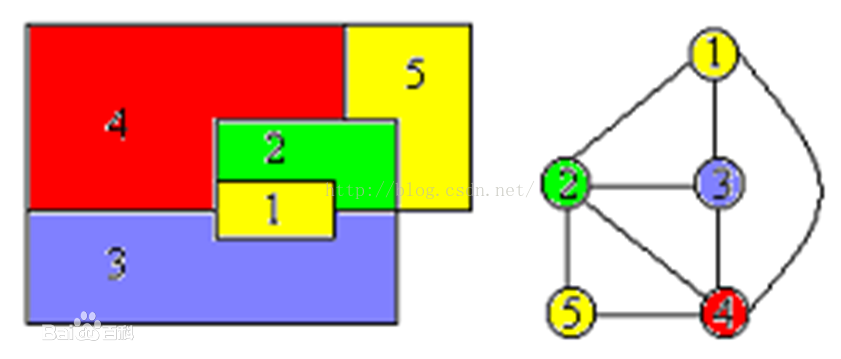
可以通过回溯的方法，不断的为每一个节点着色，在前面n-1个节点都合法的着色之后，开始对第n个节点进行着色，这时候枚举可用的m个颜色，通过和第n个节点相邻的节点的颜色，来判断这个颜色是否合法，如果找到那么一种颜色使得第n个节点能够着色，那么说明m种颜色的方案是可行的。

【问题分析】

如果把每一个区域收缩为一个顶点，把相邻两个区域用一条边相连接，就可以把一个区域图抽象为一个平面图。

用m种颜色为图中的每个顶点着色，要求每个顶点着一种颜色，并使相邻两顶点之间有着不同的颜色

抽象地图及其相应的平面图

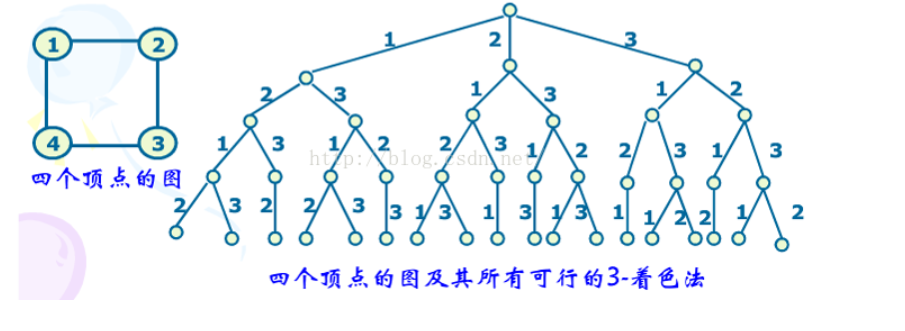


可以通过回溯的方法，不断的为每一个节点着色，在前面n-1个节点都合法的着色之后，开始对第n个节点进行着色，这时候枚举可用的m个颜色，通过和第n个节点相邻的节点的颜色，来判断这个颜色是否合法，如果找到那么一种颜色使得第n个节点能够着色，那么说明m种颜色的方案是可行的。

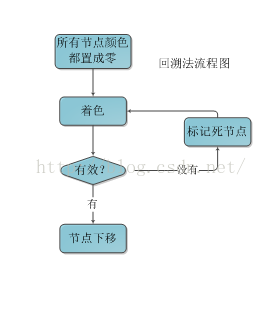
【算法设计】

我们讨论一般连通图的可着色问题，而不只是平面图。给定一个平面图G=（V，E）和M种颜色，如果这个图不是M可着色的，就回答‘no’，如果是M可着色的，则要找出所有不同的着色方法，目前我们认为，此方法是解决该问题的最佳选择

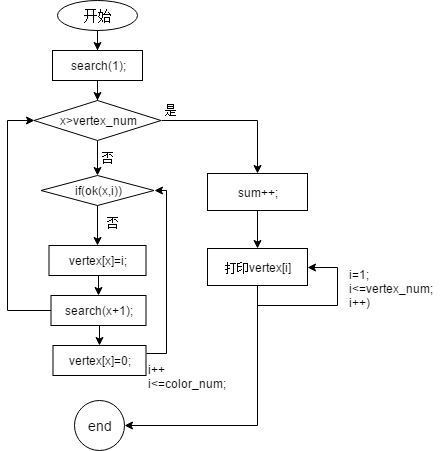
如下图方案所示：



我们假设一个实例来解决，如上图所示，平面图G=（V,E），V={V1,V2,V3,V4}，|E|=4;用M=3种颜色去着色，分别把这三种颜色的序号标为1、2、3。先把所有节点颜色设置零，从根节点开始，对根节点用第一种颜色进行着色，第一个肯定着色有效，接下去接着对下一个节点着色，先用第一种颜色对其节点着色，然后判断这个节点与相邻节点是否相同，有相同颜色则判定为无效着色，继续用第二种颜色着色，直到着色有效为止，如果与相邻节点颜色没有相同，则判定为有效着色，继续下一节点着色。



设计图



【核心代码及解释】

int graphColor(int n, int m, int c[][100])

{

int cnt = 0;

memset(color, 0, sizeof(color));

int k = 1;

while (k >= 1)

{

color[k] += 1;///染第一种颜色

while (color[k] <= m)

{

if (ok(k, c))

break;

else

color[k]++;///搜索下一个颜色

}///挑选合适颜色

if (color[k] <= m && k == n)///找完 输出

{

for (int i = 1; i <= n; i++)

printf("%d ", color[i]);

printf("\n");

cnt++;

}

else if (color[k] <= m && k < n)

{

k++;///染下一个顶点

}

else

{

color[k] = 0;///回溯 找其他方法

k--;

}

}

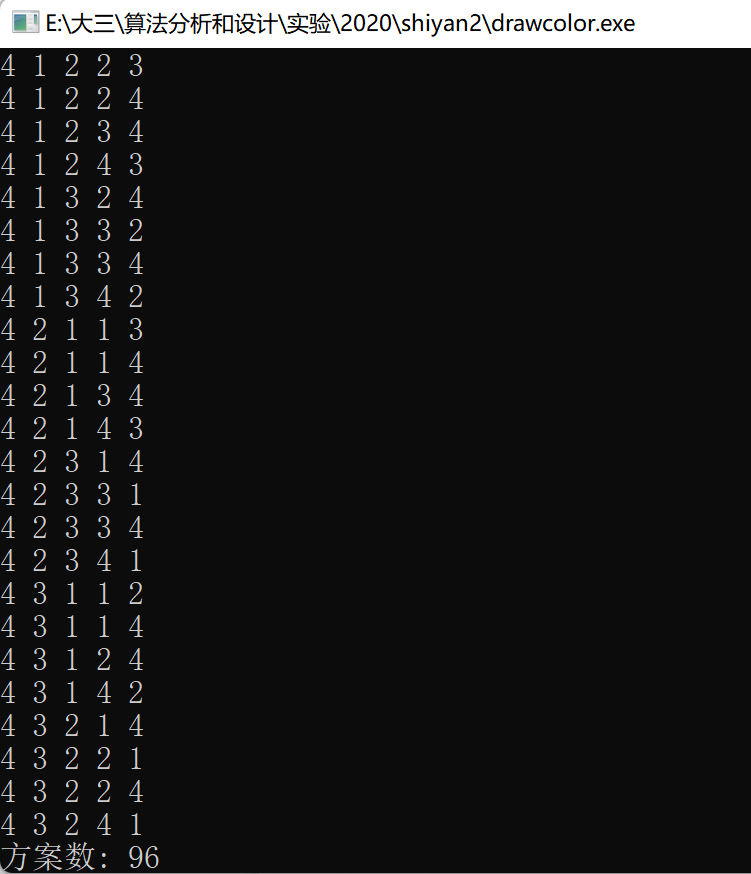
return cnt;

