·第 3 章·
回溯法

常见考点

- 回溯法设计的基本过程。
- 回溯法的性能分析和改进。
- 利用回溯法的深度优先搜索算法求解问题的方法。
- 灵活运用回溯法解决一些较复杂的应用问题。

3.1 回溯法要点归纳

3.1.1 回溯法概述

回溯法采用类似暴力或者枚举的搜索尝试过程，在包含问题的所有解的解空间树中按照深度优先搜索的策略从根结点出发深度探索。当探索到某一结点时要先判断该结点是否包含问题的解，如果包含问题的解，称为活结点，就从该结点出发继续探索下去；如果该结点不包含问题的解，称为死结点，则回溯到最近的活结点继续搜索，也就是逐层向其祖先结点回溯。

若用回溯法求问题的所有解，要回溯到根，且根结点的所有可行的子树都已被搜索遍才结束。若使用回溯法求任一个解，只要搜索到问题的一个解就可以结束。

许多复杂的、规模较大的问题都可以使用回溯法求解，它有“通用解题方法”的美称。用回溯法解题的一般步骤如下：

❶ 针对所给问题确定问题的解空间。首先应明确定义问题的解空间，问题的解空间应至少包含问题的一个（最优）解。

❷ 确定结点的扩展搜索规则。

❸ 以深度优先方式搜索解空间，并在搜索过程中用剪枝函数避免无效搜索。

设问题的解是一个 n 维向量 $a[]=(a_1,a_2,\cdots,a_n)$，其中分量 a_i 表示第 i 步的操作。所有满足约束条件的解向量组构成了问题的解空间。约束条件是 a_i（$1\leqslant i\leqslant n$）之间满足某种条件，记为 $Cond(a_i)$。在很多情况下，问题的解是单个元素，此时问题简化为求满足 Cond(x)的结果 x。

3.1.2　回溯法算法设计

1. 基本回溯法算法设计

基本回溯法算法设计主要有两个关键点。

- 结点扩展：类似暴力法的深度优先搜索的扩展方式。
- 剪枝：用约束函数剪除不满足约束条件的子树；用限界函数剪除不能得到最优解的子树。

回溯法求解的非递归框架如下：

```
int x[n],i;                                  //x 存放解向量,全局变量
初始化数组 x[];
i=1;
while(i>0(有路可走) and (未达到目标))        //还未回溯到头
{   if(i>n)                                  //搜索到叶结点
        搜索到 1 个解，输出;
    else                                     //处理第 i 个元素
    {   x[i]取第 1 个可能的值;
        while(x[i]不满足约束条件且在搜索空间内)
            x[i]下一个可能的值;
        if(x[i]在搜索空间内)
        {   标识占用的资源;
            i=i+1;                           //扩展下一个结点
        }
        else
        {   清理所占的状态空间;              //回溯
            i=i-1;
        }
    }
}
```

【例 3-1】采用回溯法的非递归框架求解 n 皇后问题的所有解。

解：用 STL 的 vector<int>向量 a 存放解向量，其元素为 a[0]、a[1]、…，其中 a[k]表示第 k（0≤k≤n−1）个皇后的列号（列号从 0 到 n−1）。

首先设置 a[0]=−1（每个新皇后的初始列号均为−1），当 a 向量存在元素时循环：取最近放入的皇后（a 的末尾元素）k，若 k==n，表示所有皇后放置完毕，输出 a 作为一个解，否则从 a[k]+1 到 n−1 试探其合适的列号 j，即 Cond(k,j)表示该位置是否与 a[0]～a[k−1]的皇后冲突，当 Cond(k,j)为 true 时修改 a[k]=j，并放置第 k+1 个皇后（初始列号为−1）；如果第

k 行不能放置第 k 个皇后，将其删除（退栈回溯），再试探第 k-1 个皇后。

对应的完整程序如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <vector>
using namespace std;
vector<int> a;                  //定义一个解向量（起到栈的作用）
int count=0;                    //用于统计解个数
void dispasolution()            //输出一个解
{   printf("  第%d 个解:",++count);
    for(int i=0;i<a.size()-1;i++)
        printf("(%d,%d) ",i+1,a[i]+1);
    printf("\n");
}
bool Cond(int k,int j)      //测试第k个皇后的位置(k,j)是否与1～k-1的皇后有冲突
{   if(k==0) return true;       //放第一个皇后时没有冲突
    for(int i=0;i<k;i++)
    {   if((a[i]==j) || (abs(j-a[i])==abs(i-k)))
            return false;       //同列或者同对角线有冲突时返回假
    }
    return true;                //没有冲突时返回真
}
void Queen(int n)               //求解 n 皇后问题
{   bool find;
    a.push_back(-1);            //第 1 个皇后进栈 a,其列号初始值为-1
    while(a.size()>0)           //栈不空时循环
    {   int k=a.size()-1;       //试探栈顶的第 k 个皇后（a 中最后一个皇后）
        if(k==n)                //若所有皇后均放好,输出一个解
            dispasolution();
        find=false;             //尚未找到第 k 个皇后的位置，find 设置为假
        for(int j=a[k]+1;j<n;j++)   //为第 k 个皇后找一个合适的列号
            if(Cond(k,j))       //在第 k 行找到一个放皇后的位置(k,j)
            {   a[k]=j;         //修改第 k 个皇后的位置（新列号）
                find=true;      //找到第 k 个皇后的位置，find 设置为真
                break;          //找到后退出 for 循环
            }
        if(find)                //在第 k 行找到一个放皇后的位置(k,j)
            a.push_back(-1);    //新的第 k+1 皇后进栈（其列号初始值为-1）
        else                    //若第 k 个皇后没有合适的位置，回溯
            a.pop_back();       //即将第 k 个皇后退栈
```

```
    }
}
void main()
{   int n;                          //n 存放实际皇后个数
    printf("皇后问题(n<20) n=");
    scanf("%d",&n);
    if(n>20)
        printf("n 值太大\n");
    else
    {   printf(" %d 皇后问题求解如下:\n",n);
        Queen(n);
    }
}
```

上述程序求解 6 皇后的结果如图 3.1 所示。

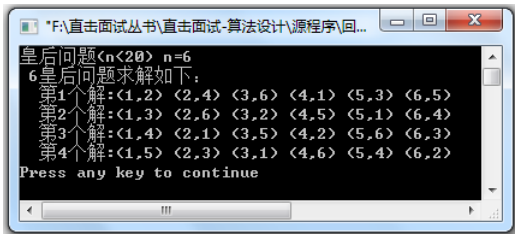


图 3.1　程序的执行结果

回溯法求解的递归算法框架如下：

```
int x[n];                           //x 存放解向量,全局变量
void backtrack(int i)               //递归框架
{   if(i>n)                         //搜索到叶结点,输出一个可行解
        输出解;
    else
    {   for(j=下界;j<=上界;j++)      //用 j 枚举 i 所有可能的路径
        {   x[i]=j;                 //产生一个可能的解分量
            …                       //其他操作
            if(Cond(i))             //满足约束条件
                backtrack(i+1);
        }
    }
}
```

【例 3-2】采用回溯法的递归框架求解 n 皇后问题的所有解。

解：用一维数组 a 存放解向量，其元素为 a[0]、a[1]、…、a[n-1]，a[k]表示第 k（0≤k≤n-1）个皇后的列号（列号从 0 到 n-1）。k 从 0 开始，当 k≥n 时输出一个解，否则在第 k 行查找一个合适的列号 j 放置第 k 个皇后，再递归调用放置第 k+1 个皇后。由于是递归算法，在执行中自动回溯并保存状态。对应的完整程序如下：

```
#include <stdio.h>
#include <stdlib.h>
#define MAXN 21                     //最多皇后个数
int a[MAXN];                        //存放各皇后所在的列号
int count=0;                        //用于统计解个数
void dispasolution(int n)           //输出一个解
{   printf("  第%d 个解: ",++count);
    for(int i=0;i<n;i++)
        printf("(%d, %d) ",i+1,a[i]+1);
    printf("\n");
}
bool Cond(int k,int j)      //测试第 k 个皇后的位置(k,j)是否与 1～k-1 的皇后有冲突
{   if(k==0) return true;           //放第 1 个皇后时没有冲突
    for(int i=0;i<k;i++)
    {   if((a[i]==j)||(abs(j-a[i])==abs(i-k)))
            return false;           //同列或者同对角线有冲突时返回假
    }
    return true;                    //没有冲突时返回真
}
void Queen(int k,int n)             //求解 n 皇后问题
{   if (k>=n)                       //若所有皇后均放好,输出一个解
        dispasolution(n);
    else
    {   for(int j=0;j<n;j++)        //在第 k 行上穷举每一个位置
            if(Cond(k,j))           //在第 k 行上找到一个合适的位置(k,j)
            {   a[k]=j;             //放好第 k 个皇后
                Queen(k+1,n);       //递归放置第 k+1 个皇后
            }
    }
}
void main()
{   int n;                          //n 存放实际皇后个数
    printf(" 皇后问题(n<20) n:");
    scanf("%d",&n);
    if (n>20)
        printf("n 值太大,不能求解\n");
```

```
        else
        {   printf("%d 皇后问题求解如下:\n",n);
            Queen(0,n);
            printf("\n");
        }
    }
```

2. 子集树和排列树

在回溯法中常见的两类典型的解空间是子集树和排列树。

❶ 当所给的问题是从 n 个元素的集合 S 中找出满足某种性质的子集时，相应的解空间树称为子集树。例如，0/1 背包问题的解空间就是一棵子集树。回溯算法搜索子集树的一般递归框架如下：

```
int x[n];
void backtrack(int i)           //i 为树的根度,1 或者 0 为根
{   if(i>n)                     //达到叶结点
        找到一个解;
    else
    {   for(int j=0;j<=1;j++)   //每个结点只有两棵子树
        {   x[i]=j;
            if(Cond(i)) backtrack(i+1);
        }
    }
}
```

❷ 当所给的问题是确定 n 个元素满足某种性质的排列时，相应的解空间树称为排列树。例如，调度问题的解空间就是一棵排列树。回溯算法搜索排列树的一般递归框架如下：

```
int x[n];
void backtrack(int i)           //i 为树的根度,1 或者 0 为根
{   if(i>n)                     //达到叶结点
        找到一个解;
    else
    {   for(intj=i;j<=n;j++)    //每个结点只有两棵子树
        {   //为了保证排列中的每一个元素不同,通过 x[i]、x[j]交换来实现
            swap(x[i],x[j]);
            if(Cond(i)) backtrack(i+1);
            swap(x[i],x[j]);    //恢复
        }
    }
}
```

3.1.3 回溯法算法的时间性能分析

通常以回溯算法的解空间树中的结点数作为算法的时间分析依据，假设解空间树共有 n 层，第 1 层有 m_0 个满足约束条件的结点，每个结点有 m_1 个满足约束条件的结点，第 2 层有 m_0m_1 个满足约束条件的结点，第 3 层有 $m_0m_1m_2$ 个满足约束条件的结点，以此类推，若第 n 层有 $m_0m_1\cdots m_{n-1}$ 个满足约束条件的结点，则采用回溯法求所有解的算法的执行时间为 $T(n)=m_0+m_0m_1+m_0m_1m_2+\cdots+m_0m_1m_2\cdots m_{n-1}$。

例如，采用回溯法求有向图中顶点 s 到顶点 t 的最短路径，对应的解空间数是以顶点 s 为根的一棵树，时间复杂度为 $O(k^n)$，k 为图中顶点出度的最大值。

一般情况下，当解空间为子集树时通常有 2^n 个叶结点，遍历整个子集树的时间为 $O(2^n)$；当解空间为排列树时通常有 n!个叶结点，遍历整个排列树的时间为 O(n!)，所以需要通过剪枝提高时间性能。

3.2 面试题解析

【面试题 3-1 ★★★】求解部分元素的全排列问题。设计一个算法求 1～n 的 n 个整数中取出 m 个整数的全排列，要求在每个排列中每个整数只能取一次。

解：采用递归回溯方法，用 x[0..m−1]存放一个排列，x[i]为排列中的第 i 个整数，依次搜索每一个 x[i]，设计一个布尔数组 used[MAXN]，used[i]表示第 i 个整数是否在当前排列中，达到每个整数只能取一次的要求。对应的程序如下：

