第5章 回溯法

回溯法采用类似穷举法的搜索尝试过程，在搜索尝试过程中寻找问题的解，当发现已不满足求解条件时就“回溯”（即回退），尝试其他路径，所以回溯法有“通用解题法”之称。本章介绍用回溯法求解问题的一般方法，并给出一些用回溯法求解的经典示例。本章的学习要点和学习目标如下：

（1）掌握问题解空间的结构和深度优先搜索过程。

（2）掌握回溯法的原理和算法框架。

（3）掌握剪支函数（约束函数和限界函数）设计的一般方法。

（4）掌握各种回溯法经典算法的设计过程和分析方法。

（5）综合运用回溯法解决一些复杂的实际问题。

5.1 回溯法概述

5.1.1 问题的解空间

在 3.1 节讨论穷举法时简要介绍过解空间，由于解空间是回溯法等的核心概念，这里作进一步讨论。一个复杂问题的解决方案往往是由若干个小的决策(即选择)步骤组成的决策序列，所以一个问题的解可以表示成解向量 $\boldsymbol{x}=(x_0,x_1,\cdots,x_{n-1})$，其中分量 x_i 对应第 i 步的选择，通常可以有两个或者多个取值，表示为 $x_i\in S_i(0\leqslant i\leqslant n-1)$，$S_i$ 为 x_i 的取值候选集，即 $S_i=(v_{i,0},v_{i,1},\cdots,v_{i,|S_i|-1})$。$\boldsymbol{x}$ 中各个分量 x_i 所有取值的组合构成问题的解向量空间，简称为**解空间**，解空间一般用树形式来组织，树中每个结点对应问题的某个状态，所以解空间也称为解空间树或者状态空间树。

解空间的一般结构如图 5.1 所示，根结点(为第 0 层)的每个分支对应分量 x_0 的一个取值(或者说 x_0 的一个决策)，若 x_0 的候选集为 $S_0=\{v_{0,1},\cdots,v_{0,a}\}$，即根结点的子树个数为 $|S_0|$，例如 $x_0=v_{0,0}$ 时对应第 1 层的结点 A_0，$x_0=v_{0,1}$ 时对应第 1 层的结点 A_1，A_2，…。对于第 1 层的每个结点 A_i，A_i 的每个分支对应分量 x_1 的一个取值，若 x_1 的取值候选集为 $S_1=\{v_{1,0},\cdots,v_{1,b}\}$，$A_i$ 的分支数为 $|S_1|$，例如对于结点 A_0 当 $x_1=v_{1,0}$ 时对应第 2 层的结点 B_0，…。以此类推，最底层是叶子结点层，叶子结点的层次为 n，解空间的高度为 $n+1$。从中看出第 i 层的结点对应 x_i 的各种选择，从根结点到每个叶子结点有一条路径，路径上每个分支对应一个分量的取值，这是理解解空间的关键。

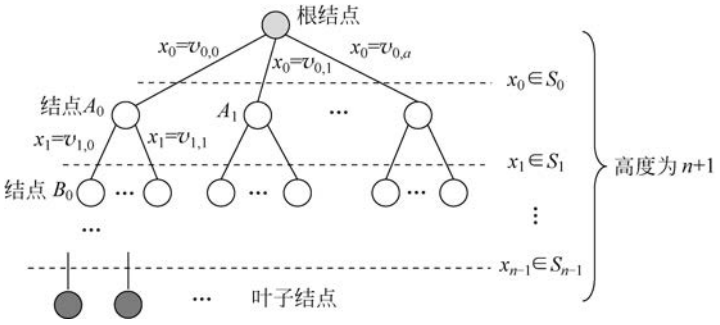

图 5.1 解空间的一般结构

从形式化角度看，解空间是 $S_0\times S_1\times\cdots\times S_{n-1}$ 的笛卡儿积，例如当 $|S_0|=|S_1|=\cdots|S_{n-1}|=2$ 时解空间是一棵高度为 $n+1$ 的满二叉树。问题的解包含在解空间中，剩下的问题就是在解空间中搜索满足问题要求的解，回溯法就是这样的一种搜索方法，但不是唯一的搜索方法。

需要注意的是，问题的解空间是虚拟的，并不需要在算法运行中真正地构造出整个树结构，然后在该解空间中搜索问题的解。实际上，有些问题的解空间因过于复杂或结点过多难以画出来。

5.1.2 什么是回溯法

先看一下求解问题的类型，通常求解问题分为两种类型，一种类型是给定一个约束函数，需要求所有满足约束条件的解，称为求所有解类型。例如鸡兔同笼问题中，所有鸡兔头数为 a、腿数为 b，求其中的鸡兔各有多少只？设鸡兔数分别为 x 和 y，则约束函数是 $x+y=a$，$2x+4y=b$，该问题需求的有解类型。另外一种类型是除了约束条件外还包含目标函数，最后是求使目标函数最大或者最小的**最优解**，称为求最优解类型。例如鸡兔同笼问题中，求所有鸡兔头数为 a、腿数为 b 并且鸡最少的解，这就是一个求最优解问题，除了前面的约束函数外还包含目标函数 $\min(x)$。

这两类问题都可以采用回溯法求解，实际上它们本质上相同，因为只有求出所有解，再按目标函数进行比较才能求出最优解。

回溯法是在解空间树中按照深度优先搜索方法从根结点出发搜索解，与树的先根遍历类似，当搜索到某个叶子结点时对应一个可能解，如果同时又满足约束条件，则该可能解是一个**可行解**。所以一个可行解就是从根结点到对应叶子结点的路径上所有分支的取值，例如一个可行解为$(a_0,a_1,\cdots,a_{n-1})$，如图 5.2 所示，在解空间中搜索到可行解的部分称为搜索空间。简单地说，回溯法采用深度优先搜索方法寻找根结点到每个叶子结点的路径，判断对应的叶子结点是否满足约束条件，如果满足该路径就构成一个解（可行解）。

回溯法在搜索解时首先让根结点成为活结点，所谓**活结点**是指自身已生成但其孩子结点没有全部生成的结点，同时也成为当前的扩展结点，所谓**扩展结点**是指正在产生孩子结点的结点。在当前扩展结点处沿着纵深方向移至一个新结点，这个新结点又成为新的活结点，并成为当前扩展结点。如果在当前扩展结点处不能再向纵深方向移动，则当前扩展结点就成为死结点，所谓**死结点**是指其所有子结点均已产生的结点，此时应往回移动（回溯）至最近的一个活结点处，并使这个活结点成为当前的扩展结点。也就是说在回溯法中从根结点开始沿着某个分支一直搜索下去，到达叶子结点后再退回搜索，直到没有活结点为止。

如图 5.3 所示，当从状态 s_i 搜索到状态 s_{i+1} 后，如果 s_{i+1} 变为死结点，则从状态 s_{i+1} 回退到 s_i，再从 s_i 找其他可能的路径，所以回溯法体现出走不通就退回再走的思路。若用回溯法求问题的所有解，需要回溯到根结点，且根结点的所有可行的子树都已被搜索完才结束。若使用回溯法求任意一个解，只要搜索到问题的一个解就可以结束。

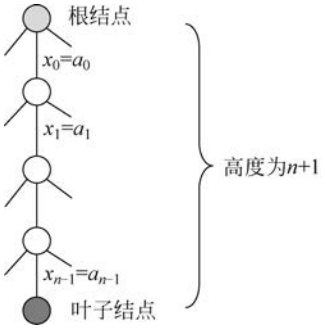

图 5.2 求解的搜索空间

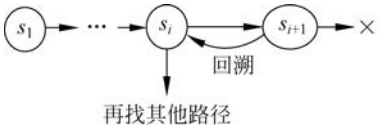

图 5.3 回溯过程

从上看出，寻找问题解的过程就是在解空间中搜索满足约束条件和目标函数的解，所以

搜索算法设计的关键点有以下3个：

① 根据问题的特性确定结点是如何扩展的，不同的问题扩展方式是不同的。例如，在有向图中搜索从顶点 s 到顶点 t 的一条路径，其扩展十分简单，就是从一个顶点找所有相邻顶点。

② 在解空间中按什么方式搜索解，实际上树的遍历主要有先根遍历和层次遍历，前者就是深度优先搜索(DFS)，后者就是广度优先搜索(BFS)。回溯法就是采用深度优先搜索解，第6章介绍的分支限界法则是采用广度优先搜索解。

③ 解空间通常十分庞大，如果要高效地找到问题的解，通常采用一些剪支的方法实现。

所谓剪支就是在解空间中搜索时提早终止某些分支的无效搜索，减少搜索的结点个数，但不影响最终结果，从而提高了算法的时间性能。常用的剪支策略如下。

① 可行性剪支：在扩展结点处剪去不满足约束条件的分支。例如，在鸡兔同笼问题中，若 $a=3,b=8$，兔数的取值范围只能是0～2，因为3只或者更多只兔子时腿数就超过8了，不再满足约束条件。

② 最优性剪支：用限界函数剪去得不到最优解的分支。例如，在求鸡最少的鸡兔同笼问题中，若已经求出一个可行解的鸡数为3，后面就不必搜索鸡数大于3的结点。

③ 交换搜索顺序：在搜索中改变搜索的顺序，比如原先是递减顺序，可以改为递增顺序，或者原先是无序，可以改为有序，这样可能减少搜索的总结点。

严格来说交换搜索顺序并不是一种剪支策略，而是一种对搜索方式的优化。前两种剪支策略采用的约束函数和限界函数统称为剪支函数。归纳起来，回溯法可以简单地理解为深度优先搜索加上剪支。因此用回溯法求解的一般步骤如下：

① 针对给定的问题确定其解空间，其中一定包含所求问题的解。

② 确定结点的扩展规则。

③ 采用深度优先搜索方法搜索解空间，并在搜索过程中尽可能采用剪支函数避免无效搜索。

【例5-1】 农夫(人)过河问题是这样的，在河东岸有一个农夫、一只狼、一只鸡和一袋谷子，只有当农夫在现场时狼不会吃鸡，鸡也不会吃谷子，否则会出现狼吃鸡或者鸡吃谷子的冲突。另有一条小船，该船只能由农夫操作，且最多只能载下农夫和另一样东西。设计一种将农夫、狼、鸡和谷子借助小船运到河西岸的过河方案。

解 在该问题中用东、西两岸的人或物品表示状态，开始状态为人和物品在东岸，西岸是空的，此时人可以带任何一个物品驾船到西岸去，这样扩展出3个状态，对于每种状态，又根据题目规则扩展出一个或多个状态，所有状态及其关系构成了本问题的解空间。

该问题的部分搜索空间如图5.4所示，图中每个方框表示一种状态，带阴影的框表示终点，带⊠的框表示有冲突，即出现狼吃鸡或鸡吃谷子的情况，带×的框表示与以前的状态重复。在解空间中采用深度优先搜索找到的一种可行的方案(可行解)如下：

① 农夫驾船带鸡从河东岸到西岸。

② 农夫驾船不带任何东西从河西岸到东岸。

③ 农夫驾船带狼从河东岸到西岸。

④ 农夫驾船带鸡从河西岸到东岸。

⑤ 农夫驾船带谷子从河东岸到西岸。

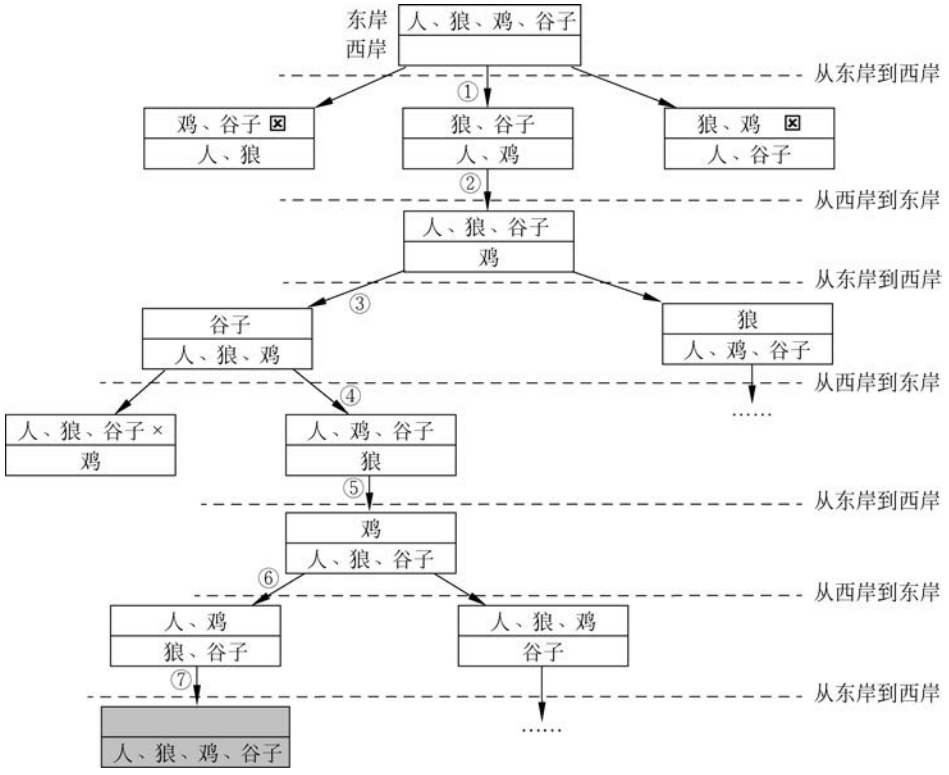

图 5.4　农夫过河的部分搜索空间

⑥ 农夫驾船不带任何东西从河西岸到东岸。

⑦ 农夫驾船带鸡从河东岸到西岸。

5.1.3　回溯法算法的框架

通常解空间有两种类型。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**，在子集树中每个结点的扩展方式是相同的，也就是说每个结点的子结点个数相同。例如在整数数组 a 中求和为目标值 target 的所有解，每个元素 $a[i]$只有选择和不选择两种方式，对应的解空间就是子集树。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**，例如求全排列问题的解空间就是典型的排列树。

