第5章 回溯法

5.1 单项选择题及其参考答案

1. 回溯法是在问题的解空间中按_____策略从根结点出发搜索的。

A. 广度优先　　B. 活结点优先

C. 扩展结点优先　　D. 深度优先

答：回溯法采用深度优先搜索在解空间中搜索问题的解。答案为 D。

2. 下列算法中_____通常以深度优先方式搜索问题的解。

A. 回溯法　　B. 动态规划

C. 贪心法　　D. 分支限界法

答：回溯法采用深度优先搜索在解空间中搜索问题的解，分支限界法采用广度优先搜索在解空间中搜索问题的解。答案为 A。

3. 关于回溯法以下叙述中不正确的是_____。

A. 回溯法有通用解题法之称，可以系统地搜索一个问题的所有解或任意解

B. 回溯法是一种既带系统性又带跳跃性的搜索算法

C. 回溯法算法需要借助队列来保存从根结点到当前扩展结点的路径

D. 回溯法算法在生成解空间的任一结点时，先判断该结点是否可能包含问题的解，如果肯定不包含，则跳过对以该结点为根的子树的搜索，逐层向祖先结点回溯

答：回溯算法是采用深度优先遍历的，需要借助栈保存从根结点到当前扩展结点的路径。答案为 C。

4. 回溯法的效率不依赖于下列因素_____。

A. 确定解空间的时间　　B. 满足显式约束的值的个数

C. 计算约束函数的时间　　D. 计算限界函数的时间

答：回溯法解空间是虚拟的，不必事先确定整个解空间。答案为 A。

5. 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. 剪支函数

C. 随机数函数　　D. 搜索函数

答：剪支函数包括约束函数（在扩展结点处剪除不满足约束条件的路径）和限界函数（剪去得不到问题解或最优解的路径）。答案为 B。

6. 对于含 n 个元素的子集树问题（每个元素二选一），最坏情况下解空间树的叶子结点个数是_____。

A. $n!$　　B. 2^n　　C. $2^{n+1}-1$　　D. 2^{n-1}

答：这样的解空间树是一棵高度为 $n+1$ 的满二叉树，叶子结点恰好有 2^n 个。答案为 B。

7. 用回溯法求解 0/1 背包问题时的解空间是_____。

A. 子集树　　B. 排列树

C. 深度优先生成树　　D. 广度优先生成树

答：在 0/1 背包问题中每个物品是二选一（要么选中要么不选中），与物品的顺序无关，对应的解空间为子集树类型。答案为 A。

8. 用回溯法求解 0/1 背包问题时最坏时间复杂度是_____。

A. $O(n)$　　B. $O(n\log_2 n)$　　C. $O(n\times 2^n)$　　D. $O(n^2)$

答：0/1 背包问题的解空间为一棵高度为 $n+1$ 的满二叉树，结点个数为 $2^{n+1}-1$，最坏情况是搜索全部结点。答案为 C。

9. 用回溯法求解 TSP 问题时的解空间是_____。

A. 子集树　　B. 排列树

C. 深度优先生成树　　D. 广度优先生成树

答：TSP 问题的解空间属于典型的排列树，因为路径与顶点顺序有关。答案为 B。

10. n 个学生每个人有一个分数，求最高分的学生的姓名，最简单的方法是_____。

A. 回溯法　　B. 归纳法　　C. 迭代法　　D. 以上都不对

答：最简单的方法是依次迭代比较求最高分数。答案为 C。

11. 求中国象棋中马从一个位置到另外一个位置的所有走法，采用回溯法求解时对应的解空间是_____。

A. 子集树　　B. 排列树

C. 深度优先生成树　　D. 广度优先生成树

答：每一步马从相邻可走的位置中选择一个位置走下去。答案为 A。

12. n 个人排队在一台机器上做某个任务，每个人的等待时间不同，完成他的任务的时间是不同的，求完成这 n 个任务的最小时间，采用回溯法求解时对应的解空间是_____。

A. 子集树　　B. 排列树

C. 深度优先生成树　　D. 广度优先生成树

答：该问题是求 1～n 的某个排列对应的 n 个任务完成的最小时间。答案为 B。

5.2 问答题及其参考答案

1. 回溯法的搜索特点是什么？

答：回溯法的搜索特点是深度优先搜索＋剪支。深度优先搜索可以尽快地找到一个解，剪支函数可以终止一些路径的搜索，提高搜索性能。

2. 有这样一个数学问题，x 和 y 是两个正实数，求 $x+y=3$ 的所有解，请问能否采用回溯法求解，如果改为 x 和 y 是两个均小于或等于 10 的正整数，又能否采用回溯法求解，如果能，请采用解空间画出求解结果。

答：当 x 和 y 是两个正实数时，理论上讲两个实数之间有无穷个实数，所以无法枚举 x 和 y 的取值，不能采用回溯法求 $x+y=3$ 的所有解。

当 x 和 y 是两个均小于或等于 10 的正整数时，它们的枚举范围是有限的，可以采用回溯法求 $x+y=3$ 的所有解，采用剪支仅扩展 $x,y\in[1,2]$的结点。解向量是(x,y)，对应的解空间如图 5.1 所示，找到的两个解是(1,2)和(2,1)。

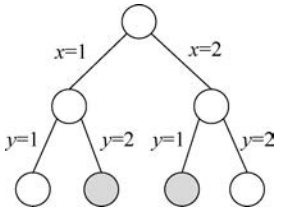

图 5.1　求 $x+y=3$ 的解空间

3. 对于 $n=4,a=(11,13,24,7),t=31$ 的子集和问题，利用左、右剪支的回溯法算法求解，给出求出的所有解，并且画出在解空间中的搜索过程。

答：利用左、右剪支的回溯法算法求出两个解如下。

第 1 个解：选取的数为 11 13 7

第 2 个解：选取的数为 24 7

在解空间中的搜索过程如图 5.2 所示，图中每个结点为(cs,rs)，其中 cs 为考虑第 i 个整数时选取的整数和，rs 为剩余整数和。题中实例搜索的结点个数是 11，如果不剪支，需要搜索 31 个结点。

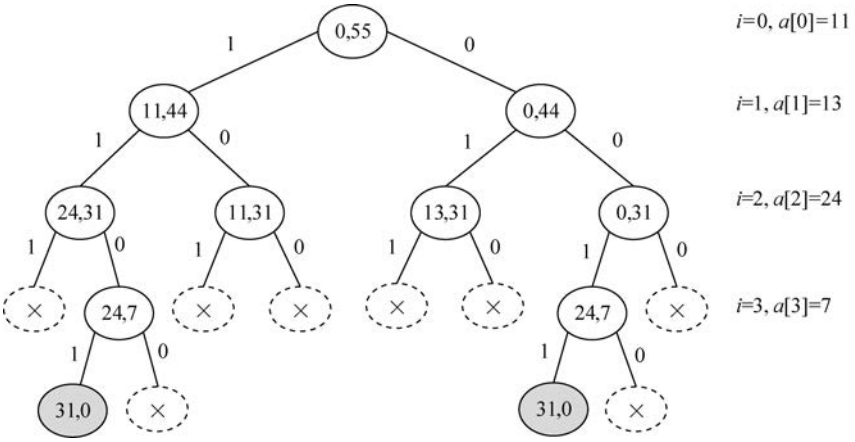

图 5.2　子集和问题的搜索过程

4. 对于 n 皇后问题，通过解空间说明 $n=3$ 时是无解的。

答： $n=3$ 时的解向量为 (x_1,x_2,x_3)，x_i 表示第 i 个皇后的列号，对应的解空间如图 5.3 所示，所有的叶子结点均不满足约束条件，所以无解。

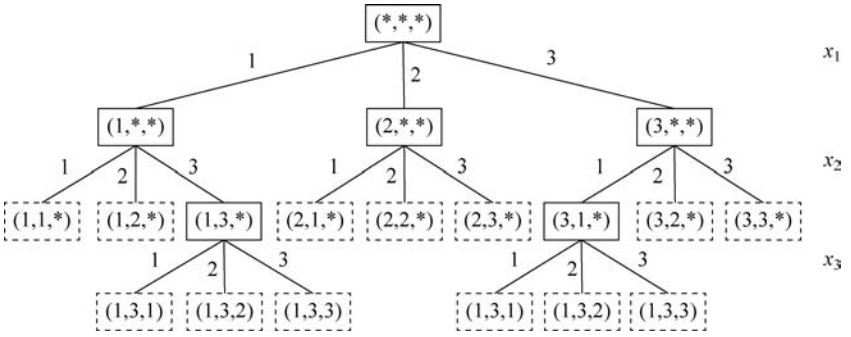

图 5.3　3 皇后问题的解空间

5. 对于 n 皇后问题，有人认为当 n 为偶数时其解具有对称性，即 n 皇后问题的解个数恰好为 $n/2$ 皇后问题的解个数的两倍，这个结论正确吗？

答： 这个结论错误。因为两个 $n/2$ 皇后问题的解合并起来不是 n 皇后问题的解。

6. 《教程》5.2.4 节采用解空间为子集树求解 n 皇后问题，请问能否采用解空间为排列树的回溯框架求解？如果能，请给出剪支操作，说明最坏情况下的时间复杂度，按照最坏情况下的时间复杂度比较哪个算法更好？

答： 设 n 皇后问题的解向量为 $(x_1,x_2,\cdots,x_n)$，x_i 表示第 i 个皇后的列号，显然每个解一定是 $1\sim n$ 的某个排列，所以可以采用解空间为排列树的回溯框架求解 n 皇后问题。其剪支操作是任何两个皇后不能同行、同列和同两条对角线。最坏情况下该算法的时间复杂度为 $O(n\times n!)$，由于 $O(n!)$ 好于 $O(n^n)$，所以按照最坏情况下的时间复杂度比较，解空间为排列树的回溯算法好于解空间为子集树的回溯算法。实际上可以通过进一步剪支使得《教程》5.2.4 节的算法的最坏时间复杂度达到 $O(n\times n!)$。

7. 对应如图 5.4 所示的无向连通图，假设颜色数 $m=2$，给出 m 着色的所有着色方案，并且画出对应的解空间。

答： 这里 $n=4$，顶点编号为 0～3，$m=2$，颜色编号为 0 和 1，解向量为 (x_0,x_1,x_2,x_3)，x_i 表示顶点 i 的着色，对应的解空间如图 5.5 所示，着色方案有两种，分别是(0,1,1,1)和(1,0,0,0)。

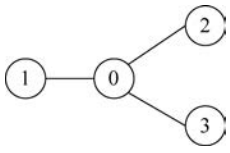

图 5.4 一个无向连通图

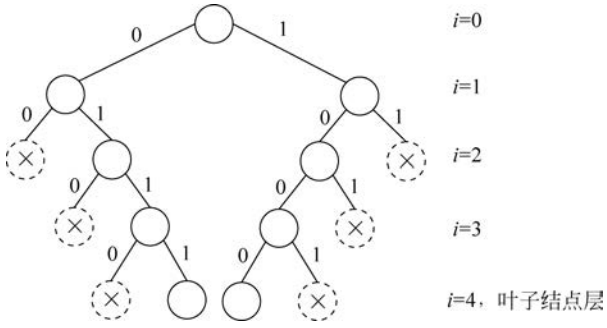

图 5.5 m 着色问题的解空间

8. 有一个 0/1 背包问题，物品个数 $n=4$，物品编号为 0～3，它们的重量分别是 3、1、2 和 2，价值分别是 9、2、8 和 6，背包容量 $W=3$。利用左、右剪支的回溯法算法求解，并且画出在解空间中的搜索过程。