```
#include <stdio.h>
#include <string.h>
#define MAXN 20
#define MAXM 10
int m,n;
int x[MAXM];
bool used[MAXN];                //used[i]表示第 i 个元素是否在当前排列中
int count=0;                    //统计部分全排列的个数
void Partperm(int i)            //求 n 个元素中 m 个元素的全排列
{   int j;
    if(i>m)
    {   printf("  [");
        for(j=1;j<=m;j++)
            printf("%d ",x[j]); //输出一个排列
        printf("]  ");
        if(++count%5==0)        //每行输出 5 个排列
```

```
            printf("\n");
        }
        else
        {   for(j=1;j<=n;j++)
            {   if(!used[j])
                {   used[j]=true;   //修改 used[i]
                    x[i]=j;         //记录 x[i]
                    Partperm(i+1);  //继续搜索排列的下一个元素
                    used[j]=false;  //回溯：还原 used[i]
                }
            }
        }
    }
    int main()
    {   m=2,n=5;
        memset(used,0,sizeof(used));
        Partperm(1);                //从 1 开始搜索
        printf("count=%d\n",count);
        return 0;
    }
```

上述程序的执行结果如图 3.2 所示。

图 3.2　程序的执行结果

【面试题 3-2 ★★★★★】求解填字游戏问题。在 3×3 个方格的方阵中要填入数字 1～10 的某 9 个数字，每个方格填一个整数，使所有相邻两个方格内的两个整数之和为素数。试求出所有满足这个要求的各种数字填法。

解：采用回溯法求解的基本框架如下。

```
void solve()
{   int pos=0
    bool ok=true;
    int n=8;
    do
```

```
        {   if(ok)
            {   if(pos==n)
                {   输出解;
                    调整;
                }
                else 扩展;
            }
            else 调整;
            ok=检查前 pos 个整数填放的合理性;
        } while(pos>=0);
    }
```

首先是 3×3 个方格的布局，采用一个一维数组 a[9]存放一种填字方案，与 3×3 个方格 b 的对应关系如图 3.3 所示，即 b[0][0]存放在 a[0]中、b[0][1]存放在 a[1]中、…、b[2][2]存放在 a[8]中。在这种存储结构中如何确定 a 的 i 位置的相邻方格呢？采用向前判断的方式：

- a 的 0 位置没有前向的相邻方格。
- a 的 1 位置的前向的相邻方格是 a[0]。
- a 的 2 位置的前向的相邻方格是 a[1]。
- a 的 3 位置的前向的相邻方格是 a[0]。
- a 的 4 位置的前向的相邻方格是 a[1]和 a[3]。
- a 的 5 位置的前向的相邻方格是 a[2]和 a[4]。
- a 的 6 位置的前向的相邻方格是 a[3]。
- a 的 7 位置的前向的相邻方格是 a[4]和 a[6]。
- a 的 8 位置的前向的相邻方格是 a[5]和 a[7]。

为此，采用一个二维数组表示相邻方格如下（a 的 i 位置的相邻方格对应第 i 行，以-1 表示结尾）：

```
int Checkmatrix[][3]={ {-1},{0,-1},{1,-1},{0,-1},{1,3,-1},
                       {2,4,-1},{3,-1},{4,6,-1},{5,7,-1} };
```

	0	1	2
0	0	1	2
1	3	4	5
2	6	7	8

图 3.3　3×3 个方格 b 采用一维数组 a 表示

另外，用 used[1..10]表示对应数字是否使用过，used[i]=true 表示数字 i 没有使用，后面可以使用它填字，used[i]=false 表示数字 i 已经填入，不能再使用它。

对应的完整程序如下：

```
#include <stdio.h>
#include <math.h>
#define N 10
bool used[N+1];                    //used[i]=true 表示数字 i 可以使用
int a[9];                          //存放 3×3 个方格
int count=0;                       //统计解个数
int Checkmatrix[][3]={ {-1},{0,-1},{1,-1},{0,-1},{1,3,-1},
                       {2,4,-1},{3,-1},{4,6,-1},{5,7,-1} };
void dispasolution(int a[])        //输出一个解
{   int i,j;
    printf("解%d\n",++count);
    for(i=0;i<3;i++)
    {   for(j=0;j<3;j++)
            printf("%3d",a[3*i+j]);
        printf("\n");
    }
}
bool isPrime(int m)                //判断 m 是否为素数
{   bool flag=true;
    for (int i=2;i<=sqrt(m);i++)
        if (m%i==0)
            return false;
    return true;
}
bool Check(int pos)        //检查 a 中 pos 位置的相邻两个方格内的数字之和是否为素数
{   int i,j;
    if(pos<0) return 0;
    for(i=0;(j=Checkmatrix[pos][i])>=0;i++)
        if(!isPrime(a[pos]+a[j]))   //有一个不是素数,则返回 false
            return false;
    return true;
}
int selectnum(int start)           //从 start 位置开始选择一个没有使用的数字
{   for(int j=start;j<=N;j++)
        if(used[j]) return j;
    return 0;                      //没有合适的数字返回 0
}
int extend(int pos)                //扩展：为 pos 位置选择一个没有使用的数字,pos++
{   a[++pos]=selectnum(1);         //扩展过程都是从 1 开始选择数字的
```

```
    used[a[pos]]=0;             //标识该数字已使用
    return pos;
}
int change(int pos)             //调整：从 pos 开始回溯
{   int j;
    //为 pos 位置选择另外一个数字,为了避免重复,是从原数字的下一个数字开始选取的
    //若不能为 pos 选择一个数字,则回溯,即恢复 a[pos]为可以使用的,再执行 pos--
    while(pos>=0 && (j=selectnum(a[pos]+1))==0)
        used[a[pos--]]=true;
    if(pos<0) return -1;        //全部回溯完毕,返回-1 算法结束
    used[a[pos]]=true;          //为 pos 位置找到一个没有使用的数字 j
    a[pos]=j;                   //pos 位置放置数字 j
    used[j]=false;              //标识数字 j 已经使用过
    return pos;                 //返回该回溯的新位置
}
void solve()                    //求解算法
{   bool ok=true;               //当前填数是否有效
    int pos=0;                  //从位置 0 开始
    a[pos]=1;                   //在 pos 位置填入 1
    used[a[pos]]=0;             //标识数字 1 已经使用过
    do
    {   if(ok)
        {   if(pos==8)
            {   dispasolution(a);
                pos=change(pos);
            }
            else pos=extend(pos);
        }
        else
            pos=change(pos);
        ok=Check(pos);
    } while(pos>=0);
}
int main()
{   for(int i=1;i<=N;i++)       //初始化,所有数字均可以使用
        used[i]=true;
    solve();
    printf("count=%d\n",count);//输出数字填法总数
    return 0;
}
```

上述程序修改 N 可以得到不同的结果，这里 N=10，结果有 128 种填字方案，如果 N=12，得到 768 种填字方案。

【面试题 3-3 ★★★★】求解 0/1 背包问题。有 n 个重量分别为 w_0、w_1、…、w_{n-1} 的物品，它们的价值分别为 v_0、v_1、…、v_{n-1}，给定一个容量为 W 的背包，设计从这些物品中选取一部分物品放入该背包的方案，每个物品要么选中要么不选中，要求选中的物品不仅能够放到背包中，而且具有最大的价值。

解：设最优解为 $X=(x_0, x_1,\cdots,x_{n-1})$，$x_i=1$ 表示选取物品 i，$x_i=0$ 表示不选取物品 i。在其解空间树中，从某个状态（结点）出发扩展分为两种情况：

- 如果背包中可以放下 i 位置的物品，则选择该物品，下一个状态为(i+1,tw+w[i],tv+v[i],op)（置 op[i]=1）。
- 不选择 i 位置的物品，下一个状态为(i+1,tw,tv,op)。

第 1 种情况对应左分支，即扩展后指向左孩子的边表示选择物品 i 放入背包，使背包中物品的总重量增加 w[i]、总价值增加 v[i]，用 tw 表示背包的剩余容量、tv 表示当前背包中所有物品的总价值。左分枝可以通过 tw+w[i]<=W 的剪枝条件。

第 2 种情况对应右分支，即扩展后指向右孩子的边表示不选择物品 i 放入背包，背包中物品的总重量和总价值保持不变。可能的解结点都在最底层的叶结点中，每个叶结点表示在考虑了 n 个物品的取舍之后的一种最终状态。找出满足条件 tw≤W && tv>maxv 的最大叶结点，根据从根结点到叶结点的路径信息即可求出问题的解。

对应的程序如下：

```
#include <stdio.h>
#define MAXN 20                          //最多物品数
//问题表示
int w[]={5,3,2,1};                       //存放 4 个物品重量
int v[]={4,4,3,1};                       //存放 4 个物品价值
int n=4;                                 //4 种物品
int W=7;                                 //限制重量不超过 7
//求解结果表示
int maxv;                                //存放最优解的总价值
int maxw;                                //存放最优解的总重量
int x[MAXN];                             //存放最终解
void Knap(int i,int tw,int tv,int op[])//从物品 i 开始深度优先搜索
{   int j;
    if(i==n)                             //找到一个叶结点
    {   if(tw<=W && tv>maxv)             //找到一个满足条件的更优解,保存它
        {   maxv=tv;
            maxw=tw;
            for(j=0;j<n;j++)
```

```
                    x[j]=op[j];
            }
        }
        else                                //尚未找完所有物品
        {   if(tw+w[i]<=W)                  //剪枝
            {   op[i]=1;                    //选取物品 i
                Knap(i+1,tw+w[i],tv+v[i],op);
            }
            op[i]=0;                        //不选取物品 i,回溯
            Knap(i+1,tw,tv,op);
        }
    }
    void dispasolution(int x[],int n)  //输出一个解
    {   int i;
        printf("最佳装填方案是:\n");
        for(i=0;i<n;i++)
            if(x[i]==1)
                printf("  选取第%d 个物品\n",i);
        printf("总重量=%d,总价值=%d\n",maxw,maxv);
    }
    int main()
    {   //如果 w、v、n 和 W 由输入得到，这里添加相应的代码
        int op[MAXN];                       //存放临时解
        Knap(0,0,0,op);                     //从物品 0 开始搜索最优解
        dispasolution(x,n);
        return 0;
    }
```

上述程序的执行结果如图 3.4 所示。

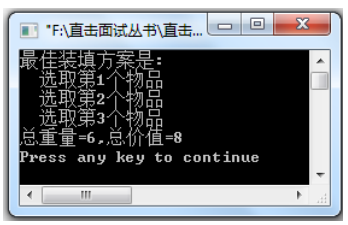


图 3.4　程序的执行结果

【面试题 3-4 ★★★★】求解子集和问题。给定 n 个不同的正整数集 $w=(w_0,w_1,\cdots,w_{n-1})$ 和一个正数 W，要求找出 w 的子集 s，使该子集中所有元素的和为 W。例如，当 n=4 时 w=(1,5,2,3)，W=6，则满足要求的子集为(1,5)和(1,2,3)。

解：本题与 0/1 背包问题类似，只是这里需要搜索所有的解。设问题解 $x=(x_0,x_1,\cdots,x_{n-1})$，$x_i$ 为 1 时表示取 w_i 整数，x_i 为 0 时表示不取 w_i 整数。

首先求出 w 的所有整数和 rw，从 i=0 开始搜索，当前状态是选择的整数和为 tw=0，剩余的整数和为 rw。左分枝表示选择 w_i，满足的条件是 tw+w[i]<=W（剪枝：剪除选择 w_i 超过 W 的结点扩展）；右分枝表示不选择 w_i，满足的条件是 tw+rw>=W（剪枝：剪除已经选择的整数和加上剩余的整数和小于 W 的结点扩展）。

对应的程序如下：