由于回溯法基于深度优先搜索，而深度优先搜索特别适合采用递归实现，在递归算法中参数（非引用参数）具有自动回退（回溯）的能力，所以大多数回溯算法都采用递归实现（回溯算法也可以采用迭代实现，但递归回溯算法比对应的迭代算法设计起来更加简便）。

设问题的解是一个 n 维向量$(x_1,x_2,\cdots,x_n)$，约束函数为 constraint(i,j)，限界函数为 bound(i,j)。根据解空间类型将递归框架分为子集树和排列树两种类型。

1. 解空间为子集树

一般地，解空间为子集树的递归回溯框架如下：

```
int x[n];                                   //x 存放解向量,这里作为全局变量
void dfs(int i)                             //求解子集树的递归框架
```

```
{   if(i>n)                                    //搜索到叶子结点,输出一个可行解
        输出一个解;
    else
    {   for (j=下界;j<=上界;j++)               //用 j 表示 x[i]的所有可能候选值
        {   x[i]=j;                            //产生一个可能的解分量
            ...                                //其他操作
            if (constraint(i,j) && bound(i,j))
                dfs(i+1);                      //满足约束条件和限界函数,继续下一层
            回溯 x[i];
            ...
        }
    }
}
```

在采用上述算法框架时有以下几点注意事项：

① 如果 i 从 1 开始调用上述递归框架，此时根结点为第 1 层，叶子结点为第 $n+1$ 层。当然 i 也可以从 0 开始，这样根结点为第 0 层，叶子结点为第 n 层，所以需要将上述代码中的“if (i>n)”改为“if(i>=n)”。

② 在上述递归框架中通过 for 循环用 j 枚举 x_i 的所有可能候选值，如果扩展路径只有两条，可以改为两次递归调用(例如求解 0/1 背包问题、子集和问题等都是如此)。

③ 这里递归框架只有 i 一个参数，在实际应用中可以根据具体情况设置多个参数。

【例 5-2】 有一个含 n 个整数的数组 a，所有元素均不相同，设计一个算法求其所有子集(幂集)。例如，$a=\{1,2,3\}$，所有子集是{},{3},{2},{2,3},{1},{1,3},{1,2},{1,2,3}(输出顺序无关)。

扫一扫

视频讲解

解 本问题的解空间是典型的子集树，集合 a 中的每个元素只有两种选择，要么选取，要么不选取。设解向量为 x，$x[i]=1$ 表示选取 $a[i]$，$x[i]=0$ 表示不选取 $a[i]$。用 i 遍历数组 a，i 从 0 开始(与解空间中根结点层次为 0 相对应)，根结点为初始状态($i=0$，x 的元素均为 0)，叶子结点为目标状态($i=n$，x 为一个可行解，即一个子集)。从状态(i，x)可以扩展出两个状态：

① 选择 $a[i]$元素⇨下一个状态为($i+1$，$x[i]=1$)。

② 不选择 $a[i]$元素⇨下一个状态为($i+1$，$x[i]=0$)。

这里 i 总是递增的，所以不会出现状态重复的情况。如图 5.5 所示为求{1,2,3}幂集的解空间，每个叶子结点对应一个子集，所有子集构成幂集。

对应的递归回溯算法如下：

```
vector<int> x;                                 //解向量,全局变量
void disp(vector<int> &a)                      //输出一个解
{   printf("{");
    for (int i=0;i<x.size();i++)
        if (x[i]==1)
            printf("%d",a[i]);
    printf("}");
}
void dfs(vector<int> &a,int i)                 //递归回溯算法
{   if (i>=a.size())                           //到达一个叶子结点
        disp(a);
```

```
    else                                    //没有到达叶子结点
    {   x[i]=1;
        dfs(a,i+1);                         //选择 a[i]
        x[i]=0;
        dfs(a,i+1);                         //不选择 a[i]
    }
}
void subsets(vector<int> &a)                //求 a 的幂集
{   int n=a.size();
    x.resize(n);                            //初始化 x 的长度为 n
    dfs(a,0);
}
```

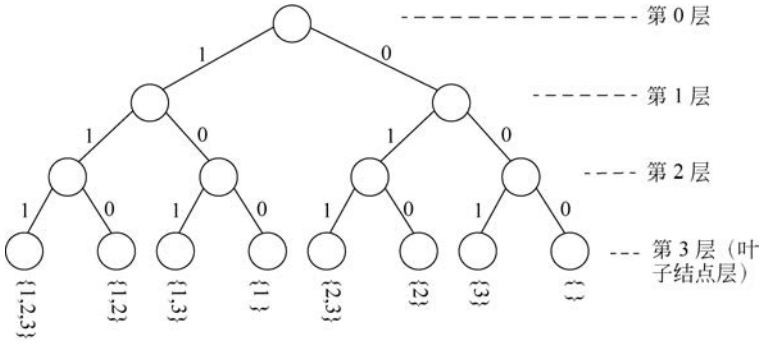

图 5.5　求 $a=\{1,2,3\}$ 幂集的解空间

扫一扫

视频讲解

【例 5-3】　设计一个算法在 1,2,3,4,5(顺序不能变)任意两个数字之间插入'+'或者'-'运算符,使得表达式的计算结果为 5,并输出所有可能的表达式。

解　用数组 $a[0..N-1](N=5)$存放 1～5 的整数,用字符数组 x 存放插入的运算符(解向量),其中 $x[i]$表示在 $a[i]$前面插入的运算符(i 从 1 开始),$x[i]$只能取值'+'或者'-'(两选一)。采用回溯法产生和为 5 的表达式,解空间的高度是 $N+1$,根结点的层次为 0。

设计函数 dfs(a,sum,x,i),其中 sum 表示到达整数 $a[i]$时计算出的表达式和(初始时置 sum=$a[0]$),i 从 1(因为 $a[0]$之前没有运算符)开始,当到达叶子结点($i=N$)时,如果 sum=5,得到一个解。对应的递归算法如下:

```
#define N 5
void dfs(vector<int> &a,int sum,vector<char> &x,int i)
{   if (i==N)                               //到达一个叶子结点(可能解)
    {   if (sum==5)                         //找到一个可行解
        {   printf(" %d",a[0]);             //输出一个解
            for (int j=1;j<N;j++)
            {   printf("%c",x[j]);
                printf("%d",a[j]);
            }
            printf("=5\n");
        }
    }
    else
    {   x[i]='+';                           //在位置 i 插入'+'
```

```
            sum+=a[i];                    //计算结果
            dfs(a,sum,x,i+1);             //继续搜索
            sum-=a[i];                    //回溯
            x[i]='-';                     //在位置i插入'-'
            sum-=a[i];                    //计算结果
            dfs(a,sum,x,i+1);             //继续搜索
            sum+=a[i];                    //回溯
        }
    }
    void solve(vector<int> &a)            //求解算法
    {   vector<char> x(a.size());         //定义解向量
        dfs(a,a[0],x,1);
    }
```

上述算法的求解结果如下：

```
1+2+3+4-5=5
1-2-3+4+5=5
```

说明：上述算法与例 5-2 的算法相比，将解向量设计为 dfs 的引用参数而不是全局变量，实际上由于算法中包含 x 的回溯，所以设计为引用参数或者全局变量都是正确的。

2. 解空间为排列树

解空间为排列树的递归框架是以求全排列为基础的，下面先通过一个示例讨论一种不同于第 3 章求全排列的递归算法。

【例 5-4】 有一个含 n 个整数的数组 a，所有元素均不相同，求其所有元素的全排列。例如，$a=\{1,2,3\}$，得到的结果是(1,2,3)，(1,3,2)，(2,3,1)，(2,1,3)，(3,1,2)，(3,2,1)。

扫一扫

视频讲解

解 用数组 a 存放初始数组 a 的一个排列，采用递归法求解。设 $f(a,n,i)$ 表示求 $a[i..n-1]$(共 $n-i$ 个元素)的全排列，为大问题，$f(a,n,i+1)$ 表示求 $a[i+1..n-1]$(共 $n-i-1$ 个元素)的全排列，为小问题，如图 5.6 所示。

显然 i 越小求全排列的元素个数越多，当 $i=0$ 时求 $a[0..n-1]$的全排列。当 $i=n-1$ 时求 $a[n-1..n-1]$的全排列，此时序列只有一个元素(单个元素的全排序就是该元素)，再合并 $a[0..n-2]$($n-1$ 个元素的排列)就得到 n 个元素的一个排列。当 $i=n$ 时求 $a[n..n-1]$的全排列，此时序列为空，说明 $a[0..n-1]$是一个排列，后面两种情况均可以作为递归出口。所以求 a 中全排列的过程是 $f(a,n,0)\rightarrow f(a,n,1)\rightarrow f(a,n,2)\rightarrow\cdots\rightarrow f(a,n,n-1)$。

那么如何由小问题 $f(a,n,i+1)$求大问题 $f(a,n,i)$呢？假设 $f(a,n,i+1)$求出了 $a[i+1..n-1]$的全排列，考虑 a_i 位置，该位置可以取 $a[i..n-1]$中的任何一个元素，但是排列中元素不能重复，为此采用交换方式，即 $j=i$ 到 $n-1$ 循环，每次循环将 $a[i]$与 $a[j]$交换，合并子问题解得到一个大问题的排列，再恢复成循环之前的顺序，即将 $a[i]$与 $a[j]$再次交换，然后进入下一次求其他大问题的排列。注意，如果不做再次交换会出现重复的排列情况，例如 $a=\{1,2,3\}$，结果为(1,2,3)，(1,3,2)，(3,1,2)，(3,2,1)，(1,2,3)，(1,3,2)，显然是错误的。

归纳起来，求 a 的全排列的递归模型 $f(a,n,i)$如下：

$$
\begin{aligned}
&f(a,n,i) \equiv \text{输出产生的解} && \text{当 } i=n-1 \text{ 时}\\
&f(a,n,i) \equiv \text{对于 } j=i\sim n-1: && \text{其他}\\
&\qquad\quad a[i]\text{与 } a[j]\text{交换位置;}
\end{aligned}
$$

$f(a,n,i+1)$;
将 $a[i]$ 与 $a[j]$ 交换位置(回溯)

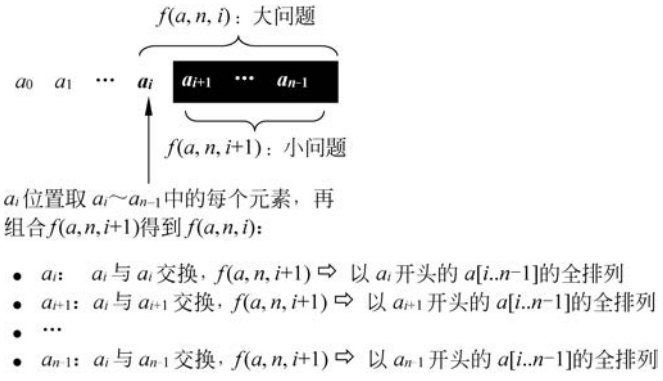

图 5.6　求 $f(a,n,i)$的过程

例如 $a=\{1,2,3\}$时，求全排列的解空间如图 5.7 所示，数组 a 的下标从 0 开始，所以根结点“$a=\{1,2,3\}$”的层次为 0，它的子树分别对应 $a[0]$位置选择 $a[0]$、$a[1]$和 $a[2]$元素。实际上对于第 i 层的结点，其子树分别对应 $a[i]$位置选择 $a[i]$、$a[i+1]$、……、$a[n-1]$元素。树的高度为 $n+1$，叶子结点的层次是 n。

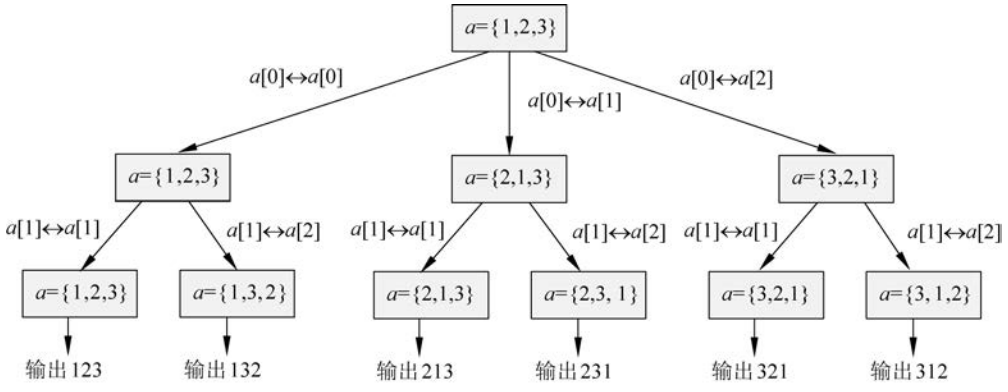

图 5.7　求 $a=\{1,2,3\}$全排列的解空间(1)

