第5章 回溯法

5.1 LintCode1353——根结点到叶子结点求和★★

问题描述：给定仅包含数字 0～9 的二叉树，每个根结点到叶子结点的路径可以表示为数字。例如 root-to-leaf 路径 1-> 2-> 3，它代表数字 123。设计一个算法求所有根结点到叶子结点的数字的总和。要求设计如下成员函数：

```
int sumNumbers(TreeNode * root) { }
```

解：将二叉树看成一棵解空间树，从根结点开始搜索所有的结点，用 cursum 累计路径表示的数字，每次到达一个叶子结点时将 cursum 累计到 ans 中，最后返回 ans 即可。对应的回溯算法如下：

```
class Solution {
  int ans;
public:
  int sumNumbers(TreeNode * root) {
     if(root == NULL) return 0;
     ans = 0;
     dfs(root,root -> val);
     return ans;
  }
  void dfs(TreeNode * root,int cursum) {
     if(root -> left == NULL && root -> right == NULL)
        ans += cursum;
     else {
        if(root -> left!= NULL) {
           cursum = cursum * 10 + root -> left -> val;
           dfs(root -> left,cursum);
           cursum = (cursum - root -> left -> val)/10;
        }
        if(root -> right!= NULL) {
           cursum = cursum * 10 + root -> right -> val;
           dfs(root -> right,cursum);
           cursum = (cursum - root -> right -> val)/10;
        }
     }
  }
};
```

上述程序提交后通过，执行用时为 41ms，内存消耗为 5.52MB。

5.2 LintCode802——数独★★★

问题描述：数独是一种逻辑性的数字填充游戏，玩家需将数字填进每一格，而每行、每列和每个宫（即 3×3 的大格）恰好有 1～9 的所有数字。游戏设计者会提供一部分的数字，使谜题只有一个答案。一个已解答的数独其实是一种多了宫的限制的拉丁方阵，因为同一个数字不可能在同一行、列或宫中出现多于一次。编写一个程序，通过填充空单元来解决数独谜题，空单元由数字 0 表示，可以认为只有一个唯一的解决方案。例如，给定的数独谜题如下：

```
{{0,0,9,7,4,8,0,0,0},
 {7,0,0,0,0,0,0,0,0},
 {0,2,0,1,0,9,0,0,0},
 {0,0,7,0,0,0,2,4,0},
 {0,6,4,0,1,0,5,9,0},
 {0,9,8,0,0,0,3,0,0},
 {0,0,0,8,0,3,0,2,0},
 {0,0,0,0,0,0,0,0,6},
 {0,0,0,2,7,5,9,0,0}}
```

返回结果如下：

```
{{5,1,9,7,4,8,6,3,2},
 {7,8,3,6,5,2,4,1,9},
 {4,2,6,1,3,9,8,7,5},
 {3,5,7,9,8,6,2,4,1},
 {2,6,4,3,1,7,5,9,8},
 {1,9,8,5,2,4,3,6,7},
 {9,7,5,8,6,3,1,2,4},
 {8,3,2,4,9,1,7,5,6},
 {6,4,1,2,7,5,9,8,3}}
```

要求设计如下成员函数：

```
void solveSudoku(vector < vector < int >> &b) { }
```

解法 1：从位置(i,j)为(0,0)开始搜索，先搜索第 i 行的每个列，当 $j=9$ 时转向下一行，每个 $b[i][j]=0$ 的位置检测是否可以放置 $d(1\leqslant d\leqslant 9)$，相当于每个位置 9 选择 1，当 $i=9$ 时说明找到一个解，返回 true。对应的程序如下：

```
class Solution {
public:
  void solveSudoku(vector < vector < int >> &b) {
     dfs(b,0,0);
  }
  bool dfs(vector < vector < int >> &b,int i,int j) {
     if (i == 9) {
        return true;
     }
     if (j == 9) {
        return dfs(b,i + 1,0);
     }
     if (b[i][j]!= 0) {
        return dfs(b,i,j + 1);
     }
     for (int d = 1;d < = 9;d++) {
        if (!valid(b,i,j,d)) {
           continue;
        }
        b[i][j] = d;
        if (dfs(b,i,j + 1)) {
           return true;
        }
        b[i][j] = 0;
     }
     return false;
  }
  bool valid(vector < vector < int >> &b, int i,int j,int d) { //检测 b[i][j]是否能够放置 d
```

```
            for (int x = 0;x < 9;x++) {                          //检测同列是否有 d
                if (b[x][j] == d) {
                    return false;
                }
            }
            for (int y = 0;y < 9;y++) {                          //检测同行是否有 d
                if (b[i][y] == d) {
                    return false;
                }
            }
            int row = i - i % 3,col = j - j % 3;                 //大格的左上角(row,col)
            for (int x = 0;x < 3;x++) {                          //检测大格是否有 d
                for (int y = 0;y < 3;y++) {
                    if (b[row + x][col + y] == d) {
                        return false;
                    }
                }
            }
            return true;
        }
    };
```

上述程序提交后通过，执行用时为 81ms，内存消耗为 5.49MB。

解法 2：采用一维序号，将 9×9 的棋盘方格按行/列编号为 0～80，即(i,j)的一维序号 $k=9i+j$，反过来，$i=k/9$，$j=k\%9$。i 从 0 开始搜索，当 $i\geqslant 81$ 时得到一个解，存放在 ans 中，最后置 b=ans(如果不做这样的置换，返回时 b 仍然为初始值)。对应的程序如下：

```
class Solution {
    vector < vector < int >> ans;
    bool flag;
public:
    void solveSudoku(vector < vector < int >> &b) {
        flag = false;
        dfs(b,0);
        b = ans;
    }
    void dfs(vector < vector < int >> &b,int k) {
        if(k > = 81) {
            ans = b;
            flag = true;
        }
        else if(!flag) {
            int i = k/9,j = k % 9;                               //序号 k 转换成行/列编号
            if(b[i][j]!= 0)
                dfs(b,k + 1);
            else {
                for(int d = 1;d < = 9;d++) {
                    if(valid(b,i,j,d)){
                        b[i][j] = d;
                        dfs(b,k + 1);
                        b[i][j] = 0;
                    }
                }
            }
        }
    }
    bool valid(vector < vector < int >> &b,int i,int j,int d) { //检测 b[i][j]是否能够放置 d
```

```
        for (int x = 0;x < 9;x++) {                          //检测同列是否有 d
            if (b[x][j] == d) {
                return false;
            }
        }
        for (int y = 0;y < 9;y++) {                          //检测同行是否有 d
            if (b[i][y] == d) {
                return false;
            }
        }
        int row = i - i % 3,col = j - j % 3;                 //大格的左上角(row,col)
        for (int x = 0;x < 3;x++) {                          //检测大格是否有 d
            for (int y = 0;y < 3;y++) {
                if (b[row + x][col + y] == d) {
                    return false;
                }
            }
        }
        return true;
    }
};
```

上述程序提交后通过,执行用时为 82ms,内存消耗为 5.45MB。

5.3 LintCode135——数字组合★★

问题描述:给定一个候选整数的集合 candidates 和一个目标值 target,所有整数(包括 target)都是正整数,设计一个算法求 candidates 中所有和为 target 的组合,在同一个组合中,candidates 中的某个整数出现的次数不限,返回的每一个组合内的整数必须是非降序的,返回的所有组合之间可以是任意顺序,解集不能包含重复的组合。例如,candidates=[2,3,6,7],target=7,答案是[[7],[2,2,3]]。要求设计如下成员函数:

```
vector < vector < int > > combinationSum(vector < int > &candidates, int target) { }
```