```
#include <stdio.h>
#define MAXN 20                     //最多整数个数
//问题表示
int w[]={1,5,2,3};                  //存放所有整数
int n=4,W=6;
//求解结果表示
int count=0;                        //累计解个数
void dispasolution(int x[])         //输出一个解
{   printf("第%d 个解:\n",++count);
    printf("  选取的整数为");
    for(int i=0;i<n;i++)
        if(x[i]==1)
            printf("%d ",w[i]);
    printf("\n");
}
void subSum(int tw,int rw,int x[],int i)//从整数 i 开始深度优先搜索
{   //tw 为当前选取整数的和，rw 为剩余的整数和
    if(i==n)                        //找到一个叶结点
    {   if(tw==W)                   //找到一个满足条件的解,输出它
            dispasolution(x);
    }
    else                            //尚未找完所有物品
    {   rw-=w[i];                   //求剩余的整数和
        if(tw+w[i]<=W)              //左孩子结点剪枝：选取满足条件的整数 w[i]
        {   x[i]=1;                 //选取第 i 个整数
            subSum(tw+w[i],rw,x,i+1);
        }
        if(tw+rw>=W)                //右孩子结点剪枝：剪除不可能存在解的结点
        {   x[i]=0;                 //回溯：不选取第 i 个整数
            subSum(tw,rw,x,i+1);
        }
    }
```

```
}
int main()
{   //如果 w、v、n 和 W 由输入得到，这里添加相应的代码
    int x[MAXN];                        //存放解向量
    int rw=0;
    for(int i=0;i<n;i++)                //求所有整数和 rw
        rw+=w[i];
    subSum(0,rw,x,0);                   //从 w0 开始搜索,剩余整数和为 rw
    return 0;
}
```

【面试题 3-5 ★★★★★】求解集合中的加法等式问题。给定 n（2≤n≤30）个互不相同的正整数，其中可能有两个或者两个以上整数之和等于另外一个整数。例如，n=6，a[]={1,2,3,5,4,6}，结果有 1+2=3、1+3=4、1+4=5、1+5=6、2+3=5、2+4=6、1+2+3=6，你需要将所有这样的加法等式找出来，并输出其个数。

解：首先对 a[0..n−1]整数序列进行递增排序，用 num 表示求几个元素之和，num 从 2 到 n−2 循环，pos 数组存放构成加法等式的元素位置。

以 num=2 为例，对于 1+3=4，有 pos[0]=0（a[0]=1），pos[1]=2（a[2]=3）。采用深度优先搜索在 a[0..n−1]中找出任何两个元素 a[i]、a[j]（j>i），求出它们的和 sum，此时 pos={i,j}，如果 sum 在 a 中 pos[num−1]+1（=j+1）位置及其后面出现，得到一个解。

剪枝：若当前选择的元素和为 sum，对于考虑的第 i 个元素 a[pos[i]]，若 sum+a[pos[i]]>a[n−1]，说明 sum 加上 a[pos[i]]的和一定不是 a 中的元素（因为 a 是递增的）。

对应的完整程序如下：

```
#include <stdio.h>
#include <algorithm>
using namespace std;
#define MAX 31
//问题表示
int a[MAX]={1,2,3,5,4,6};           //输入的整数
int n=6;                            //输入的整数个数
//求解结果表示
int pos[MAX];                       //存放构成加法等式的元素位置
int num;                            //当前搜索到的位置
int Count=0;
void dispasolution(int sum)         //输出一个解
{   Count++;
    for(int i=0;i<num;i++)
    {   if(i!=0)
            printf("+");
```

```
        printf("%d",a[pos[i]]);
    }
    printf("=%d\n",sum);
}
bool isSum(int sum)                    //判断 sum 是否为 a[pos[num-1]]后面的元素
{   for(int i=pos[num-1]+1;i<n;i++)
        if(sum==a[i])
            return true;
    return false;
}
void solve(int i,int sum)              //在 a[0..pos[num-1]]范围内搜索第 i 个元素
{   if(i==num)
    {
        if(isSum(sum)) dispasolution(sum);
    }
    else
    {   int j,b;
        if(i==0) b=0;                  //b 为第 i 个整数的起始搜索位置
        else b=pos[i-1]+1;
        for(j=b;j<n-1;j++)
        {   pos[i]=j;
            if(sum+a[pos[i]]<=a[n-1])//剪枝
                solve(i+1,sum+a[pos[i]]);
        }
    }
}
int main()
{   sort(a,a+n);                       //递增排序
    for(int j=2;j<n-1;j++)
    {   num=j;
        solve(0,0);
    }
    printf("Count=%d\n",Count);
    return 0;
}
```

【面试题 3-6 ★★★★】求解棋盘问题。在一个给定形状的棋盘（形状可能是不规则的）上面摆放棋子，棋子没有区别，要求摆放时任意两个棋子不能放在棋盘中的同一行或者同一列，请编程求解对于给定形状和大小的棋盘摆放 k 个棋子的所有可行的摆放方案数 C。

输入：输入含有多组测试数据。每组数据的第 1 行是两个正整数 n 和 k，用一个空格

隔开，表示将在一个 n×n 的矩阵内描述棋盘，k 为摆放棋子的数目。n≤80，k≤n。当输入 n=−1 和 k=−1 时表示输入结束。随后的 n 行描述了棋盘的形状，每行有 n 个字符，其中'#'表示棋盘区域，'.'表示空白区域（数据保证不出现多余的空白行或者空白列）。

输出：对于每一组数据给出一行输出，输出摆放的方案数 C（数据保证 $C<2^{31}$）。

样例输入：

```
2 1          //第 1 组测试数据
#.
.#
4 4          //第 2 组测试数据
...#
..#.
.#..
#...
-1 -1
```

样例输出：

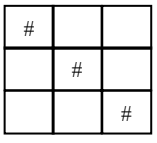
```
2
1
```

解：本题类似 N 皇后问题，但有几点不同。

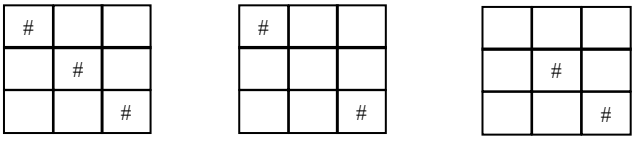
❶ 这里是给定一种棋盘布局，已经摆放了 n 个棋子，以此为基础通过回溯求摆放 k 个棋子有多少种摆放方案，而一般的 N 皇后问题是从空白棋盘开始摆放 N 个皇后的。

❷ 这里仅要求任意两个棋子不能放在同一行或者同一列，设置一个 visited 数组，visited[j]=1 表示第 j 列摆放了棋子。

例如，n=3、k=2 的求解结果如图 3.5 所示。采用回溯的深度优先遍历方法时对应的程序如下：



（a）n=3、k=2 的初始棋盘



（b）3 种方案

图 3.5　n=3、k=2 的求解结果

```
#include <stdio.h>
#include <string.h>
#define MAX 8
//问题表示
int n,k,ans;
char mat[MAX][MAX];                 //棋盘
//求解结果表示
int sum=0;
int visited[MAX];                   //标记数组
void dfs(int start,int num)
{   if(num==k)                      //摆放了 k 个棋子
    {   ans++;                      //方案数增 1
        return;
    }
    for(int i=start;i<n;i++)
        for(int j=0;j<n;j++)
            if(mat[i][j]=='#' && visited[j]==0)
            {   visited[j]=1;       //设置为 1
                dfs(i+1,num+1);     //放置第 i+1 行的棋子
                visited[j]=0;       //回溯：重置为 0
            }
}
int main()
{   while(scanf("%d %d%*c",&n,&k) && n!=-1 && k!=-1)
    {   memset(visited,0,sizeof(visited));
        for(int i=0;i<n;i++)
            gets(mat[i]);
        ans=0;
        dfs(0,0);
        printf("%d\n",ans);
    }
    return 0;
}
```

在 dfs(start,num)函数中，对于第 i 行，在搜索下一个棋子（即放置第 num+1 个棋子）时的条件是 mat[i][j]=='#' && visited[j]==0，表示考虑第 i 行原来放置的棋子，因为根据题目规定该行一定有一个棋子，如果有 mat[i][j]='#'并且 visited[j]=1，则需要忽略这一行（即该行不放置棋子，本题图中的第 2、3 种结果）。当然，也可以等价地改为如下：

```
void dfs(int start,int num)
{   if(num==k)                      //摆放了 k 个棋子
```

```
        {   ans++;                          //方案数增 1
            return;
        }
        for(int i=start;i<n;i++)
            for(int j=0;j<n;j++)
            {   if(mat[i][j]=='.' || visited[j]==1) continue;
                visited[j]=1;               //设置为 1
                dfs(i+1,num+1);             //放置下一个棋子
                visited[j]=0;               //回溯：重置为 0
            }
    }
```

【面试题 3-7 ★★★★】求解拜访亲戚问题。某个城市有 n 个车站，m 条双向公路连接其中的某些车站。每两个车站最多用一条公路直接相连，从任何一个车站出发都可以经过一条或多条公路到达其他车站，但不同的路径需要花费的时间可能不同。在一条路上花费的时间等于路径上所有公路需要的时间之和。佳佳的家在一个车站，他有 5 个亲戚，分别住在车站 a、b、c、d、e。过年了，他需要从自己的家出发拜访每个亲戚（顺序任意），给他们送去节日的祝福，怎样走需要的时间最少？

输入：第 1 行为 n（n≤50000）和 m（m≤100000），n 表示车站数目（假设所有车站编号为 0～n-1，佳佳的家所在车站的编号为 0），m 表示公路的数目。以下 m 行，每行 3 个整数 x、y、t（1≤x，y≤n，1≤t≤100），分别为公路连接的两个车站的编号和时间。

输出：仅一行，包含一个整数为最少的总时间。

样例输入：

```
6  6
0  1  8
1  2  3
2  3  4
3  4  5
4  5  2
0  5  7
```

样例输出：

```
21
```

解：本题假设佳佳拜访全部亲戚（顺序任意）时没有行走重复的公路，即路径上的边不重复出现，而且存在这样的路径，从而问题转化为从顶点 0 开始一笔画的过程。

采用方法是通过输入创建一个图的邻接表，从顶点 0 出发深度优先遍历所有顶点，记录路径长度，通过比较查找出最短路径长度。对应的完整程序如下：

```
#include <stdio.h>
```

```
#include <string.h>
#include <malloc.h>
#define MAXN 50000
#define INF 0x3f3f3f3f
typedef struct node
{   int adjvex;
    int weight;
    struct node *nextarc;
} ArcNode;
typedef struct
{
    ArcNode *firstarc;
} VNode;
typedef struct
{
    VNode adjlist[MAXN];
} AdjGraph;                     //声明邻接表类型
//问题表示
AdjGraph *G;                    //全局变量：邻接表指针
int n,m;                        //全局变量：顶点数和边数
//求解结果表示
int minpathlen=INF;             //全局变量：最短路径长度,初始值为 INF
int visited[MAXN];              //全局变量：顶点访问标记数组
void CreateGraph()              //创建空邻接表
{   G=(AdjGraph *)malloc(sizeof(AdjGraph));
    for(int i=0;i<n;i++)
        G->adjlist[i].firstarc=NULL;
}
void Addedge(int x,int y,int t)//在邻接表中添加一条无向边
{   ArcNode *p,*q;
    p=(ArcNode *)malloc(sizeof(ArcNode));
    p->adjvex=y;
    p->weight=t;
    p->nextarc=G->adjlist[x].firstarc;
    G->adjlist[x].firstarc=p;
    q=(ArcNode *)malloc(sizeof(ArcNode));
    q->adjvex=x;
    q->weight=t;
    q->nextarc=G->adjlist[y].firstarc;
    G->adjlist[y].firstarc=q;
}
```