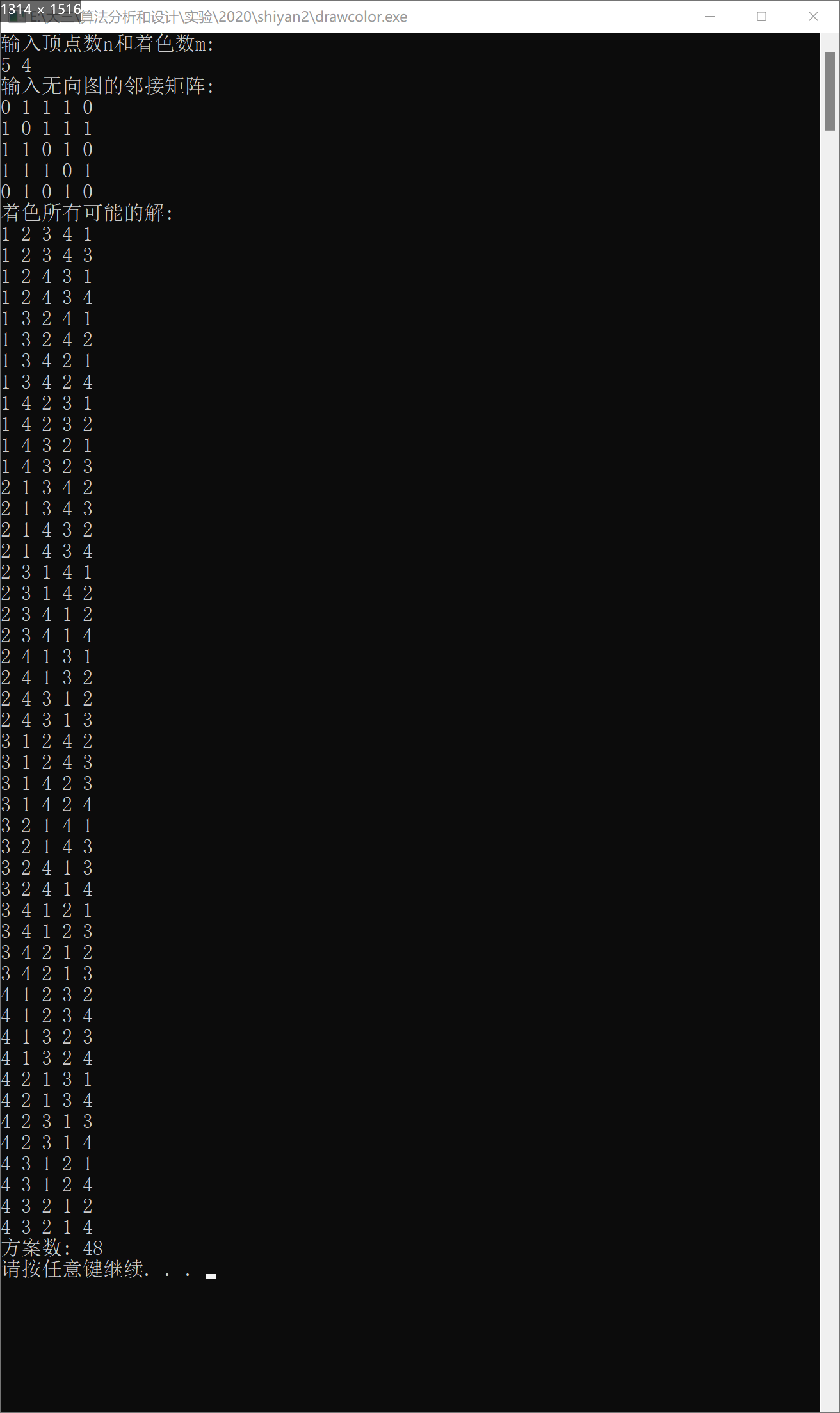
}

【实验结果】

测试1：





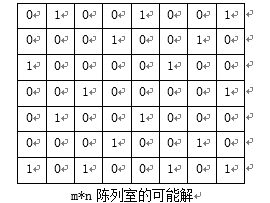


时间复杂度分析：假设有n个节点，则最坏情况下，每个节点都有n-1种颜色选择，因此回溯树最多有(n-1)^n个叶子节点，因此回溯算法的时间复杂度是O(n^2)。

**项目2：警卫问题（选做）**

【整体思路】

本问题的m\*n的陈列室的解可表示如下图所示。其中1代表在该陈列室设置警卫机器人哨位，0表示未在该陈列室设置警卫机器人哨位。



     最为极端的情况是所有元素的值为1。那什么情况下是最优解呢？就是设置警卫机器人哨位数最少即为最优。因为每个矩阵中的值都可以为1或0，有m\*n个元素，有 种可能满足约束条件的矩阵，要从 种可能中遍历找到满足约束条件的1的个数最小的矩阵。由此可见这是一个NP问题。这里的约束条件就是当某一个元素为1时，相邻的4个方向上的元素值可以为0。

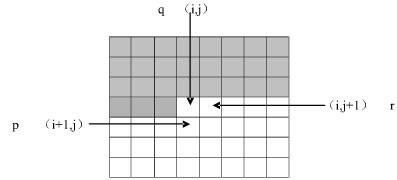
【算法设计】

 从上到下、从左到右的顺序依次考查每一个陈列室设置警卫机器人哨位的情况，以及该陈列室受到监视的情况，用[i,j]表示陈列室的位置，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，表示陈列室[i,j]设置了警卫机器人，当x[i][j]=0时，表示陈列室[i,j]没有设置了警卫机器人。用y[i][j]表示陈列室[i,j]当前受到监视的的警卫机器人的数量。当y[i][j]>0时，表示陈列室[i,j]受到监视的警卫机器人的数量，当y[i][j]=0时，表示陈列室[i,j]没有受到监视。设当前已经设置的警卫机器人的哨位数为k，已经受到监视的陈列室的数量为t，当前最优警卫机器人哨位数为bestc。

设回溯搜索时，当前关注的陈列室是[i,j]，假设该陈列室已经受到监视，即y[i][j]==1，

此时在陈列室[i,j]处设置一个警卫机器人哨位，即x[i][j]==1，相应于解空间树的一个节点q，在陈列室[i+1,j]处设置一个机器人哨位，x[i+1][j]==1，相应于解空间树的另一个节点p。容易看出，以q为根的子树的解，不优于以p为根的子树的解，以q为根的子树可以剪去。因此，在以从上到下，从左到右的顺序依次考察每一个陈列室时，已受监视的陈列室不必设置警卫机器人哨位。

设陈列室[i,j]是从上到下、从左到右搜索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处设置哨位的解空间树中的结点分别为p、q、r。



当y[i][j+1]==1时，以q为根的子树的解，不优于以p为根的子树的解，当y[i][j+1]==1且y[i][j+2]==1时，以r为根的子树的解，不优于以p为根的子树的解。搜索时应按照p、q、r的顺序来扩展结点，并检测节点p对节点q和节点r的控制条件。

解题思路:

* 可以证明,放置的机器人个数不会超过n\*m/3+1个(按每个机器人仅辐射左右或上下考虑).以n\*m/3+2为初始最优值,当放置的个数超过当前最优值时,剪去；
* 可以证明,当前访问的格点(i,j)已被监视时,放置在(i,j)的情况一定不会比放置在(i+1,j+1)的情况好.当(i+1,j+1)不在网格中时,(i+1,j)和(i,j+1)同理；所以,如果(i,j)已被监视,则不需要在此处放置机器人,直接跳过即可；
* 当(i,j)未被监视时,若(i,j+1)已被监视,则在(i,j)放置情况q结点一定不会比在(i+1,j)放置的情况p结点好（q受控于p），所以当且仅当(i,j)在网格右下角或者(i,j+1)未被监视时才考虑放置在(i,j)的情况；
* 当(i,j)未被监视时,若(i,j+1)和(i,j+2)均被监视,则在(i,j+1)放置的情况r结点一定不会比在(i+1,j)放置的情况p结点好,所以当且仅当(i,j+1)或(i,j+2)未被监视时才考虑放置在(i,j+1)的情况；

即考虑顺序为p->q->r；

* 当i=n时,不考虑放置在(i+1,j)的情况；
* 记录已经监视的格点数,剩余未监视格点数/5（还需哨兵数的下界） + 已放置哨兵数如果大于最优值,则一定达不到比当前最优值更好的情况,剪去.
* 类似于(6),考虑更紧的情况,并非每个机器人都能独立监视5个格点,至少会有m/4+1的冗余,这个剪枝仅适用于i<n-1的情况,因为最后两行由于最优值和已放置个数非常接近,总是达不到这个值.