解法 1:采用完全背包的求解思路,首先对 candidates 数组递增排序,用 i 遍历该数组,每个元素有 3 种选择,即不选择 candidates[i]、选择 candidates[i]后下一次仍然选择 candidates[i]、选择 candidates[i]后下一次不再选择 candidates[i]。对应的程序如下:

```
class Solution {
public:
  set < vector < int >> myset;
  vector < int > x;
  vector < int > a;
  int t,n;
  vector < vector < int >> combinationSum(vector < int > &candidates,int target) {
      a = candidates;
      t = target;
      n = a.size();
      sort(a.begin(),a.end());
      dfs(0,0);
      vector < vector < int >> ans;
      for(auto e:myset)
          ans.push_back(e);
      return ans;
```

```
    }
    void dfs(int cursum,int i) {                    //回溯算法
      if(i>=n) {
         if(cursum==t)                              //找到一个解
            myset.insert(x);
      }
      else {
         dfs(cursum,i+1);                           //不选择 a[i]
         if(cursum+a[i]<=t) {                       //剪支
            x.push_back(a[i]);
            dfs(cursum+a[i],i);                     //选择 a[i],然后继续选择 a[i]
            x.pop_back();
         }
         if(cursum+a[i]<=t) {                       //剪支
            x.push_back(a[i]);
            dfs(cursum+a[i],i+1);                   //选择 a[i],然后选择下一个元素
            x.pop_back();
         }
      }
   }
};
```

上述程序提交后通过,执行用时为 41ms,内存消耗为 5.74MB。

解法 2:由 candidates 去重得到数组 a,利用《教程》中 5.2.2 节求幂集的解法 2 的思路,在 a 中求子集和为 target 的序列。对应的回溯法程序如下:

```
class Solution {
public:
   vector<vector<int>> ans;
   vector<int> x;
   vector<int> a;
   int n;
   vector<vector<int>> combinationSum(vector<int> &candidates,int target) {
      if(candidates.size()==0)
         return ans;
      sort(candidates.begin(),candidates.end());    //排序
      a.push_back(candidates[0]);
      for(int i=1;i<candidates.size();i++) {        //candidates 去重得到 a
         if(candidates[i]!=a.back())
            a.push_back(candidates[i]);
      }
      n=a.size();
      dfs(target,0);
      return ans;
   }
   void dfs(int rt,int i) {                         //回溯算法
      if(rt==0) {
         ans.push_back(x);
      }
      else {
         for (int j=i;j<n;j++) {
            if(rt<a[j]) continue;                   //剪支
            x.push_back(a[j]);
            dfs(rt-a[j],j);
            x.pop_back();
         }
      }
   }
};
```

上述程序提交后通过，执行用时为 40ms，内存消耗为 4.14MB。

5.4 LintCode1915——举重★★★

问题描述：奥利第一次来到健身房，她正在计算她能举起的最大重量。杠铃所能承受的最大重量为 maxCapacity($1 \leqslant \text{maxCapacity} \leqslant 10^6$)，健身房里有 n($1 \leqslant n \leqslant 42$)个杠铃片，第 i 个杠铃片的重量为 weights[i]($1 \leqslant \text{weights}[i] \leqslant 10^6$)。奥利现在需要选一些杠铃片加到杠铃上，使得杠铃的重量最大，但是所选的杠铃片的重量总和又不能超过 maxCapacity，请计算杠铃的最大重量是多少。例如，$n=3$，weights={1,3,5}，maxCapacity=7，答案是 6。要求设计如下成员函数：

```
int weightCapacity(vector<int> &weights, int maxCapacity) { }
```

解：该问题与简单装载问题类似，用 i 遍历 w(存放杠铃片的重量)，tw 表示当前选择的杠铃片的重量和，rw 表示剩余杠铃片的重量和($w[i+1]+\cdots+w[n-1]$)，左剪支是结束 $\text{tw}+w[i]>\text{W}$ 的左分支的搜索，右剪支是结束 $\text{tw}+\text{rw}-w[i] \leqslant \text{ans}$ 的右分支的搜索，当到达叶子结点时将 tw 的最大值存放在 ans 中，搜索完毕返回 ans 即可。采用回溯法求解的程序如下：

```
class Solution {
    int ans;                                          //存放答案
    vector<int> w;                                    //存放杠铃片的重量
    int W;
public:
    int weightCapacity(vector<int> &weights, int maxCapacity) {
        w = weights;
        W = maxCapacity;
        ans = 0;
        int rw = 0;
        for(int e:w) rw += e;
        dfs(0,rw,0);
        return ans;
    }
    void dfs(int tw, int rw, int i) {                 //回溯算法
        if(i >= w.size()) {
            ans = max(ans,tw);
        }
        else {
            if(tw + w[i]<= W)
                dfs(tw + w[i],rw - w[i],i + 1);
            if(tw + rw - w[i]> ans)
                dfs(tw,rw - w[i],i + 1);
        }
    }
};
```

上述程序提交后通过，执行用时为 41ms，内存消耗为 2.14MB。

5.5 LintCode680——分割字符串★★

问题描述：给定一个字符串 s，设计一个算法可以选择在一个字符或两个相邻字符之后拆分字符串，使字符串仅由一个字符或两个字符组成，求所有可能的结果。例如，s="123"，

答案是{{"1","2","3"},{"12","3"},{"1","23"}}。要求设计如下成员函数：

```
vector<vector<string>> splitString(string& s) { }
```

解：采用回溯法求解，对于字符串 s，用解向量 x 表示它的一个分割字符串，用 i 表示当前处理的字符的序号（i 从 0 开始），对应的选择操作有两种，一是分割出 $s[i]$，二是分割出 $s[i]s[i+1]$，当 $i=n$ 时得到一个分割字符串 x，将其添加到答案 ans 中，最后返回 ans 即可。对应的程序如下：

```
class Solution {
    vector<vector<string>> ans;
    vector<string> x;
public:
    vector<vector<string>> splitString(string& s) {
        dfs(s,0);
        return ans;
    }
    void dfs(string &s,int i) {                    //回溯算法
        if(i>s.size()) return;
        if(i==s.size()) ans.push_back(x);
        else {
            string str=s.substr(i,1);
            x.push_back(str);
            dfs(s,i+1);
            x.pop_back();
            str=s.substr(i,2);
            x.push_back(str);
            dfs(s,i+2);
            x.pop_back();
        }
    }
};
```

上述程序提交后通过，执行用时为 654ms，内存消耗为 16.66MB。

5.6 LintCode136——分割回文串★★

问题描述：给定字符串 s，设计一个算法将它分割成一些子串，使得每个子串都是回文串，返回所有可能的分割方案。不同方案之间的顺序可以是任意的。例如，s="aab"，答案是{{"aa","b"},{"a","a","b"}}，注意单个字符构成的字符串是回文。要求设计如下成员函数：

```
vector<vector<string>> partition(string &s) { }
```

解：利用《教程》中 5.2.2 节求幂集的解法 2 的思路进行分割。对应的回溯法程序如下：

```
class Solution {
    vector<vector<string>> ans;
    vector<string> x;
public:
    vector<vector<string>> partition(string &s) {
        if (s.empty()) return {};
        dfs(s,0);
        return ans;
    }
    void dfs(string& s,int i) {                    //回溯算法
```

```
        if (i == s.size()) {
            ans.push_back(x);
        }
        else {
            for (int j = i;j < s.size();j++) {
                string tmp = s.substr(i,j - i + 1);
                if (!isPali(tmp)) continue;
                x.push_back(tmp);
                dfs(s,j + 1);
                x.pop_back();
            }
        }
    }
    bool isPali(string& s) {
        int i = 0;
        int j = s.size() - 1;
        while(i < j) {
            if (s[i++]!= s[j -- ]) return false;
        }
        return true;
    }
};
```

上述程序提交后通过,执行用时为 553ms,内存消耗为 5.59MB。

5.7 LintCode816——旅行商问题★★★

问题描述:给定 n 个城市,编号为 1～n,城市之间的无向道路用三元组[A,B,C]表示,即城市 A 和城市 B 之间有一条成本是 C 的无向道路,全部道路用 roads 数组存放。设计一个算法求从城市 1 开始旅行所有城市的最小成本,一个城市只能通过一次,可以假设一定能够到达所有的城市。例如,n=3,roads={{1,2,1},{2,3,2},{1,3,3}},答案是 3,对应的最短路径为 1-> 2-> 3。要求设计如下成员函数:

```
int minCost(int n,vector < vector < int >> &roads) { }
```

