HW05 回溯算法 实验报告

日期：2024 年 5 月 6 日

1. 背景知识

本次作业涉及回溯法的相关知识。目的在于理解回溯法的深度优先搜索策略，掌握用回溯法解题的算法框架，例如递归回溯，迭代回溯，子集树算法框架和排列树算法框架等。

回溯法有“通用解题法”之称，是一种系统地搜索答案的解答方法。回溯法的基本思想为为问题定义一个解空间，该空间至少包含问题的一个解，并可以组织成一棵树，在解空间树中，以深度优先策略搜索，判断当前节点是否包含问题的解，如果不包含，则跳过该结点，回到祖先结点，称为回溯，如果包含，则继续进行深度优先遍历，进入该节点的子树。可以概括为：能进则进，反之则退。

使用回溯法解决问题，首先要确定搜索范围，需要明确以下信息：

- 问题的解向量：回溯法的解可以表示为一个 n 元组$(x1, x2, \dots, xn)$;
- 问题的解空间：满足显示约束的解向量组构成一个解空间，回溯法的解空间可以组织成一棵树
- 问题的可行解：解空间中满足隐式约束的解向量就是一个可行解
- 问题的最优解：对问题给定的目标，所有可行解中目标达到最优的可行解
- 显式约束：解向量的分量 Xi 的取值范围
- 隐式约束：问题给定的约束条件

解空间通常有两种典型的解空间树：

1) 子集树

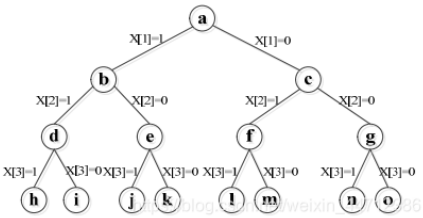
从 n 个物体的集合 r 中选择满足某种性质的子集，这类问题的解空间树称为子集树。n 个物品集合的子集个数为 2n。

解向量：对于 n 个物品的集合，其解向量也有 n 个分量，可表示为$(x1, x2, \dots, xn)$

显式约束：对于子集树问题，通常取$xi \in \{0,1\}$，其中 0 表示物体不在解向量对应的子集中；1 表示物体在解向量对应的子集中。

隐式约束：由问题决定。例如 0-1 背包问题中为：子集中的物体总重量不超过背包的容量。

显然子集树是一棵满二叉树，其结点总数为：$2n + 1 - 1$。遍历的时间复杂度为$O(2^n)$。



2) 排序树

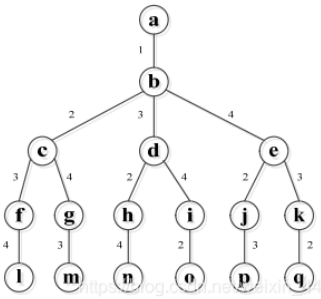
从 n 个元素的集合 r 中找出满足某种性质的排列，这类问题的解空间树称为排列树。n 个元素集合的排列个数为$n!$

解向量：对于 n 个元素的集合，其解向量也有 n 个分量，可表示为$(x1, x2, \dots, xn)$

显式约束：对于排列树问题，有$xi \in r$，且对任意的 i≠j，都有$xi \neq xj$

隐式约束：由问题决定。

排列树的每一个结点的子节点个数都为它所在层次的结点个数-1，叶子结点总数为$n!$。遍历的时间复杂度为$O(n!)$



2. 实验内容

2.1. 子集和问题（5-1）

2.1.1. 问题描述

子集和问题的一个实例为<S,t>。其中，S={s1,x2,...,xn}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，使得集合中所有元素的和等于 c。试设计一个解子集和问题的回溯法。

