5-6　旅行售货员问题的上界函数。

设 G 是一个有 n 个顶点的有向图，从顶点 i 发出的边的最小费用记为 $\min(i)$。

（1）证明图 G 的所有前缀为 $x[1:i]$的旅行售货员回路的费用至少为：

$$\sum_{j=2}^{i} a(x_{j-1}, x_j) + \sum_{j=i}^{n} \min(x_j)$$

式中，$a(u, v)$是边(u, v)的费用。

（2）利用上述结论设计一个高效的上界函数，重写旅行售货员问题的回溯法，并与主教材中的算法进行比较。

答:

（1）证明

旅行售货员回路需经过所有顶点并返回起点。对于前缀 x[1:i]，已确定的边费用为 $\sum_{j=2}^{i} a(x_{j-1}, x_j)$。剩余部分需从 xi 出发，经过 n−i 个未完全访问的顶点。由于从每个顶点 xj 出发的边最小费用为 min(xj)，剩余路径费用至少为 $\sum_{j=i}^{n} \min(x_j)$（每个顶点至少贡献一条最小费用边）。因此，整个回路费用至少为 $\sum_{j=2}^{i} a(x_{j-1}, x_j) + \sum_{j=i}^{n} \min(x_j)$，得证。

（2）基于上述结论的回溯法设计与分析

上界函数设计：在回溯过程中，对当前扩展节点，计算 $c = \sum_{j=2}^{i} a(x_{j-1}, x_j) + \sum_{j=i}^{n} \min(x_j)$。若 c 大于当前已知最小回路费用（初始可设为一个较大值，后续更新），则剪枝该分支。

算法步骤:

初始化：计算每个顶点 i 的 min(i)，设初始上界 u=+∞。

回溯搜索：从起点开始，递归扩展路径。每扩展一步 xi，计算当前路径费用与剩余 $\sum_{j=i}^{n} \min(x_j)$ 之和 c。

剪枝判断：若 c≥u，跳过该分支；否则继续搜索。若完成一个回路且费用小于 u，更新 u。

与教材算法比较：该方法利用更紧的下界，能更早剪枝。教材中的回溯法若未采用此下界，可能搜索更多无效分支。此方法通过 $\sum_{j=i}^{n} \min(x_j)$ 快速评估剩余路径最小代价，减少不必要的搜索，提高效率。

5-1　子集和问题。

问题描述：子集和问题的一个实例为<S, c>。其中，$S=\{x_1, x_2, \cdots, x_n\}$是一个正整数的集合，$c$ 是一个正整数。子集和问题判定是否存在 S 的一个子集 $S1$，使得 $\sum_{x\in S1} x=c$。试设计一个解子集和问题的回溯法。

算法设计：对于给定的正整数的集合 $S=\{x_1, x_2, \cdots, x_n\}$和正整数 c，计算 S 的一个子集 S_1，使得 $\sum_{x\in S_1} x=c$。

数据输入：由文件 input.txt 提供输入数据。文件第 1 行有 2 个正整数 n 和 c，n 表示 S 的大小，c 是子集和的目标值。接下来的 1 行中，有 n 个正整数，表示集合 S 中的元素。

结果输出：将子集和问题的解输出到文件 output.txt。当问题无解时，输出“No Solution!”。

输入文件示例	输出文件示例
input.txt	output.txt
5 10	2 2 6
2 2 6 5 4	

答：思路是通过回溯法遍历子集树，尝试所有可能的子集组合，判断是否存在和为 c 的子集，符合子集和问题的求解要求。

回溯函数设计：

backtrack(int i, int sum)：i 是当前处理的元素索引，sum 是当前子集和。

若 sum == c，找到解，输出到 output.txt 并结束程序。

若 i >= n（超出范围）或 sum > c（和超过目标），回溯。

递归处理选择 S[i] 和不选择 S[i] 两种情况。

代码如下：

suanfa.cpp > backtrack(int, int)

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
using namespace std;

vector<int> subset;
int n, c;
vector<int> S;

void backtrack(int i, int sum) {
    if (sum == c) {
        ofstream out("output.txt");
        for (int num : subset) {
            out << num << " ";
        }
        out.close();
        exit(0);
    }
    if (i >= n || sum > c) return;

    // 选择 S[i]
    subset.push_back(S[i]);
    backtrack(i + 1, sum + S[i]);
    subset.pop_back();