【核心代码及解释】

下面对每个函数和变量进行解释：

void change(int i, int j)

函数change用于在(i, j)位置设置一个警卫，并更新周围受监控情况。其中x[i][j]表示该位置是否设置了警卫，k表示当前已设置的警卫数量，y[p][q]表示该位置是否受到监控，若受到监控则其值为非零。d是一个二维数组，用于表示当前位置的上下左右四个方向。该函数的具体实现包括在当前位置设置警卫，更新周围位置的受监控状态，以及更新警卫数量。

void restore(int i, int j)

函数restore用于撤销在(i, j)处设置的警卫，并更新周围受监控情况。该函数的具体实现与change函数相似，但是是将受监控状态和警卫数量还原回原来的值。

void search(int i, int j)

函数search是核心函数，实现了回溯搜索的过程。该函数用于枚举所有可能的解，同时通过剪枝来减少搜索时间。具体实现包括从左至右、从上至下搜索未被监控的位置，递归搜索下一个点，并在搜索完当前点后恢复现场。在搜索过程中，会比较当前警卫数量与当前最优解，若大于等于最优解，则直接返回。同时，通过计算还需要设置的最少警卫数以及当前已经设置的警卫数来判断当前节点是否需要剪枝。搜索完成后，将最优解保存在bestx中。

void compute()

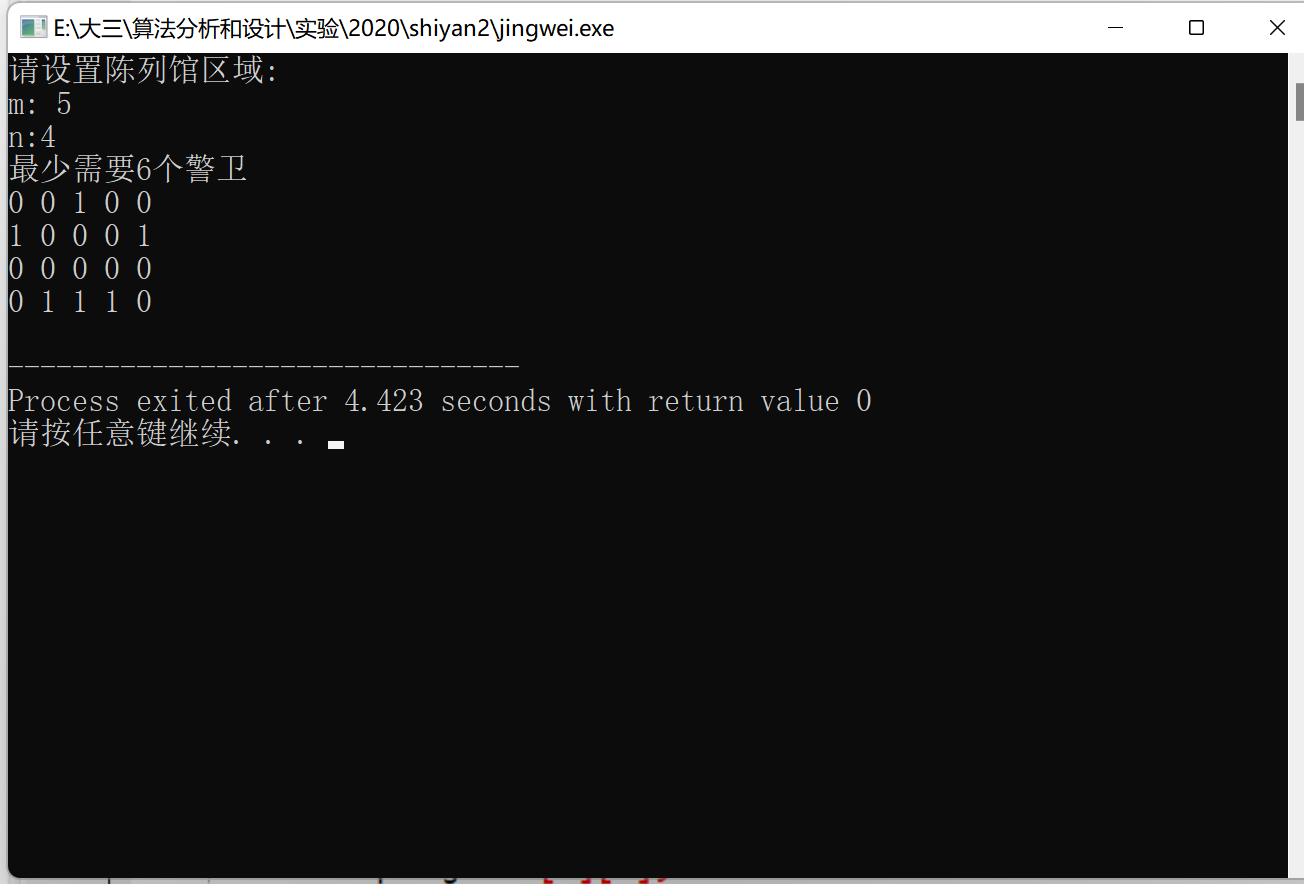
函数compute用于初始化数据，调用搜索函数并输出结果。其中，more表示一个修正参数，t1和t2表示下界和上界，best表示当前最优解的警卫数量。如果地图大小为1\*1，则直接输出结果并返回。为了方便处理边界情况，y数组在整个外面加上了一圈。最后调用search函数进行搜索，得到最优解。

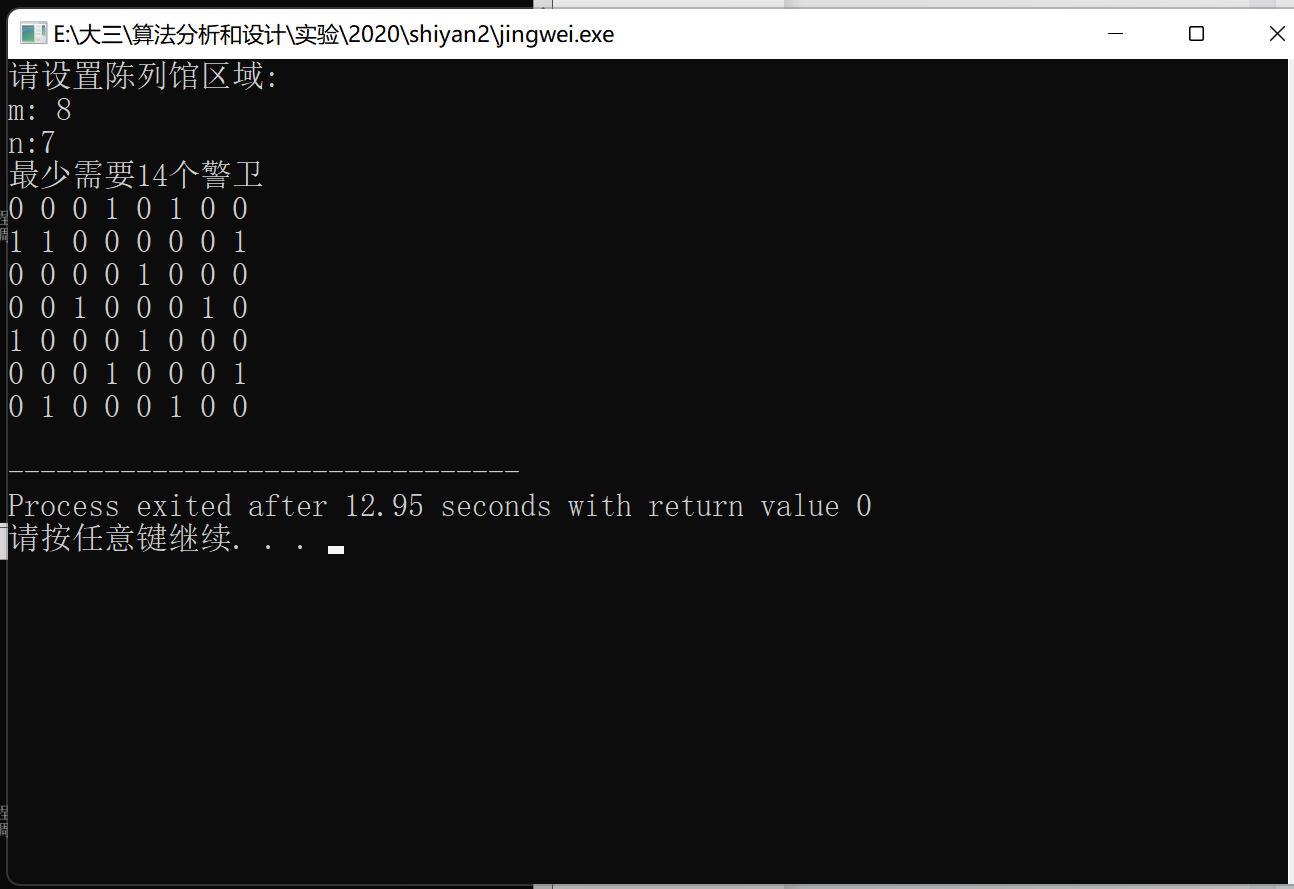
void change(int i, int j)
  