2.1.2. 问题分析

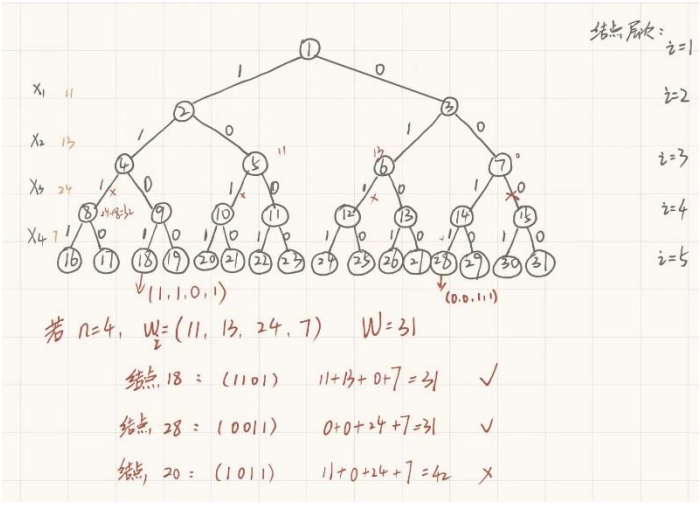
子集和问题简化就是给出一个数组和一个值，输出满足和为该值的数组子集。可以看做是 01 背包问题的子问题，0 代表不选择该元素，1 代表选择该元素。所以可以使用回溯法来解决，求解该问题需要搜索整个空间树，设解向量$X=(x1,x2,\ldots,xn)$，本问题求出其中一个解即可，所以一旦搜索到叶子结点，即 i=n+1 时，如果相应的子集和为 W，则输出 x 解向量。回溯法借助深度优先遍历实现，也符合不断试错的思想，即能进则进，不行则退。回溯法的步骤为：修改当前节点状态，递归子节点状态，回改当前节点状态。

为了更高效地实现试错和回溯，可以定义两个变量用于存储已经选取的子集和和剩余的整数和。例如，当搜索到第 i(1<=i<=n)个节点时,tw 用于表示选取的整数和，rw 表示余下的整数和，rw=w[j]（i+1<=j<=n）。

约束函数：检查当前整数 w[i]加入到子集和中是否超过 W，如果超过则说明不可以选择该路径，执行剪枝操作，用于左孩子节点剪枝。

限界函数：一个节点满足 tw+rw<W,即可以选择剩余的所有整数也找不到可行解，所以也需要剪枝，用于右孩子剪枝。

解空间：同 01 背包问题类似，从根节点到叶子结点的所有路径定义了解空间。当 n=4 时，解空间如下



变量声明为：

```
int a[1000];        // a[i] 表示集合 S 中的第 i 个元素
bool X[1000];       // X[i] 如果为 true，表示 a[i] 在子集 S1 中
int tw = 0, rw = 0; // tw: 子集 S1 的元素之和，rw: 剩余元素的和
bool flag = false;  // 标志位，用于表示是否找到解
int n, c;           // n: 集合 S 的元素数量，c: 目标和
```

2.1.3. 算法设计

回溯算法依赖于深度优先遍历，所以可以按照深度优先遍历的思路来设计。由上述解空间可知这棵子集树是一棵满二叉树，父节点到左孩子标记为 1 代表选择，父节点到右孩子标记为 0，代表不选择。解空间树中有 2^(n+1)-1 个节点，时间复杂度为$O(2^n)$.

在算法设计上，需要编写 backtrack 回溯函数，用于执行回溯与剪枝操作，回溯函数的关键在于状态转移以及怎么恢复原状。状态转移可以利用递归函数传递新的参数，例如 backtrack(p+1),其中 p 代表当前遍历的元素，伴随着递进函数和回溯函数的还有 tw 和 rw 这两个辅助变量。

当在尝试加入的时候，如果 tw+a[i]<=c,说明可以进入左子树继续寻找，所以将该元素的标志位 X[i]设置为 true，tw=tw+a[i],rw=rw-a[i],然后继续递进即可，但要注意如果递进以后没有找到结果，需要返回到这里，也就意味着这个子树不可行，需要返回原状，所以还需要 tw=tw-a[i],rw=rw+a[i]。

如果不选择该元素，也就是不走左子树，此时应该转向右子树。这是需要判断剩余元素的和 rw 加上当前已选元素的和仍然能够到达目标和，尝试不选择当前元素，当满足 tw+rw-a[i]>=c 时，进入右子树，此时需要将 a[i]的标志位设置为 false,在 rw 中删除 a[i],但是不需要修改 tw，因为没有添加也没有去 a[i]，这时，继续调用 backtrack 递进函数，并设置这个函数的返回位置下一操作为 rw+=a[i]，用于返回原状。