解空间树更清晰的描述如图 5.8 所示，对于第 i 层的结点，其扩展仅考虑 $a[i]$及后面的元素，而不必考虑前面已经选择的元素。例如第 2 层的“1,3,2”结点，前面的“1,3”不必考虑，仅扩展 $a[2]$，此时 $a[2]$只能取从根结点到该结点的路径上没有取过的值 2，从而得到“1,3,2”的一个排序。

对应的递归算法如下：

```
int cnt=0;                                  //累计排列的个数
void disp(vector<int> &a)                   //输出一个解
{   printf(" %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=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
        }
    }
}
void perm(vector<int> & a)                         //求 a 的全排列
{
    dfs(a,0);
}
```

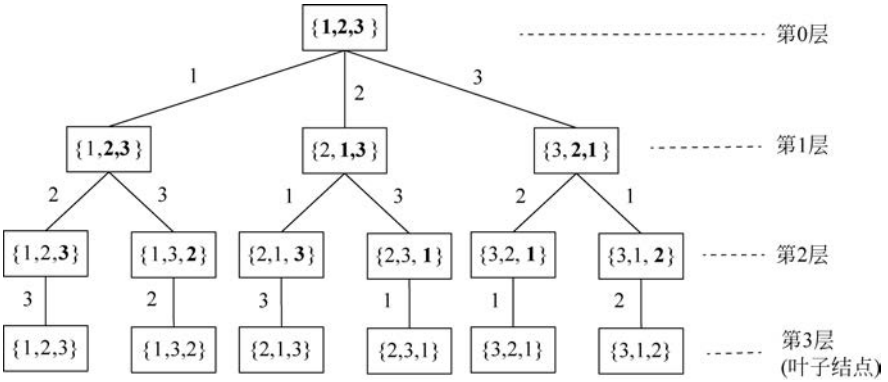

图 5.8 求 $a=\{1,2,3\}$全排列的解空间(2)

现在证明算法的正确性。实际上在递归算法中求值顺序与递推顺序相反,求 a 的全排列是从 $f(a,n,0)$开始的,求值顺序是 $f(a,n,n-1)\rightarrow f(a,n,n-2)\rightarrow\cdots\rightarrow f(a,n,1)\rightarrow f(a,n,0)$。循环不变量是 $f(a,n,i)$用于求 $a[i..n-1]$的全排列,证明如下。

初始化:在循环的第一轮迭代开始之前,即 $i=n-1$ 表示求 $a[n-1..n-1]$的全排列,而一个元素的全排列就是该元素,显然是正确的。

保持:若前面 $f(a,n,i+1)$正确,表示求出了 $a[i+1..n-1]$的全排列,将 $a[i]$与 $a[i..n-1]$中的每个元素交换,合并 $a[i+1..n-1]$的一个排列得到 $f(a,n,i)$的一个排列,再恢复后继续做完,从而得到 $f(a,n,i)$的全排列。

终止:当求值结束时 $i=0$,得到 $f(a,n,0)$即 a 的全排列。

从上述求 a 的全排列的示例可以归纳出解空间为排列树的递归回溯框架如下:

```
int x[n];                                          //x 存放解向量,并初始化
void dfs(int i)                                    //求解排列树的递归框架
{   if(i>n)                                        //搜索到叶子结点,输出一个可行解
        输出结果;
    else
    {   for (j=i;j<=n;j++)                         //用 j 枚举 x[i]的所有可能候选值
```

```
        {   ...                          //第 i 层的结点选择 x[j]的操作
            swap(x[i],x[j]);             //为保证排列中每个元素不同,通过交换来实现
            if (constraint(i,j) && bound(i,j))
                dfs(i+1);                //满足约束条件和限界函数,进入下一层
            swap(x[i],x[j]);             //恢复状态: 回溯
            ...                          //第 i 层的结点选择 x[j]的恢复操作
        }
    }
}
```

如何进一步理解上述算法呢？假设解向量为$(x_0,x_1,\cdots,x_i,\cdots,x_j,\cdots,x_{n-1})$，当从解空间的根结点出发搜索到达第 i 层的某个结点时，对应的部分解向量为$(x_0,x_1,\cdots,x_{i-1})$，其中每个分量已经取好值了，现在为该结点的分支选择一个 x_i 值（每个不同的取值对应一个分支，x_i 有 $n-i$ 个分支），前一个 swap($x[i]$,$x[j]$)表示为 x_i 取 x_j 值，后一个 swap($x[i]$,$x[j]$)用于状态恢复，这一点是利用排列树的递归回溯框架求解实际问题的关键。另外几点需要注意的说明事项与解空间为子集树的递归回溯框架相同。

5.1.4 回溯法算法的时间分析

通常以回溯法的解空间中的结点个数作为算法的时间分析依据。假设解空间树共有 $n+1$ 层（根结点为第 0 层，叶子结点为第 n 层），第 1 层有 m_0 个结点，每个结点有 m_1 个子结点，则第 2 层有 m_0m_1 个结点，同理，第 3 层有 $m_0m_1m_2$ 个结点，以此类推，第 n 层有 $m_0m_1\cdots m_{n-1}$ 个结点，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，在子集树中有 $m_0=m_1=\cdots=m_{n-1}=c$，对应算法的时间复杂度为 $O(c^n)$，在排列树中有 $m_0=n$，$m_1=n-1$，$\cdots$，$m_{n-1}=1$，对应算法的时间复杂度为 $O(n!)$。

这是一种最坏情况下的时间分析方法，在实际中可以通过剪支提高性能。为了估算得更精确，可以选取若干条不同的随机路径，分别对各随机路径估算结点总数，然后再取这些结点总数的平均值。在通常情况下，回溯法的效率高于穷举法。

5.2 基于子集树框架的问题求解

扫一扫

视频讲解

5.2.1 子集和问题

1. 问题描述

给定 n 个不同的正整数集合 $a=(a_0,a_1,\cdots,a_{n-1})$和一个正整数 t，要求找出 a 的子集 s，使该子集中所有元素的和为 t。例如，当 $n=4$ 时，$a=(3,1,5,2)$，$t=8$，则满足要求的子集 s 为(3,5)和(1,5,2)。

2. 问题求解

与求幂集问题一样，该问题的解空间是一棵子集树（因为每个整数要么选择要么不选择），并且是求满足约束函数的所有解。

1）无剪支

设解向量 $\boldsymbol{x}=(x_0,x_1,\cdots,x_{n-1})$，$x_i=1$ 表示选择 a_i 元素，$x_i=1$ 表示不选择 a_i 元素。在解空间中按深度优先方式搜索所有结点，并用 cs 累计当前结点之前已经选择的所有整数和，一旦到达叶子结点（即 $i\geqslant n$），表示 a 的所有元素处理完毕，如果相应的子集和为 t（即约束函数 cs$=t$ 成立），则根据解向量 $\boldsymbol{x}$ 输出一个解。当解空间搜索完后便得到所有解。

例如 $a=(3,1,5,2)$，$t=8$，其解空间如图 5.9 所示，图中结点上的数字表示 cs，利用深度优先搜索得到两个解，解向量分别是(1,0,1,0)和(0,1,1,1)，对应图中两个带阴影的叶子结点，图中共有 31 个结点，每个结点都要搜索。实际上，解空间是一棵高度为 5 的满二叉树，从根结点到每个叶子结点都有一条路径，每条路径就是一个决策向量，满足约束函数的决策向量就是一个解向量。

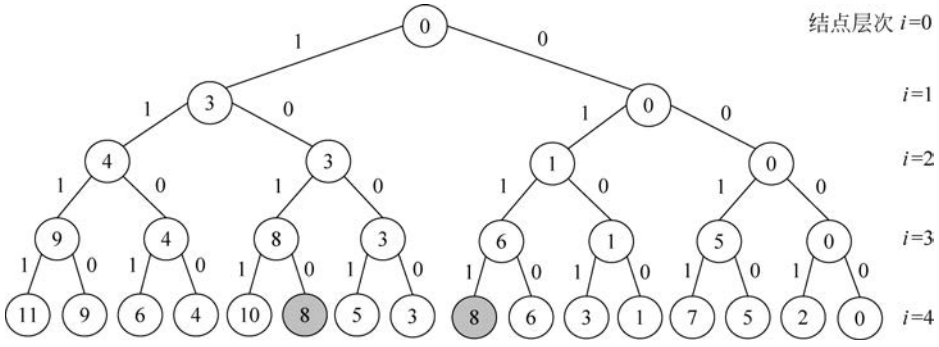

图 5.9 求 $a=(3,1,5,2)$，$t=8$ 的子集和的解空间

对应的递归算法如下：

```
int n=4,t=8;                                    //一个测试实例
vector<int> a={3,1,5,2};                        //存放所有整数
int cnt=0;                                      //累计解个数
void disp(vector<int> &x)                       //输出一个解
{   printf("   第%d个解   ",++cnt);
    printf("选取的数为");
    for (int i=0;i<n;i++)
        if (x[i]==1)
            printf("%d ",a[i]);
    printf("\n");
}
void dfs(int cs,vector<int> &x,int i)           //递归算法
{   if (i>=n)                                   //到达一个叶子结点
    {   if (cs==t)                              //找到一个满足条件的解,输出
            disp(x);
    }
    else                                        //没有到达叶子结点
    {   x[i]=1;                                 //选取整数 a[i]
        dfs(cs+a[i],x,i+1);
        x[i]=0;                                 //不选取整数 a[i]
        dfs(cs,x,i+1);
    }
}
void subs1()                                    //求解子集和问题
```

```
{    vector<int> x(n);                    //定义解向量
     dfs(0,x,0);                          //i 从 0 开始
}
```

上述算法的求解结果如下：

```
第 1 个解   选取的数为 3 5
第 2 个解   选取的数为 1 5 2
```

【算法分析】 上述算法的解空间是一棵高度为 $n+1$ 的满二叉树，共有 $2^{n+1}-1$ 个结点，递归调用 $2^{n+1}-1$ 次，每找到一个满足条件的解就调用 disp()输出，执行 disp()的时间为 $O(n)$，所以算法的时间复杂度为 $O(n\times 2^n)$。

2）左剪支

由于 a 中所有元素是正整数，每次选择一个元素时 cs 都会变大，当 cs>t 时沿着该路径继续找下去一定不可能得到解。利用这个特点减少搜索的结点个数。当搜索到第 i（$0\leqslant i<n$）层的某个结点时，cs 表示当前已经选取的整数和（其中不包含 $a[i]$），判断选择 $a[i]$ 是否合适：

① 若 cs+$a[i]>t$，表示选择 $a[i]$ 后子集和超过 t，不必继续沿着该路径求解，终止该路径的搜索，也就是左剪支。

② 若 cs+$a[i]\leqslant t$，沿着该路径继续下去可能会找到解，不能终止。

简单地说，仅扩展满足 cs+$a[i]\leqslant t$ 的左孩子结点。

例如 $a=(3,1,5,2)$，$t=8$，其搜索空间如图 5.10 所示，图 5.10 中共有 29 个结点，除去两个被剪支的结点（用虚框结点表示），剩下 27 个结点，也就是说递归调用 27 次，性能得到了提高。

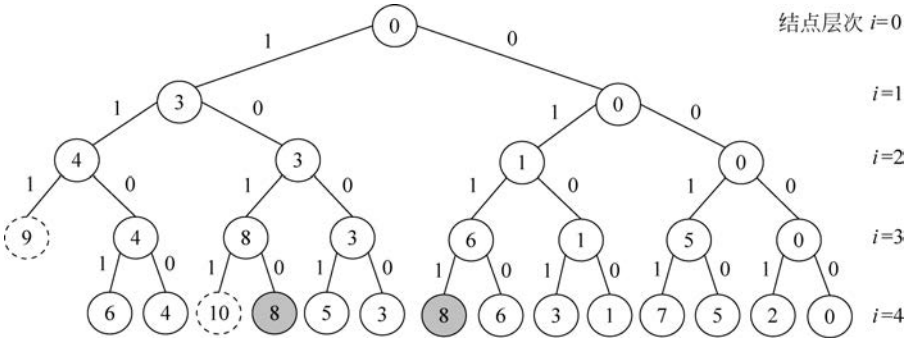

图 5.10 求 $a=(3,1,5,2)$，$t=8$ 的子集和的搜索空间(1)

对应的递归算法如下：

```
void dfs(int cs,vector<int> &x,int i)          //递归算法
{    if (i>=n)                                 //找到一个叶子结点
     {    if (cs==t)                           //找到一个满足条件的解,输出
               disp(x);
     }
     else                                      //没有到达叶子结点
     {    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],x,i+1);
```

```
        }
        x[i]=0;                          //不选取整数 a[i]
        dfs(cs,x,i+1);
    }
}
void subs2()                             //求解子集和问题
{   vector<int> x(n);                    //定义解向量
    dfs(0,x,0);                          //i 从 0 开始
}
```