答： 求解过程如下。

① 4 个物品按 v/w 递减排序后的结果如表 5.1 所示。

② 从 $i=0$ 开始搜索对应的解空间如图 5.6 所示。

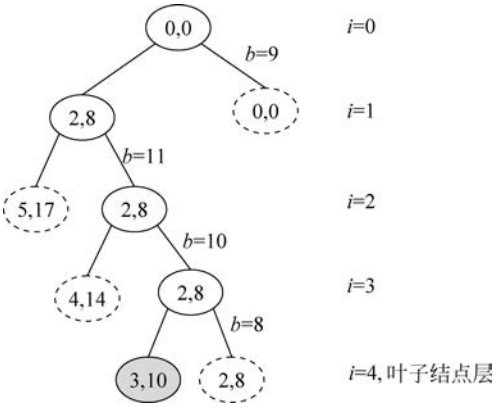

图 5.6 0/1 背包问题的解空间

最后得到的最优解是选择编号为 1 和 2 的物品，总重量为 3，总价值是 10。

表 5.1 4 个物品按 v/w 递减排序后的结果

序号 i	物品编号 no	重量 w	价值 v	v/w
0	2	2	8	4
1	0	3	9	3

续表

序号 i	物品编号 no	重量 w	价值 v	v/w
2	3	2	6	3
3	1	1	2	2

9. 以下算法用于求 n 个不同元素 a 的全排序，当 $a=(1,2,3)$时，请给出算法输出的全排序的顺序。

```
int cnt=0;                              //累计排列的个数
void disp(vector<int> &a)               //输出一个解
{   printf(" 排列%2d: (",++cnt);
    for (int i=0;i<a.size()-1;i++)
        printf("%d,",a[i]);
    printf("%d)",a.back());
    printf("\n");
}
void dfs(vector<int> & a,int i)         //递归算法
{   int n=a.size();
    if (i>=n-1)                         //递归出口
        disp(a);
    else
    {   for (int j=n-1;j>=i;j--)
        {   swap(a[i],a[j]);            //交换 a[i]与 a[j]
            dfs(a,i+1);
            swap(a[i],a[j]);            //交换 a[i]与 a[j]:恢复
        }
    }
}
void perm(vector<int> & a)              //求 a 的全排列
{
    dfs(a,0);
}
```

答： 当 $a=(1,2,3)$时调用 perm(a)的输出结果及其顺序如下。

```
排列 1: (1,3,2)
排列 2: (1,2,3)
排列 3: (2,1,3)
排列 4: (2,3,1)
排列 5: (3,1,2)
排列 6: (3,2,1)
```

10. 假设问题的解空间为$(x_0,x_1,\cdots,x_{n-1})$，每个 x_i 有 m 种不同的取值，所有 x_i 取不同的值，该问题既可以采用子集树递归回溯框架求解，也可以采用排列树递归回溯框架求解，考虑最坏时间性能应该选择哪种方法？

答： 一般情况下，这样的问题采用子集树递归回溯框架求解时最坏时间复杂度为 $O(m^n)$，采用排列树递归回溯框架求解时最坏时间复杂度为 $O(n!)$，如果 $m=2$，由于 $O(2^n)<O(n!)$，采用前者较好，若 m 接近 n，由于 $O(n^n)>O(n!)$，采用后者较好。

11. 以下两个算法都是采用排列树递归回溯框架求解任务分配问题，判断其正确性，如果不正确，请指出其中的错误。

(1) 算法 1：

```
void dfs(vector<int> &x,int cost,int i)                    //递归算法
```

```
{   if (i > n)                                  //到达叶子结点
    {   if (cost < bestc)                       //比较求最优解
        {   bestc=cost;
            bestx=x;
        }
    }
    else                                        //没有到达叶子结点
    {   for (int j=1;j<=n;j++)                  //为人员 i 试探任务 x[j]
        {   if (task[x[j]]) continue;           //若任务 x[j]已经分配,则跳过
            task[x[j]]=true;
            cost+=c[i][x[j]];
            swap(x[i],x[j]);                    //为人员 i 分配任务 x[j]
            if(bound(x,cost,i)< bestc)          //剪支
                dfs(x,cost,i+1);                //继续为人员 i+1 分配任务
            swap(x[i],x[j]);
            cost-=c[i][x[j]];                   //cost 回溯
            task[x[j]]=false;                   //task 回溯
        }
    }
}
```

(2) 算法 2：

```
void dfs(vector < int > &x,int cost,int i)      //递归算法
{   if (i > n)                                  //到达叶子结点
    {   if (cost < bestc)                       //比较求最优解
        {   bestc=cost;
            bestx=x;
        }
    }
    else                                        //没有到达叶子结点
    {   for (int j=1;j<=n;j++)                  //为人员 i 试探任务 x[j]
        {   if (task[x[j]]) continue;           //若任务 x[j]已经分配,则跳过
            swap(x[i],x[j]);                    //为人员 i 分配任务 x[j]
            task[x[j]]=true;
            cost+=c[i][x[j]];
            if(bound(x,cost,i)< bestc)          //剪支
                dfs(x,cost,i+1);                //继续为人员 i+1 分配任务
            cost-=c[i][x[j]];                   //cost 回溯
            task[x[j]]=false;                   //task 回溯
            swap(x[i],x[j]);
        }
    }
}
```

答：算法 1 是正确的。算法 2 不正确，在执行第一个 swap(x[i],x[j])后表示已经为人员 i 分配了任务 $x[j]$，应该置 task[x[i]]=true，cost+=c[i][x[i]]，后面的回溯恢复过程也是如此。

5.3 算法设计题及其参考答案

1. 给定含 n 个整数的序列 a(其中可能包含负整数)，设计一个算法从中选出若干整数，使它们的和恰好为 t。例如，$a=(-1,2,4,3,1)$，$t=5$，求解结果是(2,3,1,-1)，(2,3)，

(2,4,-1)和(4,1)。

解：由于 a 中可能包含负整数，甚至 t 有可能是负数，无法剪支，采用求 a 的幂集的思路，相当于求出 a 的所有子集并且累计子集和 cs，当到达一个叶子结点时若满足 cs=t 就是一个解。对应的算法如下：

```
vector<int> a={2,3,4,1,-1};                    //存放所有整数
int n=a.size(),t=5;
vector<int> x;                                 //解向量,全局变量
int cs;                                        //累计选择的元素和
int sum;                                       //累计组合个数
void dfs(int cs,int i)                         //递归回溯算法
{   if (i>=n)                                  //到达一个叶子结点
    {   if(cs==t)                              //找到一个解
        {   printf(" (%d): i=%d ",++sum,i);
            for (int j=0;j<x.size();j++)
                if (x[j]==1)
                    printf("%d ",a[j]);
            printf("\n");
        }
    }
    else                                       //没有到达叶子结点
    {   x[i]=1; cs+=a[i];
        dfs(cs,i+1);                           //选择 a[i]
        cs-=a[i];                              //回溯 cs
        x[i]=0;
        dfs(cs,i+1);                           //不选择 a[i]
    }
}
void subs4()                                   //求解子集和问题
{   x.resize(n,0);                             //指定 x 的长度为 n
    sum=0;
    cs=0;
    dfs(0,0);
}
```

说明：有人说一旦找到了 cs=t 就输出一个解(看成不再选择后面的元素)，所以将输出解的条件改为满足 i<n && cs=t。这样是错误的，因为这样做的结果是输出一个解后就停止了该结点的扩展，后面的解就找不到了，例如(2,3)是一个解，这样修改就找不到(2,3,1,-1)这个解了。

2. 给定含 n 个正整数的序列 a，设计一个算法从中选出若干整数，使它们的和恰好为 t 并且所选元素个数最少的一个解。

解：该问题属于典型的解空间为子集树的问题，采用子集树的回溯算法框架。当找到一个解后通过对选取的元素个数进行比较求最优解 bestx 和 bestm，剪支原理见《教程》5.2.1 节中求解子集和问题的算法。对应的算法如下：

```
vector<int> a={4,2,3,1,5};                     //存放所有整数
int n=5,t=9;
vector<int> bestx;                             //最优解向量
int bestm=n;                                   //最多选择 n 个元素
void dfs(int cs,int rs,vector<int> &x,int m,int i)   //递归算法
{   if (i>=n)                                  //到达一个叶子结点
    {   if (cs==t)                             //找到一个解
```

```
        {   if (m<bestm)                          //找更优解
            {   bestm=m;
                bestx=x;
            }
        }
    }
    else                                          //没有到达叶子结点
    {   rs-=a[i];                                 //求剩余的整数和
        if (cs+a[i]<=t)                           //左孩子结点剪支
        {   x[i]=1;                               //选取整数 a[i]
            dfs(cs+a[i],rs,x,m+1,i+1);
        }
        if (cs+rs>=t)                             //右孩子结点剪支
        {   x[i]=0;                               //不选取整数 a[i]
            dfs(cs,rs,x,m,i+1);
        }
        rs+=a[i];                                 //恢复剩余整数和
    }
}
void subs5()                                      //求解子集和问题
{   bestx.resize(n);
    vector<int> x(n);                             //解向量
    int rs=0;                                     //表示所有整数和
    for (int j=0;j<n;j++)                         //求 rs
        rs+=a[j];
    int m=0;
    dfs(0,rs,x,m,0);                              //i 从 0 开始
}
```

上述程序的执行结果如下：

```
最优解是选取的整数: 4 5 共选取 2 个整数
```

3. 给定一个含 n 个不同整数的数组 a，设计一个算法求其中所有 $m(m\leqslant n)$ 个元素的组合。例如，$a=(1,2,3)$，$m=2$，输出结果是(1,2)，(1,3)和(2,3)。

解：与求 a 的幂集类似，相当于求出 a 的所有子集，在满足题目的解中恰好选择 m 个元素。对应的算法如下：

```
int n;
vector<int> x;                                    //解向量,全局变量
int k;                                            //累计选择的元素个数
int sum;                                          //累计组合个数
void dfs(vector<int> &a,int m,int i)              //递归回溯算法
{   if (i>=n)                                     //到达一个叶子结点
    {   if(k==m)                                  //找到一个解
        {   printf(" (%d): ",++sum);
            for (int j=0;j<x.size();j++)
                if (x[j]==1)
                    printf("%d ",a[j]);
            printf("\n");
        }
    }
    else                                          //没有到达叶子结点
    {   x[i]=1; k++;
        dfs(a,m,i+1);                             //选择 a[i]
        k--;                                      //回溯 k
```

```
        x[i]=0;
        dfs(a,m,i+1);                    //不选择 a[i]
    }
}
void comb(vector<int> &a,int m)          //求 a 中 m 个元素的组合
{   n=a.size();
    x.resize(n);                         //指定 x 的长度为 n
    sum=0;
    k=0;
    dfs(a,m,0);
}
```

4. 设计一个算法求 1～n 中 $m(m\leqslant n)$个元素的排列，要求每个元素最多只能取一次。例如，$n=3$，$m=2$ 的输出结果是(1,2)，(1,3)，(2,1)，(2,3)，(3,1)，(3,2)。

解：用解向量 $x=(x_0,x_1,\cdots,x_{m-1})$表示 m 个整数的排列，每个 x_i 是 1～n 中的一个整数，并且所有 x_i 不相同，i 从 0 开始搜索，当 $i\geqslant m$ 时到达一个叶子结点，输出一个解，即一个排列。为了避免元素重复，设计 used 数组，其中 used[j]=0 表示没有选择整数 j，used[j]=1 表示已经选择整数 i。对应的算法如下：

```
int m,n;
int x[MAXM];                             //解向量 x[0..m-1]存放一个排列
bool used[MAXN];
int sum;                                 //累计解个数
void dfs(int i)                          //递归算法
{   if (i>=m)
    {   printf(" (%d): ",++sum);
        for (int j=0;j<m;j++)            //输出一个排列
            printf("%d ",x[j]);
        printf("\n");
    }
    else
    {   for (int j=1;j<=n;j++)
        {   if (!used[j])
            {   used[j]=true;            //修改 used[i]
                x[i]=j;                  //x[i]选择 j
                dfs(i+1);                //继续搜索排列的下一个元素
                used[j]=false;           //回溯:恢复 used[j]
            }
        }
    }
}
```