这个算法涉及两个剪枝函数：一个用于左子树剪枝，一个用于右子树剪枝。实际是间接剪枝的思想，

如果满足条件则进入继续递进或者回溯，反之则不进入，不执行回溯，也就自然相当于剪掉了这一支。

如果加入当前元素不会超过目标和，就尝试加入

```
    if (tw + a[i] <= c) // 进入左子树，剪枝
    {
        //添加 a[i]后的操作
        X[i] = true;//标记位设置为 true
        tw += a[i];
        rw -= a[i];
        backtrack(i + 1);//递进
        //回溯，并恢复原状
        tw -= a[i];
        rw += a[i];
    }
```

如果即使不选择当前元素，剩余元素的和加上当前已选元素的和仍然能够达到目标和，就尝试不选择当前元素

```
    if (tw + rw - a[i] >= c) // 进入右子树，剪枝
    {
        //设置标志位
        X[i] = false;
        //跳过 a[i]
        rw -= a[i];
        backtrack(i + 1);
        //回溯，并恢复原状
        rw += a[i];
    }
}
```

2.1.4. 算法代码

```
#include <iostream>
using namespace std;
int a[1000];        // a[i] 表示集合 S 中的第 i 个元素
bool X[1000];       // X[i] 如果为 true，表示 a[i] 在子集 S1 中
int tw = 0, rw = 0; // tw: 子集 S1 的元素之和，rw: 剩余元素的和
bool flag = false;  // 标志位，用于表示是否找到解
int n, c;           // n: 集合 S 的元素数量，c: 目标和
void backtrack(int i) // 回溯函数，i 表示当前考虑到的元素索引
{
    if (i >= n || flag == true) // 如果已经考虑完所有元素，或者已经找到解，就返回
        return;
    if (tw == c) // 如果子集 S1 的元素之和等于目标和，就输出解，并设置标志位为 true
    {
        for (int j = 0; j < i; ++j){
            if (X[j] == true)
                cout << a[j] << " ";
        }
        flag = true;
        return;
    }
    // 如果加入当前元素不会超过目标和，就尝试加入
    if (tw + a[i] <= c) // 进入左子树，剪枝
    {
        X[i] = true;
        tw += a[i];
        rw -= a[i];
        backtrack(i + 1);
        tw -= a[i];
        rw += a[i];
    }

    // 如果即使不选择当前元素，剩余元素的和加上当前已选元素的和仍然能够达到目标和，就尝试不选择当前元素
    if (tw + rw - a[i] >= c) // 进入右子树，剪枝
    {
        X[i] = false;
```

```
        rw -= a[i];
        backtrack(i + 1);
        rw += a[i];
    }
}

int main(){
    cin >> n >> c;              // 输入集合 S 的元素数量和目标和
    for (int i = 0; i < n; ++i){ // 输入每个元素的值，并计算所有元素的总和

        cin >> a[i];
        rw += a[i];
    }
    backtrack(0);      // 从第一个元素开始考虑，进行回溯
    if (flag == false) // 如果没有找到解，就输出 "No Solution!"
        cout << "No Solution!" << endl;
    return 0;
}
```

backtrack 是算法实现的关键，最坏情况下需要考虑所有节点，时间复杂度为$O(2^n)$.

```
5 10
2 2 6 5 4
2 2 6
请按任意键继续. . .
```

2.1.5. 调试分析

这道题很好地体现了动态规划与回溯法的区别。我最开始不是很理解两者的区别，两者都用于求解多阶段决策问题，即求解一个问题分为很多步骤（阶段），每一个步骤（阶段）可以有多种选择。不同点在于动态规划只需要求我们评估最优解是多少，最优解对应的具体解是什么并不要求。因此很适合应用于评估一个方案的效果； 回溯算法可以搜索得到所有的方案（当然包括最优解），但是本质上它是一种遍历算法，时间复杂度很高。

回溯法的伪代码

```
result = []
def backtrack(路径, 选择列表){
    if (满足结束条件){
        result.add(路径);
        return;
    }
    for (选择 : 选择列表){
        做选择;
        backtrack(路径, 选择列表);
        撤销选择;
    }
]
```

动态规划的伪代码

```
result = []
def dp(原问题, 选择列表){
    if (子问题是 base case){
        直接返回 base case 的解
    }
    if (子问题有记忆){
        直接查询记忆
    }}
    for (选择 : 选择列表){
        求解子问题，并记忆
    }
    根据子问题的解，计算原问题的解
]
```