3）右剪支

左剪支仅考虑是否扩展左孩子结点，可以进一步考虑是否扩展右孩子结点。当搜索到第 $i(0\leqslant i<n)$ 层的某个结点时，用 rs 表示余下的整数和，即 $rs=a[i]+\cdots+a[n-1]$（其中包含 $a[i]$），因为右孩子结点对应不选择整数 $a[i]$ 的情况，如果不选择 $a[i]$，此时剩余的所有整数和为 $rs=rs-a[i](a[i+1]+\cdots+a[n-1])$，若 $cs+rs<t$ 成立，说明即便选择所有剩余整数，其和都不可能达到 t，所以右剪支就是仅扩展满足 $cs+rs\geqslant t$ 的右孩子结点，注意在左、右分支处理完后需要恢复 rs，即执行 $rs=+a[i]$。

例如 $a=(3,1,5,2)$，$t=8$，其搜索过程如图 5.11 所示，图中共有 17 个结点，除去 7 个被剪支的结点（用虚框结点表示），剩下 10 个结点，也就是说递归调用 10 次，性能得到更有效的提高。

说明： 本例给定 a 中所有整数为正整数，如果 a 中有负整数，这样的左、右剪支是不成立的，因此无法剪支，算法退化为基本深度优先搜索。

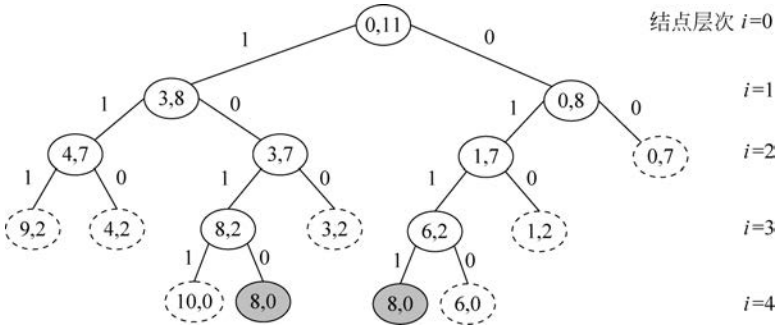

图 5.11 求 $a=(3,1,5,2)$，$t=8$ 的子集和的搜索空间(2)

求解子集和问题的递归算法如下：

```
void dfs(int cs,int rs,vector<int> &x,int i)          //递归算法
{                          //cs 为考虑整数 a[i]时选取的整数和,rs 为剩余整数和
    if (i>=n)                                         //找到一个叶子结点
    {   if (cs==t)                                    //找到一个满足条件的解,输出
            disp(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;                                   //选取第 i 个整数 a[i]
            dfs(cs+a[i],rs,x,i+1);
        }
```

```
            if (cs+rs>=t)                    //右孩子结点剪支
            {   x[i]=0;                      //不选取第 i 个整数 a[i]
                dfs(cs,rs,x,i+1);
            }
            rs+=a[i];                        //恢复剩余整数和(回溯)
        }
    }
    void subs3()                             //求解子集和问题
    {   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];
        dfs(0,rs,x,0);                       //i 从 0 开始
    }
```

【算法分析】 尽管通过剪支提高了算法的性能，但究竟剪去了多少结点与具体的实例数据相关，所以说上述算法最坏情况下的时间复杂度仍然为 $O(n\times 2^n)$。从上述实例中可以看出剪支在回溯算法中的重要性。

扫一扫

视频讲解

5.2.2 简单装载问题

1. 问题描述

有 n 个集装箱要装上一艘载重量为 t 的轮船，其中集装箱 $i(0\leqslant i\leqslant n-1)$ 的重量为 w_i。不考虑集装箱的体积限制，现要选出重量和小于或等于 t 并且尽可能重的若干集装箱装上轮船。例如，$n=5$，$t=10$，$w=\{5,2,6,4,3\}$ 时，其最佳装载方案有两种，即(1,1,0,0,1)和(0,0,1,1,0)，对应集装箱重量和达到最大值 t。

2. 问题求解

同样与求幂集问题一样，该问题的解空间树是一棵子集树(因为每个集装箱要么选择要么不选择)，但要求最佳装载方案，属于求最优解类型。设当前解向量 $\boldsymbol{x}=(x_0,x_1,\cdots,x_{n-1})$，$x_i=1$ 表示选择集装箱 i，$x_i=1$ 表示不选择集装箱 i，最优解向量用 bestx 表示，最优重量和用 bestw 表示(初始为 0)，为了简洁，将 bestx 和 bestw 设计为全局变量。

当搜索到第 $i(0\leqslant i<n)$ 层的某个结点时，cw 表示当前选择的集装箱的重量和(其中不包含 $w[i]$)，rw 表示余下集装箱的重量和，即 rw=$w[i]+\cdots+w[n-1]$(其中包含 $w[i]$)，此时处理集装箱 i，先从 rw 中减去 $w[i]$，即置 rw−=$w[i]$，采用的剪支函数如下。

① 左剪支：判断选择集装箱 i 是否合适。检查当前集装箱被选中后总重量是否超过 t，若是则剪支，即仅扩展满足 cw+$w[i]\leqslant t$ 的左孩子结点。

② 右剪支：判断不选择集装箱 i 是否合适。如果不选择集装箱 i，此时剩余的所有整数和为 rw，若 cw+rw≤bestw 成立(bestw 是当前找到的最优解的重量和)，说明即便选择所有剩余集装箱，其重量和都不可能达到 bestw，所以仅扩展满足 cw+rw>bestw 的右孩子结点。

说明：由于深度优先搜索是纵向搜索的，可以比广度优先搜索更快地找到一个解，以此作为 bestw 进行右剪支是非常合适的。

当第 i 层的这个结点扩展完成后需要恢复 rs，即置 rs+=$a[i]$(回溯)。如果搜索到某个叶子结点(即 $i\geqslant n$)，得到一个可行解，其选择的集装箱重量和为 cw(由于左剪支的原因，

cw 一定小于或等于 t），若 cw＞bestw，说明找到一个满足条件的更优解，置 bestw＝cw，bestx＝x。全部搜索完毕后，bestx 就是最优解向量。

对应的递归算法如下：

```
int n=5,t=10;                                  //一个测试用例
int w[]={5,2,6,4,3};                           //各集装箱重量,不用下标为 0 的元素
vector<int> bestx;                             //存放最优解向量
int bestw=0;                                   //存放最优解的总重量,初始化为 0
void dfs(int cw,int rw,vector<int> &x,int i)   //递归算法
{   if (i>=n)                                  //找到一个叶子结点
    {   if (cw>bestw)                          //找到一个满足条件的更优解
        {   bestw=cw;                          //保存更优解
            bestx=x;
        }
    }
    else                                       //没有到达叶子结点
    {   rw-=w[i];                              //求剩余集装箱的重量和
        if (cw+w[i]<=t)                        //左孩子结点剪支
        {   x[i]=1;                            //选取集装箱 i
            cw+=w[i];                          //累计当前所选集装箱的重量和
            dfs(cw,rw,x,i+1);
            cw-=w[i];                          //恢复当前所选集装箱的重量和(回溯)
        }
        if (cw+rw>bestw)                       //右孩子结点剪支
        {   x[i]=0;                            //不选择集装箱 i
            dfs(cw,rw,x,i+1);
        }
        rw+=w[i];                              //恢复剩余集装箱的重量和(回溯)
    }
}
void disp()                                    //输出最优解
{   for (int i=0;i<n;i++)
        if (bestx[i]==1)
            printf(" 选取第%d 个集装箱\n",i);
    printf(" 总重量=%d\n",bestw);
}
void loading()                                 //求解简单装载问题
{   bestx.resize(n);
    vector<int> x(n);
    int rw=0;
    for (int i=0;i<n;i++)
        rw+=w[i];
    dfs(0,rw,x,0);
}
```

上述算法的求解结果如下。实际上还有另外一个最优解，即选择第 2 个和第 3 个集装箱，它们的重量和是相同的。

```
选取第 0 个集装箱
选取第 1 个集装箱
选取第 4 个集装箱
总重量=10
```

说明： 在上述 dfs 算法的左结点扩展中，cw+=w[i]、dfs(cw,rw,x,i+1)和 cw-=

w[i]3 条语句可以用一条语句(即 dfs(cw+w[i],rw,x,i+1))等价地替换。

【算法分析】 该算法的解空间树中有 $2^{n+1}-1$ 个结点，每找到一个更优解时复制到 bestx 的时间为 $O(n)$，所以最坏情况下算法的时间复杂度为 $O(n\times 2^n)$。前面的实例中，$n=5$，解空间树中结点个数应为 63，采用剪支后结点个数为 16(不计带×的被剪支的结点)，如图 5.12 所示。

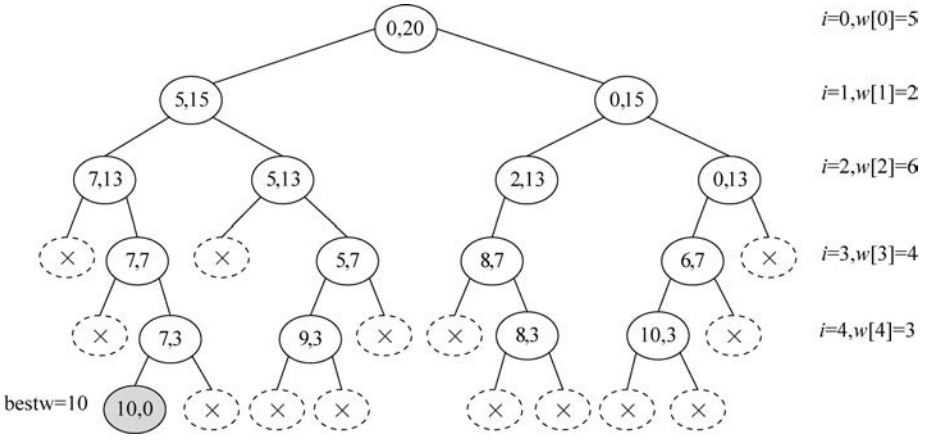

图 5.12 装载实例的搜索空间

扫一扫

视频讲解

5.2.3 0/1 背包问题

1. 问题描述

有 n 个编号为 0～$n-1$ 的物品，重量为 $w=\{w_0,w_1,\cdots,w_{n-1}\}$，价值为 $v=\{v_0,v_1,\cdots,v_{n-1}\}$，给定一个容量为 W 的背包。从这些物品中选取全部或者部分物品装入该背包中，每个物品要么选中要么不选中，即物品不能被分割，找到选中物品不仅能够放到背包中而且价值最大的方案，并对表 5.1 所示的 4 个物品求出 $W=6$ 时的一个最优解。

表 5.1 4 个物品的信息

物品编号	重量	价值	物品编号	重量	价值
0	5	4	2	2	3
1	3	4	3	1	1

2. 问题求解

该问题的解空间树是一棵子集树(因为每个物品要么选择要么不选择)，要求求价值最大的装入方案，属于求最优解类型。

1) 存储结构设计

每个物品包含编号、重量和价值，为此采用结构体数组存放所有物品，后面涉及按单位重量价值递减排序，所以设计物品结构体类型如下：

```
struct Goods                    //物品结构体类型
{   int no;                     //物品的编号
    int w;                      //物品的重量
    int v;                      //物品的价值
```

```
    Goods(int no1,int w1,int v1)                    //构造函数
    {   no=no1;
        w=w1;
        v=v1;
    }
    bool operator<(const Goods&s) const             //用于按 v/w 递减排序
    {
        return (double)v/w>(double)s.v/s.w;
    }
};
```

例如，表 5.1 所示的 4 个物品采用向量 $\boldsymbol{g}$ 存放：

```
vector<Goods> g={Goods(0,5,4),Goods(1,3,4),Goods(2,2,3),Goods(3,1,1)};   //一个测试实例
```

设当前解向量 $\boldsymbol{x}=(x_0,x_1,\cdots,x_{n-1})$，$x_i=1$ 表示选择物品 i，$x_i=1$ 表示不选择物品 i，最优解向量用 bestx 表示，最大价值用 bestv 表示(初始为 0)，为了简洁，将 n、W、bestx 和 bestv 均设计为全局变量。

2) 左剪支

由于所有物品重量为正数，采用与子集和问题类似的左剪支。当搜索到第 $i(0\leqslant i<n)$ 层的某个结点时，cw 表示当前选择的物品重量和(其中不包含 $w[i]$)。检查当前物品被选中后总重量是否超过 W，若超过则剪支，即仅扩展满足 cw+$w[i]\leqslant W$ 的左孩子结点。

3) 右剪支

这里右剪支相对复杂一些，题目求的是价值最大的装入方案，显然优先选择单位重量价值大的物品，为此将 g 中所有物品按单位重量价值递减排序，例如表 5.1 中物品排序后的结果如表 5.2 所示，序号 i 发生了改变，后面改为按 i 而不是按物品编号 no 的顺序依次搜索。

表 5.2 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	3	1.5
1	1	3	4	1.3
2	3	1	1	1
3	0	5	4	0.8