```
void DestroyGraph()                    //销毁邻接表
{   ArcNode *pre,*p;
    for(int i=0;i<n;i++)
    {   if(G->adjlist[i].firstarc!=NULL)
        {   pre=G->adjlist[i].firstarc;
            p=pre->nextarc;
            while(p!=NULL)
            {   free(pre);
                pre=p;
                p=p->nextarc;
            }
            free(pre);
        }
    }
    free(G);
}
void dfs(int u,int num,int pathlen)//深度优先遍历查找最短路径
{   //num 表示访问过的顶点个数,初始为 0,pathlen 记录当前路径的长度
    int w;
    ArcNode *p;
    num++;                             //访问顶点 u,num 增 1
    visited[u]=1;                      //置已访问标记
    if(num==n)                         //找到一条访问过全部顶点的路径
    {   if(pathlen<minpathlen)         //比较找最优解
            minpathlen=pathlen;
    }
    p=G->adjlist[u].firstarc;          //p 指向顶点 u 的第 1 个邻接点
    while(p!=NULL)
    {   w=p->adjvex;                   //w 为顶点 u 的邻接点
        if(visited[w]==0)              //若 w 顶点未访问，递归访问它
            dfs(w,num,p->weight+pathlen);
        p=p->nextarc;                  //p 指向顶点 u 的下一个邻接点
    }
    visited[u]=0;                      //回溯：恢复环境,使该顶点可重新使用
}
int main()
{   int i,x,y,t;
    scanf("%d%d",&n,&m);
    CreateGraph();
    for(i=0;i<m;i++)
    {   scanf("%d%d%d",&x,&y,&t);
```

```
            Addedge(x,y,t);
        }
        memset(visited,0,sizeof(visited)); //初始化 visited 数组
        dfs(0,0,0);                        //从顶点 0 出发查找最短路径长度
        printf("%d\n",minpathlen);         //输出结果
        DestroyGraph();                    //销毁邻接表 G
        return 0;
    }
```

【面试题 3-8 ★★★★】求解货郎担（TSP）问题。有 n 个顶点，顶点编号为 0～n−1，通过一个邻接矩阵 A 来表示（A[i][i]=0，A[i][j]表示顶点 i 到顶点 j 的边长，A[i][j]=∞表示顶点 i 到顶点 j 没有边）。设计一个算法求从顶点 0 出发经过其他所有顶点再回到顶点 0 的最短路径长度（路径上的顶点不重复），例如如图 3.6 所示的带权有向图的求解结果为 23。

解：这里采用回溯法求解。TSP(i,V)算法求从顶点 i 出发经过 V（一个顶点的集合，用 set<int>整数集合表示）中的各个顶点一次且仅一次，最后回到出发点 s（本题中 s=0）的最短路径长度。初始时 V 包含除了 s 顶点外的其他顶点。

用 minpathlen 保存最短 TSP 路径长度（初始值为 INF），minpath 向量保存最短 TSP 路径。从顶点 i（初始为 0）开始搜索，path 存放当前的一条搜索路径（没有存放最后到顶点 0 的回边），pathlen 为其路径长度。当找到一个叶结点时，如果最尾的顶点 i 到顶点 0 有一条边，将顶点 0 添加到 path 中，求出其路径长度 pathlen，通过比较，将最短路径长度保存在 minpathlen 中、最短路径保存在 minpath 中。

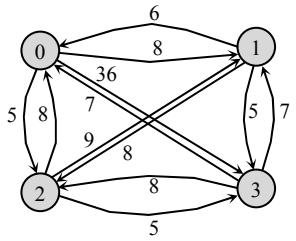


图 3.6　一个带权有向图

采用的剪枝规则：对于当前结点 i，如果对应的路径长度 pathlen+A[i][0]大于或等于 minpathlen，该结点变成死结点，不再扩展，否则需要进行扩展。注意不同于前面的大多数示例，这里一个活结点扩展出子结点的个数可能大于 2。

对应的完整程序如下：

```
#include <stdio.h>
#include <vector>
#include <set>
```

```
using namespace std;
#define MAX 100
#define INF 0x3f3f3f3f
typedef set<int> SET;                          //采用 set 表示集合
//问题表示
int A[][MAX]={                                 //一个带权有向图
        {0,8,5,6},{6,0,8,5},
        {8,9,0,5},{7,7,8,0} };
int n=4;
//求解结果表示
int minpathlen=INF;                            //保存最短 TSP 路径长度
vector<int> minpath;                           //保存最短 TSP 路径
int getpathlen(vector<int> &path)              //求 path 中路径的长度
{   int sum=0;
    for(int i=1;i<path.size();i++)
        sum+=A[path[i-1]][path[i]];
    return sum;
}
void dispapath(vector<int> &path)              //输出一条路径
{   for(int i=0;i<path.size();i++)
        printf("%d ",path[i]);
    printf("\n");
}
void TSP(int i,SET V,vector<int> path)             //求 TSP 路径
{   path.push_back(i);                             //在路径中添加顶点 i
    if(V.size()==0)                                //找到一个叶结点
    {   if(A[i][0]!=0 && A[i][0]!=INF)             //该顶点到起点 0 有边
        {   path.push_back(0);                     //在路径中添加到起点的回边
            int pathlen=getpathlen(path);          //求最短路径长度
            printf("  一条长度为%d 的路径: ",pathlen);
            dispapath(path);
            if(pathlen<minpathlen)
            {   minpathlen=pathlen;
                minpath=path;
            }
        }
    }
    else                                           //对于非叶结点
    {   int pathlen=getpathlen(path)+A[i][0];      //求出当前路径长度
        if(pathlen<minpathlen)                     //剪枝
        {   SET::iterator it;
```

```
            for(it=V.begin();it!=V.end();++it)//扫描集合 V 中的顶点 j
            {   SET tmpV=V;
                int j=*it;
                tmpV.erase(j);
                TSP(j,tmpV,path);                //递归调用
            }
        }
    }
}
int main()
{   vector<int> path;
    SET V;
    for(int i=1;i<n;i++) V.insert(i);            //将 1～n-1 插入到集合 V 中
    printf("求解过程\n");
    TSP(0,V,path);                    //用回溯法求从顶点 0 出发的 TSP 路径和长度
    printf("TSP 路径长度: %d\n",minpathlen);
    printf("最短路径: ");                        //输出最短路径
    dispapath(minpath);
    return 0;
}
```

上述程序的执行结果如图 3.7 所示。

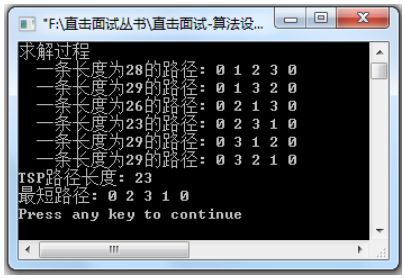


图 3.7　程序的执行结果

【面试题 3-9 ★★★★】求解组合问题。采用回溯法输出自然数 1～n 中任取 r 个数的所有组合。

解：这里采用基于深度优先搜索方法求解，用 path[0..d-1]数组存放一个组合（d 从-1 即空组合开始起），start 表示当前扫描的元素（从 1 开始起），cur 表示当前已经选择的元素个数（从 0 开始起）。算法思路如下：

```
void dfs(int path[],int d,int start,int cur)
{   if(cur==r)
        输出 path 中的一个组合;
    for(i=start;i<=n;i++)
```

```
        {   将元素 i 添加到 path 数组中;
            dfs(path,d,i+1,cur+1);                  //选择元素 i
            d--;                                    //回溯,表示不选择元素 i
        }
    }
```

对应的完整程序和测试数据如下：

```
#include <stdio.h>
#define MAX 100
//问题表示
int n=5,r=3;                                        //全局变量
void disppath(int path[])                           //输出一个组合
{   for(int j=0;j<=r-1;j++)
        printf("  %d",path[j]);
    printf("\n");
}
void dfs(int path[],int d,int start,int cur)//深度优先搜索所有组合
{   if(cur==r)
        disppath(path);
    for(int i=start;i<=n;i++)
    {   d++; path[d]=i;
        dfs(path,d,i+1,cur+1);
        d--;                                        //回溯：不选择 i
    }
}
int main()
{   int path[MAX];
    printf("n=%d,r=%d 的所有组合如下:\n",n,r);
    dfs(path,-1,1,0);
    return 0;
}
```

上述程序的执行结果如图 3.8 所示。

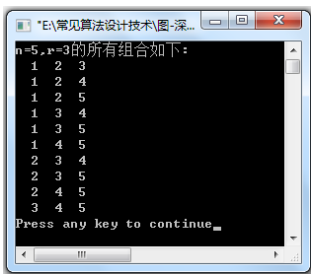


图 3.8　程序的执行结果

【面试题 3-10 ★★★】有一个递增有序整数序列，其中元素可能重复出现，求其所有的子集（结果中不能有两个相同的子集），例如 s={1,2,2}，输出结果是“{},{1},{1,2},{1,2,2},{2},{2,2}”（顺序不重要）。

解：采用第 2 章例 2-3 的思路，只是例 2-3 中的元素没有重复，相当于每个元素只能选择 0 次或者 1 次，这里每个元素可以选择 0 次或者多次，更加突出回溯的思路。对应的程序如下：

```
#include <stdio.h>
#define MAX 100
//问题表示
int s[MAX]={1,2,2};                    //s[0..n-1]为递增序列,设置为全局变量
int n=3;
//求解结果表示
int Count=0;                           //累计子集个数
void disppath(int path[,int d)         //输出一个子集path
{   printf("  输出一个子集: {");
    Count++;
    if(d!=-1)
    {   for(int i=0;i<d;i++)
            printf("%d,",path[i]);
        printf("%d",path[d]);
    }
    printf("}\n");
}
void dfs(int path[],int d,int start)   //求s[start..n-1]的不重复的子集
{   disppath(path,d);                  //输出一个子集path[0..d]
    for(int i=start;i<n;i++)           //i从start到n-1循环
    {   if(i!=start && s[i]==s[i-1])
            continue;                  //从start+1开始跳过重复的元素
        d++; path[d]=s[i];             //选择s[i]
        dfs(path,d,i+1);
        d--;                           //回溯：不选择s[i]
    }
}
int main()
{   int path[MAX];                     //path存放一个子集
    int d=-1;                          //d存放path中最后一个元素的下标
    printf("s的子集\n");
    dfs(path,d,0);                     //调用dfs(path,-1,0)
    printf("子集个数=%d\n",Count);
```

```
    return 0;
}
```

上述程序的执行结果如图 3.9 所示。

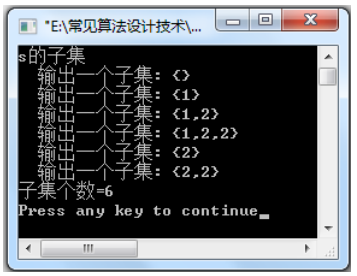


图 3.9　程序的执行结果

【面试题 3-11　★★★★】求含重复元素的序列的全排序。设集合 $R=\{r_1,r_2,\cdots,r_n\}$是要进行排列的 n 个元素，其中 r_1、r_2、…、r_n 可能相同。试着设计一个算法，列出 R 的所有不同排列，即给定 n 以及待排的 n 个可能重复的元素，计算输出 n 个元素的所有不同排列。

输入：第 1 行是元素的个数 n，$1\leqslant n\leqslant 15$。接下来的一行是待排列的 n 个元素，元素中间不要加空格。

输出：程序运行结束时，将计算输出 n 个元素的所有不同排列，最后一行是排列的总数（说明：此题计算出的所有排列顺序任意）。

样例输入：