{ //在(i, j)处设置一个警卫，并改变其周围受监控情况
  
 x[i][j] = 1;
  
 k++;
  
 for (int r = 0; r <= 4; r++)
  
 {
  
 //在自己本身跟上下左右五个地方设置受控
  
 int p = i + d[r][0];
  
 int q = j + d[r][1];
  
 y[p][q]++;
  
 if (y[p][q] == 1)
  
 t++;
  
 }
  
}
  
void restore(int i, int j)
  
{ //撤销在(i, j)处设置的警卫，并改变其周围受监控情况
  
 x[i][j] = 0;
  
 k--;
  
 for (int r = 0; r <= 4; r++)
  
 {
  
 int p = i + d[r][0];
  
 int q = j + d[r][1];
  
 y[p][q]--;
  
 if (y[p][q] == 0)
  
 t--;
  
 }
  
}
  
void search(int i, int j)
  
{ //回溯搜索
  
 while (!((y[i][j] == 0) || (i > n)))
  
 { //从上到下，从左至右搜索没被监控的位置
  
 j++;
  
 if (j > m)
  
 {
  
 i++;
  
 j = 1;
  
 }
  
 }
  
 if (i > n)
  
 {
  
 if (k < best)
  
 { //刷新警卫值
  
 best = k;
  
 for (int p = 1; p <= n; p++)
  
 for (int q = 1; q <= m; q++)
  
 bestx[p][q] = x[p][q];
  
 return;
  
 }
  
 }
  
 if (k + (t1 - t) / 5 >= best) return; //警卫数下界 = 还需设置的最少警卫数 + 现有的警卫数
  
 if ((i < n - 1) && (k + (t2 - t) / 5 >= best)) return; //如果比最优警卫数多的话，就剪去这一分枝
  
 if (i < n)
  
 { //结点p
  
 change(i + 1, j);
  
 search(i, j); //递归搜索下一个点
  
 restore(i + 1, j); //恢复
  
 }
  
 if (y[i][j + 1] == 0)
  
 { //结点q
  
 change(i, j);
  
 search(i, j);
  
 restore(i, j);
  
 }
  
 if ((j < m) && ((y[i][j + 1] == 0) || (y[i][j + 2] == 0)))
  
 { //结点r
  
 change(i, j + 1);
  
 search(i, j);
  
 restore(i, j + 1);
  
 }
  
}
  
  
void compute()
  
{
  
 more = m / 4 + 1;
  
 if (m % 4 == 3)
  
 more++;
  
 else if (m % 4 == 2)
  
 more += 2;
  
 t2 = m \* n + more + 4;
  
 t1 = m \* n + 4;
  
 best = INT\_MAX;
  
 if (m == 1 && n == 1)
  
 {
  
  
 cout << "1" << endl;
  
 cout << "1" << endl;
  
  
 }
  
 for (int i = 0; i <= m + 1; i++)
  
 {
  
 //在整个外面加上一圈，便于处理边界情况
  
 y[0][i] = 1;
  
 y[n + 1][i] = 1;
  
 }
  
 for (int i = 0; i <= n + 1; i++)
  
 {
  
 y[i][0] = 1;
  
 y[i][m + 1] = 1;
  
 }
  
 search(1, 0);
  
}

t1 和 t2 分别表示对于当前监控范围内未被监控的位置，至少需要设置的警卫数量下界和上界。它们的计算方式如下：

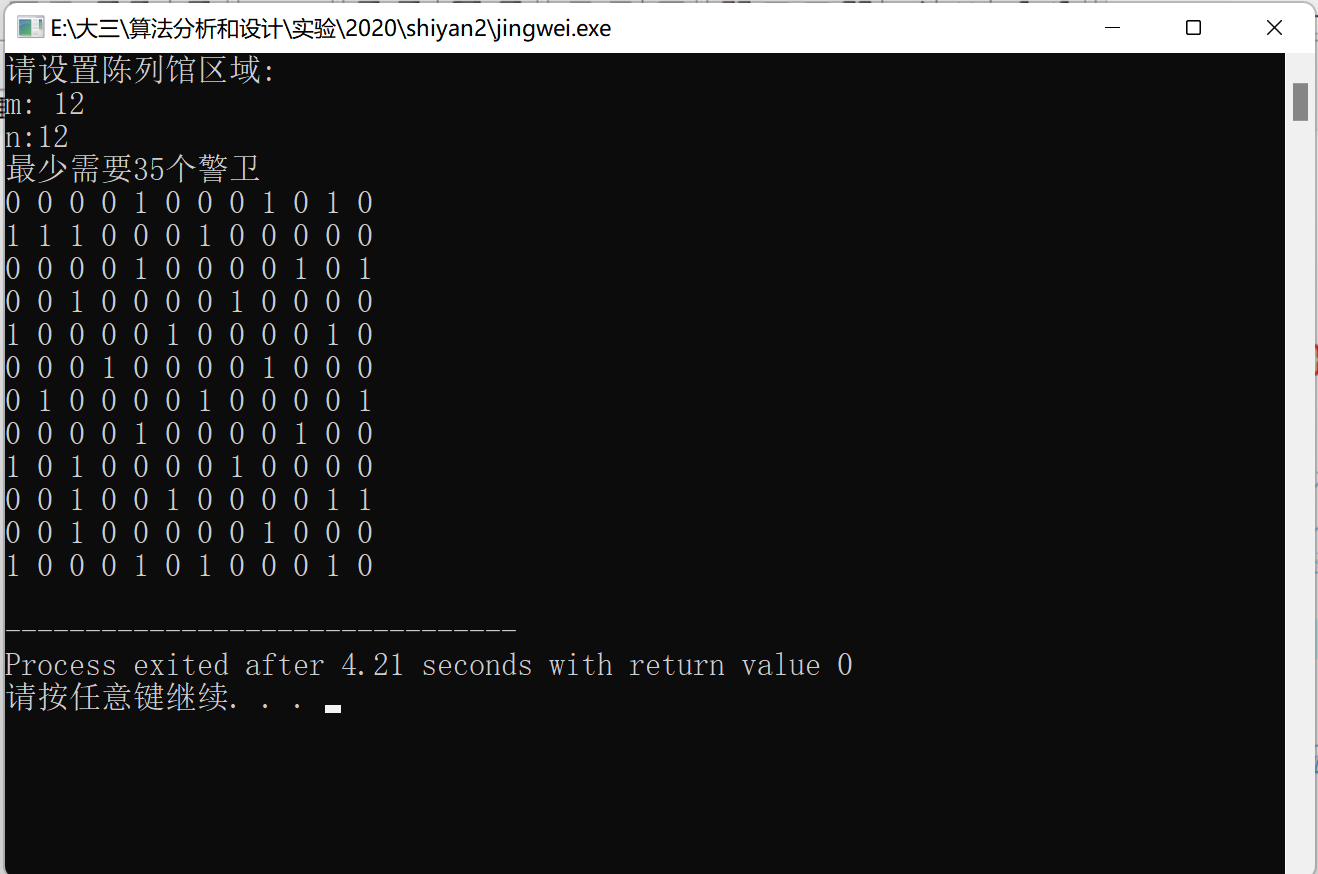
* t1 = m \* n + 4，其中 m 和 n 分别表示矩形区域的宽度和高度。在没有障碍物的情况下，每个位置都需要至少一个警卫来进行监控，另外加上四个角落的位置，因此 t1 = m \* n + 4。
* t2 = m \* n + more + 4，其中 more 表示当 m 不能被 4 整除时需要额外增加的警卫数量。在此问题中，每个警卫可以向四个方向进行监控，因此每行最多可以覆盖 m / 4 个位置，但如果 m 不能被 4 整除，则需要额外增加一个警卫来监控不能被整除的位置。另外还需要加上四个角落的位置，因此 t2 = m \* n + more + 4。