先看这样的问题，对于第 i 层的某个结点，cw 表示当前选择的物品重量和(其中不包含 $w[i]$)，cv 表示当前选择的物品价值和(其中不包含 $v[i]$)，那么继续搜索下去能够得到的最大价值是多少？由于所有物品已按单位重量价值递减排序，显然在背包容量允许的前提下应该依次连续地选择物品 i、物品 $i+1$、……，这样做直到物品 k 装不进背包，假设再将物品 k 的一部分装进背包直到背包装满，此时一定会得到最大价值。从中看出从物品 i 开始选择的物品价值和的最大值为 $r(i)$，其中有：

$$r(i)=\sum_{j=i}^{k-1}v_j+\left(\mathrm{rw}-\sum_{j=i}^{k-1}w_j\right)(v_k/w_k)$$

再回过来讨论右剪支，右剪支是判断不选择物品 i 时是否能够找到更优解。如果不选择物品 i，按上述讨论可知在背包容量允许的前提下依次选择物品 $i+1$、物品 $i+2$、……可

以得到最大价值，且从物品 $i+1$ 开始选择的物品价值和的最大值为 $r(i+1)$。如果之前已经求出一个最优解 bestv，当 cv+$r(i+1)$≤bestv 时说明不选择物品 i 时后面无论如何也不能够找到更优解。设计如下限界函数 bound(cw,cv,i)：

```
double bound(int cw,int cv,int i)          //计算第 i 层结点的上界函数值
{   int rw=W-cw;                           //背包的剩余容量
    double b=cv;                           //表示物品价值的上界值
    int j=i;
    while (j<n && g[j].w<=rw)
    {   rw-=g[j].w;                        //选择物品 j
        b+=g[j].v;                         //累计价值
        j++;
    }
    if (j<n)                               //最后物品(此时的 j 就是 r(i)公式中的 k)只能部分装入
        b+=(double)g[j].v/g[j].w*rw;
    return b;
}
```

这样当搜索到第 i 层的某个结点时，右剪支就是仅扩展满足 bound(cw,cv,$i+1$)＞bestv 的右孩子结点。

例如，对于根结点，cw＝0，cv＝0，若不选择物品 0(对应根结点的右孩子结点)，剩余背包容量 rw＝W＝6，b＝cv＝0，考虑物品 1，$g[1].w$＜rw，可以装入，$b=b+g[1].v=4$，rw＝rw－$g[1].w=3$；考虑物品 2，$g[2].w$＜rw，可以装入，$b=b+g[2].v=5$，rw＝rw－$g[2].w=2$；考虑物品 3，g[3].w＞rw，只能部分装入，$b=b+$rw×($g[3].v/g[3].w$)＝6.6。

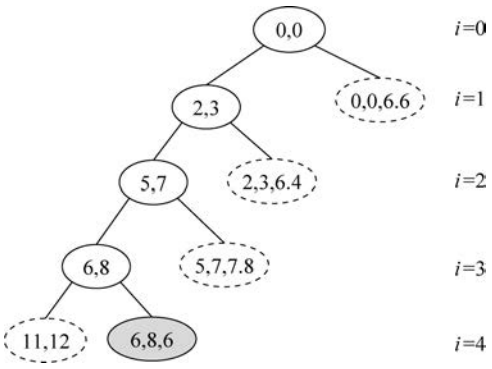

图 5.13　0/1 背包问题实例的搜索空间

右剪支是求出第 i 层的结点的 b＝bound(cw,cv,i)，若 b≤bestv，则停止右分支的搜索，也就是说仅扩展满足 b＞bestv 的右孩子结点。

对于表 5.1 所示的实例，n＝4，按 v/w 递减排序后为表 5.2，初始时 bestv＝0，求解过程如图 5.13 所示，图 5.13 中两个数字的结点为(cw,cv)，只有右结点标记为(cw,cv,ub)，为“×”的虚结点表示被剪支的结点，带阴影的结点是最优解结点，其求解结果与回溯法的完全相同，图中结点的数字为(cw,cv)，求解步骤如下：

① i＝0，根结点为(0,0)，cw＝0，cv＝0，cw＋$w[0]$≤W 成立，扩展左孩子结点，cw＝cw＋$w[0]$＝2，cv＝cv＋$v[0]$＝3，对应结点(2,3)。

② i＝1，当前结点为(2,3)，cw＋$w[1]$(5)≤W 成立，扩展左孩子结点，cw＝cw＋$w[1]$＝5，cv＝cv＋$v[1]$＝7，对应结点(5,7)。

③ i＝2，当前结点为(5,7)，cw＋$w[2]$(6)≤W 成立，扩展左孩子结点，cw＝cw＋$w[2]$＝6，cv＝cv＋$v[1]$＝7，对应结点(6,8)。

④ i＝3，当前结点为(6,8)，cw＋$w[2]$(6)≤W 不成立，不扩展左孩子结点。

⑤ i＝3，当前结点为(6,8)，不选择物品 3 时计算出 b＝cv＋0＝8，而 b＞bestv(0)成立，扩展右孩子结点。

⑥ $i=4$，当前结点为(6,8)，由于 $i \geqslant n$ 成立，它是一个叶子结点，对应一个解 bestv=8。

⑦ 回溯到 $i=2$ 层次，当前结点为(5,7)，不选择物品 2 时计算出 $b=7.8$，$b>$bestv 不成立，不扩展右孩子结点。

⑧ 回溯到 $i=1$ 层次，当前结点为(2,3)，不选择物品 1 时计算出 $b=6.4$，$b>$bestv 不成立，不扩展右孩子结点。

⑨ 回溯到 $i=0$ 层次，当前结点为(0,0)，不选择物品 0 时计算出 $b=6.6$，$b>$bestv 不成立，不扩展右孩子结点。

解空间搜索完，最优解为 bestv=8，装入方案是选择编号为 2、1、3 的 3 个物品。从中看出如果不剪支搜索的结点个数为 31，剪支后搜索的结点个数为 5。

对应的递归算法如下：

```
void dfs(int cw,int cv,vector<int> &x,int i)          //回溯算法
{   if (i>=n)                                         //找到一个叶子结点
    {   if (cw<=W && cv>bestv)                        //找到一个满足条件的更优解,保存它
        {   bestv=cv;
            bestx=x;
        }
    }
    else                                              //没有到达叶子结点
    {   if(cw+g[i].w<=W)                              //左剪支
        {   x[i]=1;                                   //选取物品 i
            dfs(cw+g[i].w,cv+g[i].v,x,i+1);
        }
        double b=bound(cw,cv,i+1);                    //计算上界时从物品 i+1 开始
        if(b>bestv)                                   //右剪支
        {   x[i]=0;                                   //不选取物品 i
            dfs(cw,cv,x,i+1);
        }
    }
}
void knap()                                           //求 0/1 背包问题
{   bestx.resize(n);
    vector<int> x(n);
    sort(g.begin(),g.end());                          //按 v/w 递减排序
    dfs(0,0,x,0);                                     //i 从 0 开始
}
```

【算法分析】 上述算法在不考虑剪支时解空间树中有 $2^{n+1}-1$ 个结点，求上界函数值和保存最优解的时间为 $O(n)$，所以最坏情况下算法的时间复杂度为 $O(n\times 2^n)$。

扫一扫

视频讲解

5.2.4 n 皇后问题

1. 问题描述

在 $n\times n(n\geqslant 4)$的方格棋盘上放置 n 个皇后，并且每个皇后不同行、不同列、不同左右对角线(否则称为有冲突)。如图 5.14 所示为 6 皇后问题的一个解。要求给出 n 个皇后的全部解。

2. 问题求解

本问题的解空间是一棵子集树(每个皇后在 1～n 列中找到一个适合的列号，即 n 选一)，并且要求所有解。采用整数数组 $q[N]$存放 n 皇后问题的求解结果，因为每行只能放

一个皇后，$q[i]$(1≤i≤n)的值表示第 i 个皇后所在的列号，即第 i 个皇后放在(i,$q[i]$)的位置上。对于图 5.14 的解，$q[1..6]$={2,4,6,1,3,5}(为了简便，不使用 $q[0]$元素)。

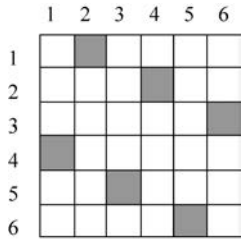

图 5.14　6 皇后问题的一个解

若在(i,j)位置上放第 i 个皇后，是否与已放好的 $i-1$ 个皇后(k,$q[k]$)(1≤k≤$i-1$)有冲突？显然它们是不同行的(因为皇后的行号 i 总是递增的)，所以不必考虑行冲突，是否存在列冲突和对角线冲突的判断如下：

① 如果(i,j)位置与前面的某个皇后同列，则有 $q[k]==j$ 成立。

② 如果(i,j)位置与前面的某个皇后同对角线，如图 5.15 所示，则恰好构成一个等腰直角三角形，即有 $|q[k]-j|==|i-k|$ 成立。

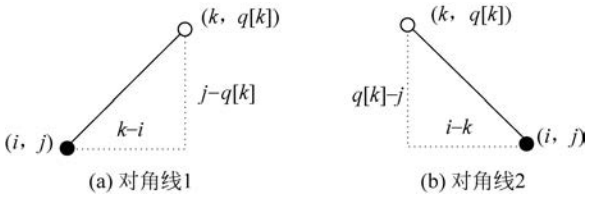

图 5.15　两个皇后构成对角线的情况

归纳起来只要(i,j)位置满足以下条件就存在冲突，否则不冲突：

$(q[k]==j)\ \|\ (\mathrm{abs}(q[k]-j)==\mathrm{abs}(i-k))$　　1≤k≤$i-1$

由此得到测试(i,j)位置能否放置第 i 个皇后的算法如下：

```
bool place(int i,int j)          //测试(i,j)位置能否放置皇后
{   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;
}
```

现在采用递归回溯框架求解。设 queen(i,n)是在 1～$i-1$ 行上已经放好了 $i-1$ 个皇后，用于在 i～n 行放置剩下的 $n-i+1$ 个皇后，为大问题；queen($i+1$,n)表示在 1～i 行上已经放好了 i 个皇后，用于在 $i+1$～n 行放置 $n-i$ 个皇后，为小问题，则求解皇后问题所有解的递归模型如下：

queen(i,n) ≡ n 个皇后放置完毕，输出一个解　　若 $i>n$
queen(i,n) ≡ 在第 i 行找到一个合适的位置(i,j)，放置一个皇后；　　其他
　　queen($i+1$,n)；

对应的递归回溯算法如下：

```
int q[MAXN];                                    //存放各皇后所在的列号,为全局变量
int cnt=0;                                      //累计解的个数
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");
}
void queen11(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=1;j<=n;j++)                  //在第i行上试探每一个列j
        {   if (place(i,j))                     //在第i行上找到一个合适位置(i,j)
            {   q[i]=j;
                queen11(n,i+1);
                q[i]=0;                         //回溯
            }
        }
    }
}
void queen1(int n)                              //用递归法求解n皇后问题
{
    queen11(n,1);
}
```

利用上述算法求出6皇后问题的4个解如下：

```
第1个解:(1,2) (2,4) (3,6) (4,1) (5,3) (6,5)
第2个解:(1,3) (2,6) (3,2) (4,5) (5,1) (6,4)
第3个解:(1,4) (2,1) (3,5) (4,2) (5,6) (6,3)
第4个解:(1,5) (2,3) (3,1) (4,6) (5,4) (6,2)
```

另外也可以采用迭代方式求解 n 皇后问题。同样用数组 q 存放皇后的列位置，$(i,q[i])$ 表示第 i 个皇后放置的位置，n 皇后问题的一个解是$(1,q[1]),(2,q[2]),\cdots,(n,q[n])$，数组 q 的下标为0的元素不用。

由于在第 i 行中找第 i 个皇后的列号时总是先执行 $q[i]$++再判断$(i,q[i])$位置是否合适，所以 $q[i]$的初始值必须置为0。

先放置第1个皇后，然后以2、3、……、n 的次序放置其他皇后，当第 n 个皇后放置好后产生一个解。为了找到所有解，此时算法还不能结束，继续试探第 n 个皇后的下一个位置。

第 $i(i<n)$个皇后放置后，接着放置第 $i+1$ 个皇后，在试探第 $i+1$ 个皇后的位置时都是从第1列开始的。当第 i 个皇后试探了所有列都不能放置时，回溯到第 $i-1$ 个皇后，此时与第 $i-1$ 个皇后的位置$(i-1,q[i-1])$有关，如果第 $i-1$ 个皇后的列号小于 n，即 $q[i-1]<n$，则将其移到下一列，继续试探；否则再回溯到第 $i-2$ 个皇后，以此类推。

对应的迭代回溯算法如下：

```
void queen2(int n)                              //用迭代法求解n皇后问题
{   int i=1;                                    //i表示当前行,i=1表示从第1个皇后开始
    q[i]=0;                                     //q[i]是当前列,在试探之前q[i]置为0
    while (i>=1)                                //重复试探
```

```
        {   q[i]++;                                  //总是先将列号增 1
            while (q[i]<=n && !place(i,q[i]))        //试探一个位置(i,q[i])是否合适
                q[i]++;
            if (q[i]<=n)                             //为第 i 个皇后找到了一个合适位置(i,q[i])
            {   if (i==n)                            //若放置了所有皇后,输出一个解
                    disp(n);
                else                                 //皇后没有放置完
                {   i++;                             //转向下一个皇后的放置
                    q[i]=0;                          //每次试探一个新皇后,q[i]总是从 0 开始
                }
            }
            else i--;                                //若第 i 个皇后找不到合适位置,则回溯到前一个皇后
        }
    }
```