```
4
aacc
```

样例输出：

```
aacc
acac
acca
caac
caca
ccaa
6
```

解：先看看求含 n 个不同字符的字符串 s 的全排列。设 f(s,n,k)表示输出 s[k..n−1]（共 n−k 个字符）所有字符的全排列，而 f(s,n,k+1)表示输出 s[k+1..n−1]（共 n−k−1 个字符）所有字符的全排列，前者是大问题，后者为小问题，对应的递归模型 f(str,k,n)如下：

```
f(s,n,k)≡输出产生的解                    若 k=n-1：只有一个字符时
f(s,n,k)≡对于 i=k～n-1:                  其他情况
```

```
                str[i]与 str[k]交换位置;
                f(str,n,k+1);
                将 str[k]与 str[i]交换位置（恢复环境）
```

对于本题，字符串 s 中含相同的字符，那么如何避免出现结果相同的排列呢？若 s="122"，采用的求全排列的过程如图 3.10 所示，显然出现了多个相同的排列。问题出现在 i=k～n−1 循环中，当 s[i]在 s[k.i−1]中出现时就跳过该轮循环，这样即可避免出现相同的排列，图中带阴影的部分就是修改后的结果，其中没有产生相同的排列。

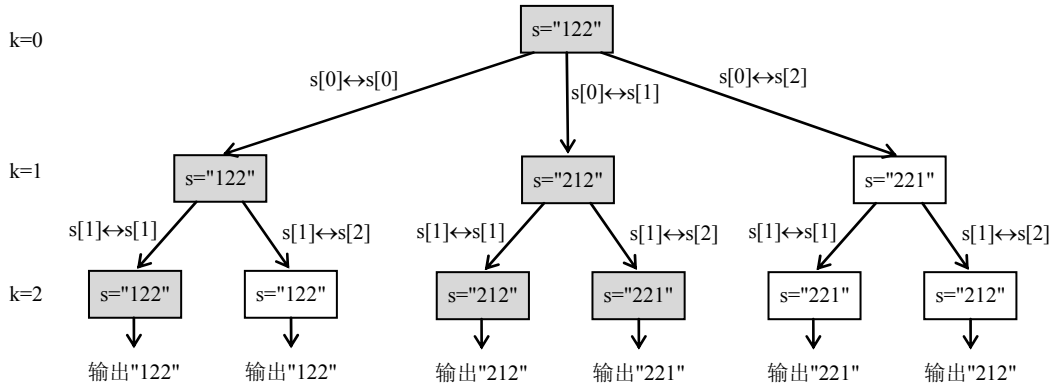


图 3.10　求"122"的全排列的过程

求含重复字符的字符串 s[0..n−1]的所有全排列的递归模型如下：

```
f(s,n,k)≡输出产生的解                    若 k=n-1：只有一个字符时
f(s,n,k)≡对于 i=k～n-1:                  其他情况
            if(s[i]不在 s[k..i-1]中)
            {   str[i]与 str[k]交换位置;
                f(str,n,k+1);
                将 str[k]与 str[i]交换位置（恢复环境）
            }
```

实际上，这也是一种深度优先搜索的暴力思路。对应的完整程序和测试数据如下：

```
#include <stdio.h>
#define MAX 20
//问题表示
char s[MAX]="1223";                 //s[0..n-1]为含重复字符的字符串
int n=4;
void swap(char &x,char &y)          //交换 x 和 y
{   char tmp=x;
    x=y;y=tmp;
}
int contains(int k,int i)           //判断 s[i]算法在 s[k..i-1]中
```

```
{   for(int j=k;j<i;j++)
        if(s[i]==s[j])
            return true;
    return false;
}
void dfs(int k)                         //产生 s[k..n-1]的所有排列
{   int i;
    if(k==n-1)
    {   printf("  一个排列: ");
        for(i=0;i<n;i++)                //只剩下一个元素
            printf("%c",s[i]);
        printf("\n");
    }
    else                                //还有多个元素待排列,递归产生排列
    {   for(i=k;i<n;i++)
        {   if(!contains(k,i))          //若 s[i]不在 s[k..i-1]中
            {   swap(s[k],s[i]);
                dfs(k+1);
                swap(s[k],s[i]);        //回溯：恢复到交换前的状态
            }
        }
    }
}
int main()
{   printf("%s 的全排列\n",s);
    dfs(0);
    return 0;
}
```

上述程序的执行结果如图 3.11 所示。

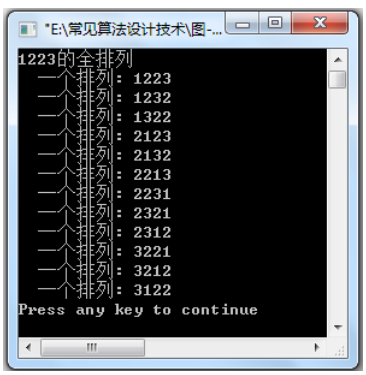


图 3.11　程序的执行结果

【面试题 3-12　★★★】求解图的 m 着色问题。给定一个含 n 个顶点（图顶点的编号为

0～n−1）的无向连通图和 m 种不同的颜色（颜色编号为 1～m），求所有的着色方案数，要求相邻两个顶点的颜色不能相同。

例如，n=5，m=4，输入图的边为(0,1)、(0,2)、(0,3)、(1,2)、(1,3)、(1,4)、(2,3)、(3,4)、(0,0)，输入边以(0,0)表示结束，对应的着色方案数为 48。

解：用二维数组 a 作为图的邻接矩阵。采用典型的回溯法，从顶点 i=0 开始，可以着色 1～m 中的一种，当出现颜色冲突时回溯，否则继续下去。当 i=n 时表示产生一种着色方案。对应的完整程序如下：

```
#include <stdio.h>
#include <string.h>
#define MAX 100
//问题表示
int a[MAX][MAX];
int n;                              //顶点个数
int m;                              //颜色数
//求解结果表示
int x[MAX];                         //解向量
int count=0;                        //着色的方案数
bool Same(int i)                    //判断顶点 i 是否与所有邻接点的着色都不相同
{   for(int j=0;j<n;j++)
        if(a[i][j]==1 && x[j]==x[i])
            return false;
    return true;
}
void solve(int i)                   //从顶点 i 开始搜索
{   if (i==n)                       //搜索的叶结点
        count++;                    //着色方案数增加 1
    else
    {   for(int c=1;c<=m;c++)       //扫描所有的颜色
        {   x[i]=c;                 //顶点 i 着色
            if(Same(i)) solve(i+1);//如果没有冲突,继续
            x[i]=0;                 //回溯
        }
    }
}
int main()
{   int i,j;
    memset(a,0,sizeof(a));
    memset(x,0,sizeof(x));
```

```
    scanf("%d%d",&n,&m);            //输入 n 和 m
    while(true)                     //输入无向图的边
    {   scanf("%d%d",&I,&j);
        if(i==0&&j==0)
            break;
        a[i][j]=1;                  //无向图的边是对称的
        a[j][i]=1;
    }
    solve(0);                       //从顶点 0 开始搜索
    printf("%d\n",count);
    return 0;
}
```

【面试题 3-13 ★★★★★】求解流水作业调度问题。有 n 个作业（编号为 1～n）要在由两台机器 M1 和 M2 组成的流水线上完成加工，每个作业加工的顺序都是先在 M1 上加工，然后在 M2 上加工，M1 和 M2 加工作业 i 所需的时间分别为 a_i 和 b_i（1≤i≤n）。

流水作业调度问题要求确定这 n 个作业的最优加工顺序，使得从第 1 个作业在机器 M1 上开始加工到最后一个作业在机器 M2 上加工完成所需的时间最少。可以假定任何作业一旦开始加工就不允许被中断，直到该作业被完成，即非优先调度。

输入：输入包含若干个用例，每个用例的第 1 行是作业数 n（1≤n≤1000），接下来 n 行，每行两个非负整数，第 i 行的两个整数分别表示第 i 个作业在第 1 台机器和第 2 台机器上加工的时间，以输入 n=0 结束。

输出：每个用例输出一行，表示采用最优调度所用的总时间，即从第 1 台机器开始到第 2 台机器结束的时间。

样例输入：

```
4
5 6
12 2
4 14
8 7
0
```

样例输出：

```
33
```

解：采用回溯法，其解空间是一个排列树，作业的编号是 1～n，用数组 x[]作为解向量，即调度方案（初值为元素分别是 1～n），最优解向量用 bestx[]存储，对应的最优调度时间用 bestf 表示。

f1 表示在机器 1 上执行当前作业 i 的总时间（含前面作业的执行时间），f2 数组表示机

器 2 的执行时间，元素 f2[i]表示机器 2 执行当前作业 i 的总时间（含前面作业的执行时间和可能的等待时间），所以 f2[n]就是执行全部作业的总时间。

类似递归求全排列问题，i 从第 1 层开始搜索，对于每个层次 i，前面已经处理了 x[1]～x[i−1]的作业，这一层考虑 j 从 i 到 n 循环处理后继作业。

当在第 i 层执行当前作业 x[j]时考虑机器 1 的时间，f1[i]=f1[i−1]+a[x[j]]，由于每个作业都是从机器 1 开始的（机器 1 不需要等待），所以 f1 不需要用数组表示，直接用 f1 表示，即 f1=f1+a[x[j]]。

考虑机器 2 的时间，这分为以下两种情况。

- 机器 2 空闲：即当前作业在 M_1 上执行后立即可在 M_2 上执行（满足 f2[i−1]≤f1 条件），则有 f2[i]=f1+b[x[j]]。
- 机器 2 不空闲：即当前作业在 M_1 上执行后，由于前一个作业还在 M_2 上执行，所以需要等到前一个作业全部执行完后才能执行（满足 f2[i−1]>f1 条件），则有 f2[i]=f2[i−1]+b[x[j]]。

剪枝的条件：当第 i 层求出的 f2[i]（即执行第 i 个作业 x[i]的总时间）已经大于等于 bestf（当前求出的执行全部作业的最优总时间）时就没有必要从该结点向下扩展了，让其成为死结点。

对应的完整程序如下：

```
 #include <stdio.h>
#include <string.h>
#define INF 0x3f3f3f3f
#define MAX 1001                //最多的作业数
//问题表示
int n=4;                        //作业数
int a[MAX]={0,5,12,4,8};        //M1 上的加工时间,不用下标 0 的元素
int b[MAX]={0,6,2,14,7};        //M2 上的加工时间,不用下标 0 的元素
//求解结果表示
int bestf;                      //存放最优调度时间
int f1;                         //机器 1 的执行时间
int f2[MAX];                    //机器 2 的执行时间
int x[MAX];                     //当前调度
int bestx[MAX];                 //存放当前作业最佳调度
void swap(int &x,int &y)        //交换 x 和 y
{   int tmp=x;
    x=y; y=tmp;
}
void disparr(int x[])           //输出一个数组的元素
{   for(int i=1;i<=n;i++)
        printf("%d ",x[i]);
}
void solve(int i)               //从 i 开始搜索
{   if(i>n)                     //到达叶结点,产生一种调度方案
```