为了针对回溯遍历全部解空间，我们可以引入剪枝策略，将不可能存在解的情况连通其后续情况一块跳过，这样可以大大提高效率。

2.2. 最小重量机器设计问题（5-3）

2.2.1. 问题描述

设某一机器由 n 个部件组成，每一种部件都可以从 m 个不同的供应商处购得。设 wij 是从供应商 j 处购来的部件 i 的重量，cij 是相应的价格。试设计一个算法，给出总价格不超过 d 的最小重量机器设计。

2.2.2. 问题分析

这道题实际是一个组合优化问题，解决此类问题可以考虑使用贪心算法，动态规划算法或者回溯等方法，对于本本题，如果使用贪心算法，每种部件都选择重量最小的供应商，大概率会因为没有考虑价格而导致无法正确求解。同时，由于本题不符合优化原则即当选择 n 种部件时的最优解不一定是选择 n+1 种

部件的最优解，所以不能用动态规划算法。由于本题符合多米诺性质即如果某一选择方案选择了 n nn 个部件时不符合约束条件，那么无论第n+k(k>0)种部件如何选择，该选择方案都不符合约束条件，所以可以使用回溯算法，同时为了加快回溯速度，引入剪枝，分支限界方法。

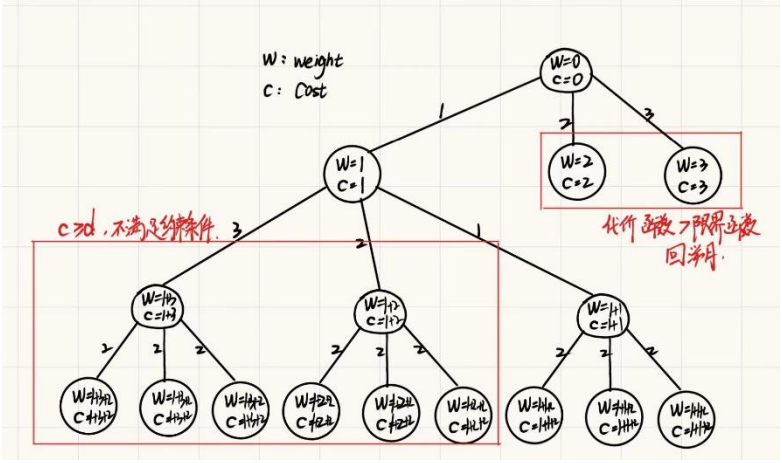
分析题目可知，共 n 个部件，m 个供应商，总价格不超过 d，每个供应商编号分别为 1,2,3，…，m.解向量为(x1,x2…,xn)其中 Xi 代表第 i 个部件应该从哪个供应商处购买。

搜索空间：$\{< x1,x2,x3,\cdots xn > \mid xi \in N, 1 \le xi \le m\}$

约束条件：所有部件的总价格<=d

限界函数：当前已经得出的所有可行解对应的重量的最小值

代价函数：这道题由于有价格，所以可以引入代价函数用于限制。当前已经选择的部件总重量+剩余的部件可能得最小重量，不考虑价格。



数据结构定义：

```
struct State
{
    int weight = 0; // 当前机器的总重量
    int cost = 0;   // 当前机器的总价格
};
const int MAXN = 100; //范围限制
int m, n, d, minWeight =INT_MAX;//供应商数量，部件数量，总价格限制，最小重量
State currentState;              // 当前状态
// 部件的价格和重量
vector<vector<int>> cost(MAXN, vector<int>(MAXN)), weight(MAXN, vector<int>(MAXN));
// 最小重量和结果
vector<int> minSum(MAXN), result(MAXN), tempResult(MAXN);
```

2.2.3. 算法设计

使用深度优先搜索(DFS)策略，在跟节点处，没有选择任何部件，总重量和总价格都是 0，代价函数的值为所有部件的最低价格之和，限界函数的初值为无穷大，代价函数小于限界函数，所以往下搜索根节点的 m 个分支，对于所有节点，都进行如下操作：