解:为了简便,将 n 个城市的编号由 1～n 改为 0～n－1,这样起点城市为 0,题目是求从顶点 0 开始经过其他全部顶点并且每个顶点仅经过一次的最短路径长度。该问题与 TSP 问题类似,但有以下几点区别:

(1) 这里的路径不需要回到起点 0。

(2) 图中两个顶点之间的道路是无向的,但两个顶点之间可能存在多条道路,应该取最小成本。

(3) 仅需要求最短路径长度,不必求一条最短路径。

采用基于排列树的回溯法对应的程序如下:

```
class Solution {
    const int INF = 0x3f3f3f3f;
    vector < vector < int >> A;
    int ans;
    vector < int > x;
public:
    int minCost(int n,vector < vector < int >> &roads) {
        if(n == 1) return 0;
```

```
        A = vector < vector < int >>(n, vector < int >(n, INF));
        for(int i = 0;i < roads.size();i++) {          //建立邻接表 A
            int a = roads[i][0] - 1;
            int b = roads[i][1] - 1;
            int w = roads[i][2];
            A[a][b] = min(A[a][b],w);
            A[b][a] = A[a][b];
        }
        x = vector < int >(n);
        for(int i = 0;i < n;i++) x[i] = i;             //将 0～n - 1 添加到 x 中
        ans = INF;
        int d = 0;
        dfs(d,n,1);
        return ans;
    }
    void dfs(int d,int n,int i) {                      //回溯算法
        if(i > = n) {                                  //到达一个叶子结点
            ans = min(ans,d);
        }
        else {
            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]]!= INF) {         //若 x[i - 1]到 x[j]有边
                    if(d + A[x[i - 1]][x[j]]< ans) {   //剪支
                        swap(x[i],x[j]);
                        dfs(d + A[x[i - 1]][x[i]],n,i + 1);
                        swap(x[i],x[j]);
                    }
                }
            }
        }
    }
};
```

上述程序提交后通过，执行用时为 41ms，内存消耗为 5.57MB。

5.8 LeetCode784——字母大小写全排列★★

问题描述：给定一个含 $n(1 \leqslant n \leqslant 12)$ 个字符的字符串 s，s 中仅含小写字母、大写字母和数字，通过将字符串 s 中的每个字母转变大小写可以获得一个新的字符串，设计一个算法求可能得到的所有字符串集合，以任意顺序返回输出。例如，s ="a1b2"，答案是{"a1b2","a1B2","A1b2","A1B2"}。要求设计如下成员函数：

```
vector < string > letterCasePermutation(string s) { }
```

解：用 x 表示解向量(一个可能得到的字符串)，ans 存放答案。首先置 x 为 s。用 i 遍历 x，分为两种情况：

(1) 若 $x[i]$为数字，只有不转变一种选择。

(2) 若 $x[i]$为字母，有不转变和转变两种选择。

当 $i=n$ 时得到一个可能的字符串 x，将其添加到 ans 中，最后返回 ans 即可。对应的程序如下：

```
class Solution {
    vector < string > ans;
    string x;
public:
    vector < string > letterCasePermutation(string s) {
      x = s;
      dfs(0);
      return ans;
    }
    void dfs(int i) {                                    //回溯算法
      if (i == x.size())
         ans.push_back(x);
      else {
         dfs(i + 1);                                     //不转变
         if (isalpha(x[i])) {                            //s[i]是字母
            x[i] ^ = 32;                                 //大小写转换
            dfs(i + 1);
            x[i] ^ = 32;                                 //回溯(恢复)
         }
      }
    }
};
```

上述程序提交后通过,执行用时为 8ms,内存消耗为 10.5MB。

5.9 LeetCode1079——活字印刷★★

问题描述:给定一个含 $n(1\leqslant n\leqslant 7)$个活字字模的 s,其中每个字模上都刻有一个字母 $s[i]$,设计一个算法求可以印出的非空字母序列的数目,注意每个活字字模只能使用一次。例如,s="AAB",可能的序列为"A"、"B"、"AA"、"AB"、"BA"、"AAB"、"ABA"和"BAA",答案为 8。要求设计如下成员函数:

```
int numTilePossibilities(strings) { }
```

解法 1:用解向量 x 表示一个可以印出的非空字母序列,myset 存放所有可以印出的非空字母序列,为了达到去重(例如,s="AA"时避免在可以印出的非空字母序列中出现两个"AA")的目的,将 myset 设计为 set < string >类型的容器,用数组 used 表示每个位置的字母是否使用过。与其他回溯算法不同的是,这里每一个非空的 x 都是一个解。对应的程序如下:

```
class Solution {
  int ans = 0;
  vector < int > used;
  set < string > myset;                                  //用 myset 来去重
  string x;
public:
  int numTilePossibilities(string s) {
     used = vector < int >(s.size(),0);
     x = "";
     dfs(s);
     return ans;
  }
  void dfs(string &s){                                   //回溯算法
     if(x!= "") {
        if(myset.count(x) == 0) {
```

```
                myset.insert(x);
                ans++;
            }
        }
        for(int i = 0;i < s.size();i++) {
            if(used[i] == 0){
                x.push_back(s[i]);
                used[i] = 1;
                dfs(s);
                used[i] = 0;
                x.pop_back();
            }
        }
    }
};
```

上述程序提交后通过，执行用时为 116ms，内存消耗为 14.9MB。

解法 2：题目不必求所有可以印出的非空字母序列，仅需要求数目 ans(初始为 0)。采用回溯法，假设直接在 s 上操作得到可以印出的非空字母序列，每次变动则递增一次 ans。这样的关键是去重，这里采用对 s 排序的方法，用数组 used 表示每个位置的字母是否使用过，当处理字母 $s[i]$时跳过($i>0$ && $s[i]==s[i-1]$ && used$[i-1]==0$)的情况，从而保证原字符数组的前一个字符(例如'A')永远放在后一个相同字符(例如'A')的前面。对应的程序如下：

```
class Solution {
public:
    int ans = 0;
    vector < int > used;
    int numTilePossibilities(string s) {
        used = vector < int >(s.size(),0);
        sort(s.begin(),s.end());
        dfs(s);
        return ans;
    }
    void dfs(string &s){                                    //回溯算法
        for (int i = 0;i < s.size();i++) {
            if (i > 0 && s[i] == s[i - 1] && used[i - 1] == 0)
                continue;
            if(used[i] == 0) {
                used[i] = 1;
                ans++;
                dfs(s);
                used[i] = 0;
            }
        }
    }
};
```

上述程序提交后通过，执行用时为 4ms，内存消耗为 6MB。

5.10 LeetCode93——复原 IP 地址★★

问题描述：给定一个只包含数字的字符串 s($0\leqslant s.\text{length}\leqslant 3000$，$s$ 仅由数字组成)，用于表示一个 IP 地址，返回所有可能从 s 获得的有效 IP 地址，可以按任何顺序返回答案。有

效 IP 地址正好由 4 个整数(每个整数位于 0～255)组成,且不能含有前导零,整数之间用'.'分隔,如"0.1.2.201"和"192.168.1.1"是有效 IP 地址,但是"0.011.255.245"、"192.168.1.312"和"192.168@1.1"是无效 IP 地址。例如,s="010010",结果为{"0.10.0.10","0.100.1.0"}。要求设计如下成员函数:

```
vector<string> restoreIpAddresses(string s) { }
```

解:假设 s 中含 n 个数字符,用数组 x 存放一个 IP 地址的 4 个整数,用 i 遍历 s(初始 i=0 对应解空间中的根结点),cnt 累计找到的有效整数的个数。对于解空间中第 i 层的结点,考虑 $s[i]$的决策,剩余的数字个数为 $n-i$,若 $n-i>(4-\text{cnt})\times 3$,说明剩余的数字个数太多了,若 $n-i<4-\text{cnt}$,说明剩余的数字个数太少了。如果 cnt=4 且 $i=n$,说明找到一个解 x,将 x 转换为 IP 字符串 tmp 添加到结果 ans 中。