```
        {   if(f2[n]<bestf)             //找到更优解
            {   bestf=f2[n];
                printf("   一个解: bestf=%d",bestf);
                printf(", 调度方案: ");disparr(x);
                printf(", f2: "); disparr(f2);
                printf("\n");
                for(int j=1;j<=n;j++)
                    bestx[j]=x[j];
            }
            return;
        }
        for(int j=i;j<=n;j++)           //没有到达叶结点,考虑 i 到 n 的作业
        {   f1+=a[x[j]];                //机器 1 当前的时间
            if(f2[i-1]>f1)              //如果机器 2 不空闲
                f2[i]=f2[i-1]+b[x[j]];
            else                        //如果机器 2 空闲
                f2[i]=f1+b[x[j]];
            if(f2[i]<bestf)             //剪枝：仅搜索当前总时间小于 bestf 的结点
            {   swap(x[i],x[j]);
                solve(i+1);
                swap(x[i],x[j]);
            }
            f1-=a[x[j]];                //回溯
        }
    }
    int main()
    {   //根据题目需要添加输入 n 和作业时间的代码
        f1=0;
        bestf=INF;
        memset(f2,0,sizeof(f2));
        for(int k=1;k<=n;k++)           //设置初始调度为作业 1、2、…、n 的顺序
            x[k]=k;
        printf("求解过程:\n");
        solve(1);                       //从作业 1 开始搜索
        printf("求解结果:\n");
        printf("    最少时间: %d",bestf);
        printf(", 最优调度方案: ");
        disparr(bestx);printf("\n");
        return 0;
    }
```

上述程序的执行结果如图 3.12 所示。

说明：本题采用回溯法的求解思路就是求 x[]={1,2,…,n}的全排列，对于每一种排列求出总时间，通过比较求出最短时间，其中通过剪枝条件进行剪枝。由于最优调度方案可能有多个，读者可以照此方法设计求最优调度方案的个数。

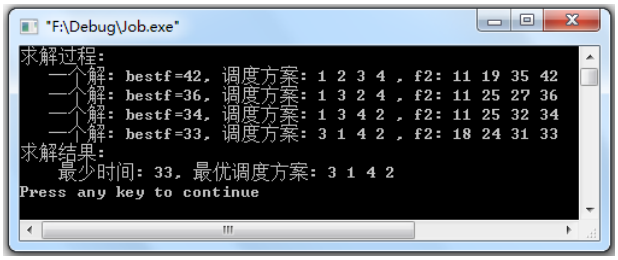


图 3.12　程序的执行结果

【面试题 3-14 ★★★★】求解进程调度问题。有 n 个进程（编号为 1～n），每个进程包含就绪时间 a_i 和服务时间 b_i，现在在一个 CPU 上执行，任何时刻只能执行一个进程，而且任何进程一旦开始执行就不允许被中断，直到该进程被完成（非优先调度）。在一个进程执行完毕后可以从等待的一个或者多个进程中选择一个执行，也可以没有进程执行。设计一种好的调度算法使得所有进程的等待时间最少，并给出一种进程调度顺序。例如，n=3，a[]={0,2,2}，b[]={10,1,6}，结果是所有进程的最少等待时间为 10，一种进程调度顺序是 2、3、1。

解：与前一个面试题类似，也是基于全排列思路。用数组 x 存放当前调度、数组 bestx 存放当前最优调度、wt 存放当前调度的所有进程等待时间、bestwt 存放最优等待时间，用数组 f[MAX]存放每个进程的执行时间，f[i]表示第 i 个进程执行完毕的时间。

当在第 i 层执行当前进程 x[j]时分为两种情况：

- 已经超过了进程 x[j]的就绪时间（a[x[j]]<f[i−1]），立即执行进程 x[j]，即 f[i]=f[i−1]+b[x[j]]，此时进程 x[j]存在等待时间。
- 尚未到达进程 x[j]的就绪时间（a[x[j]]≥f[i−1]），即 f[i]=a[x[j]]+b[x[j]]（进程 x[j]的执行完毕时间），此时进程 x[j]没有等待时间。

剪枝的条件：如果第 i 层求出的等待时间 wt 已经大于等于 bestwt（当前求出的执行全部进程的等待时间），就没有必要从该结点向下扩展了，让其成为死结点。

对应的完整程序如下：

```
#include <stdio.h>
#include <string.h>
```

```
#define INF 0x3f3f3f3f
#define MAX 1001                    //最多的进程数
//问题表示
int n=3;
int a[MAX]={0,0, 2,2};              //进程的就绪时间
int b[MAX]={0,10,1,6};              //进程的服务时间
//求解结果表示
int wt;                             //当前调度的等待时间
int bestwt;                         //存放最优等待时间
int f[MAX];                         //每个进程的执行时间
int x[MAX];                         //当前调度
int bestx[MAX];                     //存放当前最优调度
void swap(int &x,int &y)            //交换 x 和 y
{   int tmp=x;
    x=y; y=tmp;
}
void disparr(int x[])               //输出一个数组的元素
{   for(int i=1;i<=n;i++)
        printf("%3d",x[i]);
}
void solve(int i)                   //从 i 开始搜索
{   if(i>n)                         //到达叶结点,产生一种调度方案
    {   if(wt<bestwt)               //找到更优解
        {   bestwt=wt;
            printf("   一个解: bestwt=%d",bestwt);
            printf(", 调度方案: "); disparr(x);
            printf(", f: "); disparr(f);
            printf("\n");
            for(int j=1;j<=n;j++)
                bestx[j]=x[j];
        }
        return;
    }
    for(int j=i;j<=n;j++)           //没有到达叶结点,考虑 i 到 n 的作业
    {   int tmpf=f[i];
        if(a[x[j]]<f[i-1])          //立即执行进程 x[j]
            f[i]=f[i-1]+b[x[j]];
        else                        //进程 x[j]没有到达就绪时间
            f[i]=a[x[j]]+b[x[j]];
        int tmpwt=wt;
        wt+=f[i]-(a[x[j]]+b[x[j]]);
```

```
            if(bestwt>wt)           //剪枝：仅搜索当前等待时间小于 bestwt 的结点
            {   swap(x[i],x[j]);
                solve(i+1);
                swap(x[i],x[j]);
            }
            f[i]=tmpf;              //回溯
            wt=tmpwt;
        }
    }
    int main()
    {   //根据题目需要添加输入 n 和进程时间的代码
        bestwt=INF;
        wt=0;
        memset(f,0,sizeof(f));
        for(int k=1;k<=n;k++)       //设置初始调度为进程 1、2、…、n 的顺序
            x[k]=k;
        printf("求解过程:\n");
        solve(1);                   //从进程 1 开始搜索
        printf("求解结果:\n");
        printf("    最少等待时间: %d",bestwt);
        printf(", 最优调度方案: ");
        disparr(bestx);printf("\n");
        return 0;
    }
```

上述程序的执行结果如图 3.13 所示。

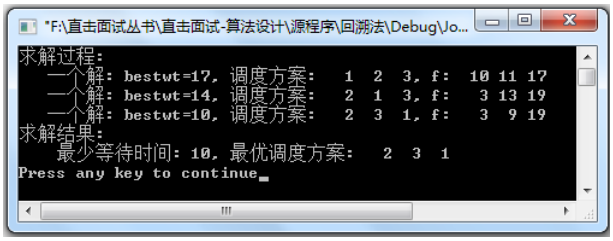


图 3.13　程序的执行结果

【面试题 3-15 ★★★】求解手机数字键盘字母组合问题。手机中数字对应的字母组合如图 3.14 所示，求用户按键的数字串对应的所有字母组合。例如，若用户按键的数字串 s="23"，输出[ad]、[ae]、[af]、[bd]、[be]、[bf]、[cd]、[ce]、[cf]。

解：用二维数组 keyboard[][MAX]存放数字键盘，例如 keyboard[2]="abc"表示数字 2 对应字母 a、b、c。

采用暴力思路，用基本的深度优先搜索方法，start 扫描 s 中的数字字符（从 0 开始），s[start]-'0'为对应的数字，扩展方案是选择 keyboard[s[start]-'0'][i]字符和不选择 keyboard[s[start]-'0'][i]字符两种，当 start≥n 时产生一种组合。对应的程序如下：

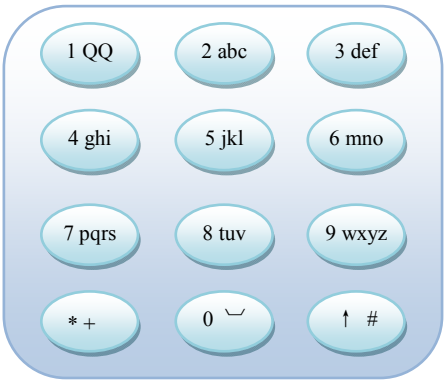


图 3.14　手机数字键盘

```
#include <stdio.h>
#define MAX 10
//问题表示
char s[MAX]="23";                //s[0..n-1]为给定的字符串,设置为全局变量
int n=2;
char keyboard[][MAX]={" ","","abc","def","ghi","jkl","mno","pqrs","tuv","wxyz"};
void disppath(char path[],int d)//输出一个组合
{   printf("  一个组合: [");
    for(int i=0;i<=d;i++)
        printf("%c",path[i]);
    printf("]\n");
}
void dfs(char path[],int d,int start)//求 s[start..n-1]对应的字母组合
{   if (start>=n)
        disppath(path,d);        //输出一个组合 path[0..d]
    else
    {   for(int i=0;keyboard[s[start]-'0'][i];i++)
                                 //处理 keyboard[s[start]-'0']字符串
        {   d++;path[d]=keyboard[s[start]-'0'][i];
                                 //选择 keyboard[s[start]-'0'][i]字符
            dfs(path,d,start+1);
            d--;                 //回溯：不选择 keyboard[s[start]-'0'][i]字符
        }
    }
```

```
}
int main()
{   int d=-1;                          //d 存放 path 中最后一个元素的下标
    char path[MAX];                    //path 存放一个组合
    printf("%s 对应的所有组合:\n",s);
    dfs(path,d,0);                     //调用 dfs(path,-1,0)
    return 0;
}
```

上述程序的执行结果如图 3.15 所示。

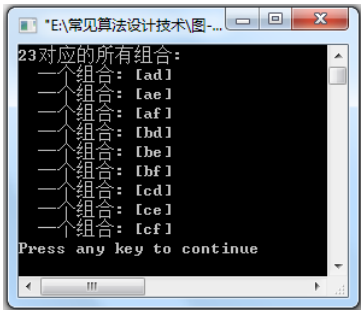


图 3.15　程序的执行结果

【面试题 3-16 ★★★★】求解任务分配的最优方案问题。有 4 个工人、4 个任务，每个工人完成不同的任务需要的时间不同，求任务分配的最优方案，即总时间最少的方案。

解：用二维数组 Cost[MAXN][MAXN]存放各任务中各个人需要的时间，行号为任务序号（从 0 到 n−1）、列号为工人序号（从 0 到 n−1）。

采用回溯法，bestcost 存放最优分配方案的时间（初始时为∞）；一维数组 bestx 存放最优分配方案，bestx[i]表示任务 i 分配的工人；一维数组 x 存放临时方案；一维数组 Worker 存放工人是否分配了任务，Worker[i]=1 表示工人 i 已经分配了任务，Worker[i]=0 表示工人 i 还没有分配任务（因为每个工人只能完成一个任务）。

剪枝：对于第 i 个任务，前面任务的累计时间为 cost，考虑分配给第 j 个工人（满足 Worker[j]=0），若 cost+Cost[i][j]≥bestcost，则该结点变为死结点（最基本的剪枝方法）。

对应的完整程序如下：

```
#include <stdio.h>
#include <string.h>
#define INF 0x3f3f3f3f
#define MAXN 10
//问题表示
int n=4;
```