【实验结果】









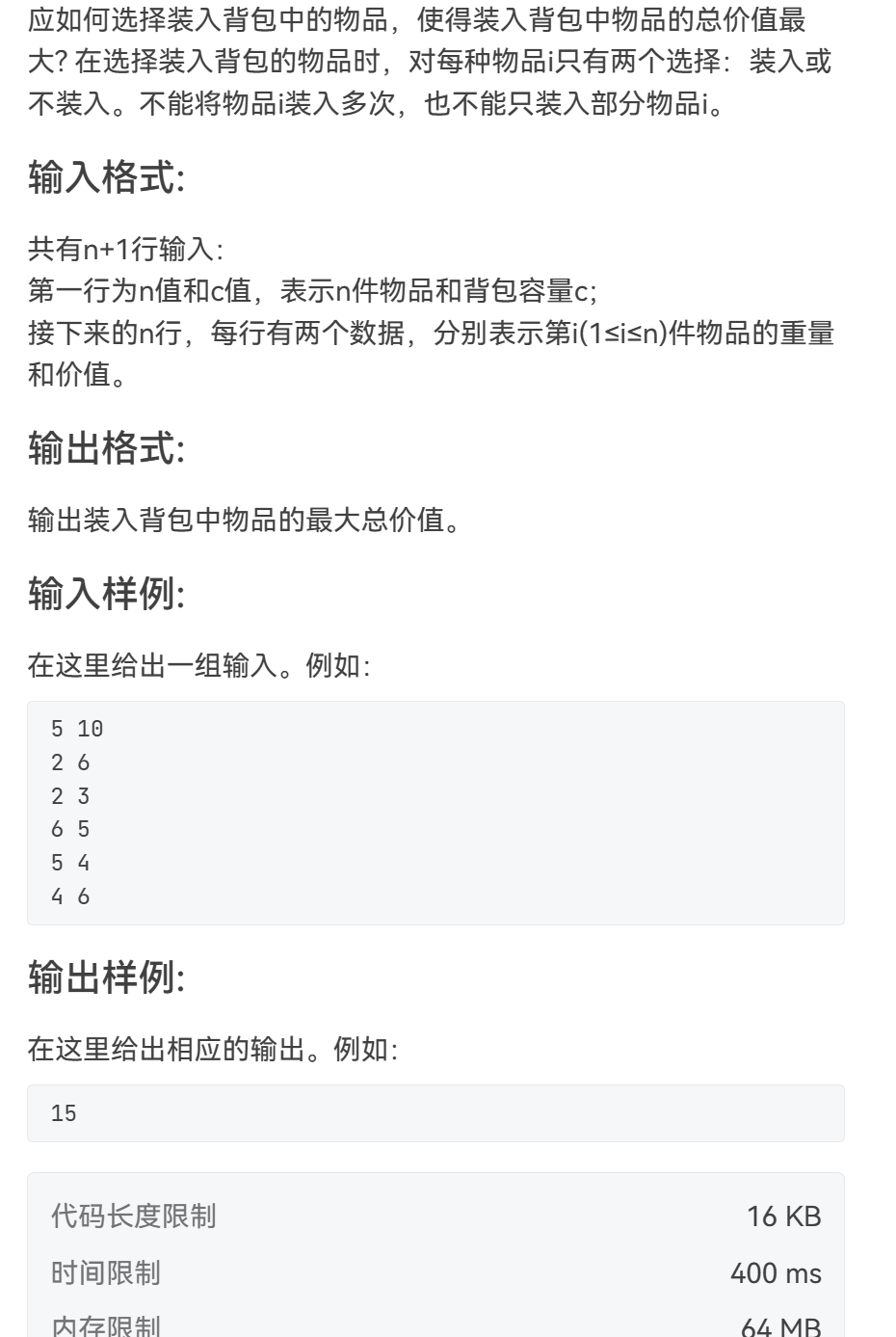
时间复杂度分析：

 回溯法求解子集树框架的时间复杂度为O（2 m \* n）；

 空间复杂度为矩阵大小O（m\*n）；

**项目3: PTA**

**第1题：PTA-1**



【核心代码及解释】

代码中的核心算法是 findbest 函数，它使用递归的方式枚举所有可能的子集，并在每个子集中计算重量和价值，如果该子集可以放入背包且价值比之前的最优子集要高，则更新最优子集。

但是，该算法的时间复杂度非常高，是指数级别的，因为它需要枚举所有可能的子集，时间复杂度为 O(2^n)。对于较大的 n，这种方法的计算时间将非常长，甚至无法完成。因此，它不适用于解决大规模背包问题。

如果需要解决大规模背包问题，可以使用动态规划算法或贪心算法等更高效的算法。

//记录当前子集
  
int record(int sum\_v)
  
{
  
 int i;
  
 int count = 0;
  
 res = sum\_v;
  
 ans.clear();
  
 for (i = 0; i <= n; i++)
  
 {
  
 if (flag[i])
  
 ans.push\_back(i) ;
  
 }
  
 return count;
  
}
  
  
//计算最优解
  
void findbest(int x)
  
{
  
 int i;
  
 if (sum\_w > b)
  
 return;
  
 if (sum\_v > res)
  
 nos = record(sum\_v);
  
 for (i = x; i <= n; i++)
  
 {
  
 sum\_v = sum\_v + v[i];
  
 sum\_w = sum\_w + w[i];
  
 flag[i] = 1;
  
  
 findbest(i + 1);
  
  
 sum\_v = sum\_v - v[i];
  
 sum\_w = sum\_w - w[i];
  
 flag[i] = 0;
  
 }
  
}
  
  
int main()
  
{
  
 int i;
  
 printf("请输入物品总数： ");
  
 cin >> n ;
  
 printf("请输入背包承重： ");
  
 cin >> b;
  
 printf("请输入各个物体的重量以及价值： ");
  
 for (i = 0; i < n; i++)
  
 {
  
 cin>>w[i]>>v[i];
  
  
 flag[i] = 0;
  
 }
  
  
// ans=findbest2();
  
 findbest(0);
  
 printf("可以获得的最大价值的子集为：\n");
  
 for (auto x:ans)
  
 {
  
 cout<<x<<" ";
  
 }
  
 cout<<endl;
  
 printf("总价值为 %d\n", res);
  
 return 0;
  
  
}
  
/\*
  
4
  
10
  
7 42
  
3 12
  
4 40
  
5 25
  
\*/
  
//3
  
//25
  
//20 20
  
//15 30
  
//10 25

标准解法：（动态规划）

#include <bits/stdc++.h>
  
using namespace std;
  
int dp[1010];
  
int w[1010],v[1010];
  
int n=0,m=0;
  
int main()
  
{
  
 cin>>n>>m;
  
 for(int i=1;i<=n;i++)
  
 {
  
 cin>>v[i]>>w[i];
  
 }
  
 for(int i=1;i<=n;i++)
  
 {
  
 for(int j=m;j>=v[i];j--)
  
 {
  
 dp[j]=max(dp[j],dp[j-v[i]]+w[i]);
  
 }
  
 }
  
 cout<<dp[m]<<endl;
  