在其他情况下,若遇到 $s[i]$='0',由于 IP 中的各个整数不能有前导零,那么这段 IP 地址只能为 0;否则扩展 $s[i..i+2]$的每个位置 j 作为分割点,求出对应的整数 d,若 d 有效则作为 IP 地址的一段,从 $j+1$ 开始继续向下搜索,若 d 无效则返回。对应的程序如下:

```
class Solution {
    vector<string> ans;                    //存放答案
    int x[4];
public:
    vector<string> restoreIpAddresses(string s) {
        dfs(s,0,0);
        return ans;
    }
    void dfs(string& s,int cnt,int i) {    //回溯算法
        int n=s.size();
        if(n-i>(4-cnt)*3)                  //找到 4 段 IP 地址并且 s 遍历完
            return;
        if(n-i<(4-cnt))
            return;
        if (cnt==4 && i==n) {
            string tmp="";
            for(int j=0;j<4;j++) {
                tmp+=to_string(x[j]);
                if(j!=3) tmp+='.';
            }
            ans.push_back(tmp);
        }
        else {
            if (s[i]=='0') {               //不能有前导零,若当前为'0',则这段 IP 地址只能为 0
                x[cnt]=0;
                dfs(s,cnt+1,i+1);
            }
            int d=0;
            for (int j=i;j<min(i+3,n);j++) {
                d=d*10+(s[j]-'0');
                if (d>0 && d<=255) {
                    x[cnt]=d;
                    dfs(s,cnt+1,j+1);
                }
                else return;               //d 无效时回溯
            }
        }
    }
};
```

上述程序提交后通过，执行用时为 4ms，内存消耗为 6.6MB。

5.11 LeetCode22——括号的生成★★

问题描述：给定一个整数 n（$1 \leqslant n \leqslant 8$）表示要生成括号的对数，设计一个算法求生成的所有可能的并且有效的括号组合。例如，$n=3$，答案为{"((()))","(()())","(())()","()(())","()()()"}。要求设计如下成员函数：

```
vector<string> generateParenthesis(int n) { }
```

解：用 x 表示当前解向量，从空串开始最多添加 $2n$ 个括号，$x[i]$要么选择'('，要么选择')'，用 left 累计'('的个数，用 right 累计')'的个数。对于解空间中的某个结点，左分支对应选择'('，右分支对应选择')'，叶子结点是满足条件 $x.\text{size}()=2n$ 的结点，若叶子结点同时满足 left==n && right==n，则 x 是一个有效括号串，将其添加到 ans 中。采用的剪支操作是终止满足 right > left || left > n || right > n 条件的结点继续扩展。对应的回溯法程序如下：

```
class Solution {
    vector<string> ans;                    //存放全部结果串
    string x;                              //解向量(一个有效括号串)
public:
    vector<string> generateParenthesis(int n) {
        if(n == 1)
            ans.push_back("()");
        else{
            x = "";
            dfs(n,0,0);
        }
        return ans;
    }
    void dfs(int n,int left,int right) { //回溯算法
        if (right > left || left > n || right > n)     //剪支
            return;
        if (x.size() == n * 2 && left == n && right == n) {
            ans.push_back(x);              //找到一个有效括号串，添加到 ans 中
        }
        else {
            x.push_back('(');              //选择'('
            dfs(n,left + 1,right);
            x.pop_back();                  //回溯
            x.push_back(')');              //选择')'
            dfs(n,left,right + 1);
            x.pop_back();                  //回溯
        }
    }
};
```

上述程序提交后通过，执行用时为 4ms，内存消耗为 11.5MB。

5.12 LeetCode89——格雷编码★★

问题描述：格雷编码是一个二进制数字系统，n 位格雷编码序列是一个由 2^n 个整数组成的序列。

(1) 每个整数都在[0,2^n-1]的范围内(含 0 和 2^n-1)。

(2) 第一个整数是 0。

(3) 一个整数在序列中的出现不超过一次。

(4) 每对相邻整数的二进制表示恰好一位不同,且第一个和最后一个整数的二进制表示恰好一位不同。

给定一个整数 $n(1\leqslant n\leqslant 16)$,设计一个算法求一个有效的 n 位格雷编码序列。例如,$n=2$ 时答案为{0,1,3,2}或者{0,2,3,1},前者的二进制数是{00,01,11,10},后者的二进制数是{00,10,11,01}。要求设计如下成员函数:

```
vector<int> grayCode(int n) { }
```

解:用 ans 存放 n 位格雷编码序列,将 g 初始为 0(n 位格雷编码序列的第一个整数是 0),也就是将 g 看成 n 位二进制数,二进制位 0～二进制位 $n-1$ 均为 0,在此基础上做 0/1 翻转。例如,$n=3$ 时,3 位格雷编码序列的产生过程如图 5.1 所示,答案为{0,1,3,7,5,2,6,4}。i 对应 g 的二进制起始位,j 从 i 到 $n-1$ 位翻转,每一个结点值都是 ans 的一个元素,其顺序恰好是深度优先搜索顺序,最后返回 ans 即可。对应的程序如下:

```
class Solution {
    vector<int> ans;
public:
    vector<int> grayCode(int n) {
        ans.push_back(0);
        dfs(0,0,n);
        return ans;
    }
    void dfs(int g,int i,int n) {          //依次翻转 g 的第 i 到 n-1 位
        if (i>=n) return;
        for (int j=i;j<n;j++) {
            int k=g^(1<<j);                //翻转 g 的二进制位 j 得到 k
            ans.push_back(k);
            dfs(k,j+1,n);                  //k 自动回溯
        }
    }
};
```

上述程序提交后通过,执行用时为 12ms,内存消耗为 12.2MB。

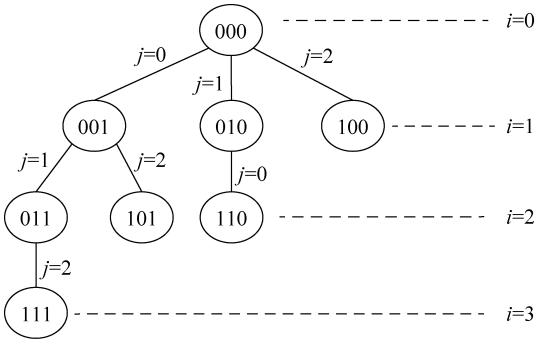


图 5.1 3 位格雷编码序列的产生过程

5.13 LeetCode301——删除无效的括号★★★

问题描述：给定一个由 $n(1\leqslant n\leqslant 25)$ 个括号和字母组成的字符串 s，删除最小数量的无效括号，使得输入的字符串有效，设计一个算法求所有可能的结果，答案可以按任意顺序返回。例如，s="(a)())()"，答案是{"(a())()","(a)()()"}。要求设计如下成员函数：

```
vector<string> removeInvalidParentheses(string s) { }
```

解：若一个字符串中的括号匹配，必须满足两点，一是任意前缀中右括号的个数不少于左括号的个数，二是总的左、右括号个数相同。采用回溯法求解。首先通过遍历 s 求出剩余左、右括号的个数 left 和 right(初始均为 0)，其过程是在遍历中遇到左括号时 left 增 1，遇到右括号时如果 left≠0 则将 right 减 1，否则 right 增 1。