从上看出迭代回溯算法远不如递归回溯算法清晰，这就是为什么在一般情况下回溯算法都是采用递归回溯算法的原因。

【算法分析】 该算法中每个皇后都要试探 n 列，共 n 个皇后，其解空间是一棵子集树，每个结点可能有 n 棵子树，而每个皇后试探一个合适位置的时间为 $O(n)$，所以最坏情况下算法时间复杂度为 $O(n\times n^n)$。

扫一扫

视频讲解

5.2.5 任务分配问题

1. 问题描述

有 $n(n\geqslant 1)$ 个任务需要分配给 n 个人执行，每个任务只能分配给一个人，每个人只能执行一个任务。第 i 个人执行第 j 个任务的成本是 $c[i][j](0\leqslant i,j\leqslant n-1)$。求出总成本最小的一种分配方案。如表 5.3 所示为 4 个人员、4 个任务的信息。

表 5.3 4 个人员、4 个任务的信息

人员	任务 0	任务 1	任务 2	任务 3
0	9	2	7	8
1	6	4	3	7
2	5	8	1	8
3	7	6	9	4

2. 问题求解

n 个人和 n 个任务的编号均用 0～$n-1$ 表示。所谓一种分配方案就是由第 i 个人执行第 j 个任务，也就是说每个人从 n 个任务中选择一个任务，即 n 选一，所以本问题的解空间树可以看成一棵子集树，并且要求总成本最小的解（最优解是最小值），属于求最优解类型。

设计解向量 $\boldsymbol{x}=(x_0,x_1,\cdots,x_{n-1})$，这里以人为主，即人找任务（也可以以任务为主，即任务找人），也就是第 i 个人执行第 x_i 个任务（$0\leqslant x_i\leqslant n-1$）。bestx 表示最优解向量，bestc 表示最优解的成本（初始值为∞），$\boldsymbol{x}$ 表示当前解向量，cost 表示当前解的总成本（初始为 0），另外设计一个 used 数组，其中 used[j]表示任务 j 是否已经分配（初始时所有元素均为 false），为了简单，将这些变量均设计为全局变量。

解空间中根结点的层次 i 为 0，当搜索到第 i 层的每个结点时，表示为第 i 个人分配一

个没有分配的任务，即选择满足 used[j]=0($0 \leqslant j \leqslant n-1$)的任务 j。对应的递归回溯算法如下：

```
int n=4;                                    //一个测试实例
int c[MAXN][MAXN]={{9,2,7,8},{6,4,3,7},{5,8,1,8},{7,6,9,4}};
vector<int> bestx;                          //最优解向量
int bestc=INF;                              //最优解的成本,初始值为∞
bool used[MAXN];                            //used[j]表示任务j是否已经分配
void dfs(vector<int> &x,int cost,int i)     //为第i个人员分配任务
{   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=0;j<n;j++)               //为人员i试探分配任务
        {   if (used[j]==0)                 //若任务j还没有被分配
            {   x[i]=j;                     //将任务j分配给人员i
                used[j]=true;               //表示任务j已经被分配
                cost+=c[i][j];              //累计成本
                dfs(x,cost,i+1);            //继续为人员i+1分配任务
                used[j]=false;              //used[j]回溯
                x[i]=-1;                    //x[i]回溯
                cost-=c[i][j];              //cost回溯
            }
        }
    }
}
void alloction()                            //求解算法
{   memset(used,false,sizeof(task));
    vector<int> x(n,0);                     //当前解向量,n个元素初始化为0
    int cost=0;                             //当前解的成本
    dfs(x,cost,0);                          //从人员0开始分配任务
}
```

上述算法的执行结果如下：

```
第0个人安排任务1
第1个人安排任务0
第2个人安排任务2
第3个人安排任务3
总成本=13
```

【算法分析】 算法的解空间是一棵 n 叉树(子集树)，再考虑叶子结点处复制更优解的时间为 $O(n)$，所以最坏的时间复杂度为 $O(n \times n^n)$。例如，表5.3的实例中 $n=4$，经测试搜索的结点个数为65。

现在考虑采用剪支提高性能，该问题是求最小值，所以设计下界函数。当搜索到第 i 层的某个结点时，如果选择了任务 j(即执行 $x[i]=j$，cost+=$c[i][j]$)，此时部分解向量 $\boldsymbol{P}=(x_0,x_1,\cdots,x_i)$，那么后面如何分配才能使得该路径(一条从根到叶子结点的路径对应一个分配方案)的 cost 尽可能小呢？如果后面编号为 $i+1 \sim n-1$ 的每个人都分配一个尚未分配的最小成本的任务，其累计成本为

$$\text{minsum}=\sum_{i1=i+1}^{n-1}\min_{j1\notin P}\{c_{i1,j1}\}$$

则 b=cost+minsum 一定是该路径的最小成本，如图 5.16 所示。如果 $b\geqslant$bestc(bestc 是当前已经求出的一个最优成本)，说明 $x[i]=j$ 这条路径走下去一定不可能找到更优解，所以停止该分支的搜索。这里的剪支就是仅扩展 $b<$bestc 的孩子结点。

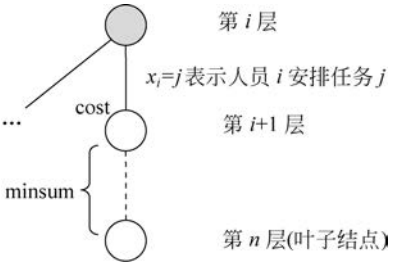

图 5.16　人员 i 安排任务 j 的情况

带剪支的递归回溯算法如下：

```
int bound(int cost,int i)                    //求下界算法
{   int minsum=0;
    for (int i1=i;i1<n;i1++)                 //求 c[i..n-1]行中未分配任务的最小成本和
    {   int minc=INF;                        //置为∞
        for (int j1=0;j1<n;j1++)
            if (used[j1]==false && c[i1][j1]<minc)
                minc=c[i1][j1];
        minsum+=minc;
    }
    return cost+minsum;
}
void dfs(vector<int> &x,int cost,int i)      //为第 i 个人员分配任务
{   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=0;j<n;j++)                //为人员 i 试探任务 j
        {   if (used[j]==0)                  //若任务 j 还没有被分配
            {   used[j]=true;
                x[i]=j;                      //任务 j 分配给人员 i
                cost+=c[i][j];
                if(bound(cost,i+1)<bestc)    //剪支(考虑 c[i+1..n-1]行中的最小成本)
                    dfs(x,cost,i+1);         //继续为人员 i+1 分配任务
                used[j]=false;               //回溯
                x[i]=-1;
                cost-=c[i][j];
            }
        }
    }
}
```

对于表 5.3 的实例，经测试搜索的结点个数为 9，算法的时间性能得到明显提高。但求下界的时间为 $O(n^2)$，带剪支的回溯算法的最坏时间复杂度为 $O(n^2\times n^n)$。

扫一扫

视频讲解

5.2.6　出栈序列

1. 问题描述

有一个含 n 个不同元素的进栈序列 a，求通过一个栈得到的所有合法的出栈序列。例如 a={1,2,3}时产生的 5 个合法的出栈序列是{3,2,1}、{2,3,1}、{2,1,3}、{1,3,2}、{1,2,3}。

2. 问题求解

设计一个栈 st，用 i 遍历 a 的元素(初始时 $i=0$)，解向量为 $\boldsymbol{x}=(x_0,x_1,\cdots,x_{n-1})$，每个解向量对应一个合法的出栈序列，$j$ 表示产生的出栈序列中的元素个数。显然一种状态是由(a，st，$\boldsymbol{x}$)确定的，实际上由 a 和 $\boldsymbol{x}$ 可以确定 st 的状态，所以可以简化为用(a，$\boldsymbol{x}$)表示状态，而 a 的状态可以用其遍历变量 i 表示，$\boldsymbol{x}$ 的状态可以用 j 表示，这样实际状态用(i，j)表示。如图 5.17 所示，在(i，j)状态下可以选择以下两种操作：

① a_i 进栈($i<n$)，状态变为($i+1$，j)。

② 出栈一个元素 tmp(栈 st 非空)并且添加到 $\boldsymbol{x}$ 中，状态变为(i，$j+1$)。注意 $i=j$ 时表示栈空，此时不能做出栈操作。

由此看出该问题类似于子集和问题，每次在两种操作中选择一个操作执行，所以可以采用基于子集树的回溯算法求解。

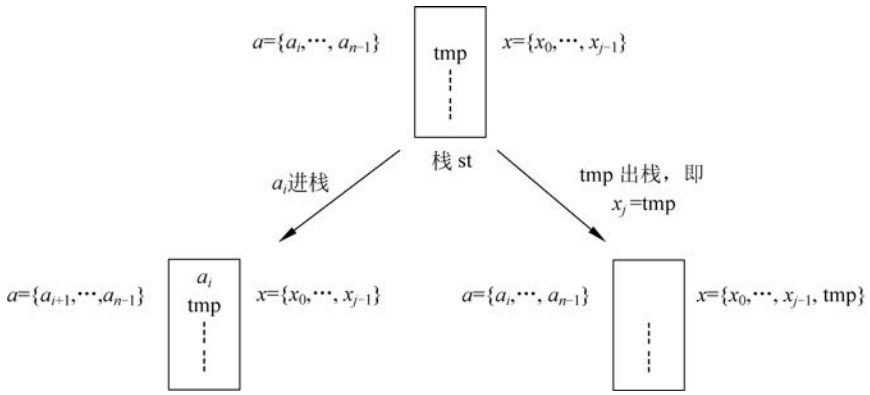

图 5.17 在(i，j)状态下的两种操作

显然解空间中的叶子结点的状态是(n，n)，表示 a 中所有元素遍历完成，并且产生的出栈序列包含 n 个元素，此时的 x 就是一个合法的出栈序列。例如，$a=\{1,2,3\}$的求解过程如图 5.18 所示。

对应的算法如下：

```
int sum=0;                                  //累计出栈序列的个数
vector<int> a={1,2,3};                      //进栈序列
int n=a.size();                             //进栈序列的元素个数
stack<int> st;
void disp(vector<int> & x)                  //输出一个解
{   printf(" 出栈序列%2d: ",++sum);
    for (int i=0;i<n;i++)
        printf("%d ",x[i]);
    printf("\n");
}
void dfs(vector<int> & x,int i,int j)       //递归算法
{   if (i==n && j==n)                       //输出一种可能的方案
        disp(x);
    else
    {   if (i<n)                            //剪支：i<n 时 a[i]进栈
        {   st.push(a[i]);                  //a[i]进栈
            dfs(x,i+1,j);
```

```
            st.pop();                                   //回溯：出栈
        }
        if (!st.empty())                                //剪支：栈不空时出栈 tmp
        {   int tmp=st.top(); st.pop();                 //出栈 tmp
            x[j]=tmp;                                   //将 tmp 添加到 x 中
            j++;                                        //j 增加 1
            dfs(x,i,j);
            j--;                                        //回溯：j 减少 1
            st.push(tmp);                               //回溯：x 进栈以恢复环境
        }
    }
}
void solve()                                            //求 a 的所有合法的出栈序列
{   vector<int> x(n);
    dfs(x,0,0);                                         //i、j 均从 0 开始
}
```

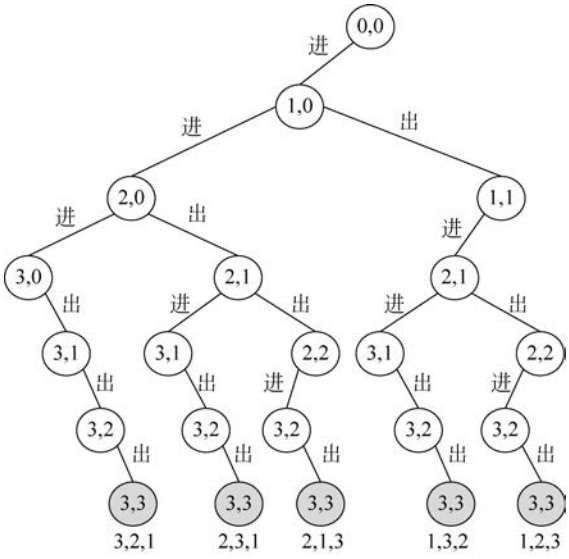

图 5.18　求 $a=\{1,2,3\}$ 的所有出栈序列的过程

【算法分析】　类似子集和问题，但解空间的高度大约为 $i+j=2n$，输出一个解的时间为 $O(n)$，所以最坏情况下的时间复杂度为 $O(n\times 2^{2n})$ 即 $O(n\times 4^n)$，由于存在剪支操作（保证 $i\geqslant j$，即出栈序列中的元素个数一定不大于进栈的元素个数），所以实际时间性能远好于 $O(n\times 4^n)$。

扫一扫

视频讲解

5.2.7　图的 m 着色

1. 问题描述

给定无向连通图 G 和 m 种不同的颜色。用这些颜色为图 G 的各顶点着色，每个顶点着一种颜色。如果有一种着色法使 G 中每条边的两个顶点着不同颜色，则称这个图是 m 可着色的。图的 m 着色问题是对于给定图 G 和 m 种颜色，找出所有不同的着色方案的数目。