5. 在《教程》5.2.4 节求 n 皇后问题的算法中每次放置第 i 个皇后时，其列号 x_i 的试探范围是 1～n，实际上前面已经放好的皇后的列号是不必试探的，请根据这个信息设计一个更高效的求解 n 皇后问题的算法。

解：place(i,j)算法用于测试第 i 行第 j 列是否可以放置第 i 个皇后，时间复杂度为 $O(i)$，调用次数越多性能越差。现在设计一个 used 数组，used[j]=0 表示列号 j 没有被占用，used[j]=0 表示列号 j 已经被占用，在试探第 i 个皇后的列号 x_i 时仅对 used[j]=0 的列号 j 调用 place(i,j)，这样大大减少了调用 place(i,j)的次数。对应的算法如下：

```
int q[MAXN];                             //存放各皇后所在的列号,为全局变量
bool used[MAXN];
```

```
int sum=0;                                  //累计解个数
void disp(int n)                            //输出一个解
{   printf("  第%d个解:",++sum);
    for (int i=1;i<=n;i++)
        printf("(%d,%d) ",i,q[i]);
    printf("\n");
}
bool place(int i,int j)                     //测试(i,j)位置能否摆放皇后
{   if (i==1) return true;                  //第一个皇后总是可以放置
    int k=1;
    while (k<i)                             //k=1～i-1是已放置了皇后的行
    {   if (abs(q[k]-j)==abs(i-k))          //不必检测同列的情况
            return false;
        k++;
    }
    return true;
}
int cnt=0;                                  //执行place算法的次数
void queen31(int n,int i)                   //回溯算法
{   if (i>n)
        disp(n);                            //所有皇后放置结束时输出一个解
    else
    {   for (int j=1;j<=n;j++)              //在第i行上试探每一个列j
        {   if(used[j]) continue;           //跳过前面已经放置皇后的列号
            cnt++;
            if (place(i,j))                 //在第i行上找到一个合适的位置(i,j)
            {   q[i]=j;
                used[j]=true;               //列号j已经放置皇后
                queen31(n,i+1);
                used[j]=false;              //回溯
                q[j]=0;
            }
        }
    }
}
void queen3(int n)                          //递归求解n皇后问题
{   memset(used,false,sizeof(used));
    queen31(n,1);
}
```

例如,$n=4$ 时上述算法共调用 place(i,j)算法 32 次,如果不改进则调用 60 次,$n=6$ 时从 894 次降低为 356 次。

6. 请采用基于排列树的回溯框架设计求解 n 皇后问题的算法。

解:用 q 数组表示 n 皇后问题的一个解中所有皇后的列号,显然 q 一定是 1～n 的某个排列,所以可以采用基于排列树的回溯框架设计。对应的算法如下:

```
int q[MAXN];                                //存放各皇后所在的列号,为全局变量
int cnt=0;                                  //累计解个数
void disp(int n)                            //输出一个解
{   printf("  第%d个解:",++cnt);
    for (int i=1;i<=n;i++)
        printf("(%d,%d) ",i,q[i]);
    printf("\n");
}
bool place(int i,int j)                     //测试(i,j)位置能否摆放皇后
```

```
{   if (i==1) return true;                  //第一个皇后总是可以放置
    int k=1;
    while (k<i)                             //k=1～i-1是已放置了皇后的行
    {   if ((q[k]==j) || (abs(q[k]-j)==abs(i-k)))
            return false;
        k++;
    }
    return true;
}
void queen41(int n,int i)                   //回溯算法
{   if (i>n)
        disp(n);                            //所有皇后放置结束
    else
    {   for (int j=i;j<=n;j++)              //在第i行上试探每一个列j
        {   swap(q[i],q[j]);                //第i个皇后放置在q[j]列
            if(place(i,q[i]))               //剪支操作
                queen41(n,i+1);
            swap(q[i],q[j]);                //回溯
        }
    }
}
void queen4(int n)                          //用递归法求解n皇后问题
{   for(int i=1;i<=n;i++)
        q[i]=i;
    queen41(n,1);
}
```

7. 一棵整数二叉树采用二叉链 b 存储，设计一个算法求根结点到每个叶子结点的路径。

解： 这里的二叉树 b 看成解空间，所谓结点 p 的扩展就是搜索它的左、右孩子结点。解向量 x 存放根结点到一个叶子结点的路径（由于路径的长度可能不同，所以不同解的解向量的长度可能不同）。从根结点出发搜索到叶子结点，每次遇到一个叶子结点就输出 x。对应的算法如下：

```
vector<int> x;                                  //解向量
int sum;                                        //累计解个数
void dfs(TreeNode * b)
{   if (b->left==NULL && b->right==NULL)        //到达一个叶子结点
    {   printf(" (%d): ",++sum);
        for(int j=0;j<x.size();j++)
            printf("%d ",x[j]);
        printf("\n");
    }
    else                                        //没有到达叶子结点
    {   if(b->left!=NULL)                       //结点b有左孩子
        {   x.push_back(b->left->val);          //在x的末尾添加b的左孩子结点值
            dfs(b->left);
            x.pop_back();                       //回溯
        }
        if(b->right!=NULL)                      //结点b有右孩子
        {   x.push_back(b->right->val);         //在x的末尾添加b的右孩子结点值
            dfs(b->right);
            x.pop_back();                       //回溯
        }
    }
```

```
}
void allpath(TreeNode * b)                          //求解算法
{   if(b==NULL) return;
    x.push_back(b->val);
    dfs(b);
}
```

8. 一棵整数二叉树采用二叉链 b 存储，设计一个算法求根结点到叶子结点的路径中路径和最小的一条路径，如果这样的路径有多条，求其中的任意一条。

解： 与第 7 题的解题思路类似，这里是求最短路径(最优解)，增加 sum 表示当前路径 x 的路径和，bestx 和 bestsum 用于存放最优解，分别是最优路径与最优路径和。在找到一个解时通过比较路径长度求最短路径。对应的算法如下：

```
vector<int> x;                                      //解向量
int sum;                                            //路径和
vector<int> bestx;                                  //最优解向量
int bestsum=INF;                                    //最短路径和
void dfs(TreeNode * b)
{   if (b->left==NULL && b->right==NULL)  //到达一个叶子结点
    {   if(sum<bestsum)                             //通过比较求更优解
        {   bestsum=sum;
            bestx=x;
        }
    }
    else                                            //没有到达叶子结点
    {   if(b->left!=NULL)                           //结点 b 有左孩子
        {   x.push_back(b->left->val);              //在 x 的末尾添加 b 的左孩子结点值
            sum+=b->left->val;
            dfs(b->left);
            sum-=b->left->val;
            x.pop_back();                           //回溯
        }
        if(b->right!=NULL)                          //结点 b 有右孩子
        {   x.push_back(b->right->val);             //在 x 的末尾添加 b 的右孩子结点值
            sum+=b->right->val;
            dfs(b->right);
            sum-=b->right->val;
            x.pop_back();                           //回溯
        }
    }
}
void minpath(TreeNode * b)                          //求解算法
{   if(b==NULL) return;
    x.push_back(b->val);
    sum=b->val;
    dfs(b);
    printf(" 最短路径: ");
    for(int j=0;j<bestx.size();j++)
        printf("%d ",bestx[j]);
    printf("路径和=%d\n",bestsum);
}
```

9. 一棵整数二叉树采用二叉链 b 存储，设计一个算法产生每个叶子结点的编码。假设从根结点到某个叶子结点 a 有一条路径，从根结点开始，路径走左分支时用 0 表示，走右分支时用 1 表示，这样的 0/1 序列就是 a 的编码。

解：与第 7 题的解题思路类似，这里解向量 x 表示叶子结点的编码(初始为空)，从根结点开始，走左分支时添加 0，走右分支时添加 1。在到达一个叶子结点时输出 x。对应的算法如下：

```
vector<int> x;                                  //解向量
void dfs(TreeNode * b)
{   if (b->left==NULL && b->right==NULL)        //到达一个叶子结点
    {   printf(" %d 的编码: ",b->val);
        for(int j=0;j<x.size();j++)
            printf("%d ",x[j]);
        printf("\n");
    }
    else                                        //没有到达叶子结点
    {   if(b->left!=NULL)                       //结点 b 有左孩子结点
        {   x.push_back(0);                     //在 x 的末尾添加 0
            dfs(b->left);
            x.pop_back();                       //回溯
        }
        if(b->right!=NULL)                      //结点 b 有右孩子结点
        {   x.push_back(1);                     //在 x 的末尾添加 1
            dfs(b->right);
            x.pop_back();                       //回溯
        }
    }
}
void leafcode(TreeNode * b)                     //求解算法
{   if(b==NULL) return;
    dfs(b);
}
```

10. 假设一个含 n 个顶点(顶点编号为 0～$n-1$)的不带权图采用邻接矩阵 $\boldsymbol{A}$ 存储，设计一个算法判断其中顶点 u 到顶点 v 是否有路径。

解：将从图中顶点 u 出发的全部搜索看成解空间，所谓顶点 u 的扩展就是搜索它相邻的尚未访问的顶点。用 flag 变量表示 u 到 v 是否有路径(看成解向量，初始设置为 false)，解空间中的起始点 u 对应根结点，叶子结点对应顶点 v。从顶点 u 出发搜索，为了避免在一条路径中重复访问顶点，设置 visited 数组(初始时所有元素置为 0)，visited[j]=0 表示顶点 j 未访问，visited[j]=1 表示顶点 j 已访问。由于是从根结点开始搜索其相邻顶点的，所以必须先置 visited[u]=1，当访问到顶点 v 时说明 u 到 v 有路径，置 flag 为 true，一旦 flag 为 true，便终止后面的结点扩展。采用深度优先搜索的回溯算法如下：

```
vector<vector<int>> A={{0,1,1,0,0},{1,0,1,1,0},{1,1,0,1,0},{0,1,1,0,1},{0,0,0,1,0}};
int n=5;
vector<int> visited(n,0);                       //访问标记
bool flag;                                      //表示 u 到 v 是否有路径
void dfs(int u,int v)
{   if (u==v)                                   //到达顶点 v
        flag=true;
    else if(!flag)                              //没有到达顶点 v 且 flag 为假
    {   for(int j=0;j<n;j++)
        {   if(A[u][j]==1 && visited[j]==0)     //顶点 u 到 j 有边并且 j 没有访问过
            {   visited[j]=1;
                dfs(j,v);
                visited[j]=0;                   //回溯
```

```
            }
        }
    }
}
bool haspath(int u,int v)                    //求解算法
{   flag=false;
    visited[u]=1;                            //标记起始点 u 已访问
    dfs(u,v);
    return flag;
}
```

11. 假设一个含 n 个顶点(顶点编号为 $0\sim n-1$)的不带权图采用邻接矩阵 $\boldsymbol{A}$ 存储,设计一个算法求其中顶点 u 到顶点 v 的所有路径。