再尝试遍历所有可能的去掉非法括号的方案，即尝试在原字符串 s 中去掉 left 个左括号和 right 个右括号，然后检测剩余的字符串是否为合法匹配，如果是合法匹配，则得到一个可能的结果。需要注意的是，这样得到的结果字符串可能存在重复，可以利用哈希集合实现去重。对应的程序如下：

```
class Solution {
public:
  set<string> ans;                                    //存放结果字符串
  vector<string> removeInvalidParentheses(string s) {
    int left=0;
    int right=0;
    for(int i=0;i<s.size();i++) {
       if (s[i]=='(') left++;
       else if (s[i]==')') {
          if (left==0) right++;
          else left--;
       }
    }
    dfs(s,0,left,right);
    vector<string> anss;
    for(auto e:ans) anss.push_back(e);
    return anss;
  }
  void dfs(string s,int i,int left,int right) {       //回溯算法
    if (left==0 && right==0) {
       if (valid(s)) ans.insert(s);                   //自动去重
    }
    else {
       for (int j=i;j<s.size();j++) {
          if (left+right>s.size()-j) return;          //剪支
          if (left>0 && s[j]=='(')                    //尝试去掉一个左括号
             dfs(s.substr(0,j)+s.substr(j+1),j,left-1,right);
          if (right>0 && s[j]==')')                   //尝试去掉一个右括号
             dfs(s.substr(0,j)+s.substr(j+1),j,left,right-1);
       }
    }
  }
  bool valid(string& s) {                             //判断 s 中的括号是否匹配
    int cnt=0;
```

```
        for (int i = 0;i < s.size();i++) {
            if (s[i] == '(') cnt++;
            else if(s[i] == ')') {
                cnt -- ;
                if (cnt < 0) return false;
            }
        }
        return cnt == 0;
    }
};
```

上述程序提交后通过,执行用时为 36ms,内存消耗为 8.3MB。可以改为这样去重,在每次进行搜索时,如果遇到连续相同的括号只需要搜索一次即可。例如当前遇到的字符串为"((((()))",去掉前 4 个左括号中的任意一个生成的结果字符串是一样的,均为"(((()))",因此在这样的情况下只需要去掉一个左括号后进入下一轮搜索,不需要将前 4 个左括号都尝试一遍。对应的程序如下:

```
class Solution {
public:
    vector < string > ans;                                  //存放答案
    vector < string > removeInvalidParentheses(string s) {
        int left = 0;
        int right = 0;
        for(int i = 0;i < s.size();i++) {
            if (s[i] == '(') left++;
            else if (s[i] == ')') {
                if (left == 0) right++;
                else left -- ;
            }
        }
        dfs(s,0,left,right);
        return ans;
    }
    void dfs(string s,int i,int left,int right) {          //回溯算法
        if (left == 0 && right == 0) {
            if (valid(s))
                ans.push_back(s);
        }
        else {
            for (int j = i;j < s.size();j++) {
                if (j > i && s[j] == s[j - 1])              //去重
                    continue;
                if (left + right > s.size() - j)            //剪支
                    return;
                if (left > 0 && s[j] == '(')                //尝试去掉一个左括号
                    dfs(s.substr(0,j) + s.substr(j + 1),j,left - 1,right);
                if (right > 0 && s[j] == ')')               //尝试去掉一个右括号
                    dfs(s.substr(0,j) + s.substr(j + 1),j,left,right - 1);
            }
        }
    }
    bool valid(string& s) {                                 //判断 s 中的括号是否匹配
        int cnt = 0;
        for (int i = 0;i < s.size();i++) {
            if (s[i] == '(') cnt++;
            else if(s[i] == ')') {
                cnt -- ;
```

```
                if (cnt < 0) return false;
            }
        }
        return cnt == 0;
    }
};
```

上述程序提交后通过，执行用时为 4ms，内存消耗为 7.4MB。

5.14 POJ3050——跳房子

时间限制：1000ms，空间限制：65 536KB。

问题描述：奶牛玩跳房子游戏，不是要跳入一组线性编号的方格，而是跳入一个 5×5 的网格，其中每个方格有一个数字。奶牛熟练地跳到网格中的任何数字上，并向前、向后、向右或向左(不能斜向)跳到网格中的另一个数字，这样跳过 6 个方格，这些方格的数字合并起来得到一个 6 位数的整数(可能有前导零，例如 000201)。求可以用这种方式创建的不同整数的数量。

输入格式：共 5 行，每行 5 个整数表示一个网格。

输出格式：输出可以创建的不同整数的数量。

输入样例：

```
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 2 1
1 1 1 1 1
```

输出样例：

```
15
```

解：网格中的每个方格最多有上、下、左、右相邻方格，从网格中的每个位置出发搜索，走 5 步得到一个整数，将其添加到 ans 中，由于添加的整数可能重复，需要去重，为此将 ans 设计为 set 容器。对应的程序如下：

```
#include<iostream>
#include<set>
using namespace std;
int a[5][5];
set<int> ans;                                    //存放结果整数
int dx[4]={-1, 1, 0, 0};
int dy[4]={0, 0, -1, 1};
void dfs(int x,int y,int k,int d) {              //回溯算法
   if(k==6) {
      ans.insert(d);
   }
   else {
      for(int i=0;i<4;i++) {
       int nx=x+dx[i];
         int ny=y+dy[i];
         if(nx>=0 && nx<5 && ny>=0 && ny<5) {
            k++;
            d=d*10+a[nx][ny];
```

```
                dfs(nx,ny,k,d);
                d = (d - a[nx][ny])/10;                    //回溯
                k -- ;
            }
        }
    }
}
int main() {
    for(int i = 0;i < 5;i++) {
        for(int j = 0;j < 5;j++)
            scanf(" % d", &a[i][j]);
    }
    for(int i = 0;i < 5;i++) {                             //将每个位置作为起点试探
        for(int j = 0;j < 5;j++)
            dfs(i,j,1,a[i][j]);
    }
    printf(" % d\n", ans.size());
    return 0;
}
```

上述程序提交后通过，执行用时为 32ms，内存消耗为 448KB。

5.15 POJ1724——道路

时间限制：1000ms，空间限制：65 536KB。

问题描述：编号为 1～N 的 N 个城市之间通过单向道路相连，每条道路都有两个与之相关的参数，即道路长度和过路费。现在 Bob 在城市 1，Alice 在城市 N，Bob 有 K 元钱，请帮助 Bob 选择可以找到 Alice 并且能够负担的最短路径。

输入格式：输入的第一行包含整数 K（$0 \leqslant K \leqslant 10\,000$），第二行包含整数 N（$2 \leqslant N \leqslant 100$），表示城市总数，第三行包含整数 R（$1 \leqslant R \leqslant 10\,000$），表示道路总数。以下 R 行中的每一行由单个空格分隔的整数 S、D、L 和 T 来表示一条道路，其中 S 是源城市（$1 \leqslant S \leqslant N$），$D$ 是目的地城市（$1 \leqslant D \leqslant N$），$L$ 是道路长度（$1 \leqslant L \leqslant 100$），$T$ 是过路费（$0 \leqslant T \leqslant 100$），注意不同的道路可能具有相同的源城市和目的地城市。

输出格式：输出一行，包含从城市 1 到城市 N 的最短路径的总长度，其总费用小于或等于 K。如果这样的路径不存在，则输出－1。

输入样例：

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

输出样例：

```
11
```

解：由输入数据建立图的邻接表存储结构，用 ans 存放答案(初始为∞)。从顶点 1 开始搜索，curlen 和 curcost 分别表示从顶点 1 到达当前顶点 s 的最短路径长度和费用，找到顶点 s 的相邻顶点 v，若顶点 v 已经访问，或者 curcost+<s,v>边的费用>K 或者 curlen+<s,v>边长度>=ans 时均跳过顶点 v 的试探，否则从顶点 v 出发继续搜索，当 $s=N$ 时说明找到了一条从顶点 1 到达顶点 N 的路径，置 ans=min(ans,curlen)。最后返回 ans 即可。对应的回溯算法如下：

```
#include<iostream>
#include<vector>
#include<cstring>
using namespace std;
const int MAXN=110;
const int INF=0x3f3f3f3f;
struct Edge {                                   //边类型
    int v;
    int len;
    int cost;
    int next;
};
int head[MAXN];                                 //图的邻接表
Edge edg[100*MAXN];
int cnt;
int K,N,R;
int visited[MAXN];
int ans;                                        //存放答案
int curlen,curcost;
void addedge(int S,int D,int L,int T) {         //增加一条边
    edg[cnt].v=D;
    edg[cnt].len=L;
    edg[cnt].cost=T;
    edg[cnt].next=head[S];
    head[S]=cnt++;
}
void dfs(int s) {                               //回溯算法
    if(s==N) {
        ans=min(ans,curlen);                    //更新步数最小值
    }
    else {
        for(int j=head[s];j!=-1;j=edg[j].next) {
            int v=edg[j].v;
            int len=edg[j].len;
            int cost=edg[j].cost;
            if(visited[v]==1) continue;
            if(curcost+cost>K) continue;        //总费用剪支
            if(curlen+len>=ans) continue;       //路径长度剪支
            curlen+=len;
            curcost+=cost;
            visited[v]=1;
            dfs(v);
            visited[v]=0;
            curcost-=cost;
            curlen-=len;
        }
    }
}
int main() {
```

```
    scanf("%d%d%d",&K,&N,&R);
    int S,D,L,T;
    memset(head,0xff,sizeof(head));
    cnt = 0;
    for(int i = 0;i < R;i++) {
        scanf("%d%d%d%d",&S,&D,&L,&T);
        addedge(S,D,L,T);
    }
    memset(visited,0,sizeof(visited));
    ans = INF;
    curlen = 0; curcost = 0;
    dfs(1);
    if(ans == INF)
        printf("-1\n");
    else
        printf("%d\n",ans);
    return 0;
}
```