- **代价函数与限界函数比较：**计算当前节点的代价函数，即已选择部件的总重量加上剩余部件可能的最小重量（不考虑价格）。若当前节点的代价函数值已超过限界函数，则该路径不可能产生更优解，因此执行回溯。
- **分支限界：**检查当前节点是否满足预算约束。如果当前选择的总价格超出了预算限制 d，则放弃该路径并回溯。
- **继续搜索：**若当前节点通过了代价函数和预算约束的检查，算法将继续探索该节点的 m 个子分支，即该部件的所有供应商选项。
- **叶子节点处理：**当搜索到达叶子节点，即所有部件的供应商都已选择时，算法得到了一个完整的机器设计方案。此时，将该方案的总重量与当前限界函数进行比较：若该方案的总重量小于限界函数的值，则更新限界函数，并记录对应的供应商选择方案作为新的候选最优解;无论是否更新限界函数，搜索完成后，算法都将回溯至前一节点，以继续探索其他可能的部件组合。

我按照输入，处理，输出的思路，将数据读取整合为 input()函数，将回溯算法整合为 backtrack()函数，同时为了后续剪枝操作的方便，我写了一个 setMinSum 函数避免探索那些不可能产生比已知更小重量的解决方案。

剪枝函数：

为了避免不必要的搜索，我引入了叶子结点剪枝，最小重量剪枝，以及预算限制剪枝。

a) **叶子节点剪枝：**

条件：if (p > n)

描述：当所有部件都已经被考虑（即索引 p 大于部件数量 n）时，到达了搜索树的叶子节点。此时，如果当前总重量 currentState.weight 小于已知的最小重量 minWeight，则更新最小重量并记录结果。这一步骤也隐含了剪枝，因为如果当前总重量已经不可能得到一个更优的解，则不会继续探索更深层次的节点。

b) **最小重量剪枝：**

条件：if (minSum[p] + currentState.weight > minWeight)

描述：在搜索过程中，如果当前考虑的部件索引 p 的最小重量累积加上当前状态的总重量超过了已知的最小总重量 minWeight，则没有必要继续搜索该分支。这个剪枝操作是基于预先计算的 minSum 数组，该数组存储了在选择前 i 个部件时可能得到的最小总重量。

c) **预算限制剪枝：**

条件：if (currentState.cost + cost[p][i] <= d)

描述：在选择供应商时，如果加上当前供应商的价格后，总价格会超过预算限制 d，则跳过该供应商，不进行后续的递归调用。这是一种直接的剪枝，因为它确保了不会选择那些会导致超出预算的供应商。

SetMinSum 函数

```
/**
 * @brief 设置最小重量数组
 */
void setMinSum(){
    minSum[n + 1] = 0;
    for (int i = n; i >= 1; --i){
        int minWeight = *min_element(weight[i].begin() + 1, weight[i].begin() + 1 + m);
        minSum[i] = minSum[i + 1] + minWeight;
    }
}
```

setMinSum 函数通过逆序动态规划计算一个数组，该数组存储了在选择前 i 个部件时可能得到的最小总重量。它首先初始化数组最后一个元素为 0，然后从最后一个部件开始向前计算每个部件的最小重量，并将其累积到 minSum 数组中。这个数组将在回溯搜索中用于剪枝，以避免探索那些不可能产生更优解的路径，从而提高算法的效率。循环从 n 到 1，每次循环中使用 min_element 函数，其时间复杂度为 O(m)，因为需要在 m 个供应商中找到最小值。

因此，setMinSum 函数的总体时间复杂度是 O(nm)。

Backtrack 函数

```
/**
 * @brief 回溯函数
 * @param p 当前部件的索引
 */
void backtrack(int p)
{
    if (p == 1)
        setMinSum();
    if (p > n) // 判断是否到达叶子节点
    {
        if (currentState.weight < minWeight) // 更新最小重量
        {
            minWeight = currentState.weight;
            copy(tempResult.begin() + 1, tempResult.begin() + 1 + n, result.begin() + 1);
        }
        return;
    }
    // 该叶子结点添加后比最小重量小，剪枝
    if (minSum[p] + currentState.weight > minWeight)
    {
        return;
    }
    for (int i = 1; i <= m; ++i) // 遍历所有供应商
    {
        // 该供应商的价格小于总价格限制
        if (currentState.cost + cost[p][i] <= d)
        {
            currentState.weight += weight[p][i];
            currentState.cost += cost[p][i];
```

```
            tempResult[p] = i;                    // 记录结果
            backtrack(p + 1);                     // 继续
            currentState.weight -= weight[p][i]; // 回溯上一状态
            currentState.cost -= cost[p][i];     // 回溯上一状态
        }
    }
}
```