```
int Cost[MAXN][MAXN]={ {2,12,5,32},{8,15,7,11},
                       {24,18,9,6},{21,1,8,28} };
//求解结果表示
int bestcost=INF;
int bestx[MAXN];                    //最优方案，bestx[i]表示任务 i 分配的工人
int x[MAXN];                        //临时方案
int Worker[MAXN];                   //工人是否分配了任务
void solve(int i,int cost)          //给任务 i 分配工人
{   if(i==n)                        //达到叶结点
    {   if(cost<bestcost)           //比较查找最优解
        {   bestcost=cost;
            for(int i=0;i<n;i++)
                bestx[i]=x[i];
        }
    }
    else                            //没有达到叶结点
    {   for(int j=0;j<n;j++)        //考虑所有工人
        {   if(Worker[j]==0&&cost+Cost[i][j]<bestcost) //剪枝
            {   Worker[j]=1;x[i]=j;                    //给任务 i 分配工人 j
                solve(i+1,cost+Cost[i][j]);
                Worker[j]=0;x[i]=0;                    //回溯
            }
        }
    }
}
int main()
{   memset(Worker,0,sizeof(Worker));
    solve(0,0);                     //从任务 0 开始分配
    printf("最佳方案总时间=%d\n",bestcost);
    for(int i=0;i<n;i++)            //输出最佳方案
        printf("\t 任务%d 由工人%d 来做:%d\n",i,bestx[i],Cost[i][bestx[i]]);
    return 0;
}
```

上述程序的执行结果如图 3.16 所示。

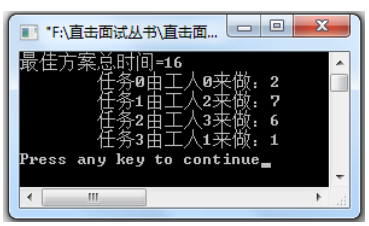


图 3.16　程序的执行结果

【面试题 3-17 ★★★★】求解最小重量机器设计 I 问题。设某一机器由 n 个部件组成，每一种部件都可以从 m（m≥n）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，要求所有部件在不同供应商处购买。

解：用二维数组 w[MAXN][MAXM]存放各部件各供应商购得的重量，行号为部件序号（从 0 到 n-1）、列号为供应商序号（从 0 到 m-1），w[i][j]表示部件 i 在供应商 j 购得的重量；用二维数组 c[MAXN][MAXM]存放各部件各供应商购得的价格，c[i][j]表示部件 i 在供应商 j 购得的价格。

采用回溯法，bestw 存放最优购买重量（初始时为∞）；bestc 存放最优购买价格（初始时为∞）；一维数组 bestx 存放最优分配方案，bestx[i]表示部件 i 分配的供应商；一维数组 x 存放临时方案；一维数组 used 存放部件是否分配了供应商，used[i]=1 表示部件 i 已经分配了供应商，used[i]=0 表示部件 i 还没有分配供应商（因为每个部件只能在一个供应商处购买）。

剪枝：对于第 i 个部件，前面部件的累计价格为 cost、前面部件的累计重量为 weight，考虑分配给第 j 个供应商（满足 used[j]=0），若 cost+c[i][j]≥C 或者 weight+w[i][j]≥bestw，则该结点变为死结点。

对应的完整程序如下：

```
#include <stdio.h>
#include <string.h>
#define INF 0x3f3f3f3f
#define MAXN 10
#define MAXM 10
//问题表示
int n=3;
int m=3;
int w[MAXN][MAXM]={{3,1,5},{1,3,5},{5,3,1}};
int c[MAXN][MAXM]={{12,4,6},{4,2,6},{6,2,4}};
int C=15;
//求解结果表示
int bestw=INF;                          //最优方案的总重量
int bestc=INF;                          //最优方案的总价格
int bestx[MAXN];                        //最优方案,bestx[i]表示部件 i 分配的供应商
int x[MAXN];                            //临时方案
int used[MAXN];                         //供应商是否分配了部件
void solve(int i,int cost,int weight)//给部件 i 分配供应商
{   if(i==n)                            //达到叶结点
    {   if(cost<=C && weight<bestw)     //比较查找最优解
```

```
            {   bestc=cost;
                bestw=weight;
                for(int i=0;i<n;i++)
                    bestx[i]=x[i];
            }
        }
        else                                    //没有达到叶结点
        {   for(int j=0;j<m;j++)                //考虑所有供应商
            {   if(used[j]==0 && cost+c[i][j]<C && weight+w[i][j]<bestw)//剪枝
                {   used[j]=1;x[i]=j;           //给部件 i 分配供应商 j
                    solve(i+1,cost+c[i][j],weight+w[i][j]);
                    used[j]=0;x[i]=0;           //回溯
                }
            }
        }
    }
    int main()
    {   memset(used,0,sizeof(used));
        solve(0,0,0);                           //从部件 0 开始分配
        printf("最佳方案:\n");
        printf("\t 总重量=%d, 总价格=%d\n",bestw,bestc);
        for(int i=0;i<n;i++)                    //输出最佳方案
            printf("\t 部件%d 在供应商%d 处购买\n",i,bestx[i]);
        return 0;
    }
```

上述程序的执行结果如图 3.17 所示。

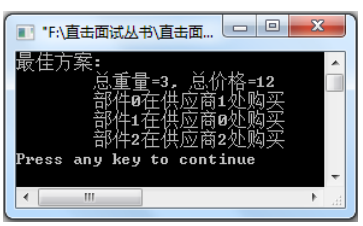


图 3.17　程序的执行结果

【面试题 3-18 ★★★★】 求解最小重量机器设计 II 问题。设某一机器由 n 个部件组成，每一种部件都可以从 m（m≥n）个不同的供应商处购得，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，可以在同一个供应商处购买多个部件。

解：与最小重量机器设计 I 问题相比，仅仅改为可以在同一个供应商处购买多个部件，

即不同的部件供应商 j 可以重复，为此将 used 一维数组改为二维数组。当 used[1][2]=1 时表示部件 1 在供应商 2 处购得，但 used[3][1]仍然为 0，不影响对部件 3 从供应商 2 处购买的判断。

对应的完整程序如下：

```
#include <stdio.h>
#include <string.h>
#define INF 0x3f3f3f3f
#define MAXN 10
#define MAXM 10
//问题表示
int n=3;
int m=3;
int w[MAXN][MAXM]={{3,1,5},{1,3,5},{5,3,1}};
int c[MAXN][MAXM]={{2,4,6},{4,2,6},{6,2,4}};
int C=10;
//求解结果表示
int bestw=INF;                          //最优方案的总重量
int bestc=INF;                          //最优方案的总价格
int bestx[MAXN];                        //最优方案，bestx[i]表示部件 i 分配的供应商
int x[MAXN];                            //临时方案
int used[MAXN][MAXM];                   //供应商是否分配了部件
void solve(int i,int cost,int weight)   //给部件 i 分配供应商
{   if(i==n)                            //达到叶结点
    {   if(cost<=C && weight<bestw)     //比较查找最优解
        {   bestc=cost;
            bestw=weight;
            for(int i=0;i<n;i++)
                bestx[i]=x[i];
        }
    }
    else                                //没有达到叶结点
    {   for(int j=0;j<m;j++)            //考虑所有供应商
        {   if(used[i][j]==0 && cost+c[i][j]<C && weight+w[i][j]<bestw)//剪枝
            {   used[i][j]=1;x[i]=j;    //给部件 i 分配供应商 j
                solve(i+1,cost+c[i][j],weight+w[i][j]);
                used[i][j]=0;x[i]=0;    //回溯
            }
        }
    }
}
```

```
int main()
{   memset(used,0,sizeof(used));
    solve(0,0,0);                    //从部件 0 开始分配
    printf("最佳方案:\n");
    printf("\t 总重量=%d,总价格=%d\n",bestw,bestc);
    for(int i=0;i<n;i++)             //输出最佳方案
        printf("\t 部件%d 在供应商%d 处购买\n",i,bestx[i]);
    return 0;
}
```

上述程序的执行结果如图 3.18 所示。

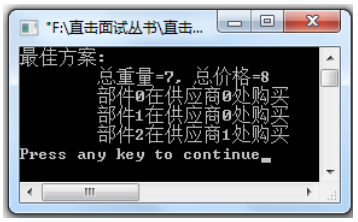


图 3.18　程序的执行结果

3.3　自测题和参考答案

3.3.1　自测题

3-1　以下算法中通常以深度优先方式搜索问题解的是（　　）。

A．备忘录法　　B．动态规划法　　C．贪心法　　D．回溯法

3-2　下面（　　）是回溯法中为避免无效搜索采取的策略。

A．递归函数　　B．剪枝函数　　C．随机数函数　　D．搜索函数

3-3　用回溯法解 0/1 背包问题时的解空间树是（　　）。

A．子集树　　B．排列树　　C．深度优先生成树　　D．广度优先生成树

3-4　用回溯法解旅行售货员（TSP）问题时的解空间树是（　　）。

A．子集树　　B．排列树　　C．深度优先生成树　　D．广度优先生成树

3-5　简要说明回溯法的求解过程。

3-6　在象棋算式中不同的棋子代表不同的数，有如图 3.19 所示的算式，设计一个算法求这些棋子各代表哪些数字。

```
    兵炮马卒
+   兵炮车卒
-----------
  车卒马兵卒
```

图 3.19　象棋算式

3-7　求解集装箱问题。有 n 个集装箱要装上一艘载重量为 W 的轮船，其中集装箱 i（1≤i≤n）的重量为 w_i。在装载体积不受限制的情况下尽可能将重的集装箱装上轮船，当总重量相同时要求选取的集装箱的个数尽可能少。例如，n=5，W=10，w={5,2,6,4,3}时，其最佳装载方案是“0,0,1,1,0”，即装载集装箱 2 和 3。

3-8　求放置炮塔个数问题。给定一个 n×n 的地图，'.'表示空地，'X'表示墙，现在往地图上放置炮塔，要求两两炮塔不能同行同列（除非之间有墙），给定一种地图，问这个地图最多可以放置多少个炮塔？例如，输入：

```
4            //n=4
.X..         //4 行 4 列的地图
....
XX..
....
```

其输出结果为 5。

3-9　给定一个含 n 个顶点的带权有向图，顶点编号为 0～n-1，每个顶点有一个风险值，用一个正整数表示，求从指定的顶点 s 到 t 的最优路径。一条路径上所有顶点的风险值之和为该路径的风险，最优路径是指风险最小的路径，当有多条这样的路径时选择路径长度最短的路径。

3-10　求网格中的黑点分布方案数。现有一个 ROWS×COLS 的网格，在某些格子中有黑点，已知各行与各列中有黑点的点数之和，分别用 Rows 和 Cols 数组表示，在这张网格中画出黑点位置的方案可能有多种，请求出对应的方案数。例如，ROWS=COLS=2，Rows[]={1,1}，Cols[]={1,1}，结果是 2。

3.3.2　参考答案

3-1　**答**：回溯法总是采用深度优先方式搜索问题解空间。答案为 D。

3-2　**答**：剪枝函数用于剪除不可能得到问题解的分枝。答案为 B。

3-3　**答**：用回溯法求解 0/1 背包问题时每个物品要么选择，要么不选择，等同于求一个集合的所有子集问题。答案为 A。

3-4　**答**：用回溯法求解旅行售货员问题时需要考虑所有顶点的排列，等同于求全排列

问题。答案为 B。

3-5　**答**：回溯法也称为试探法，该方法首先暂时放弃关于问题规模大小的限制，并将问题的候选解按某种顺序逐一枚举和检验。当发现当前候选解不可能是解时就选择下一个候选解；倘若当前候选解除了不满足问题规模要求外满足所有其他要求，此时继续扩大当前候选解的规模，并继续试探。如果当前候选解满足包括问题规模在内的所有要求，该候选解就是问题的一个解。在回溯法中，放弃当前候选解，寻找下一个候选解的过程称为回溯；扩大当前候选解的规模，以继续试探的过程称为向前试探。

3-6　**解**：设兵、炮、马、卒和车的取值分别为 a、b、c、d、e，则有 a、b、c、d、e 的取值范围为 0～9 且均不相等（即 a==b ‖ a==c ‖ a==d ‖ a==e ‖ b==c ‖ b==d ‖ b==e ‖ c==d ‖ c==e ‖ d==e 不成立）。

设 m=a×1000+b×100+c×10+d，n=a×1000+b×100+e×10+d，s=e×10000+d×1000+c×100+a×10+d，则有 m+n==s。

为了避免同一数字被重复使用，可设立逻辑数组 x，当 x[i]（0≤i≤9）的值为 1 时表示数 i 没有被使用。例如对于棋子兵，先试探它取值 a，让 x[a]=0 表示其他棋子不能再取值 a，继续其他棋子的试探，当不成功（放弃当前候选解）或输出一个解（找到一个解）后进行回溯，让 x[a]=1 表示其他棋子可以取值 a，即再试探其他候选解。对应的算法如下：