上述程序提交后通过，执行用时为 63ms，内存消耗为 272KB。

5.16 POJ1699——最佳序列

时间限制：1000ms，空间限制：10 000KB。

问题描述：基因是由 DNA 组成的，组成 DNA 的核苷酸碱基是 A(腺嘌呤)、C(胞嘧啶)、G(鸟嘌呤)和 T(胸腺嘧啶)。给定一个基因的几个片段，要求从它们构造一个最短序列，该序列应使用所有段，并且不能翻转任何片段。

例如，给定"TCGG"、"GCAG"、"CCGC"、"GATC"和"ATCG"，可以采用如图 5.2 所示的方式滑动基因片段构造出一个长度为 11 的序列，它是最短序列(但可能不是唯一的)。

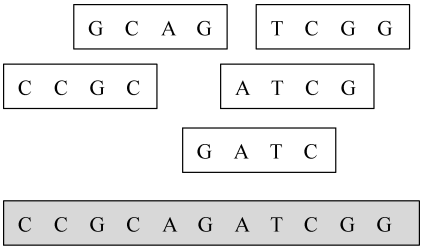


图 5.2 构造最短序列的过程

输入格式：第一行是一个整数 $T(1 \leqslant T \leqslant 20)$，表示测试用例的数量。然后是 T 个测试用例。每个测试用例的第一行包含一个整数 $n(1 \leqslant n \leqslant 10)$，它表示基因片段的数量，下面的 n 行分别表示 n 个基因片段，假设基因片段的长度为 1～20。

输出格式：对于每个测试用例，输出一行包含可以从这些基因片段构造的最短序列的长度。

输入样例：

```
1
5
TCGG
GCAG
CCGC
GATC
ATCG
```

输出样例：

```
11
```

解法 1：题目是求 n 个基因片段去掉最大相同前、后缀重叠部分后得到的字符串的最小长度。用 ans 存放答案，字符串数组 gen 存放 n 个基因片段，glen 数组存放 n 个基因片段的长度，设计二维数组 addlen，其中 addlen[i][j]表示基因片段 i 合并基因片段 j（去掉基因片段 i 的后缀和基因片段 j 的前缀的最大重叠部分）得到的字符串相对基因片段 i 增加的长度。例如，gen[i]="TCGG"，gen[j]="GCAG"，前者的后缀和后者的前缀的最大相同部分是"G"，两者合并后为"TCGGCAG"，是在 gen[i]的基础上增加了 3 个字符，所以 addlen[i][j]=3，注意 addlen 数组不是对称的。

采用基于子集树框架的回溯算法，从每个基因开始搜索，通过比较得到合并基因序列，求出最小长度 ans 并且输出。对应的程序如下：

```
#include<iostream>
#include<cstring>
using namespace std;
const int INF=0x3f3f3f3f;
char gen[11][21];
int glen[11];
int addlen[11][11];
int ans;
int used[11];
int n;
void dfs(int pre,int curlen,int step) {          //回溯算法(子集树)
    if(curlen>=ans)                              //剪支
        return;
    if(step==n) {
        ans=min(ans,curlen);
    }
    else {
        for(int j=0;j<n;j++) {
            if(used[j]==0) {
                used[j]=1;
                dfs(j,curlen+addlen[pre][j],step+1);
                used[j]=0;
            }
        }
    }
}
void add(int x,int y) {                          //求 addlen 数组
    int length=0;
    for(int i=1;i<=glen[x]&&i<=glen[y];i++) {
        bool flag=true;
        for(int j=glen[x]-i,k=0;k<i;k++,j++) {
            if(gen[x][j]!=gen[y][k]) {
                flag=false;
                break;
            }
        }
        if(flag) length=i;
    }
    addlen[x][y]=glen[y]-length;
}
int main() {
    int T;
    cin>>T;
    while(T--) {
```

```
        cin >> n;
        for(int i = 0;i < n;i++) {
            cin >> gen[i];
            glen[i] = strlen(gen[i]);
            used[i] = 0;
        }
        for(int i = 0;i < n;i++) {
            for(int j = 0;j < n;j++) {
                if(i!= j) add(i,j);
            }
        }
        ans = INF;
        for(int i = 0;i < n;i++) {                    //从每个基因片段 i 开始求解
            memset(used,0,sizeof(used));
            used[i] = 1;
            dfs(i,glen[i],1);
            used[i] = 0;
        }
        cout << ans << endl;
    }
    return 0;
}
```

上述程序提交后通过，执行用时为 47ms，内存消耗为 172KB。

解法 2：n 个基因片段的编号为 $0 \sim n-1$，将每个基因看成一个顶点，addlen[i][j]表示顶点 i 到顶点 j 的权值，这样构成一个带权有向图，求经过全部顶点的最短路径的长度（路径的首结点值为该基因片段的长度）。类似于 TSP 问题，采用基于排列树框架的回溯法程序如下：

```
#include < iostream >
#include < cstring >
using namespace std;
const int INF = 0x3f3f3f3f;
char gen[11][21];
int glen[11];
int addlen[11][11];
int ans;
int n;
int x[11];                                            //解向量
void dfs(int curlen,int i) {                          //回溯算法(排列树)
    if (i > = n) {                                    //到达一个叶子结点
        ans = min(ans,curlen);
    }
    else {
        for (int j = i;j < n;j++) {
            swap(x[i],x[j]);                          //交换 x[i]与 x[j]
            if(i == 0) {
                dfs(glen[x[0]],i + 1);
            }
            else if(curlen + addlen[x[i - 1]][x[i]]< = ans) {//剪支
                dfs(curlen + addlen[x[i - 1]][x[i]],i + 1);
            }
            swap(x[i],x[j]);                          //交换 x[i]与 x[j]:恢复
        }
    }
}
void add(int x,int y) {                               //求 addlen 数组
```

```
    int length = 0;
    for(int i = 1;i <= glen[x] && i <= glen[y];i++) {
        bool flag = true;
        for(int j = glen[x] - i,k = 0;k < i;k++,j++) {
            if(gen[x][j]!= gen[y][k]) {
                flag = false;
                break;
            }
        }
        if(flag) length = i;
    }
    addlen[x][y] = glen[y] - length;
}
int main() {
    int T;
    cin >> T;
    while(T-- ) {
        cin >> n;
        for(int i = 0;i < n;i++) {
            cin >> gen[i];
            glen[i] = strlen(gen[i]);
        }
        for(int i = 0;i < n;i++) {
            for(int j = 0;j < n;j++)
                if(i!= j) add(i,j);
        }
        for(int i = 0;i < n;i++)
            x[i] = i;
        ans = INF;
        dfs(0,0);
        cout << ans << endl;
    }
    return 0;
}
```