    // 不选择 S[i]
    backtrack(i + 1, sum);
}

int main() {
    ifstream in("input.txt");
    in >> n >> c;
    S.resize(n);
    for (int i = 0; i < n; ++i) {
        in >> S[i];
    }
    in.close();

    sort(S.begin(), S.end());
    backtrack(0, 0);

    ofstream out("output.txt");
    out << "No Solution!";
    out.close();
    return 0;
}
```

5-3　最小重量机器设计问题。

问题描述：设某一机器由 n 个部件组成，每种部件都可以从 m 个不同的供应商处购得。设 w_{ij} 是从供应商 j 处购得的部件 i 的重量，c_{ij} 是相应的价格。试设计一个算法，给出总价格不超过 c 的最小重量机器设计。

算法设计：对于给定的机器部件重量和机器部件价格，计算总价格不超过 d 的最小重量机器设计。

数据输入：由文件 input.txt 给出输入数据。第一行有 3 个正整数 n、m 和 d。接下来的 $2n$ 行，每行 n 个数。前 n 行是 c，后 n 行是 w。

结果输出：将计算的最小重量及每个部件的供应商输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3 3 4	4
1 2 3	1 3 1
3 2 1	
2 2 2	
1 2 3	
3 2 1	
2 2 2	

答：思路：用回溯法，就是递归地选每个部件的供应商。对于每个部件 i，遍历 m 个供应商 j。如果选了 j 后，总价格不超过 d，就继续选下一个部件。记录当前的总重量和总价格。当处理完所有 n 个部件时，检查总价格是否符合条件，如果符合且重量更小，就更新最优解。

代码如下：

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <climits>

using namespace std;

int n, m, d;
vector<vector<int>> c_price;
vector<vector<int>> w_weight;
int min_weight = INT_MAX;
vector<int> best_choice;

void backtrack(int index, int total_cost, int total_weight, vector<int>& current_choice) {
    if (index == n) {
        if (total_cost <= d && total_weight < min_weight) {
            min_weight = total_weight;
            best_choice = current_choice;
        }
        return;
    }
    for (int j = 0; j < m; ++j) {
        if (total_cost + c_price[index][j] > d) continue;
        current_choice[index] = j + 1;
        backtrack(index + 1, total_cost + c_price[index][j], total_weight + w_weight[index][j], current_choice);
    }
}

int main() {
    ifstream in("input.txt");
    in >> n >> m >> d;
    c_price.resize(n, vector<int>(m));
    w_weight.resize(n, vector<int>(m));
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            in >> c_price[i][j];
        }
    }
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            in >> w_weight[i][j];
        }
    }
    in.close();

    vector<int> current_choice(n, 0);
    backtrack(0, 0, 0, current_choice);
```

```cpp

    ofstream out("output.txt");
    if (min_weight != INT_MAX) {
        out << min_weight << endl;
        for (int num : best_choice) {
            out << num << " ";
        }
    } else {
        out << "No Solution!"; // 理论上输入保证有解，此为保险
    }
    out.close();
    return 0;
}
```

5-6　无和集问题。

问题描述：设 S 是正整数集合。S 是一个无和集，当且仅当 $x, y \in S$ 蕴含 $x+y \notin S$。对于任意正整数 k，如果可将 $\{1, 2, \cdots, k\}$ 划分为 n 个无和子集 $S_1, S_2, \cdots, S_n$，则称正整数 k 是 n 可分的。记 $F(n)=\max\{k \mid k$ 是 n 可分的$\}$。试设计一个算法，对任意给定的 n，计算 $F(n)$的值。

算法设计：对任意给定的 n，计算 $F(n)$的值。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n。

结果输出：将计算的 $F(n)$的值以及 $\{1, 2, \cdots, F(n)\}$的一个 n 划分输出到文件 output.txt。文件的第 1 行是 $F(n)$的值。接下来的 n 行，每行是一个无和子集 S_i。

输入文件示例	输出文件示例
input.txt	output.txt
2	8
	1 2 4 8
	3 5 6 7

答：算法思想：从 1 开始回溯，建立两个两个二维数组，一个存放中间值，一个存放最终的结果。

dfs(1)　a[0][1]　a[0][0]++　t+1

dfs(2)　a[0][2]　a[0][0]++　t+1

dfs(3)　a[0][3]不满足，a[1][1]　a[1][0]++　t+1

dfs(4)　a[0][3]或者 a[1][2]都满足 回溯尝试，找到一个 k 最大的情况

代码如下：

```c
#include<stdio.h>

#define N 100

int F[N][N],answer[N][N];

int n,maxValue;

int judge(int t,int k){
    int i,j;
    for(i=1;i<=F[k][0];i++){
        for(j=i+1;j<=F[k][0];j++){
            if(F[k][i]+F[k][j]==t)
                return 0;
        }
    }
    return 1;

}

void dfs(int t){
    int i,j;
    if(t>maxValue){
        for(i=0;i<n;i++){
            for(j=0;j<=F[i][0];j++){
                answer[i][j] = F[i][j];
            }
        }
        maxValue = t;
    }

    for(i=0;i<n;i++){
        F[i][F[i][0]+1]=t;
        if(judge(t,i)){
            F[i][0]+=1;
            dfs(t+1);
            F[i][0]-=1;
        }
    }
}