2. 问题求解

对于含 n 个顶点的无向连通图 G，顶点的编号是 $0\sim n-1$，采用 vector<int>数组的邻

接表 A 存储，其中 $A[i]$向量为顶点 i 的所有相邻顶点。例如如图 5.19 所示的无向连通图，对应的邻接表如下：

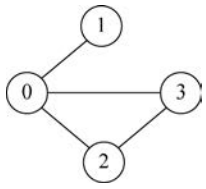

图 5.19 一个无向连通图

```
A[0..3]={{1,2,3},{0},{0,3},{0,2}}
```

m 种颜色的编号为 0～$m-1$，这里实际上就是为每个顶点 i 选择 m 种颜色中的一种(m 选一)，使得任意两个相邻顶点的着色不同，所以解空间树看成一棵子集树，并且求解个数，属于求所有解类型。

设计解向量为 $\boldsymbol{x}=(x_0,x_1,\cdots,x_{n-1})$，其中 x_i 表示顶点 i 的着色($0\leqslant x_i\leqslant m-1$)，初始时置 x 的所有元素为－1，表示所有顶点均没有着色，用 cnt 累计解个数(初始为 0)。采用递归回溯方法从顶点 0 开始试探($i=0$ 对应根结点)，当 $i\geqslant n$ 时表示找到一种着色方案(对应解空间中的一个叶子结点)。

对于顶点 i，所有可能的着色 j 为 0～$m-1$ 中的一种，如果顶点 i 的所有相邻顶点的颜色均不等于 j，说明顶点 i 着色 j 是合适的，只要有一个相邻顶点的颜色等于 j，则顶点 i 着色 j 是不合适的，需要回溯。对应的算法如下：

```
#define MAXN 30                          //最多顶点个数
int n;
vector<int> A[MAXN];                     //邻接表
int cnt;                                 //全局变量,累计解个数
int x[MAXN];                             //全局变量,x[i]表示顶点i的着色
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)                             //达到一个叶子结点
        cnt++;
    else
    {   for(int j=0;j<m;j++)
        {   x[i]=j;                      //置顶点i为颜色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 Colors(int m)                        //求图的m着色问题
{   cnt=0;
    memset(x,0xff,sizeof(x));            //所有元素初始化为-1
    dfs(m,0);                            //从顶点0开始搜索
    return cnt;
}
```

例如，对于图 5.19，若 $m=3$，求出有 12 种不同的着色方案。

【算法分析】 在该算法中每个顶点都试探编号为 0～$m-1$ 的颜色，共 n 个顶点，对应解空间树是一棵 m 叉树（子集树），每个结点调用 judge()的时间为 $O(n)$，所有算法的最坏时间复杂度为 $O(n \times m^n)$。

5.2.8 实战——救援问题(HDU1242)

1. 问题描述

扫一扫

视频讲解

A 被抓进了监狱，监狱被描述为一个有 $N \times M(N,M \leqslant 200)$ 个方格的网格，监狱里有围墙、道路和守卫。A 的朋友们想要救他（可能有多个朋友），只要有一个朋友找到 A（就是到达 A 所在的位置）那么 A 将被救了。在找 A 的过程中只能向上、向下、向左和向右移动，若遇到有守卫的方格必须杀死守卫才能进入该方格。假设每次向上、向下、向右、向左移动需要一个单位时间，而杀死一个守卫也需要一个单位时间。请帮助 A 的朋友们计算救援 A 的最短时间。

输入格式：第一行包含两个整数，分别代表 N 和 M。然后是 N 行，每行有 M 个字符，其中'#'代表墙，'.'代表道路，'a'代表 A，'r'代表 A 的朋友，'x'代表守卫。处理到文件末尾。

输出格式：对于每个测试用例，输出一个表示所需最短时间的整数。如果这样的整数不存在，输出一行包含"Poor ANGEL has to stay in the prison all his life"的字符串。

输入样例：

```
7 8
# . # # # # # .
# . a # . . r .
# . . # x . . .
. . # . . # . #
# . . . # # . .
. # . . . . . .
. . . . . . . .
```

输出样例：

```
13
```

2. 问题求解

本题与迷宫问题类似，假设距离 A 最近的是朋友 B，显然 B 到 A 的最短路径和 B 到 A 的最短路径是相同的，由于 A 只有一个人，而他的朋友可能有多个，所以这里从 A 出发搜他的最近的朋友。A 的位置用(sx,sy)表示（对应解空间的根结点），从该位置搜索路径，len 表示当前路径的长度，bestlen 表示最优解，即最短路径长度（初始置为∞），每次找到一个朋友（对应解空间的叶子结点）比较路径长度，将最短路径长度保存在 bestlen 中。与普通迷宫问题相比，每次路径搜索也是 4 个方位选一，但改为遇到道路'.'走一步，路径长度 len 增加 1，当遇到守卫'x'时除了走一步还需要杀死守卫，所以路径长度 len 增加 2。解空间搜索完毕，若 bestlen 为∞，说明没有找到任何朋友，按题目要求输出一个字符串，否则说明最少找到一个朋友，输出 bestlen 即可。对应的程序如下：

```
#include<iostream>
#include<cstring>
```

```
using namespace std;
#define INF 0x3f3f3f3f
#define MAXN 202
int dx[]={0,0,1,-1};                                    //水平方向偏移量
int dy[]={1,-1,0,0};                                    //垂直方向偏移量
char grid[MAXN][MAXN];                                  //存放网格
int visited[MAXN][MAXN];                                //访问标记数组
int n,m;
int bestlen;
void dfs(int x,int y,int len)
{   if(grid[x][y]=='r')                                 //找到朋友
    {   if(len<bestlen)                                 //比较求最短路径长度
            bestlen=len;
    }
    else
    {   for(int di=0;di<4;di++)                         //枚举4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]==1)                      //(nx,ny)已访问时跳过
                continue;
            if(grid[x][y]=='#')                         //(nx,ny)为墙时跳过
                continue;
            if(grid[nx][ny]=='x')                       //(nx,ny)为守卫的情况
            {   visited[nx][ny]=1;                      //标记(nx,ny)已经访问
                dfs(nx,ny,len+2);                       //走一步+杀死守卫
                visited[nx][ny]=0;                      //路径回溯
            }
            else                                        //(nx,ny)为道路的情况
            {   visited[nx][ny]=1;
                dfs(nx,ny,len+1);                       //走一步
                visited[nx][ny]=0;
            }
        }
    }
}
int main()
{   int sx,sy;                                          //标记A的位置
    while(cin >> n >> m)
    {   for(int i=0;i<n;i++)                            //输入矩阵
        {   for(int j=0;j<m;j++)
            {   cin >> grid[i][j];
                if(grid[i][j]=='a')
                {   sx=i;
                    sy=j;
                }
            }
        }
        bestlen=INF;
        memset(visited,0,sizeof(visited));
        dfs(sx,sy,0);
        if(bestlen==INF)                                //如果bestlen为INF说明没找到路径
            cout <<"Poor ANGEL has to stay in the prison all his life."<< endl;
        else
            cout << bestlen << endl;
```

```
    }
    return 0;
}
```

上述程序提交的结果为通过，执行时间为 187ms，内存消耗为 1596KB，满足题目的时空要求。

5.3 基于排列树框架的问题求解

扫一扫

视频讲解

5.3.1 任务分配问题

1. 问题描述

见 5.2.5 节任务分配问题的描述。

2. 问题求解

n 个人和 n 个任务的编号均用 0～$n-1$ 表示，设计解向量 $\boldsymbol{x}=(x_0,x_1,\cdots,x_{n-1})$，同样以人为主，也就是第 i 个人执行第 x_i 个任务（$0\leqslant x_i\leqslant n-1$），显然每个合适的分配方案 $\boldsymbol{x}$ 一定是 0～$n-1$ 的一个排列，可以求出 0～$n-1$ 的全排列，每个排列作为一个分配方案，求出其成本，比较找到一个最小成本 bestc 即可。

用 bestx 表示最优解向量，bestc 表示最优解的成本，$\boldsymbol{x}$ 表示当前解向量，cost 表示当前解的总成本（初始为 0），另外设计一个 used 数组，其中 used[j]表示任务 j 是否已经分配（初始时所有元素均为 false），为了简单，将这些变量均设计为全局变量。根据排列树的递归算法框架，当搜索到第 i 层的某个结点时，第一个 swap(x[i],x[j])表示为人员 i 分配任务 x[j]（注意不是任务 j），成本是 c[i][x[i]]（因为 x[i]就是交换前的 x[j]），所以执行 used[x[i]]=true，cost+=c[i][x[i]]，调用 dfs(x，cost，$i+1$)继续为人员 $i+1$ 分配任务，回溯操作是 cost-=c[i][x[i]]、used[x[i]]=false 和 swap(x[i],x[j])（正好与调用 dfs(x，cost，$i+1$)之前的语句的顺序相反）。

考虑采用剪支提高性能，设计下界函数，与 5.2.5 节的 bound 算法相同，仅需要将 j（指任务编号）改为 x[j]即可，如图 5.20 所示。

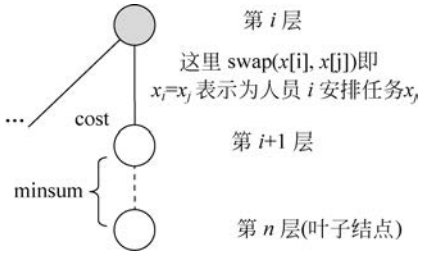

图 5.20 为人员 i 安排任务 x[j]的情况

带剪支的排列树递归回溯算法如下：

```
int n=4;                                        //一个测试实例
int c[MAXN][MAXN]={{9,2,7,8},{6,4,3,7},{5,8,1,8},{7,6,9,4}};
vector<int> bestx;                              //最优解向量
int bestc=INF;                                  //最优解的成本,初始置为∞
bool used[MAXN];                                //used[j]表示任务 j 是否已经分配人员
int bound(vector<int> &x,int cost,int i)        //求下界算法
{   int minsum=0;
    for (int i1=i;i1<n;i1++)                    //求 c[i..n-1]行中未分配的最小成本和
```

```
    {   int minc=INF;
        for (int j1=0;j1<n;j1++)
            if (used[x[j1]]==false && c[i1][x[j1]]<minc)
                minc=c[i1][x[j1]];
        minsum+=minc;
    }
    return cost+minsum;
}
void dfs(vector<int> &x,int cost,int i)          //递归回溯(排列树)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=i;j<n;j++)                    //为人员 i 试探任务 x[j]
        {   if (used[x[j]]) continue;            //若任务 x[j]已经被分配,则跳过
            swap(x[i],x[j]);                     //为人员 i 分配任务 x[j]
            used[x[i]]=true;
            cost+=c[i][x[i]];
            if(bound(x,cost,i+1)<bestc)          //剪支
                dfs(x,cost,i+1);                 //继续为人员 i+1 分配任务
            cost-=c[i][x[i]];                    //cost 回溯
            used[x[i]]=false;                    //used 回溯
            swap(x[i],x[j]);
        }
    }
}
void alloction()                                 //基于排列树的递归回溯算法
{   memset(used,false,sizeof(used));
    vector<int> x;                               //当前解向量
    for(int i=0;i<n;i++)                         //将 x[0..n-1]分别设置为 0 到 n-1 的值
        x.push_back(i);
    int cost=0;                                  //当前解的成本
    dfs(x,cost,0);                               //从人员 1 开始
}
```

【算法分析】 算法的解空间是一棵排列树,求下界的时间为 $O(n^2)$,所以最坏的时间复杂度为 $O(n^2\times n!)$。例如,上述实例中 $n=4$,经测试不剪支(除去 dfs 中的 if(bound(x,cost,i)<bestc))时搜索的结点个数为 65,而剪支后搜索的结点个数为 9。

说明:任务分配问题在 5.2.5 节采用基于子集树框架时最坏时间复杂度为 $O(n^2\times n^n)$,这里采用基于排列树框架的最坏时间复杂度为 $O(n^2\times n!)$,显然 $n>2$ 时 $O(n!)$优于 $O(n^n)$,实际上由于前者通过 used 判重,剪去了重复的分支,其解空间本质上也是一棵排列树,两种算法的最坏时间复杂度都是 $O(n^2\times n!)$,类似地有 5.2.4 节的 n 皇后问题等。

扫一扫

视频讲解

5.3.2 货郎担问题

1. 问题描述

货郎担问题又译为旅行商问题(TSP),是数学领域中的著名问题之一。假设有一个货郎担要拜访 n 个城市,他必须选择所要走的路径,路径的限制是每个城市只能拜访一次,而

且最后要回到原来出发的城市，要求路径长度最短的路径。

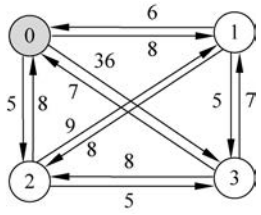

图 5.21　一个 4 城市的图

以图 5.21 所示的一个 4 城市图为例，假设起点 s 为 0，所有从顶点 0 回到顶点 0 并通过所有顶点的路径如下：

```
路径 1：0→1→2→3→0：28
路径 2：0→1→3→2→0：29
路径 3：0→2→1→3→0：26
路径 4：0→2→3→1→0：23
路径 5：0→3→2→1→0：59
路径 6：0→3→1→2→0：59
```