上述程序提交后通过，执行用时为 32ms，内存消耗为 172KB。

5.17 POJ1564——求和

时间限制：1000ms，空间限制：10 000KB。

问题描述：给定一个指定的总和 t 和一个包含 n 个整数的序列，求该序列中加起来和为 t 的不同子序列的个数。例如，$t=4$，$n=6$，序列为{4，3，2，2，1，1}，则有 4 个不同子序列之和等于 4，即 4、3+1、2+2 和 2+1+1。一个整数在序列中出现的次数可以与它在子序列中出现的次数一样多，并且单个数字算作总和。

输入格式：输入包含一个或多个测试用例，每行一个测试用例，每个测试用例包含 t（总数），后跟 n（序列中的整数个数），然后是 n 个整数 x_1、……、x_n。如果 $n=0$，则表示输入结束。否则，t 为小于 1000 的正整数，n 为 1～12（含）的整数，x_1、……、x_n 是小于 100 的正整数，所有整数之间由一个空格分隔。每个序列中的整数以非递增顺序出现，并且可能有重复。

输出格式：对于每个测试用例，首先输出一行包含"Sums of"、总数和一个冒号，然后每行输出一个和式子，如果没有（总数为 0）则输出"NONE"的行。每个和式子中的数字必须以非递增顺序出现，一个整数在和式子中的重复次数可能与在原始序列中的重复次数一样

多，和式子本身必须根据出现的整数按降序排序，换句话说，和式子必须按其第一个整数排序，具有相同第一个整数的和式子必须按其第二个整数排序，以此类推。在每个测试用例中，所有的和式子必须是不同的。

输入样例：

```
4 6 4 3 2 2 1 1
5 3 2 1 1
400 12 50 50 50 50 50 50 25 25 25 25 25 25
0 0
```

输出样例：

```
Sums of 4:
4
3 + 1
2 + 2
2 + 1 + 1
Sums of 5:
NONE
Sums of 400:
50 + 50 + 50 + 50 + 50 + 50 + 25 + 25 + 25 + 25
50 + 50 + 50 + 50 + 50 + 25 + 25 + 25 + 25 + 25 + 25
```

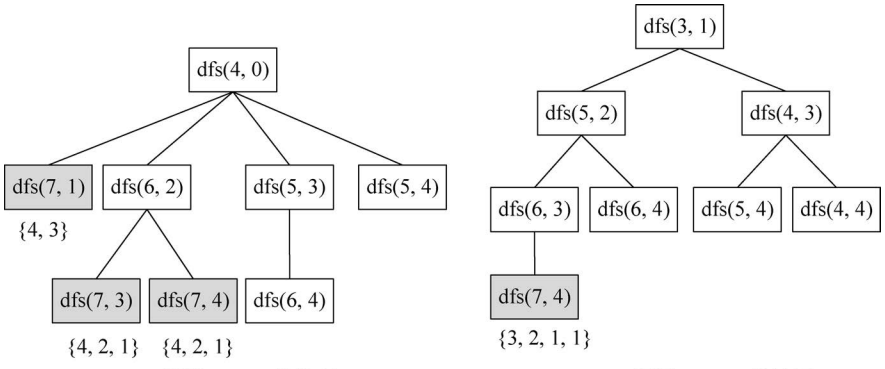
解法 1：对于每个测试用例，用 ans 存放答案，其中每个元素为 vector < int >类型，存放一个和为 t 的子序列，由于最后的结果需要去重，所以将 ans 设计为 set 容器，set 中默认是递增排列，题目要求每个和式子中的整数必须以非递增顺序出现，所以得到 ans 后反向输出 ans 中的各个和式子即可。求和式子的过程与求幂集类似，用 x 作为解向量，解空间中根结点对应 $i=0$，当 $i=n$ 时判断当前子序列和 cursum = t 是否成立，若成立将 x 添加到 ans 中。对应的回溯法程序如下：

```
#include < iostream >
#include < vector >
#include < set >
using namespace std;
int t,n;
int a[20];
set < vector < int >> ans;
vector < int > x;                                    //解向量
void dfs(int cursum,int i) {                         //回溯算法
   if(i > = n) {                                     //到达一个叶子结点
      if(cursum == t) ans.insert(x);
   }
   else {
      if(cursum + a[i]< = t) {                       //左剪支
         x.push_back(a[i]);
         dfs(cursum + a[i],i + 1);
         x.pop_back();
      }
      dfs(cursum,i + 1);                             //不选择 a[i]
   }
}
int main() {
   while(scanf("%d%d",&t,&n)!= EOF) {
      if(!n) break;
      for(int i = 0;i < n;i++)
```

```
            scanf("%d",&a[i]);
        ans.clear();
        printf("Sums of %d:\n",t);
        dfs(0,0);
        if(ans.size()==0)
            printf("NONE\n");
        else {
            set<vector<int>>::reverse_iterator rit;
            for(rit=ans.rbegin();rit!=ans.rend();rit++) {
                vector<int> e=*rit;
                for(int i=0;i<e.size();i++) {
                    if(i!=0) printf("+");
                    printf("%d",e[i]);
                }
                printf("\n");
            }
        }
    }
}
```

上述程序提交后通过，执行用时为0ms，内存消耗为128KB。

解法2：定义大问题f(cursum,i)是选择$a[i]$后(cursum=$a[i]$)在$a[i+1..n-1]$中选择若干整数得到所有以$a[i]$开头的和为t的子序列。求解过程是对于$i+1\sim n-1$内的整数j，当cursum+$a[j]\leqslant t$时，小问题是f(cursum+$a[j]$,j)。递归出口是$i\geqslant n$或者cursum=t。例如，$t=7$，$n=5$，$a[]=\{4,3,2,1,1\}$，调用dfs(4,0)和dfs(3,1)的过程如图5.3所示，得到4个解，其他调用没有解，这样的4个解中有两个存放，去重的结果是4+3、4+2+1和3+2+1+1。在求解向量x时需要回溯。



(a) 调用dfs(4, 0)的结果　　(b) 调用dfs(3, 1)的结果

图5.3　调用dfs(4,0)和dfs(3,1)的过程

对应的回溯法程序如下：

```
#include<iostream>
#include<vector>
#include<set>
using namespace std;
int t,n;
int a[20];
set<vector<int>> ans;
vector<int> x;                                    //解向量
void dfs(int cursum,int i) {                      //回溯算法
```

```
    if(i >= n) return;
    if(cursum == t) {
        ans.insert(x);
    }
    else {
        for(int j = i + 1;j < n;j++){
            if(cursum + a[j]<= t) {
                x.push_back(a[j]);
                dfs(cursum + a[j],j);
                x.pop_back();
            }
        }
    }
}
int main() {
    while(scanf("%d%d",&t,&n)!= EOF) {
        if(!n) break;
        for(int i = 0;i < n;i++)
            scanf("%d",&a[i]);
        ans.clear();
        printf("Sums of %d:\n",t);
        for(int i = 0;i < n;i++) {
            x.clear();
            x.push_back(a[i]);
            dfs(a[i],i);
        }
        if(ans.size() == 0)
            printf("NONE\n");
        else {
            set < vector < int >>::reverse_iterator rit;
            for(rit = ans.rbegin();rit!= ans.rend();rit++) {
                vector < int > e = *rit;
                for(int i = 0;i < e.size();i++) {
                    if(i!= 0) printf("+");
                    printf("%d",e[i]);
                }
                printf("\n");
            }
        }
    }
}
```

上述程序提交后通过，执行用时为 0ms，内存消耗为 128KB。

5.18 POJ2245——组合

时间限制：1000ms，空间限制：65 536KB。

问题描述：给定由 k 个整数构成的集合 S，求其中所有 6 个整数的集合。

输入格式：输入包含一个或多个测试用例，每个测试用例由一行组成，其中包含多个用空格隔开的整数，第一个整数是 k（$6 < k < 13$），然后 k 个整数指定集合 S，按照升序排列。输入以 $k=0$ 终止。

输出格式：对于每个测试用例，输出 S 中所有 6 个整数的组合，每个组合按升序排列，全部的组合按字典序输出，每个测试用例输出后空一行，最后一个测试用例之后不空行。