解:与第 10 题的解题思路类似,这里是求顶点 u 到顶点 v 的所有路径,在解空间中从根结点(对应起始点 u)开始搜索,在扩展时每访问一个顶点将其添加到 x 的末尾,当访问到叶子结点(对应顶点 v)时输出对应的路径 x。对应的算法如下:

```
vector<vector<int>> A={{0,1,1,0,0},{1,0,1,1,0},{1,1,0,1,0},{0,1,1,0,1},{0,0,0,1,0}};
int n=5;
vector<int> x;                                   //解向量(路径)
int sum=0;                                       //路径数
vector<int> visited(n,0);
void dfs(int u,int v)
{   if (u==v)                                    //到达顶点 v
    {   printf(" (%d): ",++sum);
        for(int j=0;j<x.size();j++)
            printf("%d ",x[j]);
        printf("\n");
    }
    else                                         //没有到达顶点 v
    {   for(int j=0;j<n;j++)
        {   if(A[u][j]==1 && visited[j]==0)       //顶点 u 到 j 有边并且 j 没有访问过
            {   x.push_back(j);
                visited[j]=1;
                dfs(j,v);
                visited[j]=0;                    //回溯
                x.pop_back();
            }
        }
    }
}
void allpath(int u,int v)                        //求解算法
{   x.push_back(u);                              //将起始点 u 添加到路径中
    visited[u]=1;                                //标记起始点 u 已访问
    dfs(u,v);
}
```

12. 假设一个含 n 个顶点(顶点编号为 $0\sim n-1$)的带权图采用邻接矩阵 $\boldsymbol{A}$ 存储,设计一个算法求其中顶点 u 到顶点 v 的一条路径长度最短的路径,一条路径的长度是指路径上经过的边的权值和。如果这样的路径有多条,求其中的任意一条。

解:与第 10 题的解题思路类似,这里是求最短路径(最优解),增加 len 表示当前路径 x 的路径长度,bestx 和 bestlen 用于存放最优解,分别是最优路径与最优路径长度。在找到一个解时通过比较路径长度求最短路径。对应的算法如下:

```
#define INF 0x3f3f3f3f
vector<vector<int>> A={{0,1,5,0,0},{1,0,2,4,0},{5,2,0,1,0},{0,4,1,0,2},{0,0,0,2,0}};
int n=5;
vector<int> x;                                  //解向量
int len=0;                                      //路径长度
vector<int> bestx;                              //最优解向量
int bestlen=INF;                                //最优路径长度
vector<int> visited(n,0);
void dfs(int u,int v)
{   if (u==v)                                   //到达顶点 v
    {   if(len<bestlen)                         //通过比较找更短路径
        {   bestlen=len;
            bestx=x;
        }
    }
    else                                                //没有到达顶点 v
    {   for(int j=0;j<n;j++)
        {   if(A[u][j]!=0 && A[u][j]!=INF)              //u 到 j 有边
            {   if(visited[j]==0)                       //顶点 j 没有访问过
                {   x.push_back(j);
                    visited[j]=1;
                    len+=A[u][j];
                    dfs(j,v);
                    len-=A[u][j];                       //回溯
                    visited[j]=0;
                    x.pop_back();
                }
            }
        }
    }
}
void minpath(int u,int v)                           //求解算法
{   x.push_back(u);                                 //将起始点 u 添加到路径中
    visited[u]=1;                                   //标记起始点 u 已访问
    len=0;                                          //路径长度初始化为 0
    dfs(u,v);
    printf(" 最短路径: ");
    for(int j=0;j<bestx.size();j++)
        printf("%d ",bestx[j]);
    printf(" 长度=%d\n",bestlen);
}
```

5.4 上机实验题及其参考答案

5.4.1 象棋算式

编写一个实验程序 exp5-1 求解如图 5.7 所示的算式，其中每个不同的棋子代表不同的数字，要求输出这些棋子各代表哪个数字的所有解。

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

图 5.7 象棋算式

解：解向量为(a,b,c,d,e)，分别表示兵、炮、马、卒和车的取值。

解空间是一棵高度为 6 的树，根结点对应 a 的各种选择，第 2 层结点对应 b 的各种选择，以此类推。所有解的取值范围为 0～9，并且 a～e 均不相等。根据算式设：

$$m = a \times 1000 + b \times 100 + c \times 10 + d$$
$$n = a \times 1000 + b \times 100 + e \times 10 + d$$
$$s = e \times 10000 + d \times 1000 + c \times 100 + a \times 10 + d$$

则约束条件是 $m+n==s$。

为了避免同一数字被重复使用，设计布尔型数组 used，used[j]=0($0 \leqslant j \leqslant 9$)表示数字 j 没有被使用，used[j]=0 表示数字 j 已经被使用，在搜索中仅扩展 used[j]为 0 的结点。为了简便，将(a,b,c,d,e)用 $x=(x_0,x_1,x_2,x_3,x_4)$代替，这样 i 从 0 开始搜索，当到达一个叶子结点($i \geqslant 5$)时判断上述约束条件是否成立，若成立则输出一个解。对应的实验程序如下：

```
#include<iostream>
#include<vector>
using namespace std;
vector<int> used(10,0);                    //used[j]表示数字 j 是否已使用
int sum;                                   //累计解个数
void dfs(vector<int> x,int n,int i)        //递归回溯算法
{   if(i>=n)
    {   int m=x[0]*1000+x[1]*100+x[2]*10+x[3];
        int n=x[0]*1000+x[1]*100+x[4]*10+x[3];
        int s=x[4]*10000+x[3]*1000+x[2]*100+x[0]*10+x[3];
        if (m+n==s)                        //找到一个可行解
        {   printf(" 第%d 个解 ",++sum);
            printf("兵:%d 炮:%d 马:%d 卒:%d 车:%d\n",x[0],x[1],x[2],x[3],x[4]);
        }
        return;
    }
    else
    {   for(int j=0;j<=9;j++)
        {   if(used[j]==0)
            {   x[i]=j;
                used[j]=1;
                dfs(x,n,i+1);
                x[i]=-1;
                used[j]=0;
            }
        }
    }
}
void piece()                               //求解算法
{   int n=5;
    vector<int> x(n);                      //定义解向量
    sum=0;
    dfs(x,n,0);
}
```

```
int main()
{   printf("实验结果\n ");
    piece();
    printf("共%d 个解\n",sum);
    return 0;
}
```

上述程序的执行结果如图 5.8 所示。

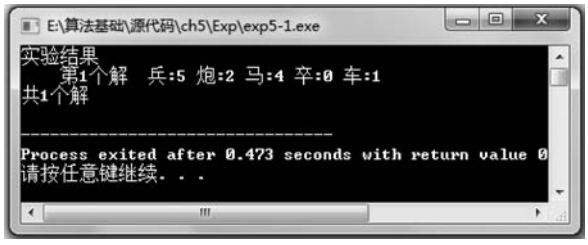

图 5.8 exp5-1 实验程序的执行结果

5.4.2 子集和

编写一个实验程序 exp5-2,给定含 n 个正整数的数组 a 和一个整数 t,如果 a 中存在若干个整数(至少包含一个整数)的和恰好等于 t,说明有解,否则说明无解。要求采用相关数据进行测试。

解:相关原理见《教程》5.2.1 节中求解子集和问题的算法。这里仅判断是否有解,用 flag 变量表示是否有解(初始设置为 false),cnt 变量表示解中选取的整数个数(初始为 0),cs 为当前选取的整数和,当到达一个叶子结点($i \geqslant n$)时,若 cs==t && cnt>=1 成立说明有解,置 flag 为 true,一旦 flag 为 true 便终止结点的扩展,全部搜索完毕 flag 仍然为 false 说明无解。对应的实验程序如下:

```
#include <iostream>
#include <vector>
using namespace std;
int n=4,t;
vector <int> a={11,13,24,7};                    //存放所有整数
int cnt;                                        //解中选取的整数个数
bool flag;                                      //是否存在解
void dfs(int cs,int rs,int i)                   //递归回溯算法
{   if (i>=n)                                   //找到一个叶子结点
    {   if (cs==t && cnt>=1)                    //找到一个满足条件的解,置 flag 为 true
            flag=true;
    }
    else if(!flag)                              //没有到达叶子结点并且 flag 为假
    {   rs-=a[i];                               //求剩余的整数和
        if (cs+a[i]<=t)                         //左孩子结点剪支
        {   cnt++;
            dfs(cs+a[i],rs,i+1);
            cnt--;                              //回溯
        }
        if (cs+rs>=t)                           //右孩子结点剪支
            dfs(cs,rs,i+1);
        rs+=a[i];                               //恢复剩余整数和
```

```
    }
}
bool subs()                                    //求解子集和问题
{   int rs=0;                                  //表示所有整数和
    for (int j=0;j<n;j++)                      //求 rs
        rs+=a[j];
    flag=false;
    cnt=0;
    dfs(0,rs,0);                               //i 从 0 开始
    return flag;
}
int main()
{   printf("a: ");
    for(int j=0;j<n;j++)
        printf("%d ",a[j]);
    printf("\n");
    t=0;
    printf("实验结果\n");
    printf(" t=%d 时%s\n",t,(subs()? "存在解":"没有解"));
    t=15;
    printf(" t=%d 时%s\n",t,(subs()? "存在解":"没有解"));
    t=21;
    printf(" t=%d 时%s\n",t,(subs()? "存在解":"没有解"));
    t=24;
    printf(" t=%d 时%s\n",t,(subs()? "存在解":"没有解"));
    return 0;
}
```

上述程序的执行结果如图 5.9 所示。

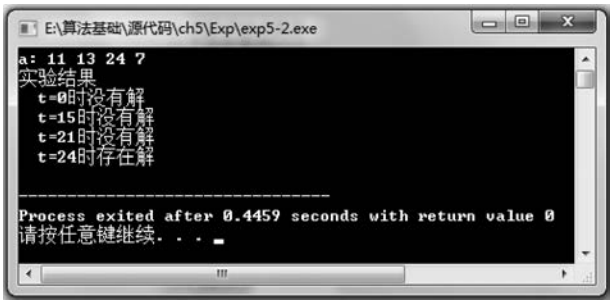

图 5.9 exp5-2 实验程序的执行结果

5.4.3 迷宫路径

编写一个实验程序 exp5-3 采用回溯法求解迷宫问题。给定一个 $m \times n$ 个方块的迷宫，每个方块值为 0 时表示空白，为 1 时表示障碍物，在行走时最多只能走到上、下、左、右相邻的方块。求指定入口 s 到出口 t 的所有迷宫路径和其中一条最短路径。

解：迷宫问题也是一个解空间为子集树的问题，实际上每个方块在四周的 4 个方位选一，入口对应根结点，出口对应叶子结点。这里需要求所有解(迷宫路径)，同时求一个最优解(最短长度的路径)，用解向量 x 表示迷宫路径，len 表示其长度，bestx 表示最短路径，bestlen 表示最短路径长度。为了避免重复，设计 visited 二维数组，visited[i][j]=0 表示[i,j]方块没有访问过，visited[i][j]=1 表示[i,j]方块已经访问过。从根结点(即入口 s)出发