```

5-13　工作分配问题。

问题描述：设有 n 件工作分配给 n 个人。将工作 i 分配给第 j 个人所需的费用为 c_{ij}。试设计一个算法，为每个人都分配 1 件不同的工作，并使总费用达到最小。

算法设计：设计一个算法，对于给定的工作费用，计算最佳工作分配方案，使总费用达到最小。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n（$1 \leqslant n \leqslant 20$）。接下来的 n 行，每行 n 个数，表示工作费用。

结果输出：将计算的最小总费用输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3	9
10 2 3	
2 3 4	
3 4 5	

答：思路：读取输入的 n 和费用矩阵 c。

初始化 minCost 为一个很大的数，比如 INT_MAX。

定义一个数组 used 来记录工作是否被分配，或者用 path 数组记录已分配的工作。

回溯函数，参数是当前处理的人的编号（比如 k），当前费用 sum，以及 path 数组。

在回溯函数中，当 k == n 时，更新 minCost。

否则，对于每个工作 i（0 到 n-1），如果未被分配，就分配给第 k 个人，递归 k+1，sum + c [k][i]。

代码如下：

suanfa.cpp > main()

```cpp
#include <iostream>
#include <vector>
#include <climits>
using namespace std;

void backtrack(int k, int sum, const vector<vector<int>>& c, vector<bool>& visited, int& minCost) {
    if (k == visited.size()) {
        if (sum < minCost) {
            minCost = sum;
        }
        return;
    }
    for (int i = 0; i < visited.size(); ++i) {
        if (!visited[i] && sum + c[k][i] < minCost) {
            visited[i] = true;
            backtrack(k + 1, sum + c[k][i], c, visited, minCost);
            visited[i] = false;
        }
    }
}

int main() {
    int n;
    cin >> n;
    vector<vector<int>> c(n, vector<int>(n));
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            cin >> c[i][j];
        }
    }
    int minCost = INT_MAX;
    vector<bool> visited(n, false);
    backtrack(0, 0, c, visited, minCost);
    cout << minCost << endl;
    return 0;
}
```

5-16　无优先级运算问题。

问题描述：给定 n 个正整数和 4 个运算符+、−、*、/，且运算符无优先级，如 2+3×5=25。对于任意给定的整数 m，试设计一个算法，用以上给出的 n 个数和 4 个运算符，产生整数 m，且用的运算次数最少。给出的 n 个数中每个数最多只能用 1 次，但每种运算符可以任意使用。

算法设计：对于给定的 n 个正整数，设计一个算法，用最少的无优先级运算次数产生整数 m。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m。第 2 行是给定的用于运算的 n 个正整数。

结果输出：将计算的产生整数 m 的最少无优先级运算次数以及最优无优先级运算表达式输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
5 25	2
5 2 3 6 7	2+3*5

答：算法思路：使用回溯法，尝试不同的数字组合和运算。每次选择两个数字进行四种运算中的一种，得到一个新的结果，然后将这个结果加入剩余数字中，继续递归。记录运算次数，找到最小的。