输入样例：

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

输出样例：

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

解：用 a 存放输入的 k 个整数，设计回溯算法 dfs(cnt,i)表示从 $a[i]$开始选择 6 个整数，用 ans 存放选择的整数。当 cnt=6 时输出 ans。对应的程序如下：

```
#include<iostream>
using namespace std;
const int MAXN=14;
int a[MAXN],ans[6];
int k;
void dfs(int cnt,int i) {                          //回溯算法
    if(cnt==6) {                                   //已经组合了6个数就输出
        printf("%d",ans[0]);
        for(int j=1;j<6;j++)
            printf(" %d",ans[j]);
        printf("\n");
    }
    else {
        for(int j=i;j<k;j++) {
            ans[cnt]=a[j];
            dfs(cnt+1,j+1);
        }
    }
}
int main() {
    bool first=true;
    while(scanf("%d",&k)!=EOF && k) {
        for(int i=0;i<k;i++)
            scanf("%d",&a[i]);
        if(!first)
            printf("\n");                           //第一个测试之前不加空行,其他加空行
        first=false;
        dfs(0,0);
    }
    return 0;
}
```

上述程序提交后通过，执行用时为 16ms，内存消耗为 88KB。

5.19 POJ1321——棋盘问题

时间限制：1000ms，空间限制：10 000KB。

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：输入含有多组测试数据。每组数据的第一行是两个正整数 n 和 k（$n \leqslant 8$，$k \leqslant n$），用一个空格隔开，表示将在一个 $n \times n$ 的矩阵内描述棋盘，以及摆放棋子的数目。当输入为－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：对于每组测试数据，给出一行输出，输出摆放的方案数目 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：简化的皇后问题，采用回溯法求解，用二维数组 map 存放棋盘，used 数组中的 used[j]表示第 j 列是否已经放过棋子。对应的程序如下：

```
#include<iostream>
#include<cstring>
using namespace std;
#define MAXN 10
using namespace std;
char map[MAXN][MAXN];
bool used[MAXN];                                    //记录一列是否已经放过棋子
int n,k,ans;
void dfs(int i,int num) {                           //回溯算法
    if (num==k) {                                   //找到一个解 ans 增 1
        ans++;
    }
    else if(i<n) {
        for (int j=0;j<n;j++){                      //搜索行 i 的每个列 j
            if (map[i][j]=='#'&& used[j]==false) {
                used[j]=true;
                dfs(i+1,num+1);                     //在行 i 的列 j 放置一个棋子
                used[j]=false;
            }
        }
        dfs(i+1,num);                               //行 i 不放置任何棋子
    }
}
int main() {
    while(scanf("%d%d",&n,&k)) {
        if (n==-1&&k==-1) break;
        for(int i=0;i<n;i++)
            scanf("%s",map[i]);
        memset(used,false,sizeof(used));
        ans=0;
```

```
            dfs(0,0);
            printf("%d\n",ans);
        }
        return 0;
    }
```

上述程序提交后通过，执行用时为47ms，内存消耗为88KB。

5.20 POJ2488——骑士之旅

时间限制：1000ms，空间限制：65 536KB。

问题描述：骑士想要周游世界，可以将这个世界看成 p 列 q 行的棋盘，骑士只能向8个方向走“日”字，而且不能重复。问骑士可以在棋盘的任何方格上开始和结束吗？

输入格式：输入的第一行包含一个正整数 n，表示有 n 个测试用例。每个测试用例输入一行，包含两个正整数 p 和 q，表示一个 $p \times q$ 棋盘($1 \leqslant p \times q \leqslant 26$)。

输出格式：每个测试用例的输出以"Scenario #i:"开始，其中 i 是从1开始的测试用例编号，这里的路径是一条从起始位置经过棋盘中全部方格的路径，路径中的每个方格用“行字母＋列数字”表示，行字母为以"A"开始的 q 个字母，列数字是 $1 \sim p$，这样每条路径用一个序列表示，本题求一条按字典序排列最小的路径，如果不存在这样的路径，则输出"impossible"。每个测试用例输出之后空一行，最后一个测试用例的后面不空行。

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：题目中第3个测试用例的图示如图5.4所示，即棋盘的行号为'A'～'C'、列号为1～4。从任意一个位置出发经过棋盘中全部方格的路径可能有多条，这些路径按题目规定的路径表示，即路径序列也不同，答案是一条按字典序排列最小的路径。

	1	2	3	4
A	(1)	(4)	(7)	(10)
B	(8)	(11)	(2)	(5)
C	(3)	(6)	(9)	(12)

图5.4 第3个测试用例的图示

对于 p 列 q 行的棋盘，行号为'A'～'A'＋q－1、列号为1～p。为了保证找到的路径是按字典序排列最小的路径，必须从(A,1)位置开始搜索，另外，由于每个位置最多有8个相邻位置，所以搜索次序也应该遵循字典序，(x,y)位置的8个相邻位置如图5.5所示，按字典序建立的偏移量数组如下：

```
int dx[8] = { - 2, - 2, - 1, - 1,1,1,2,2};
int dy[8] = { - 1,1, - 2,2, - 2,2, - 1,1};
```

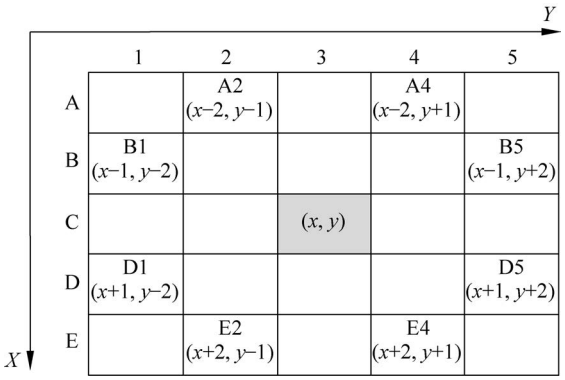


图 5.5 (x,y)位置的 8 个相邻位置

可以看成(x,y)位置的 8 个方位，在搜索时 di 依次从 0 到 7 试探。用 path 数组存放找到的第一条路径，step 表示路径长度或者 path 数组的下标，当 step＝p * q 时 path 即为所求，置 flag＝true 结束搜索过程。如果搜索完毕 flag 仍然为 false，则说明找不到按字典序排列最小的路径(注意并不表示找不到其他路径)。对应的程序如下：

```
#include < iostream >
#include < cstring >
using namespace std;
const int MAXN = 30;
int visited[MAXN][MAXN];
int p,q;
int dx[8] = { - 2, - 2, - 1, - 1, 1, 1, 2, 2};
int dy[8] = { - 1, 1, - 2, 2, - 2, 2, - 1, 1};
struct Path {
    int x,y;
} path[MAXN];
bool flag;
void dfs(int x,int y,int step) {                    //回溯算法
    if(flag) return;
    path[step].x = x;
    path[step].y = y;
    if(step == p * q) {
        flag = true;
    }
    else {
        for(int di = 0;di < 8;di++) {
            int nx = x + dx[di];
            int ny = y + dy[di];
            if(nx > = 1 && nx < = q && ny > = 1 && ny < = p && !visited[nx][ny]) {
                visited[nx][ny] = 1;
                dfs(nx,ny,step + 1);
                visited[nx][ny] = 0;
            }
        }
    }
}
int main() {
    int t;
```

```
    cin >> t;
    for(int cas = 1;cas < = t;cas++) {
        cin >> p >> q;
        memset(visited,0,sizeof(visited));
        flag = false;
        int x = 1,y = 1,step = 1;
        visited[x][y] = 1;
        dfs(x,y,step);
        cout << "Scenario #" << cas <<":" << endl;
        if(flag) {
            for(int i = 1;i < = p * q;i++) {
                cout << char(path[i].x - 1 + 'A') << path[i].y;
            }
            cout << endl;
        }
        else cout << "impossible" << endl;
        if(cas!= t) cout << endl;
    }
    return 0;
}
```

上述程序提交后通过,执行用时为 16ms,内存消耗为 180KB。