搜索，先置入口 s 的 visited 为 1，当访问到出口 t 时输出 x 构成一条迷宫路径，同时比较路径长度确定最短路径。对应的完整实验程序如下：

```
#include<iostream>
#include<vector>
using namespace std;
#define INF 0x3f3f3f3f
int n=4;
int m=4;
vector<vector<int>> A={{0,0,0,1},{0,1,0,0},{0,0,0,1},{1,0,0,0}};
int dx[]={0,0,1,-1};                          //水平方向的偏移量
int dy[]={1,-1,0,0};                          //垂直方向的偏移量
vector<vector<int>> visited(m,vector<int>(n,0));
struct Box                                    //方块类型
{   int x;
    int y;
    Box(int x1,int y1): x(x1),y(y1) {}        //构造函数
};
vector<Box> x;                                //解向量
int len;                                      //解向量表示的路径长度
vector<Box> bestx;                            //最优解向量
int bestlen=INF;                              //最优解向量表示的路径长度
int sum;                                      //表示路径数
void disp()                                   //输出一条迷宫路径
{   printf("      路径%d: ",++sum);
    for (int j=0;j<x.size();j++)
        printf("[%d,%d] ",x[j].x,x[j].y);
    printf(" 长度=%d\n",len);
}
void dfs(Box& s,Box&t)                        //回溯法
{   if(sum>10) return;
    if(s.x==t.x && s.y==t.y)
    {   disp();
        if(len<bestlen)
        {   bestlen=len;
            bestx=x;
        }
    }
    else
    {   for(int di=0;di<4;di++)               //试探四周的每个方位 di
        {   int nx=s.x+dx[di];                //相邻方块为(nx,ny)
            int ny=s.y+dy[di];
            if(nx<0 || nx>=m || ny<0 || ny>=n)
                continue;                     //跳过超界的方块
            if(A[nx][ny]==1)
                continue;                     //跳过障碍物
            if(visited[nx][ny]==1)
                continue;                     //跳过已经访问的方块
            visited[nx][ny]=1;                //访问(nx,ny)
            len++;
            Box b(nx,ny);
            x.push_back(b);                   //(nx,ny)添加到路径中
            dfs(b,t);
            x.pop_back();                     //回溯
            len--;
            visited[nx][ny]=0;
        }
    }
```

```
}
void mgallpath(Box& s,Box& t)                    //求解算法
{   x.push_back(s);                              //入口 s 添加到路径中
    visited[s.x][s.y]=1;                         //标记入口 s 已访问
    len=0;
    dfs(s,t);
}
int main()
{   Box s(0,0);
    Box t(3,3);
    printf("实验结果\n");
    printf("   从[%d,%d]到[%d,%d]的全部路径\n",s.x,s.y,t.x,t.y);
    mgallpath(s,t);
    printf("一条最短路径: ");
    for (int j=0;j<bestx.size();j++)
        printf("[%d,%d] ",bestx[j].x,bestx[j].y);
    printf(" 长度=%d\n",bestlen);
    return 0;
}
```

上述程序用于求如图 5.10 所示的迷宫中从入口(0,0)到出口(3,3)的所有迷宫路径,程序的执行结果如图 5.11 所示。

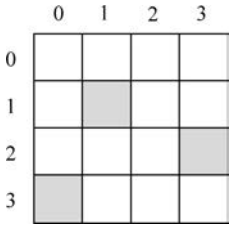

图 5.10 一个迷宫图

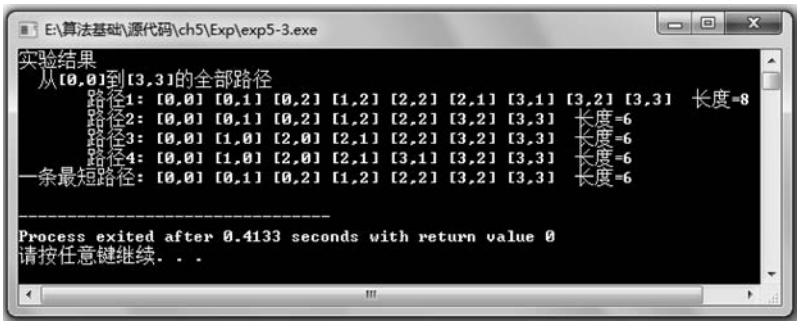

图 5.11 exp5-3 实验程序的执行结果

5.4.4 哈密顿回路

编写一个实验程序 exp5-4 求哈密顿回路。给定一个无向图,由指定的起点前往指定的终点,途中经过所有其他顶点且只经过一次,称为哈密顿路径,闭合的哈密顿路径称作哈密顿回路。设计一个回溯算法求无向图的所有哈密顿回路,并用相关数据进行测试。

解:相关原理见《教程》5.3.2 节 TSP 问题的基于排列树的回溯算法,这里求哈密顿回路更加简单。用 0/1 邻接矩阵 $\boldsymbol{A}$ 存放无向图,设计当前解向量 $\boldsymbol{x}=(x_0,x_1,\cdots,x_{n-1})$,每个 x_i 表示一个图中顶点,实际上每个 $\boldsymbol{x}$ 表示一条路径,初始时 x_0 置为起点 s,$x_1\sim x_{n-1}$ 为其他 $n-1$ 个顶点的编号,d 表示当前路径的长度,当到达一个叶子结点($i\geqslant n$)时,如果 $A[x[n-1]][s]==1$ 说明 $x[n-1]$到 s 有边,$x\cup\{s\}$就是一条从 s 出发到达 s 的哈密顿回路,输出 x 即可。对应的完整实验程序如下:

```
#include<iostream>
#include<vector>
using namespace std;
int n=5;                                         //图中顶点个数
```

```
vector<vector<int>> A={{0,1,1,1,0},{1,0,0,1,1},{1,0,0,0,1},{1,1,0,0,1},{0,1,1,1,0}};
int cnt=0;                                          //累计路径条数
void disp(vector<int> &x,int d,int s)               //输出一个解
{   printf("  第%d条回路: ",++cnt);
    for (int j=0;j<x.size();j++)
        printf("%d->",x[j]);
    printf("%d\n",s);                               //末尾加上起点 s
}
void dfs(vector<int> &x,int d,int s,int i)          //回溯法
{   if(i>=n)                                        //到达一个叶子结点
    {   if(A[x[n-1]][s]==1)                         //若 x[n-1]到 s 有边
            disp(x,d,s);
    }
    else                                            //没有到达叶子结点
    {   for(int j=i;j<n;j++)                        //试探 x[i]走到 x[j]的分支
        {   if (A[x[i-1]][x[j]]==1)                 //若 x[i-1]到 x[j]有边
            {   swap(x[i],x[j]);
                dfs(x,d+A[x[i-1]][x[i]],s,i+1);
                swap(x[i],x[j]);
            }
        }
    }
}
void Hamiltonian(int s)                             //求起始点为 s 的哈密顿回路
{   vector<int> x;                                  //定义解向量
    x.push_back(s);
    for(int i=0;i<n;i++)                            //将非 s 的顶点添加到 x 中
        if(i!=s)
            x.push_back(i);
    int d=0;
    dfs(x,d,s,1);                                   //从 x[1]顶点开始扩展
}
int main()
{   printf("实验结果\n");
    int s=1;
    printf(" 从顶点%d 出发的哈密顿回路:\n",s);
    Hamiltonian(s);
    return 0;
}
```

上述程序用于求如图 5.12 所示的无向图中 $s=1$ 的所有哈密顿回路,程序的执行结果如图 5.13 所示。

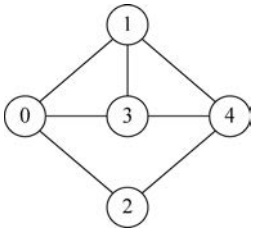

图 5.12　一个无向图

图 5.13　exp5-4 实验程序的执行结果

5.5 在线编程题及其参考答案

5.5.1 LeetCode216——组合总和Ⅲ

问题描述：找出所有相加之和为 n 的 k 个数的组合，组合中只允许含 1～9 的正整数，并且每种组合中不存在重复的数字。例如，$k=3$，$n=7$ 的结果是{1,2,4}，而 $k=3$，$n=9$ 的结果是{1,2,6}，{1,3,5}和{2,3,4}。要求设计如下函数：

```
class Solution {
public:
    vector<vector<int>> combinationSum3(int k,int n) {}
};
```

解：用类变量 ans 存放结果，相关剪支原理见《教程》5.2.1 节中求解子集和问题的算法，这里不再讨论，改为用解向量 x 存放选取的所有整数，每个分量的取值范围是 1～9，cnt 表示选取的整数个数，i 从 1 开始(对应解空间的根结点)，每个整数 i 只有选取和不选取两种可能，当到达一个叶子结点($i\geqslant 10$)时，如果 cs=N 并且 cnt=K 表示找到一个解，将 x 添加到 ans 中。对应的程序如下：

```
class Solution {
    vector<vector<int>> ans;
    int N,K;
public:
    vector<vector<int>> combinationSum3(int k,int n)
    {   N=n; K=k;
        int rs=0;                       //rs 表示所有整数和
        for (int j=1;j<=9;j++)          //求 rs
            rs+=j;
        int cnt=0;
        vector<int> x;
        dfs(0,rs,x,cnt,1);              //i 从 1 开始
        return ans;
    }
    void dfs(int cs,int rs,vector<int> & x,int cnt,int i)   //递归回溯算法
    {   if (i>=10)                                          //找到一个叶子结点
        {   if (cs==N && cnt==K)                            //找到一个满足条件的解
                ans.push_back(x);
        }
        else                                                //没有到达叶子结点
        {   rs-=i;                                          //求剩余的整数和
            if (cs+i<=N)                                    //左剪支
            {   x.push_back(i);                             //选取整数 i
                cnt++;
                dfs(cs+i,rs,x,cnt,i+1);
                cnt--;                                      //回溯
                x.pop_back();
            }
            if (cs+rs>=N)                                   //右剪支
                dfs(cs,rs,x,cnt,i+1);
            rs+=i;                                          //恢复剩余整数和
```

```
        }
    }
};
```

上述程序的提交结果为通过，执行时间为 0ms，内存消耗为 6.5MB。实际上由于测试数据比较少，即使不采用任何剪支，执行时间也只有 4ms。没有任何剪支的程序如下：

```
class Solution {
    vector<vector<int>> ans;
    int N,K;
public:
    vector<vector<int>> combinationSum3(int k,int n)
    {   N=n; K=k;
        int cnt=0;
        vector<int> x;
        dfs(0,x,cnt,1);                          //i从1开始
        return ans;
    }
    void dfs(int cs,vector<int> & x,int cnt,int i)      //递归回溯算法
    {   if (i>=10)                               //找到一个叶子结点
        {   if (cs==N && cnt==K)                 //找到一个满足条件的解
                ans.push_back(x);
        }
        else                                     //没有到达叶子结点
        {   x.push_back(i);                      //选取整数i
            cnt++;
            dfs(cs+i,x,cnt,i+1);
            cnt--;                               //回溯
            x.pop_back();
            dfs(cs,x,cnt,i+1);                   //不选取整数i
        }
    }
};
```

5.5.2 LeetCode39——组合总和

问题描述：给定一个无重复元素的数组 a 和一个目标数 t，找出 a 中所有可以使数字和为 t 的组合。a 中的数字可以无限制被重复选取，其中所有数字（包括 t）都是正整数。例如，$a=\{2,3,6,7\}$，$t=7$ 的结果为$\{\{7\},\{2,2,3\}\}$，而 $a=\{2,3,5\}$，$t=8$ 的结果为$\{\{2,2,2,2\},\{2,3,3\},\{3,5\}\}$。要求设计如下函数：

```
class Solution {
public:
    vector<vector<int>> combinationSum(vector<int> & a,int t) { }
};
```

解：与 LeetCode216 题目类似，但这里 a 中每个元素可以重复选取，同样用解向量 $\boldsymbol{x}$ 存放选取的所有整数，用 i 从 0 开始遍历 a，增加一个剩余数的参数 rt（rt 为 t 与当前选取的整数和的差，初始值为 t）。当搜索第 i 层的一个结点时，求出 cnt=rt/$a[i]$（表示最多可以选取 cnt 个 $a[i]$），这样 $a[i]$ 的选取分为 cnt+1 种情况：不选取 $a[i]$，选取一个 $a[i]$，选取两个 $a[i]$，…，选取 cnt 个 $a[i]$，如图 5.14 所示。当 $i \geqslant N$（对应解空间的一个叶子结点）并且 rt=0 时对应一个解 $\boldsymbol{x}$，将 $\boldsymbol{x}$ 添加到 ans 中。