```
void fun()
{   int x[10];
    int a,b,c,d,e,i,m,n,s;
    for(i=0;i<=9;i++) x[i]=1;                    //置初值 1
    for(a=1;a<=9;a++)
    {   x[a]=0;                                  //试探兵取值 a
        for(b=0;b<=9;b++)
            if(x[b])
            {   x[b]=0;                          //试探炮取值 b
                for(c=0;c<=9;c++)
                    if(x[c])
                    {   x[c]=0;                  //试探马取值 c
                        for(d=0;d<=9;d++)
                            if(x[d])
                            {   x[d]=0;          //试探卒取值 d
                                for(e=0;e<=9;e++)
                                    if(x[e])
                                    {   x[e]=0; //试探车取值 e
                                        m=a*1000+b*100+c*10+d;
                                        n=a*1000+b*100+e*10+d;
                                        s=e*10000+d*1000+c*100+a*10+d;
                                        if(m+n==s)
```

```
                                        cout<<"兵:"<<a<<"炮:"<<
                                        b<<"马:"<<c<<"卒:"<<d
                                        <<"车:"<<e<<endl;
                                    x[e]=1; //回溯车的取值
                                }
                                x[d]=1;     //回溯卒的取值
                            }
                            x[c]=1;             //回溯马的取值
                        }
                        x[b]=1;                 //回溯炮的取值
                    }
                    x[a]=1;                     //回溯兵的取值
        }
    }
```

3-7　**解**：采用回溯法，思路与用回溯法求解 0/1 背包问题类似，只是需要统计选择的集装箱个数 m，在重量相同时以 m 较小作为最优解。对应的程序如下：

```
#include <stdio.h>
#define MAXN 20                         //最多集装箱个数
#define INF 0x3f3f3f3f
//问题表示
int w[]={5,2,6,4,3};                    //各集装箱重量
int n=5,W=10;
//求解结果表示
int maxw=0;                             //存放最优解的总重量
int minm=INF;                           //存放最优解的集装箱个数,初值为最大值
int x[MAXN];                            //存放最优解向量
void Loading(int tw,int m,int op[],int i)//求最优方案
{   //tw 为考虑集装箱 i 时装入的总重量，m 为已经装载的数量
    if(i==n)                            //找到一个叶结点
    {
        if(tw<=W && (tw>maxw || (tw==maxw && m<minm)))
        {   maxw=tw;                    //找到一个满足条件的更优解,保存它
            minm=m;
            for(int j=1;j<=n;j++)
                x[j]=op[j];
        }
    }
    else                                //尚未找完所有物品
    {   op[i]=1;                        //选取集装箱 i
        if(tw+w[i]<=W)                  //左孩子结点剪枝：装载满足条件的集装箱
```

```
            Loading(tw+w[i],m+1,op,i+1);
        op[i]=0;                        //不选取集装箱 I,回溯
        Loading(tw,m,op,i+1);
    }
}
void dispasolution()                    //输出最优解
{   printf("选取的集装箱:\n");
    for(int i=0;i<n;i++)
        if(x[i]==1)
            printf("  选取集装箱%d\n", i);
    printf("总重量=%d\n",maxw);
}
int main()
{   int op[MAXN];                       //存放临时解
    Loading(0,0,op,0);                  //从集装箱 0 开始搜索最优解
    dispasolution();
    return 0;
}
```

3-8　**解**：这里对地图中的每个位置 k（k=0～n(n−1)）扫描，每个位置分为可以放置炮塔和不能放置炮塔两种情况，当达到叶结点（k=n×n）时产生一个解，即放置炮塔的方案，但这里只是求放置炮塔的个数，所以用 number 记录一个放置方案中的炮塔个数，并且比较求出 ans=max{ number }，ans 就是最后的结果。对应的完整程序如下：

```
#include <iostream.h>
#define MAX 51
//问题表示
int n;                              //地图大小
char map[MAX][MAX];                 //地图
//求解结果表示
int ans;                            //放置炮塔的最多个数
bool Camset(int row,int column)     //能否在(row,column)位置放置炮塔
{   for(int i=row-1;i>=0;--i)
    {   if(map[i][column]=='O') return false;
        if(map[i][column]=='X') break;
    }
    for (int j=column-1;j>=0;--j)
    {   if(map[row][j]=='O') return false;
        if(map[row][j]=='X') break;
    }
    return true;
```

```
}
void solve(int k,int number)            //从位置 k 开始搜索
{   int row,column;
    if(k==n*n)                          //达到叶结点
    {   if(number>ans) ans=number;
        return;
    }
    else
    {   row=k/n;                        //求出位置 k 对应的行号
        column=k%n;                     //求出位置 k 对应的列号
        if((map[row][column]=='.') && (Camset(row,column)))
        {   map[row][column]='O';       //改为'O'
            solve(k+1,number+1);
            map[row][column]='.';       //回溯：恢复为'.'
        }
        solve(k+1,number);
    }
}
int main()
{   int i,j;
    cin>>n;                             //输入地图大小
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            cin>>map[i][j];             //输入地图
    ans=0;
    solve(0,0);
    cout<<ans<<endl;
    return 0;
}
```

3-9　**解**：采用回溯法的深度优先方式搜索从顶点 s 到 t 的最优路径，用 curv 记录当前路径的风险，bestv 是当前找到的最优路径风险，如果 curv≥bestv，则剪枝。对应的完整程序如下：

```
#include <stdio.h>
#include <string.h>
#include <vector>
using namespace std;
#define INF 0x3f3f3f3f
#define MAX 31
//问题表示
int n;                              //顶点个数
```

```
int a[MAX][MAX];                    //长度矩阵
int v[MAX];                         //顶点的风险
//求解结果表示
bool visited[MAX];
int curv=0;                         //当前路径风险
int bestv=INF;                      //最优路径风险
int bestl=INF;                      //最优路径长度
vector<int> bestpath;               //最优路径
int s,t;                            //起点和终点
void dispapath(vector<int> &path)   //输出一条路径
{   for(int i=0;i<path.size();i++)
        printf("%d ",path[i]);
    printf("\n");
}
int getpathlen(vector<int> path)    //求 path 路径的长度
{   int len=0;
    for(int i=1;i<path.size();i++)
        len+=a[path[i-1]][path[i]];
    return len;
}
void solve(int i,vector<int> path) //从顶点 i 开始搜索
{   visited[i]=true;
    path.push_back(i);              //将顶点 i 添加到路径中
    curv+=v[i];
    if(i==t)                        //找到终点
    {   int len=getpathlen(path);
        printf("   一个解: 风险=%3d, 长度=%3d",curv,len);
        printf(" ,路径: "); dispapath(path);
        if (curv<bestv)             //找最低风险的路径
        {   bestv=curv;
            bestl=len;
            bestpath=path;
        }
        else if(curv==bestv)        //风险相同找更短的路径
        {   if(len<bestl)
            {   bestl=len;
                bestpath=path;
            }
        }
    }
    for(int j=0;j<n;j++)
        if(a[i][j]!=INF && a[i][j]!=0 && !visited[j])  //顶点 i 到 j 有边
        {   if(curv<bestv)          //剪枝
```

```
                    solve(j,path);
            }
        visited[i]=false;
        curv-=v[i];
    }
    int main()
    {   memset(visited,0,sizeof(visited));
        memset(a,INF,sizeof(a));
        memset(v,0,sizeof(v));
        n=6;
        a[0][1]=1;a[0][3]=3;a[0][5]=2;
        a[1][5]=3;a[1][2]=6;
        a[3][5]=2;a[3][4]=6;
        a[4][2]=4;
        a[5][2]=2;a[5][4]=10;
        v[0]=1;v[1]=2;v[2]=4;v[3]=3;v[4]=1;v[5]=2;
        s=0;t=2;
        vector<int> path;
        printf("求解过程:\n");
        solve(s,path);
        printf("求解结果:\n");
        printf("   最低风险=%3d, 最短路径长度=%3d,",bestv,bestl);
        printf("最优路径: ");dispapath(bestpath);
        return 0;
    }
```

上述程序的执行结果如图 3.20 所示。

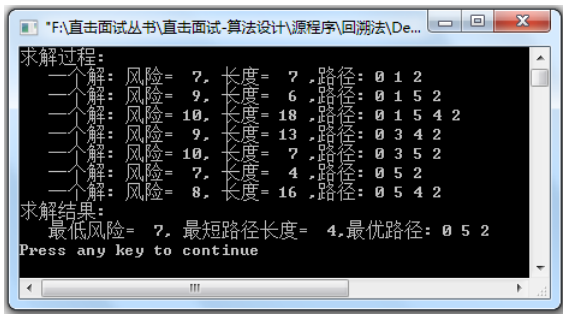


图 3.20　程序的执行结果

3-10　**解**：采用回溯法，二维数组 Grid 作为放置黑点的网格，Grid[i][j]=0 表示位置(i,j)没有黑点，Grid[i][j]=1 表示位置(i,j)有黑点。从第 i 行开始深度优先搜索，每找到一个解

Count 增 1（Count 的初始值为 0），最后输出 Count。对应的程序如下：

```
#include <stdio.h>
#include <string.h>
#define ROWS 3
#define COLS 3
//问题表示
int Grid[ROWS][COLS];                //放置黑点的网格
//求解结果表示
int Count=0;                         //累计方案数
int Rows[ROWS]={1,1,1};              //各行黑点数和的情况
int Cols[COLS]={1,1,1};              //各列黑点数和的情况
int SumR[ROWS];                      //累计各行的黑点数
int SumC[COLS];                      //累计各列的黑点数
void Set(int i)                      //从第 i 行开始搜索
{   if(i==ROWS)
    {   int k=0;
        while (k<COLS && SumC[k]==Cols[k])k++;
        if(k==COLS)
            Count++;
    }
    else
    {   for(int j=0;j<COLS;j++)
        {   if(Rows[i]==0)
                Set(i+1);            //在第 i+1 行放黑点
            else if(Grid[i][j]==0)
            {   Grid[i][j]=1;        //在(i,j)位置放一个黑点
                SumR[i]++;
                SumC[j]++;
                if(SumR[i]<Rows[i] && SumC[j]<=Cols[j])
                    Set(i);          //继续在第 i 行放黑点
                else if(SumR[i]==Rows[i] && i<ROWS)
                    Set(i+1);        //在第 i+1 行放黑点
                Grid[i][j]=0;        //回溯
                SumR[i]--;
                SumC[j]--;
            }
        }
    }
```

```
}
int main()
{   memset(Grid,0,sizeof(Grid));
    Set(0);
    printf("Count=%d\n",Count);     //输出 6
    return 0;
}
```