代码如下：

```cpp
// 检查是否可以通过运算得到 m
Operation backtrack(vector<int>& nums, int target, int currentCount, string currentExpr) {
    if (nums.size() == 1) {
        if (nums[0] == target) {
            return {currentExpr, currentCount};
        }
        return {"", INT_MAX};
    }

    Operation minOp = {"", INT_MAX};
    int n = nums.size();
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (i == j) continue;
            int num1 = nums[i];
            int num2 = nums[j];
            vector<int> newNums;
            for (int k = 0; k < n; ++k) {
                if (k != i && k != j) {
                    newNums.push_back(nums[k]);
                }
            }

            // 加法
            int sum = num1 + num2;
            newNums.push_back(sum);
            string newExpr = currentExpr.empty() ?
                "(" + to_string(num1) + "+" + to_string(num2) + ")" :
                currentExpr + "+" + to_string(num2);
            Operation op = backtrack(newNums, target, currentCount + 1, newExpr);
            newNums.pop_back();
            if (op.count < minOp.count) {
                minOp = op;
            }

            // 减法
            int sub = num1 - num2;
            if (sub > 0) { // 确保结果为正整数
                newNums.push_back(sub);
                newExpr = currentExpr.empty() ?
                    "(" + to_string(num1) + "-" + to_string(num2) + ")" :
                    currentExpr + "-" + to_string(num2);
                op = backtrack(newNums, target, currentCount + 1, newExpr);
                newNums.pop_back();
                if (op.count < minOp.count) {
                    minOp = op;
                }
```

```
            }

            // 乘法
            int mul = num1 * num2;
            newNums.push_back(mul);
            newExpr = currentExpr.empty() ?
                "(" + to_string(num1) + "*" + to_string(num2) + ")" :
                currentExpr + "*" + to_string(num2);
            op = backtrack(newNums, target, currentCount + 1, newExpr);
            newNums.pop_back();
            if (op.count < minOp.count) {
                minOp = op;
            }

            // 除法
            if (num2 != 0 && num1 % num2 == 0) {
                int div = num1 / num2;
                newNums.push_back(div);
                newExpr = currentExpr.empty() ?
                    "(" + to_string(num1) + "/" + to_string(num2) + ")" :
                    currentExpr + "/" + to_string(num2);
                op = backtrack(newNums, target, currentCount + 1, newExpr);
                newNums.pop_back();
                if (op.count < minOp.count) {
                    minOp = op;
                }
            }
        }
    }
    return minOp;
}
```

5-20　部落卫队问题。

问题描述：原始部落 byteland 中的居民们为了争夺有限的资源，经常发生冲突。几乎每个居民都有他的仇敌。部落酋长为了组织一支保卫部落的队伍，希望从部落的居民中选出最多的居民入伍，并保证队伍中任何 2 个人都不是仇敌。

算法设计：给定 byteland 部落中居民间的仇敌关系，计算组成部落卫队的最佳方案。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m，表示 byteland 部落中有 n 个居民，居民间有 m 个仇敌关系。居民编号为 $1, 2, \cdots, n$。接下来的 m 行中，每行有 2 个正整数 u 和 v，表示居民 u 与居民 v 是仇敌。

结果输出：将计算的部落卫队的最佳组建方案输出到文件 output.txt。文件的第 1 行是部落卫队的人数；第 2 行是卫队组成 x_i（$1 \leqslant i \leqslant n$）。$x_i$=0 表示居民 i 不在卫队中，x_i=1 表示居民 i 在卫队中。

输入文件示例	输出文件示例
input.txt	output.txt
7 10	3
1 2	1 0 1 0 0 0 1
1 4	
2 4	
2 3	
2 5	
2 6	
3 5	
3 6	
4 5	
5 6	

答：算法思路：

构建仇敌关系图。可以用邻接表或者邻接矩阵。这里用邻接矩阵方便，比如 bool

enemy [n+1][n+1]，记录 u 和 v 是否是仇敌。

回溯法搜索。定义一个数组 x [i] 表示第 i 个居民是否被选（1 选，0 不选）。递归过程中，尝试选或不选当前居民，同时检查是否与已选的有仇敌关系。

记录最大的人数和对应的 x 数组。

代码如下：

```cpp
int n, m;
bool enemy[21][21];
int maxCount = 0;
int bestX[21];

void backtrack(int k, int count, int x[]) {
    if (k > n) {
        if (count > maxCount) {
            maxCount = count;
            memcpy(bestX, x, (n + 1) * sizeof(int));
        }
        return;
    }
    // 尝试选择第k个居民
    bool canSelect = true;
    for (int i = 1; i < k; ++i) {
        if (x[i] && enemy[i][k]) {
            canSelect = false;
            break;
        }
    }
    if (canSelect) {
        x[k] = 1;
        backtrack(k + 1, count + 1, x);
        x[k] = 0;
    }
    // 尝试不选择第k个居民
    x[k] = 0;
    backtrack(k + 1, count, x);
}

int main() {
    cin >> n >> m;
    memset(enemy, false, sizeof(enemy));
    for (int i = 0; i < m; ++i) {
        int u, v;
        cin >> u >> v;
        enemy[u][v] = enemy[v][u] = true;
    }
    int x[21] = {0};
    backtrack(1, 0, x);
    cout << maxCount << endl;
    for (int i = 1; i <= n; ++i) {
        cout << bestX[i];
    }
    cout << endl;
    return 0;
```