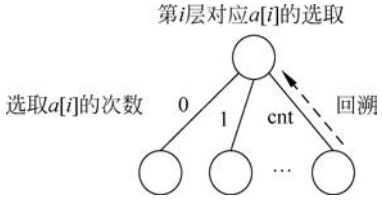

图 5.14 $a[i]$元素是在 cnt+1 种情况中选一

另外也可以将满足 rt=0 的结点作为一个解，剪去 $i \geqslant N$ 或者 rt<0 的分支，或者说仅扩展满足 $i<N$ 并且 rt>0 的结点。对应的程序如下：

```
class Solution {
    vector < vector < int >> ans;
    int N;
public:
    vector < vector < int >> combinationSum( vector < int > & a, int t)
    {   N=a.size();
        vector < int > x;
        dfs(a,t,x,0);                                   //i 从 0 开始
        return ans;
    }
    void dfs( vector < int > &a, int rt, vector < int > & x, int i)    //递归回溯算法
    {   if (rt==0)                                      //找到一个叶子结点
            ans.push_back(x);
        else if(i < N && rt > 0)
        {   dfs(a,rt,x,i+1);                            //不选择 a[i]
            int cnt=0;
            for(int j=1;a[i] * j <=rt;j++)              //枚举 a[i]可以选取的次数
            {   cnt++;
                x.push_back(a[i]);                      //包含 a[i]选取 1,2,…,cnt 次
                dfs(a,rt-a[i] * j,x,i+1);
            }
            for(int j=0;j < cnt;j++)
                x.pop_back();                           //前面 a[i]最多取 cnt 次,回溯 cnt 次
        }
    }
};
```

上述程序的提交结果为通过，执行时间为 8ms，内存消耗为 14.6MB。

5.5.3 LeetCode131——分割回文串

问题描述：给定一个字符串 s（长度范围是 1～16，均由小写英文字母组成），请将 s 分割成一些子串，使每个子串都是回文串，返回 s 所有可能的分割方案。回文串是正着读和反着读都一样的字符串。例如，s="aab"，结果是[["a","a","b"],["aa","b"]]。要求设计如下函数：

```
class Solution {
public:
    vector < vector < string >> partition(string s) {   }
};
```

解：用 ans 存放所有分割方案。i 从 0 开始，找到 $s[i..j]$的每一个回文的终止位置 j，所以该问题类似子集和问题，假设有 k 个这样的 j，扩展就是在 k 个情况中选取一个，再从

$j+1$ 位置开始继续搜索。解向量 x 存放一个分割方案，当 $i \geqslant n$ 时表示找到了 s 的一个解，将 x 添加到 ans 中，最后返回 ans。对应的程序如下：

```
class Solution {
    vector<vector<string>> ans;                          //存放所有方案
public:
    vector<vector<string>> partition(string s)
    {   vector<string> x;
        dfs(s,x,0);
        return ans;
    }
    void dfs(string s,vector<string> & x,int i)          //从 i 位置开始分割回文
    {   int n=s.size();
        if(i>=n)                                         //找到一个解
            ans.push_back(x);
        else
        {   for(int j=i;j<n;j++)                         //试探 i 开始的每一个位置 j
            {   string s1=s.substr(i,j-i+1);             //取出 s[i..j]的子串 s1
                if(isPalindrome(s1))                     //若 s1 是回文
                {   x.push_back(s1);
                    dfs(s,x,j+1);
                    x.pop_back();                        //回溯
                }
            }
        }
    }
    bool isPalindrome(string s)                          //判断 s 是否为回文
    {   int i=0, j=s.size()-1;
        while(i<j)
        {   if(s[i]!=s[j])
                return false;
            i++; j--;
        }
        return true;
    }
};
```

上述程序的提交结果为通过，执行时间为 124ms，内存消耗为 78MB。

5.5.4 HDU1027——第 k 小的排列

问题描述：给定正整数 n 和 k，求 $1 \sim n$ 的全排列中第 k 小的排列。

输入格式：输入包含多个测试用例，每个测试用例一行，由两个整数（即 n 和 k）组成（$1 \leqslant n \leqslant 1000, 1 \leqslant k \leqslant 10000$）。输入直到文件结束。

输出格式：对于每个测试用例，在一行中输出 $1 \sim n$ 中第 k 小的排列，两个数字之间输出一个空格，但不要在最后一个数字后输出任何空格。

输入样例：

```
6 4
11 8
```

输出样例：

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

1 2 3 4 5 6 7 9 8 11 10

解：采用基于子集树的回溯框架，用解向量 $x[1..n]$ 存放一个排列（初始置为 1～n），设计一个重复标记数组 used，used[j]表示数字 j 是否已经使用。解空间的根结点层次 $i=1$。对于第 i 层的结点，$x[i]$可能的取值为 1～n 中未使用过的数字 j。cnt 累计求出的排列个数，由于这样做求出的所有排列是递增的，所以当 cnt=k 时对应的 x 中的排列就是第 k 小的排列。对应的程序如下：

```
#include<iostream>
#include<cstring>
using namespace std;
#define INF 0x3f3f3f3f
#define MAXN 10005
int x[MAXN];                    //存放一个排列
bool used[MAXN];                //used[j]表示j是否使用过
int cnt;
bool flag;
bool permutation(int i,int n,int k)     //回溯算法
{   if(i==n+1)                  //到达一个叶子结点
            {   cnt++;
                if(cnt==k)              //找到第k个解时返回true
                    return true;
            }
            else
            {   for(int j=1;j<=n;j++)
                {   if(!used[j])        //整数j没有使用过
                    {   x[i]=j;
                        used[j]=true;   //标记j已经使用
                        if (permutation(i+1,n,k))
                            return true;
                        used[j]=false;  //回溯
                    }
                }
            }
    return false;
}
int main()
{   int n,k;
    while(scanf("%d%d",&n,&k)!=EOF)
    {   flag=false;
        memset(used,false,sizeof(used));
        cnt=0;
        for(int i=1;i<=n;i++)           //初始化
            x[i]=i;
        permutation(1,n,k);
        for(int i=1;i<=n;i++)           //输出结果
        {   if(i==1)
                printf("%d",x[i]);
            else
                printf(" %d",x[i]);
        }
        printf("\n");
    }
    return 0;
}
```

上述程序的提交结果为通过，执行时间为 452ms，内存消耗为 1776KB。

5.5.5 HDU2553——*n* 皇后问题

问题描述：在 $n\times n$ 个方格的棋盘上放置了 n 个皇后，使得它们不相互攻击（即任意两个皇后不允许处在同一排，同一列，也不允许处在与棋盘边框成 45°角的斜线上）。对于给定的 n，请求出有多少种合法的放置方法。

输入格式：共有若干行，每行一个正整数 n（$n\leqslant 10$），表示棋盘和皇后的数量，如果 $n=0$ 表示结束。

输出格式：共有若干行，每行一个正整数，表示对应输入行的皇后的不同放置数量。

输入样例：

```
1
8
5
0
```

输出样例：

```
1
92
10
```

解：采用《教程》5.2.4 节中求 n 皇后问题的思路，仅将输出全部解改为输出解的个数。另外，这里 n 最大为 10，但测试用例个数可能超过 10，所以先求出 1～10 皇后问题，将解个数存放在 ans 数组中，当输入的皇后个数为 n 时直接输出 ans[n]即可。采用基于子集树递归函数框架的程序如下：

```
#include<iostream>
#include<vector>
using namespace std;
#define MAXN 12
int q[MAXN];                                    //存放各皇后所在的列号,为全局变量
int cnt;                                        //累计解个数
bool place(int i,int j)                         //测试(i,j)位置能否摆放皇后
{   if (i==1) return true;                      //第一个皇后总是可以放置
    int k=1;
    while (k<i)                                 //k=1～i-1是已放置了皇后的行
    {   if ((q[k]==j) || (abs(q[k]-j)==abs(i-k)))
            return false;
        k++;
    }
    return true;
}
void queen(int n,int i)                         //回溯算法
{   if (i>n)
        cnt++;                                  //解个数增加1
    else
    {   for (int j=1;j<=n;j++)                  //在第i行上试探每一个列j
        {   if (place(i,j))                     //在第i行上找到一个合适的位置(i,j)
            {   q[i]=j;
                queen(n,i+1);
                q[i]=0;                         //回溯
```

```
            }
        }
    }
}
int main()
{   int ans[MAXN];                          //ans[n]存放 n 皇后问题的解个数
    for(int n=1;n<=10;n++)
    {   cnt=0;
        queen(n,1);
        ans[n]=cnt;
    }
    int n;
    while(~scanf("%d",&n) && n)
        printf("%d\n",ans[n]);
    return 0;
}
```

上述程序的提交结果为通过，执行时间为 31ms，内存消耗为 1372KB。当然也可以采用基于排列树的递归回溯框架求解，对应的程序如下：

```
#include<iostream>
#include<cstring>
using namespace std;
#define MAXN 12                         //最多皇后个数
int q[MAXN];                            //存放各皇后所在的列号，为全局变量
int cnt;                                //累计解个数
bool place(int i,int j)                 //测试(i,j)位置能否摆放皇后
{   if (i==1) return true;              //第一个皇后总是可以放置
    int k=1;
    while (k<i)                         //k=1～i-1 是已放置了皇后的行
    {   if ((q[k]==j) || (abs(q[k]-j)==abs(i-k)))
            return false;
        k++;
    }
    return true;
}
void queen(int n,int i)                 //回溯算法
{   if (i>n)
        cnt++;                          //解个数增加 1
    else
    {   for (int j=i;j<=n;j++)          //在第 i 行上试探每一个列 j
        {   swap(q[i],q[j]);            //第 i 个皇后放置在 q[j]列
            if(place(i,q[i]))           //剪支操作
                queen(n,i+1);
            swap(q[i],q[j]);            //回溯
        }
    }
}
int main()
{   int ans[MAXN];
    for(int n=1;n<=10;n++)
    {   cnt=0;
        for(int i=1;i<=n;i++)
            q[i]=i;
        queen(n,1);
```

```
        ans[n]=cnt;
    }
    int n;
    while(~scanf("%d",&n) && n)
        printf("%d\n",ans[n]);
    return 0;
}
```

上述程序的提交结果为通过，执行时间为 15ms，内存消耗为 1740KB。

5.5.6 HDU2616——杀死怪物

问题描述：YF 的家乡附近有一座山，山上住着一个大怪物，YF 想要消灭它。YF 有 n 个技能，怪物的血量为 m，当血量小于或等于 0 时怪物被消灭。YF 的技能在不同的时间使用时有不同的效果。现在告诉你每个技能的效果，用(A,M)描述，A 为该技能在普通时间内使用时消耗怪物的血量，M 表示当怪物的血量小于或等于 M 时使用该技能可以获得双倍效果。每种技能最多只能使用一次。

输入格式：输入包含多个测试用例。每个测试用例的第一行是两个整数 n 和 m（$2<n<10$，$1<m<10^7$），n 为 YF 的技能数量，接下来 n 行，每行的(A_i,M_i)（$0<A_i,M_i\leqslant m$）描述一个技能。

输出格式：对于每个测试用例，输出一个整数表示 YF 至少应该使用多少技能才能消灭怪物，如果 YF 不能消灭怪物则输出－1。

输入样例：

```
3 100
10 20
45 89
5 40

3 100
10 20
45 90
5 40

3 100
10 20
45 84
5 40
```

输出样例：

```
3
2
-1
```

解：n 个技能的编号为 0～n－1，每个仅能够使用一次，用 used 标记（值为 0 表示该技能没有使用，值为 1 表示已经使用）。设解向量 $\boldsymbol{x}=(x_0,x_1,\cdots,x_{i-1})$，表示使用 i 次（个）技能消灭了怪物，x_j 表示第 j 次使用的技能的编号，rm 表示怪物的剩余血量（初始为 m），叶子结点对应 rm≤0（怪物被消灭）。i 从 0 开始试探（根结点的层次为 0），解向量中的第 i 层结点表示第 i 次选择技能，可选范围是 0～n－1 中没有使用的技能，一旦到达一个叶子结点，比较求最小的 i 存放到 bestx 中，最后输出 bestx 即可。由于仅求最小的 i，所以不必定