}

int dp[1010]; 声明了一个长度为 1010 的整数数组，用于记录状态转移方程的结果。

int w[1010],v[1010]; 同样是整数数组，分别用于记录物品的重量和价值。

int n=0,m=0; 初始化两个整数，分别表示物品数量和背包容量。

cin>>n>>m; 从标准输入流中读取 n 和 m 的值。

for(int i=1;i<=n;i++) 开始一个循环，从 1 到 n 遍历每个物品。

cin>>v[i]>>w[i]; 从标准输入流中读取每个物品的价值和重量。

接下来是一个二重循环，用于计算每个背包容量下的最大价值。

首先是外层循环 for(int i=1;i<=n;i++)，从 1 到 n 遍历每个物品。

内层循环 for(int j=m;j>=v[i];j--) 从当前背包容量开始逆序遍历到当前物品的重量。

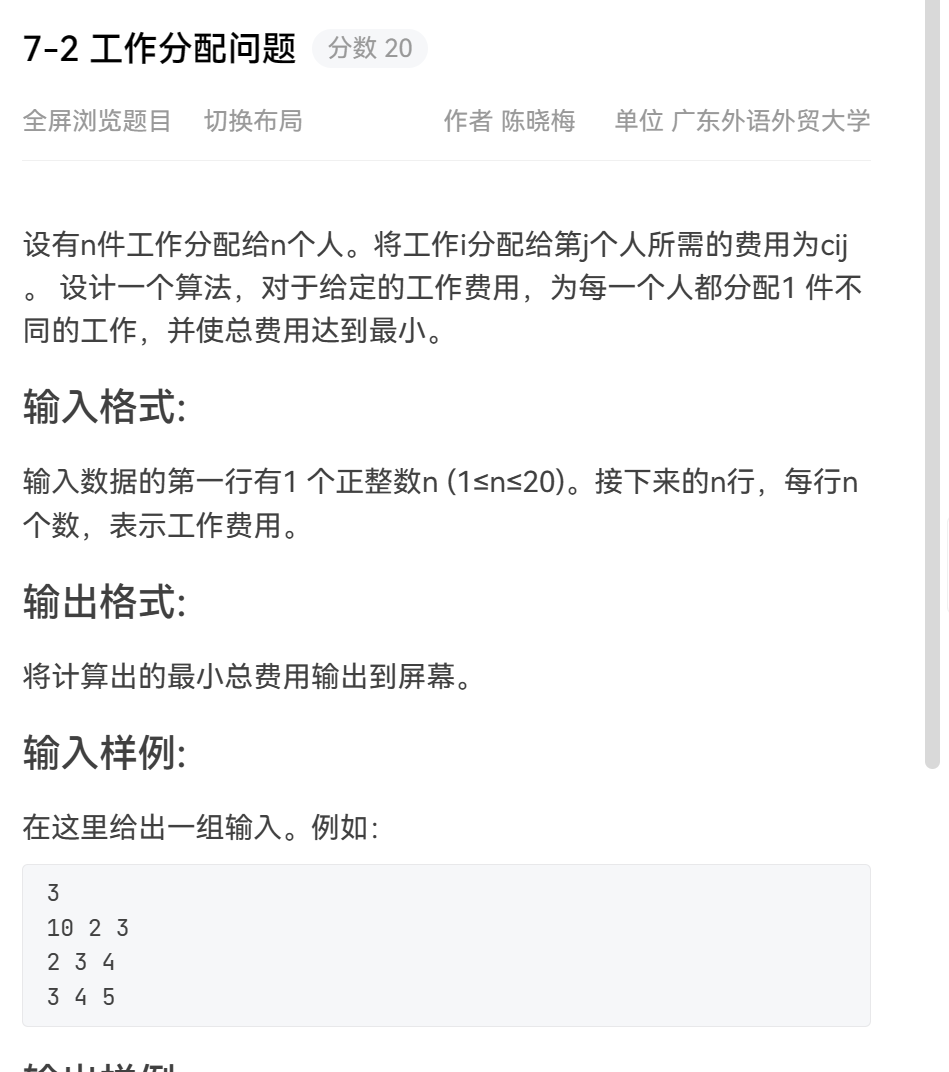
dp[j]=max(dp[j],dp[j-v[i]]+w[i]); 这一行代码是状态转移方程，将 dp[j] 的值更新为 dp[j] 和 dp[j-v[i]]+w[i] 中的最大值。

最后 cout<<dp[m]<<endl; 输出 dp 数组中最后一个元素的值，即背包容量为 m 时的最大价值。

**【实验结果】**



**第2题PTA-2**



**【核心代码及解释】**

int n;
  
int pay[21][21]; //pay[i][j]表示将工作i分配给第j个人的费用为pay[i][j]
  
int minn=INT\_MAX; //因为要求最小值，所以将minn初始化为最大整数（int型）
  
int sum=0; //记录搜索过程中得到的工作费用和
  
int book[21]; //用于标记一个人是否已被分配工作：book[i]=0表示没有被分配工作；book[i]=1表示已经被分配工作
  
  
void dfs(int t)
  
{
  
 if(t>=n) //已经到达叶子结点，继续判断是否找到了最小总费用
  
 {
  
 if(minn>sum) //没有找到最小总费用
  
 {
  
 minn=sum; //更新最小总费用
  
 return;
  
 }
  
 }
  
 for(int i=0;i<n;i++) //为第工作t安排人
  
 {
  
 if(!book[i]) //第i个人还没有被安排工作
  
 {
  
 book[i]=1; //将工作t分配给第i个人
  
 sum+=pay[t][i]; //更新总费用
  
 if(sum<minn) //如果当前得到的sum小于最小值，就向下搜索子树；否则剪枝
  
 dfs(t+1);
  
 book[i]=0; //没有得到比minn更小的和，回溯
  
 sum-=pay[t][i];
  
 }
  
 }
  
  
}
  
  
int main()
  
{
  
 cin>>n;
  
 for(int i=0;i<n;i++)
  
 {
  
 for(int j=0;j<n;j++)
  
 {
  
 cin>>pay[i][j];
  
 }
  
 book[i]=0;
  
 }
  
 dfs(0);
  
 cout<<minn<<endl;
  
 return 0;
  
}

这段代码实现了一个求解最小工作分配费用的问题，使用了深度优先搜索（DFS）和剪枝技巧。具体解释如下：

首先，代码定义了一些全局变量和数组，包括工作数目n，分配工作费用的二维数组pay，记录最小费用的变量minn，记录搜索过程中得到的费用和的变量sum，以及用于标记每个人是否已被分配工作的数组book。

然后，代码实现了一个深度优先搜索的函数dfs，用于搜索所有可能的工作分配方案并更新最小费用。具体实现如下：

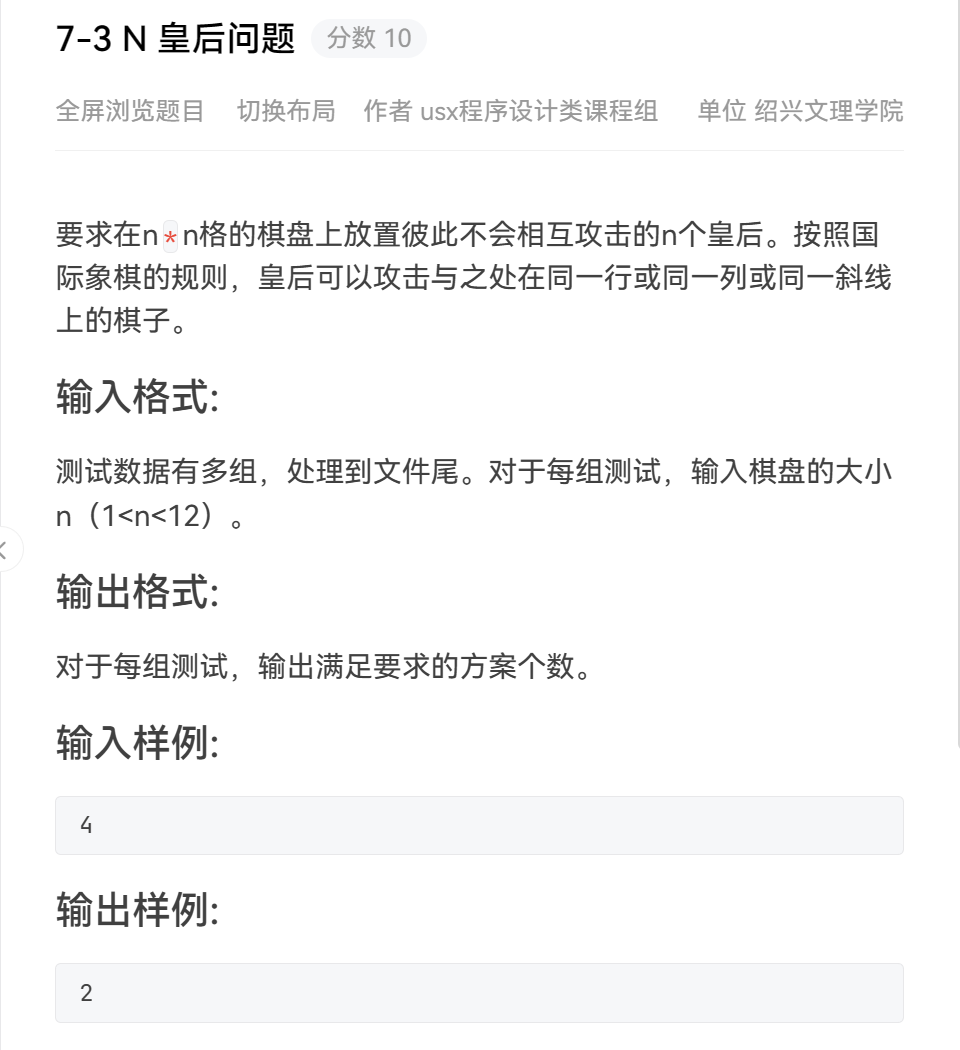
1. 如果搜索到了叶子结点（即已经为所有工作分配了人），则判断当前得到的费用和是否比已有的最小费用更小。如果是，则更新最小费用。
2. 如果还没有为所有工作分配人，则遍历所有可能的人员，并依次将当前工作分配给每个人。如果该人还没有被分配工作，则将当前工作分配给该人，并更新总费用。
3. 如果当前得到的总费用仍然小于最小费用，则继续向下搜索下一个工作的分配方案；否则剪枝，回溯到上一个状态，将当前工作分配给其他人员，继续搜索。