最后求得的最短路径长度为 23，最短路径为 0→2→3→1→0。

2. 问题求解

本问题是求路径长度最短的路径，属于求最优解类型。假设图中有 n 个顶点，顶点编号为 0～$n-1$，采用邻接矩阵 $\boldsymbol{A}$ 存储。显然 TSP 路径是简单回路（除了起始点和终点相同，其他顶点不重复），可以采用穷举法，以全排列的方式求出所有路径及其长度，再加上回边，在其中找出长度最短的回路即为 TSP 路径，但这样做难以剪支，时间性能较低。

现在采用基于排列树的递归回溯算法，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～x_{n-1} 为其他 $n-1$ 个顶点编号，d 表示当前路径的长度，用 bestx 保存最短路径，bestd 表示最短路径长度，其初始值置为∞。设计算法 dfs(x,d,s,i)的几个重点如下：

① x_0 固定作为起点 s，不能取其他值，所以不能从 $i=0$ 开始调用 dfs，应改为从 $i=1$（此时 $d=0$）开始调用 dfs。为了简单，假设 $s=0$，x 初始时为(0,1,…,$n-1$)，$i=1$ 时 x_1 会取 $x[1..n-1]$的每一个值（共 $n-1$ 种取值），如图 5.22 所示，当 $x_1=x_1$(1)时，对应路径长度为 $d+A[0][1]$，当 $x_1=x_2$(2)时，对应路径长度为 $d+A[0][2]$，以此类推。归纳起来，当搜索到解空间的第 i 层的某个结点时，x_i 取 $x[i..n-1]$中的某个值后当前路径长度为 $d+A[x[i-1]][x[i]]$。

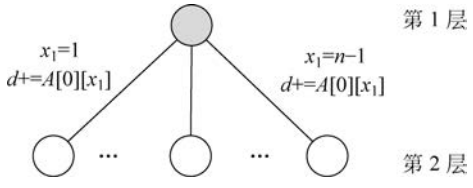

图 5.22　x_1 的各种取值情况

② 当搜索到达某个叶子结点时($i\geqslant n$)，对应的 TSP 路径长度应该是 $d+A[x[n-1]][s]$（因为 TSP 路径是闭合的回路），对应的路径是 $x\cup\{s\}$。通过比较所有回路的长度求最优解。

③ 如何剪支呢？若当前已经求出最短路径长度 bestd，如果 x_i 取 x_j 值，对应的路径长度为 $d+A[x[i-1]][x[j]]$，若 $d+A[x[i-1]][x[j]]\geqslant$bestd，说明该路径走下去不可能找到更短路径，终止该路径的搜索，也就是说仅扩展满足 $d+A[x[i-1]][x[j]]<$bestd 的路径。

对应的用回溯法求 TSP 问题的算法如下：

```
vector<vector<int>> A={{0,8,5,36},{6,0,8,5},{8,9,0,5},{7,7,8,0}};
int n=4;
int cnt=0;                                  //路径条数累计
vector<int> bestx;                          //保存最短路径
int bestd=INF;                              //保存最短路径长度,初始为∞
```

```
void disp(vector<int> &x,int d,int s)                    //输出一个解
{   printf("   第%d 条路径: ",++cnt);
    for (int j=0;j<x.size();j++)
        printf("%d->",x[j]);
    printf("%d",s);                                      //末尾加上起点 s
    printf(", 路径长度: %d\n",d+A[x[n-1]][s]);
}
void dfs(vector<int> &x,int d,int s,int i)               //回溯法算法
{   if(i>=n)                                             //到达一个叶子结点
    {   disp(x,d,s);                                     //输出一个解
        if(d+A[x[n-1]][s]<bestd)                         //同时比较求最优解
        {   bestd=d+A[x[n-1]][s];                        //求 TSP 长度
            bestx=x;                                     //更新 bestx
            bestx.push_back(s);                          //末尾添加起始点
        }
    }
    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]]!=0 && A[x[i-1]][x[j]]!=INF)   //若 x[i-1]到 x[j]有边
            {   if(d+A[x[i-1]][x[j]]<bestd)              //剪支
                {   swap(x[i],x[j]);
                    dfs(x,d+A[x[i-1]][x[i]],s,i+1);
                    swap(x[i],x[j]);
                }
            }
        }
    }
}
void TSP1(int s)                                         //用回溯法求解 TSP(起始点为 s)
{   vector<int> x;                                       //定义解向量
    x.push_back(s);
    for(int i=1;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
}
```

上述算法当 $s=1$ 时的求解结果如下。实际上共有 6 条路径，通过剪支终止了两条路径的搜索。

```
第 1 条路径: 1->0->2->3->1,   路径长度: 23
第 2 条路径: 1->2->3->0->1,   路径长度: 28
第 3 条路径: 1->3->2->0->1,   路径长度: 29
第 4 条路径: 1->3->0->2->1,   路径长度: 26
最短路径:    1->0->2->3->1,   路径长度: 23
```

【算法分析】 算法的解空间是一棵排列树，由于是从第一层开始搜索的，排列树的高度为 n（含叶子结点层），考虑输出一个解的时间为 $O(n)$，所以最坏的时间复杂度为 $O(n\times(n-1)!)$即 $O(n!)$。

思考题： TSP 问题是在一个图中查找从起点 s 经过其他所有顶点又回到顶点 s 的最短路径，在上述算法中为什么不考虑路径中出现重复顶点的情况？

5.3.3 实战——含重复元素的全排列Ⅱ(LeetCode47)

1. 问题描述

给定一个可包含重复数字的序列 nums，按任意顺序返回所有不重复的全排列。例如，nums=[1,1,2]，输出结果是[[1,1,2]，[1,2,1]，[2,1,1]]。要求设计如下函数：

扫一扫

视频讲解

```
class Solution {
public:
    vector<vector<int>> permuteUnique(vector<int>& nums) { }
};
```

2. 问题求解

该问题与求非重复元素的全排列问题类似，解空间是排列树，并且属于求所有解类型。先按求非重复元素全排列的一般过程求含重复元素的全排列，假设 $a=\{1,\boxed{1},2\}$，其中包含两个1，为了区分，后面一个1加上一个框，求其全排列的过程如图5.23所示。从中看出，1↔1的分支和1↔$\boxed{1}$的分支产生的所有排列是相同的，属于重复的排列，应该剪去后者，再看第1层的“{2,$\boxed{1}$,1}”结点，同样它扩展的两个分支分别是$\boxed{1}$↔$\boxed{1}$和$\boxed{1}$↔1，也是相同的，也应该剪去后者。这样剪去后得到的结果是{1,1,2}，{1,2,1}和{2,1,1}，也就是不重复的全排列。

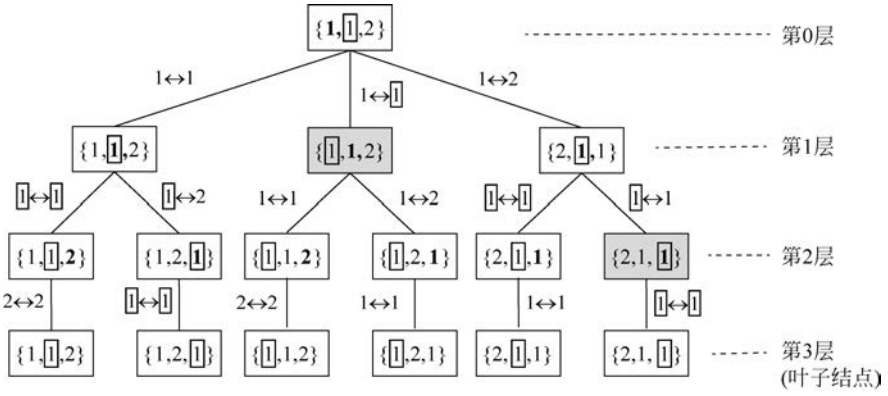

图 5.23 求 $a=\{1,\boxed{1},2\}$ 的全排列的过程

同样设解向量为 $\boldsymbol{x}=(x_0,x_1,\cdots,x_n)$，每个 $\boldsymbol{x}$ 表示一个排列，x_i 表示该排列中 i 位置所取的元素，初始时 $\boldsymbol{x}$=nums。在解空间中搜索到第 i 层的某个结点C时，如图5.24所示，C结点的每个分支对应 x_i 的一个取值，理论上讲 x_i 可以取 $x_i\sim x_{n-1}$ 的每个值，也就是说从根结点经过结点C到达第 $i+1$ 层的结点有 $n-1-i+1=n-i$ 条路径，在这些路径中从根结点到C结点都是相同的。当 x_i 取值 x_j 时（对应图中粗分支）走到B

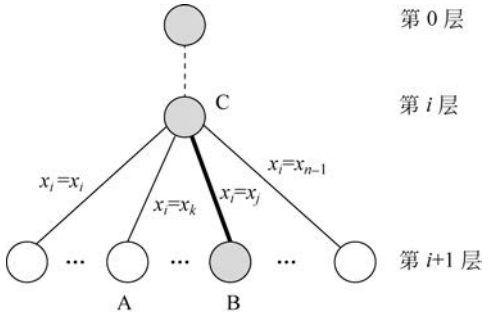

图 5.24 x_i 的各种取值

结点，如果 x_j 与前面 $x_i \sim x_{j-1}$ 中的某个值 x_k 相同，当 x_i 取值 x_k 时走到 A 结点，显然根结点到 A 和 B 结点的路径完全相同，而且它们的层次相同，后面的操作也相同，则所有到达叶子结点产生的解必然相同，属于重复的排列，需要剪去。

剪去重复解的方法是，当 j 从 i 到 $n-1$ 循环时，每次循环执行 swap($x[i]$,$x[j]$)为 i 位置选取元素 $x[j]$，如果 $x[j]$与 $x[i..j-1]$中的某个元素相同，则会出现重复的排列，跳过，也就是说在执行 swap($x[i]$,$x[j]$)之前先判断 $x[j]$是否在前面的元素 $x[i..j-1]$中出现过，如果没有出现过就继续做下去，否则跳过 $x[j]$的操作。对应的程序如下：

```
class Solution {
    vector<vector<int>> ps;                    //存放 nums 的全排列
public:
    vector<vector<int>> permuteUnique(vector<int>& nums)
    {   int n=nums.size();
        dfs(nums,n,0);
        return ps;
    }
    void dfs(vector<int>&x,int n,int i)        //递归算法
    {   if (i>=n)                              //到达叶子结点
            ps.push_back(x);
        else                                   //没有到达叶子结点
        {   for (int j=i;j<n;j++)              //遍历 x[i..n-1]
            {   if(judge(x,i,j))               //检测 x[j]
                {   swap(x[i],x[j]);           //为 i 位置选取元素 x[j]
                    dfs(x,n,i+1);              //继续
                    swap(x[i],x[j]);           //回溯
                }
            }
        }
    }
    bool judge(vector<int>& x,int i,int j)
    //判断 x[j]是否在 x[i..j-1]中出现过，若出现过，返回 false；没有出现过，返回 true
    {   if(j>i)
        {   for(int k=i;k<j;k++)               //x[j]是否与 x[i..j-1]的元素相同
                if(x[k]==x[j])                 //若相同返回 false
                    return false;
        }
        return true;                           //全部不相同返回 true
    }
};
```

上述程序提交时通过，执行时间为 4ms，内存消耗为 8.7MB。

5.4 练 习 题

5.4.1 单项选择题

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

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

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

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

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

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}

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

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. $O(n^2)$

9. 用回溯法求解旅行商问题时的解空间是_____。

A. 子集树　　B. 排列树

C. 深度优先生成树　　D. 广度优先生成树

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

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

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

5.4.2 问答题

1. 回溯法的搜索特点是什么？

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。

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。

4. 对于 n 皇后问题，通过解空间说明 $n=3$ 时是无解的。

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？

6. 本书 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？

7. 对应如图 5.25 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。

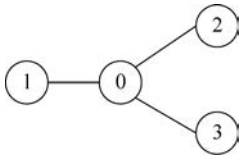

图 5.25 一个无向连通图

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。

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)             //使用递归算法
{   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);
}
```

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？

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)          //使用递归算法
{   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]已经被分配,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)          //使用递归算法
{   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]已经被分配,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]);
        }
    }
}
```

5.4.3 算法设计题

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)。

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。

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)。

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)。

5. 在 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。

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。

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。

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。

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。

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。

11. 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，设计一个算法求其中顶点 u 到顶点 v 的所有路径。

12. 假设一个含 n 个顶点(顶点编号为 0～$n-1$)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。

5.5 上机实验题

1. 象棋算式

编写一个实验程序 exp5-1 求解如图 5.26 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。

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。

兵炮马卒
\+ 兵炮车卒
车卒马兵卒

图 5.26 象棋算式

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。

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。

5.6 在线编程题

1. LeetCode216——组合总和Ⅲ
2. LeetCode39——组合总和
3. LeetCode131——分割回文串
4. HDU1027——第 k 小的排列
5. HDU2553——n 皇后问题
6. HDU2616——杀死怪物
7. POJ3187——向后数字和
8. POJ1321——棋盘问题
9. POJ2488——骑士游历
10. POJ1040——运输问题
11. POJ1129——最少频道数