义解向量 x。对应的程序如下：

```
#include<iostream>
#include<algorithm>
using namespace std;
#define MAXN 15
#define INF 0x3f3f3f3f
int n;
int bestx;                              //bestx 表示最优解
struct Spell                            //技能的类型
{   int a,b;
    int used;                           //标记该技能是否已经使用过
} s[MAXN];
void dfs(int rm,int i)
{   if(rm<=0)                           //怪物被消灭对应叶子结点
        bestx=min(bestx,i);             //比较求最小值
    else
    {   for(int j=0;j<n;j++)            //试探第 i 次使用的技能
        {   if(s[j].used)               //跳过已经使用过的技能
                continue;
            s[j].used=1;                //第 i 次使用技能 j
            if(rm<=s[j].b)              //技能力量倍增的情况
                dfs(rm-s[j].a*2,i+1);
              else                      //技能普通使用的情况
                dfs(rm-s[j].a,i+1);
            s[j].used=0;                //回溯
        }
    }
}
int main()
{   int m;
    while(cin>>n>>m)
    {   for(int i=0;i<n;i++)
        {   cin>>s[i].a>>s[i].b;
            s[i].used=0;
        }
        bestx=INF;
        dfs(m,0);
        if(bestx==INF)
              cout<<-1<<endl;
        else
            cout<<bestx<<endl;
    }
    return 0;
}
```

上述程序的提交结果为通过，执行时间为 124ms，内存消耗为 1800KB。

5.5.7 POJ3187——向后数字和

问题描述：给定一个 1～n 的某个排列，将相邻数字相加以生成一个少一个数字的新列表，重复这个操作，直到只剩下一个数字 x 为止。例如，若给定一个 $n=4$ 的排列为 3，1，2，4，第一次相邻数字相加的结果是 4，3，6，第 2 次相邻数字相加的结果是 7，9，第 3 次相邻数字相加的结果是 16，那么该排列产生的向后数字和 $x=16$。不同排列的向后数字和可能不同。

输入格式：输入有多个测试用例，每个测试用例是两个由空格分隔的整数 n 和 sum。

输出格式：对于每个测试用例，求出其向后数字和等于 sum 的某个 1～n 的排列，如果有多个这样的排列，请选择按字典顺序排列最小的一个。

输入样例：

```
4 16
```

输出样例：

```
3 1 2 4
```

提示：另外一个满足要求的答案是 3,2,1,4，但 3,1,2,4 是最小的。

解：题目就是枚举 1～n 的排列，求出其向后数字和 x，若 x＝sum，则输出该排列。求 1～n 的全排列可以采用基于排列树的递归算法框架，对应的程序如下：

```
#include <iostream>
#include <vector>
using namespace std;
#define MAXN 12
int n,sum;
vector<int> a;
vector<int> ans;
bool judge(vector<int> b)                    //判断 b 序列是否满足条件
{   for (int i=n-1;i>=1;i--)
    {   for (int j=0;j<=i-1;j++)
            b[j]=b[j]+b[j+1];
    }
    if (b[0]==sum)
        return true;
    else
        return false;
}
bool flag;
void dfs(vector<int> &a,int i)               //递归算法
{   if (i>=n)                                //到达一个叶子结点
    {   if(judge(a))
        {   ans=a;
            flag=true;
        }
    }
    else if(!flag)                           //没有到达叶子结点且 flag 为 false
    {   for (int j=i;j<n;j++)
        {   swap(a[i],a[j]);                 //交换 a[i]与 a[j]
            dfs(a,i+1);
            swap(a[i],a[j]);                 //交换 a[i]与 a[j]:恢复
        }
    }
}
int main()
{   while (cin >> n >> sum)
```

```
    {   a.resize(n);
        for (int j=0;j<n;j++)
            a[j]=j+1;
        flag=false;
        dfs(a,0);
        for (int i=0;i<n;i++)
            cout << ans[i] << " ";
        cout << endl;
    }
    return 0;
}
```

但是上述程序提交时出现答案错误，原因是这样求出的全排列并非按字典顺序递增排列的。例如，对于测试用例 4,16，找到的答案是 3,2,1,4，不满足题目的要求。这里使用 STL 中提供的一个求全排列的通用函数 next_permutation。它产生的全部排列就是按字典顺序递增排列的，利用该函数求解的程序如下：

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;
int n,sum;
bool judge(vector<int> b)                    //判断 b 序列是否满足条件
{   for (int i=n-1;i>=1;i--)
    {   for (int j=0;j<=i-1;j++)
            b[j]=b[j]+b[j+1];
    }
    if (b[0]==sum)
        return true;
    else
        return false;
}
int main()
{   while (cin >> n >> sum)
    {   vector<int> a(n);
        for (int j=0;j<n;j++)
            a[j]=j+1;
        do
        {   if(judge(a)) break;
        } while (next_permutation(a.begin(),a.end()));
        for (int i=0;i<n;i++)
            cout << a[i] << " ";
        cout << endl;
    }
    return 0;
}
```

上述程序提交时通过，执行时间为 625ms，内存消耗为 188KB，满足时空要求。

5.5.8 POJ1321——棋盘问题

问题描述：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，要求摆放时任意的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 C。

输入格式：输入包含多个测试用例。每个测试用例的第一行是两个正整数 n 和 k（$n \leqslant 8$，$k \leqslant n$），表示在一个 $n \times n$ 的矩阵内描述棋盘以及摆放棋子的数目，当为 -1 -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）。

输出格式：对于每个测试用例输出一行表示摆放的方案数目 C（数据保证 $C < 2^{31}$）。

输入样例：

```
2  1
#  .
.  #
4  4
.  .  .  #
.  .  #  .
.  #  .  .
#  .  .  .
-1  -1
```

输出样例：

```
2
1
```

解：本题类似 n 皇后问题，但有以下几点不同。

① 棋子之间的冲突更加简单，只有不同行列，设置一个 used 数组，used[j]=1 表示第 j 列已经放了棋子，used[j]=0 表示第 j 列没有放棋子（初始时所有元素设置为 0）。

② 每行中最多只能在指定的位置放棋子（'#'的位置）。

③ 有的行可能不放棋子，而 n 皇后问题中每一行都需要放置一个皇后。

这样该问题相当于子集和问题，每行中'#'就是子集和问题中的一个整数，一个解就是在 n 个整数中挑选 k 个没有冲突的整数。设计解向量 $\boldsymbol{x}=(x_0,x_1,\cdots,x_{n-1})$，$x_i$ 表示第 i 行放置棋子的列号，注意 x_i 有两种选择，第 i 行不放置棋子或者找到一个'#'位置放置一个棋子。用 tot 表示放置棋子的个数，$i \geqslant n$ 对应一个叶子结点，如果到达某个叶子结点并且 tot=k，则解的个数 cnt 增加 1，最后输出 cnt 即可。由于题目不需要输出每一个解，仅需要求解的个数，所以不必设计解向量 $\boldsymbol{x}$。对应的程序如下：

```
#include<iostream>
#include<cstring>
using namespace std;
#define MAXN 12
char map[MAXN][MAXN];
int n,k;
int cnt;                          //方案个数
int used[MAXN];                   //used[j]表示第 j 列是否放了棋子
void queen(int tot,int i)         //回溯算法
{   if(i>=n)
    {   if (tot==k)               //棋子的个数恰好为 k
            cnt++;                //解的个数增加 1
    }
```

```
        else
        {   queen(tot,i+1);                    //第 i 行不放棋子
            for (int j=0;j<n;j++)              //第 i 行放一个棋子,试探每一个列 j
            {   if (map[i][j]=='#' && !used[j])
                {   used[j]=1;                 //在第 i 行第 j 列放一个棋子
                    queen(tot+1,i+1);
                    used[j]=0;                 //回溯
                }
            }
        }
    }
    int main()
    {   while(scanf("%d%d",&n,&k) && n!=-1 && k!=-1)
        {   for (int j=0;j<n;j++)              //输入一个测试用例
                scanf("%s",map[j]);
            cnt=0;
            memset(used,0,sizeof(used));
            queen(0,0);                        //求解
            printf("%d\n",cnt);                //输出结果
        }
        return 0;
    }
```

上述程序提交时通过,执行时间为 47ms,内存消耗为 128KB,满足时空要求。

5.5.9 POJ2488——骑士游历

问题描述:给定一个 $n\times m$ 的棋盘,一个骑士希望从其中某个方格出发走遍棋盘中的所有方格,请帮助他找到一条这样的路径。如果骑士在(x,y)位置,骑士走一步的 8 个位置如图 5.15 所示。

	$x-1,y-2$		$x+1,y-2$	
$x-2,y-1$				$x+2,y-1$
		x,y		
$x-2,y+1$				$x+2,y+1$
	$x-1,y+2$		$x+1,y+2$	

图 5.15 骑士走一步的 8 个位置

输入格式:输入的第一行为正整数 t,表示测试用例的个数。每个测试用例的输入为两个正整数 n 和 m,表示一个 $n\times m(1\leqslant n\times m\leqslant 26)$的棋盘。

输出格式:每个测试用例的输出以包含"Scenario #no:"的行开头,其中 no 是从 1 开始的测试用例的编号,然后输出一行表示找到的一条路径,路径采用字符串表示。例如 A1B3C1A2B4C2A3B1C3A4B2C4 表示一条长度为 12 的路径,由 12 个方格组成,每个方格由行号和列号构成(编号从 1 开始),但是行号用大写字母表示,$A=1$,$B=2$,以此类推,列号直接用数字表示。两个测试用例的结果输出之间空一行。如果不存在这样的路径,则输出"impossible"字符串。

输入样例:

```
3
1 1
2 3
4 3
```

输出样例:

```
Scenario #1:
A1
```

```
Scenario #2:
impossible

Scenario #3:
A1B3C1A2B4C2A3B1C3A4B2C4
```

解：类似迷宫问题，但行走方向有8个方位，用以下偏移量表示。

```
int dx[]={-1, 1, -2, 2,-2,2,-1,1};                    //x方向的偏移量
int dy[]={-2,-2,-1,-1, 1,1, 2, 2};                    //y方向的偏移量
```

遍历棋盘的每个位置(i,j)，从该位置出发搜索路径，用ans数组存放一条路径，用cnt参数表示路径上经过的方格个数，当cnt=$n*m$时表示找到了一条合适的路径ans，按题目要求输出即可。由于只需要找一条路径，用flag表示是否找到了路径(初始为false)，一旦找到路径置flag为true，并且终止搜索，如果最后flag为false，说明没有路径。对应的程序如下：

```
#include<iostream>
#include<cstring>
using namespace std;
#define MAXN 33
int dx[]={-1,1,-2,2,-2,2,-1,1};                       //x方向的偏移量
int dy[]={-2,-2,-1,-1,1,1,2,2};                       //y方向的偏移量
int n,m;
struct Box                                            //方块类型
{   int x,y;
    Box() {}
    Box(int x,int y):x(x),y(y) {}                     //构造函数
} ans[MAXN*MAXN];
int visited[MAXN][MAXN];
bool flag;
void dfs(int x,int y,int cnt)                         //从(x,y)出发搜索路径
{   if(flag) return;                                  //找到一条路径后返回
    if(cnt==n*m)                                      //找到一个解后输出
    {   flag=true;
        for(int i=1;i<=cnt;i++)
            printf("%c%d",ans[i].y+'A',ans[i].x+1);
        printf("\n");
    }
    else
    {   for(int di=0;di<8;di++)                       //试探8个方位
        {   int nx=x+dx[di];
            int ny=y+dy[di];
            if(nx<0 || nx>=n || ny<0 || ny>=m)        //(nx,ny)超界时跳过
                continue;
            if(visited[nx][ny])                       //(nx,ny)已访问时跳过
                continue;
            visited[nx][ny]=1;
            ans[cnt+1]=Box(nx,ny);
            dfs(nx,ny,cnt+1);
            visited[nx][ny]=0;                        //回溯,为什么ans不必回退
        }
    }
}
int main()
{   int t;
```