最后，在主函数中，代码读入工作数目n和各个工作分配费用的二维数组pay，并调用dfs函数进行搜索。最终输出最小费用minn即可。

**【实验结果】**



**第3题PTA-3**



**【核心代码及解释】**

首先，定义三个数组col、dg、udg，表示列、对角线和反对角线是否被占用。其中col[i]表示第i列是否被占用，dg[i-u+n]表示从左上到右下的第i条对角线是否被占用，udg[u+i]表示从左下到右上的第i条对角线是否被占用。这里使用了一个技巧，将对角线的编号映射到一个新的坐标系上，使得对角线的编号从0~2n-2，方便数组存储。

然后，使用深度优先搜索（DFS）算法解决问题。首先从第0行开始放置皇后，对于每一行，从第0列开始枚举所有可能的放置位置。如果这个位置合法（不与之前的皇后冲突），就将皇后放在这里，然后更新col、dg、udg数组，进入下一行继续搜索；如果这个位置不合法，就跳过这个位置，继续搜索下一个位置。如果成功放置了N个皇后，就得到了一个解，计数器ans加1，然后回溯到上一行，继续搜索下一个位置，直到所有情况都搜索完毕。

最后输出计数器ans的值，即N皇后问题的解的个数。

int isvalid(int u,int i)
  
{
  
 if(col[i]==1||dg[i-u+n]||udg[u+i])
  
 return 0;
  
 return 1;
  
}
  
void dfs(int u)
  
{
  
 if(u==n)
  
 {
  
// for(int i=0;i<n;i++)
  
// {
  
// for(int j=0;j<n;j++)
  
// {
  
// cout<<path[i][j];
  
// }
  
// cout<<endl;
  
// }
  
// cout<<endl;
  
 ans++;
  
 return;
  
 }
  
 for(int i=0;i<n;i++)
  
 {
  
 if(!isvalid(u,i))continue;
  
 path[u][i]='Q';
  
 col[i]=1;
  
 dg[i-u+n]=1;
  
 udg[u+i]=1;
  
 dfs(u+1);
  
 path[u][i]='.';
  
 col[i]=0;
  
 dg[i-u+n]=0;
  
 udg[u+i]=0;
  
  
 }
  
}

**【实验结果】**

**源代码**

## 项目1

#include<bits/stdc++.h>
  
#define debug(a) cout<<#a<<"="<<a<<endl
  
int color[500];
  
bool ok(int k, int c[][100])
  
{
  
 for (int i = 1; i < k; i++)
  
 {
  
 if (c[k][i] == 1 && color[i] == color[k])///看已经着色的 与之相连的顶点和他同色与否？
  
 return false;
  
 }
  
 return true;
  
}
  
int graphColor(int n, int m, int c[][100])
  
{
  
 int cnt = 0;
  
 memset(color, 0, sizeof(color));
  
 int k = 1;
  
 while (k >= 1)
  
 {
  
 color[k] += 1;///染第一种颜色
  
 while (color[k] <= m)
  
 {
  
 if (ok(k, c))
  
 break;
  
 else
  
 color[k]++;///搜索下一个颜色
  
 }///挑选合适颜色
  
 if (color[k] <= m && k == n)///找完 输出
  
 {
  
 for (int i = 1; i <= n; i++)
  
 printf("%d ", color[i]);
  
 printf("\n");
  
 cnt++;
  
 }
  
 else if (color[k] <= m && k < n)
  
 {
  
 k++;///染下一个顶点
  
 }
  
 else
  
 {
  
 color[k] = 0;///回溯 找其他方法
  
 k--;
  
 }
  
 }
  
 return cnt;
  
}
  
int main()
  
{
  
 int n, m, i, j;
  
 int c[100][100];
  
 printf("输入顶点数n和着色数m:\n");
  
 scanf("%d %d", &n, &m);
  
 printf("输入无向图的邻接矩阵:\n");
  
 for (i = 1; i <= n; i++)
  
 for (j = 1; j <= n; j++)
  
 scanf("%d", &c[i][j]);
  
 printf("着色所有可能的解:\n");
  
 int cnt = graphColor(n, m, c);
  
 printf("方案数: %d\n", cnt);
  
 system("pause");
  
}
  
/\*
  
5 4
  
0 1 1 1 0
  
1 0 1 1 1
  
1 1 0 1 0
  
1 1 1 0 1
  
0 1 0 1 0
  
5 4
  
0 1 1 0 0
  
1 0 1 1 1
  
1 1 0 0 1
  
0 1 0 0 1
  
0 1 1 1 0
  
\*/

### 项目2

#include<cstdio>
  
#include<cstdlib>
  
#include<iostream>
  
using namespace std;
  
#define MAX = 1000;
  
int d[6][3] = { {0,0}, {0,-1}, {-1,0}, {0,1}, {1,0} };
  
int x[100][100];
  
int y[100][100];
  
int bestx[100][100]; //x用来设置当前警卫，y用来表示监控情况，bestx返回最终结果
  
int n, m, best, k = 0, t = 0; //当前已设置的警卫数为k，受监视的陈列室数为t，当前最少警卫数为best
  
int t1, t2, more; //判断下界剪枝的条件参数
  
bool p;
  
  
  
void change(int i, int j)
  
{ //在(i, j)处设置一个警卫，并改变其周围受监控情况
  
 x[i][j] = 1;
  
 k++;
  
 for (int r = 0; r <= 4; r++)
  
 {
  
 //在自己本身跟上下左右五个地方设置受控
  
 int p = i + d[r][0];
  
 int q = j + d[r][1];
  
 y[p][q]++;
  
 if (y[p][q] == 1)
  
 t++;
  
 }
  
}
  
void restore(int i, int j)
  
{ //撤销在(i, j)处设置的警卫，并改变其周围受监控情况
  
 x[i][j] = 0;
  
 k--;
  
 for (int r = 0; r <= 4; r++)
  
 {
  
 int p = i + d[r][0];
  
 int q = j + d[r][1];
  
 y[p][q]--;
  
 if (y[p][q] == 0)
  
 t--;
  
 }
  
}
  
void search(int i, int j)
  
{ //回溯搜索
  
 while (!((y[i][j] == 0) || (i > n)))
  
 { //从上到下，从左至右搜索没被监控的位置
  
 j++;
  
 if (j > m)
  
 {
  
 i++;
  
 j = 1;
  
 }
  
 }
  
 if (i > n)
  
 {
  
 if (k < best)
  
 { //刷新警卫值
  
 best = k;
  
 for (int p = 1; p <= n; p++)
  
 for (int q = 1; q <= m; q++)
  
 bestx[p][q] = x[p][q];
  
 return;
  
 }
  
 }
  
 if (k + (t1 - t) / 5 >= best) return; //警卫数下界 = 还需设置的最少警卫数 + 现有的警卫数
  
 if ((i < n - 1) && (k + (t2 - t) / 5 >= best)) return; //如果比最优警卫数多的话，就剪去这一分枝
  
 if (i < n)
  
 { //结点p
  
 change(i + 1, j);
  
 search(i, j); //递归搜索下一个点
  
 restore(i + 1, j); //恢复
  
 }
  
 if (y[i][j + 1] == 0)
  
 { //结点q
  
 change(i, j);
  
 search(i, j);
  
 restore(i, j);
  
 }
  
 if ((j < m) && ((y[i][j + 1] == 0) || (y[i][j + 2] == 0)))
  
 { //结点r
  
 change(i, j + 1);
  
 search(i, j);
  
 restore(i, j + 1);
  
 }
  
}
  
  
void compute()
  
{
  
 more = m / 4 + 1;
  
 if (m % 4 == 3)
  
 more++;
  
 else if (m % 4 == 2)
  
 more += 2;
  
 t2 = m \* n + more + 4;
  
 t1 = m \* n + 4;
  
 best = INT\_MAX;
  
 if (m == 1 && n == 1)
  
 {
  
  
 cout << "1" << endl;
  
 cout << "1" << endl;
  
  
 }
  
 for (int i = 0; i <= m + 1; i++)
  
 {
  
 //在整个外面加上一圈，便于处理边界情况
  
 y[0][i] = 1;
  
 y[n + 1][i] = 1;
  
 }
  
 for (int i = 0; i <= n + 1; i++)
  
 {
  
 y[i][0] = 1;
  
 y[i][m + 1] = 1;
  
 }
  
 search(1, 0);
  
}
  
  
/\*\*
  
 \* 世界名画陈列馆问题（回溯法）
  
 \*/
  
int main()
  