```
    scanf("%d",&t);
    for(int no=1;no<=t;no++)
    {   if(no!=1) printf("\n");
        scanf("%d%d",&n,&m);
        memset(visited,0,sizeof(visited));
        flag=false;
        printf("Scenario #%d:\n",no);
        for(int i=0;i<n;i++)
        {   for(int j=0;j<m;j++)
            {   visited[i][j]=1;
                ans[1]=Box(i,j);
                dfs(i,j,1);
                visited[i][j]=0;
                if(flag) break;
            }
            if(flag) break;
        }
        if(!flag) printf("impossible\n");
    }
    return 0;
}
```

上述程序提交时通过，执行时间为16ms，内存消耗为136KB，满足时空要求。

5.5.10 POJ1040——运输问题

问题描述：一条火车线路共有 $m+1$ 个车站，起点站A的编号为0，终点站B的编号为 m，其他站的编号为 $1\sim m-1$，火车按车站编号顺序依次停靠，每位乘客的车票价格是他乘坐的车站数。火车有 s 个座位，火车行驶中任何时刻不能超载。现在有 n 个订单，每个订单包含起点站、终点站和乘客人数(一个订单中的所有乘客的行程是相同的)。求火车从车站A开到车站B的最大总收入，注意一个订单要么被执行(该订单的所有乘客都可以乘车)，要么被拒绝。

输入格式：输入包含多个测试用例。每个测试用例的第一行包含3个整数，即s、m 和 n($m\leqslant 8$，$n\leqslant 22$)，接下来 n 行，每行是一个订单信息，由起点站、终点站和乘客人数3个整数组成。第一行中的3个整数都等于0时表示输入结束。

输出格式：对于每个测试用例，输出一行包含最大总收入的整数。

输入样例：

```
10 3 4
0 2 1
1 3 5
1 2 7
2 3 10
10 5 4
3 5 10
2 4 9
0 2 5
2 5 8
0 0 0
```

输出样例：

```
19
34
```

解：用 order 数组存放 n 个订单，采用回溯法求解，i 从 0 到 $n-1$ 处理所有订单，每个订单 i 的处理有 3 种情况。

① 不选取订单 i。

② 考虑订单 i，考虑后满足约束条件，则执行订单 i。

③ 考虑订单 i，考虑后不满足约束条件，则不执行订单 i。

从中看出，本问题就是三选一的子集树问题，也可以看成执行订单 i 和不执行订单 i 的二选一问题。为了确定约束条件，设计 cnt 数组，其中 cnt[j]表示当前车站 j 的乘客人数。当考虑订单 i 时，累计该订单中乘客乘坐到达的车站 j 的人数，若 cnt[j]>s，则说明不满足约束条件，不能执行订单 i。

使用 besttot 表示最大总收入(初始为 0)，当前总收入用 tot 表示。当 $i \geqslant n$ 时表示当前方案的总收入为 tot，比较将最大值存放到 besttot，最后输出 besttot 即可。对应的程序如下：

```
#include<iostream>
#include<cstring>
#include<algorithm>
using namespace std;
#define MAXN 25
int s,m,n;
struct Orders                          //车站订单类型
{   int start;                         //起点站
    int dest;                          //终点站
    int peop;                          //乘客人数
    int mon;                           //该订单的价格
} order[MAXN];
int cnt[MAXN];                         //cnt[i]表示到达车站i的总乘客人数
int besttot;                           //最优解
void dfs(int tot,int i)
{   if(i>=n)
        besttot=max(besttot,tot);
    else
    {   dfs(tot,i+1);                  //不考虑订单i
        bool flag=true;                //考虑订单i
        for(int j=order[i].start+1;j<=order[i].dest;j++)  //处理订单i
        {   cnt[j]+=order[i].peop;
            if(cnt[j]>s)               //不能执行该订单
                flag=false;
        }
        if(flag)                       //如果能够执行该订单
            dfs(tot+order[i].mon,i+1); //执行订单i
        for(int j=order[i].start+1;j<=order[i].dest;j++)  //回溯
            cnt[j]-=order[i].peop;
    }
}
int main()
{   while(cin>>s>>m>>n)
    {   if(s==0 && m==0 && n==0) break;
        memset(cnt,0,sizeof(cnt));
        besttot=0;
        for(int i=0;i<n;i++)
        {   cin>>order[i].start>>order[i].dest>>order[i].peop;
            order[i].mon=(order[i].dest-order[i].start)*order[i].peop;    //求订单的价格
```

```
        }
        dfs(0,0);
        cout << besttot << endl;
    }
}
```

上述程序提交时通过，执行时间为 625ms，消耗的空间为 212KB，满足时空要求。

5.5.11 POJ1129——最少频道数

问题描述：现在需要在一个非常大的区域建立一个广播电台，想让该区域都能够接收到信号，必须建立一些用于转播信号的中继器，但是相邻的两个中继器的频道必须不同，否则会相互干扰，不相邻的中继器可以使用相同的频道。给定一个中继网络，要求求出所需要的最少不同频道数。

输入格式：输入包含多个测试用例，每个测试用例的第一行是中继器的个数 n，接下来共 n 行，每一行表示一个中继器的邻接关系，例如 A:BCDH 表示中继器 A 与中继器 B、C、D 和 H 相邻。注意相邻是对称关系，如果 A 与 B 相邻，则 B 必然与 A 相邻。中继器采用大写字母表示，邻接关系按字母顺序列出，中继器的个数 n 最多为 26 个。以输入 $n=0$ 表示结束。

另外，由于中继器位于一个平面内，所以连接相邻中继器形成的图形没有任何交叉的线段。

输出格式：对于每个测试用例，输出一行表示所需的最少频道数。示例输出显示了该行的格式。当只需要一个频道时，注意频道是单数形式。

输入样例：

```
2
A:
B:
4
A:BC
B:ACD
C:ABD
D:BC
4
A:BCD
B:ACD
C:ABD
D:ABC
0
```

输出样例：

```
1 channel needed.
3 channels needed.
4 channels needed.
```

解：由输入建立一个无向图的邻接表，每个顶点表示一个中继器，采用《教程》5.2.7 节中图的 m 着色算法的思路，每种不同的频道就是一种不同的颜色，题目就是求使得图中所有相邻顶点着上不同颜色所需要的最少颜色数。但这里没有给出颜色数 m，并且要求最小的 m（最优解是最小值），属于问题Ⅱ类型。

相关变量设计与 m 着色算法的相同，仅增加 bestm 表示最小的 m，根据四色定理，任意这样的图最多 4 种颜色即可着色。让 m 从 1 到 4 循环，从顶点 0 开始执行 dfs 回溯算法，若 cnt>0 说明找到了最小的 m，置 bestm，最后输出 bestm。对应的程序如下：

```
#include<iostream>
#include<vector>
#include<cstring>
using namespace std;
#define INF 0x3f3f3f3f                    //表示∞
#define MAXN 30                           //最多顶点个数
int n;
int x[MAXN];
int bestm;
int cnt;                                  //全局变量,累计解个数
vector<int> A[MAXN];                      //邻接表
bool judge(int i,int j)                   //判断顶点 i 是否可以着上颜色 j
{   for(int k=0;k<A[i].size();k++)
    {   if(x[A[i][k]]==j)                 //存在相同颜色的顶点
        return false;
    }
    return true;
}
void dfs(int m,int i)                     //递归回溯算法
{   if(i>=n)                              //到达一个叶子结点
        cnt++;
    else if(cnt==0)                       //如果 cnt>0 就没有必要继续了
        {   for(int j=0;j<m;j++)
            {   x[i]=j;                   //设置顶点 i 颜色为 j
                if(judge(i,j))            //若顶点 i 可以着颜色 j
                    dfs(m,i+1);
                x[i]=-1;                  //回溯
            }
        }
}
int main()
{   char str[MAXN];
    while(scanf("%d",&n)!=EOF)
    {   if(n==0) break;
            bestm=INF;
        for(int i=0;i<=30;i++)            //初始化 A
            A[i].clear();
        for(int i=0;i<n;i++)              //由输入建立邻接表 A
        {   scanf("%s",str);
            for(int j=2;str[j]!='\0';j++) //A→0,B→1,以此类推
                A[(str[0]-'A')].push_back(str[j]-'A');
        }
        for(int m=1;m<=4;m++)             //循环找最小的 m
        {   memset(x,0xff,sizeof(x));     //所有元素初始化为-1
            cnt=0;
            dfs(m,0);                     //从顶点 i=0 开始
            if(cnt>0)
            {   bestm=m;
                break;                    //一旦找到就结束循环
            }
        }
        if(bestm==1)                      //按题目要求输出结果
```

```
            printf("1 channel needed.\n");
        else
            printf("%d channels needed.\n",bestm);
    }
    return 0;
}
```

上述算法需要调用 bestm 次 dfs 算法，也就是说若 bestm＝4，需要依次独立地调用 dfs(1,0)，dfs(2,0)，dfs(3,0)，dfs(4,0)，这样性能比较低，可以改为从 $m=1$ 开始，当没有找到合适的着色时置 m++继续搜索，这样调用一次 dfs 即可。对应的改进算法如下：

```
#include<iostream>
#include<vector>
#include<cstring>
using namespace std;
#define INF 0x3f3f3f3f                    //定义为∞
#define MAXN 30                           //最多顶点个数
int n;
int x[MAXN];
int bestm;
vector<int> A[MAXN];                      //邻接表
bool judge(int i,int j)                   //判断顶点 i 是否可以着颜色 j
{   for(int k=0;k<A[i].size();k++)
    {   if(x[A[i][k]]==j)                 //存在相同颜色的顶点
        return false;
    }
    return true;
}
void dfs(int m,int i)                     //递归回溯算法
{   if(i>=n)                              //到达一个叶子结点
        bestm=min(bestm,m);
    else
    {   for(int j=0;j<m;j++)
        {   x[i]=j;                       //设置顶点 i 颜色为 j
            if(judge(i,j))                //若顶点 i 可以着颜色 j
                dfs(m,i+1);
            x[i]=-1;                      //回溯
        }
        if(m<4)                           //执行到这里说明 m 小了，但最多 4 种颜色即可
        {   x[i]=m;                       //增加一种颜色 m(注意增加的颜色编号为 m)
            dfs(m+1,i+1);
            x[i]=-1;                      //回溯
        }
    }
}
int main()
{   char str[MAXN];
    while(scanf("%d",&n)!=EOF)
    {   if(n==0) break;
        bestm=INF;
        for(int i=0;i<=30;i++)            //初始化 A
            A[i].clear();
        for(int i=0;i<n;i++)              //由输入建立邻接表 A
        {   scanf("%s",str);
            for(int j=2;str[j]!='\0';j++)
                A[(str[0]-'A')].push_back(str[j]-'A');
```

```
            }
            memset(x,0xff,sizeof(x));          //所有元素初始化为-1
            dfs(0,0);                          //从颜色数 m=0,顶点 i=0 开始
            if(bestm==1)
                printf("1 channel needed.\n");
            else
                printf("%d channels needed.\n",bestm);
        }
        return 0;
    }
```

非常有趣的是上述两个程序在 POJ 提交时执行时间均为 0ms,说明测试数据比较小,理论上讲后面一个算法的性能较高。