{
  
 printf("请设置陈列馆区域: \n");
  
 printf("m: ");
  
 cin >> m;
  
 printf("n:");
  
 cin >> n;
  
 compute(); //计算
  
 cout << "最少需要" << best << "个警卫" << endl;
  
  
 for (int i = 1; i <= n; i++)
  
 {
  
 for (int j = 1; j <= m; j++)
  
 cout << bestx[i][j] << " ";
  
 cout<<endl;
  
 }
  
 return 0;
  
}

### 项目3

#### **dfs(TLE)**

#include<bits/stdc++.h>
  
#define debug(a) cout<<#a<<"="<<a<<endl
  
using namespace std;
  
const int N=110;
  
int n;//物品总个数
  
int b;//背包承重
  
int nos;//最优子集的物品数量
  
int t[50];
  
int sum\_v = 0, sum\_w = 0;
  
vector<int>ans;//记录该子集的物品编号及重量
  
int w[N],v[N],flag[N];
  
int res;
  
//记录当前子集
  
int record(int sum\_v)
  
{
  
 int i;
  
 int count = 0;
  
 res = sum\_v;
  
 ans.clear();
  
 for (i = 0; i <= n; i++)
  
 {
  
 if (flag[i])
  
 ans.push\_back(i) ;
  
 }
  
 return count;
  
}
  
  
//计算最优解
  
void findbest(int x)
  
{
  
 int i;
  
 if (sum\_w > b)
  
 return;
  
 if (sum\_v > res)
  
 nos = record(sum\_v);
  
 for (i = x; i <= n; i++)
  
 {
  
 sum\_v = sum\_v + v[i];
  
 sum\_w = sum\_w + w[i];
  
 flag[i] = 1;
  
  
 findbest(i + 1);
  
  
 sum\_v = sum\_v - v[i];
  
 sum\_w = sum\_w - w[i];
  
 flag[i] = 0;
  
 }
  
}
  
  
int main()
  
{
  
 int i;
  
 printf("请输入物品总数： ");
  
 cin >> n ;
  
 printf("请输入背包承重： ");
  
 cin >> b;
  
 printf("请输入各个物体的重量以及价值： ");
  
 for (i = 0; i < n; i++)
  
 {
  
 cin>>w[i]>>v[i];
  
  
 flag[i] = 0;
  
 }
  
  
// ans=findbest2();
  
 findbest(0);
  
 printf("可以获得的最大价值的子集为：\n");
  
 for (auto x:ans)
  
 {
  
 cout<<x<<" ";
  
 }
  
 cout<<endl;
  
 printf("总价值为 %d\n", res);
  
 return 0;
  
  
}
  
/\*
  
4
  
10
  
7 42
  
3 12
  
4 40
  
5 25
  
\*/
  
//3
  
//25
  
//20 20
  
//15 30
  
//10 25

#### **DP(AC)**

#include <bits/stdc++.h>
  
using namespace std;
  
int dp[1010];
  
int w[1010],v[1010];
  
int n=0,m=0;
  
int main()
  
{
  
 cin>>n>>m;
  
 for(int i=1;i<=n;i++)
  
 {
  
 cin>>v[i]>>w[i];
  
 }
  
 for(int i=1;i<=n;i++)
  
 {
  
 for(int j=m;j>=v[i];j--)
  
 {
  
 dp[j]=max(dp[j],dp[j-v[i]]+w[i]);
  
 }
  
 }
  
 cout<<dp[m]<<endl;
  
}

### 项目4

#include <bits/stdc++.h>
  
using namespace std;
  
  
int n;
  
int pay[21][21]; //pay[i][j]表示将工作i分配给第j个人的费用为pay[i][j]
  
int minn=INT\_MAX; //因为要求最小值，所以将minn初始化为最大整数（int型）
  
int sum=0; //记录搜索过程中得到的工作费用和
  
int book[21]; //用于标记一个人是否已被分配工作：book[i]=0表示没有被分配工作；book[i]=1表示已经被分配工作
  
  
void dfs(int t)
  
{
  
 if(t>=n) //已经到达叶子结点，继续判断是否找到了最小总费用
  
 {
  
 if(minn>sum) //没有找到最小总费用
  
 {
  
 minn=sum; //更新最小总费用
  
 return;
  
 }
  
 }
  
 for(int i=0;i<n;i++) //为第工作t安排人
  
 {
  
 if(!book[i]) //第i个人还没有被安排工作
  
 {
  
 book[i]=1; //将工作t分配给第i个人
  
 sum+=pay[t][i]; //更新总费用
  
 if(sum<minn) //如果当前得到的sum小于最小值，就向下搜索子树；否则剪枝
  
 dfs(t+1);
  
 book[i]=0; //没有得到比minn更小的和，回溯
  
 sum-=pay[t][i];
  
 }
  
 }
  
  
}
  
  
int main()
  
{
  
 cin>>n;
  
 for(int i=0;i<n;i++)
  
 {
  
 for(int j=0;j<n;j++)
  
 {
  
 cin>>pay[i][j];
  
 }
  
 book[i]=0;
  
 }
  
 dfs(0);
  
 cout<<minn<<endl;
  
 return 0;
  
}

### 项目5

#include <bits/stdc++.h>
  
using namespace std;
  
const int N = 10010;
  
int n;
  
int col[N],dg[N],udg[N];
  
char path[110][110];
  
int ans=0;
  
int isvalid(int u,int i)
  
{
  
 if(col[i]==1||dg[i-u+n]||udg[u+i])
  
 return 0;
  
 return 1;
  
}
  
void dfs(int u)
  
{
  
 if(u==n)
  
 {
  
// for(int i=0;i<n;i++)
  
// {
  
// for(int j=0;j<n;j++)
  
// {
  
// cout<<path[i][j];
  
// }
  
// cout<<endl;
  
// }
  
// cout<<endl;
  
 ans++;
  
 return;
  
 }
  
 for(int i=0;i<n;i++)
  
 {
  
 if(!isvalid(u,i))continue;
  
 path[u][i]='Q';
  
 col[i]=1;
  
 dg[i-u+n]=1;
  
 udg[u+i]=1;
  
 dfs(u+1);
  
 path[u][i]='.';
  
 col[i]=0;
  
 dg[i-u+n]=0;
  
 udg[u+i]=0;
  
  
 }
  
}
  
int main()
  
{
  
 while(cin>>n)
  
 {
  
 memset(path,'.',sizeof(path));
  
 ans=0;
  
 dfs(0) ;
  
  
 cout<<ans<<endl;
  
 }
  
  
 return 0;
  
}

**实验心得**

回溯法是一种暴力搜索的方法，它通常用于解决组合优化问题，例如0/1背包问题和图着色问题。下面是这两个问题的问题描述、回溯算法的解决思路以及时间复杂度分析。

***0/1背包问题*** 0/1背包问题是一个经典的组合优化问题，它的问题描述是：给定一个背包和一组物品，每个物品有自己的重量和价值，在不超过背包容量的前提下，如何选择物品放入背包，使得背包中物品的总价值最大。

回溯算法的解决思路是：从第一个物品开始，依次考虑每个物品放或者不放到背包中，直到考虑完所有物品或者背包已经满了。如果背包未满并且所有物品都已考虑过，则更新最优解。如果当前情况已经不能成为最优解的一部分，则回溯到前一个决策点。

时间复杂度分析：假设有n个物品，背包容量为W，则回溯树最多有2^n个叶子节点，因此回溯算法的时间复杂度是O(2^n)。但是，在实际应用中，可以通过剪枝等技巧来优化回溯算法的效率。

***图着色问题*** 图着色问题是一个经典的NP完全问题，它的问题描述是：给定一个无向图，如何用最少的颜色对图中的节点进行着色，使得相邻节点颜色不同。

回溯算法的解决思路是：从第一个节点开始，依次考虑每个节点的颜色，直到所有节点都被着色或者没有合法的着色方案。如果当前情况已经不能成为最优解的一部分，则回溯到前一个决策点。

时间复杂度分析：假设有n个节点，则最坏情况下，每个节点都有n-1种颜色选择，因此回溯树最多有(n-1)^n个叶子节点，因此回溯算法的时间复杂度是O((n-1)^n)。同样地，可以通过剪枝等技巧来优化回溯算法的效率。

此外，我了解到了什么是警卫问题，以及如何进行回溯设置警卫，这个问题是几个问题中最难的一个，最终通过不断尝试，得出了正确答案，对回溯的理解更深了一层。

需要注意的是，回溯算法的时间复杂度很高，因此在实际应用中，通常需要结合其他算法或者优化技巧来提高效率。