backtrack 函数是实现回溯算法的核心，其目的是通过递归探索所有可能的部件供应商组合，以找到总重量最轻且总价格不超过预算的机器设计。该函数首先接受一个参数 p，它代表当前正在考虑的部件的索引，从 1 开始。然后初始化最小重量数组，如果当前是第一个部件（p == 1），则先调用 setMinSum 函数来初始化最小重量数组 minSum，这个数组后续用于剪枝。

判断叶子节点：如果 p > n，则表示已经考虑完所有部件，当前节点是一个叶子节点。此时，函数检查当前状态的总重量 currentState.weight 是否小于已知的最小重量 minWeight：如果是，更新 minWeight 并将当前的供应商选择方案 tempResult 复制到最终结果数组 result 中。

剪枝条件：在继续探索之前，函数使用 minSum 数组和当前状态的总重量 currentState.weight 来判断是否有可能通过增加更多部件得到一个比当前最小重量还小的解。如果 minSum[p] + currentState.weight 已经大于 minWeight，则剪去这个分支，不再进一步搜索。

遍历供应商：对于当前部件 p 的每个供应商（从 1 到 m），函数检查如果选择这个供应商是否会导致总价格超出预算 d：如果不超预算，函数将该供应商的重量和价格累加到当前状态的重量和价格上，记录当前部件选择的供应商到 tempResult，然后递归调用 backtrack(p + 1) 继续探索下一个部件。

回溯：在每个供应商选择后，无论是否找到了更优解，函数都会执行回溯操作，即将当前部件的重量和价格从 currentState 中减去，以恢复到选择该供应商之前的状态。完成所有供应商的探索后，函数返回到上一个递归级别，继续探索其他可能的部件供应商组合。

时间复杂度方面，由于 backtrack 函数会为每个部件遍历所有供应商，其时间复杂度是 O(m^n)，其中 n 是部件的数量，m 是每个部件的供应商数量。然而，由于剪枝的存在，实际运行时间通常会比这个上界要好得多。

2.2.4. 算法代码

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

const int MAXN = 100;

struct State
{
    int weight = 0; // 当前机器的总重量
    int cost = 0;   // 当前机器的总价格
};
// 部件的价格和重量
vector<vector<int>> cost(MAXN, vector<int>(MAXN)), weight(MAXN, vector<int>(MAXN));
vector<int> minSum(MAXN), result(MAXN), tempResult(MAXN);   // 最小重量和结果
int m, n, d, minWeight = INT_MAX;    // 供应商数量，部件数量，总价格限制，最小重量
State currentState;         // 当前状态

/**
 * @brief 输入数据
 */
void input()
{
    cin >> n >> m >> d;
    for (int i = 1; i <= n; ++i)
    {
        for (int j = 1; j <= m; ++j)
        {
            cin >> cost[i][j];
        }
    }
    for (int i = 1; i <= n; ++i)
    {
        for (int j = 1; j <= m; ++j)
```

```
        {
            cin >> weight[i][j];
        }
    }
}
/**
 * @brief 设置最小重量数组
 */
void setMinSum()
{
    minSum[n + 1] = 0;
    for (int i = n; i >= 1; --i)
    {
        int minWeight = *min_element(weight[i].begin() + 1, weight[i].begin() + 1 + m);
        minSum[i] = minSum[i + 1] + minWeight;
    }
}
/**
 * @brief 回溯函数
 * @param p 当前部件的索引
 */
void backtrack(int p)
{
    if (p == 1)
        setMinSum();
    if (p > n) // 判断是否到达叶子节点
    {
        if (currentState.weight < minWeight) // 更新最小重量
        {
            minWeight = currentState.weight;
            copy(tempResult.begin() + 1, tempResult.begin() + 1 + n, result.begin() + 1);
        }
        return;
    }
    // 该叶子结点添加后比最小重量小，剪枝
    if (minSum[p] + currentState.weight > minWeight){
        return;
    }
    for (int i = 1; i <= m; ++i) // 遍历所有供应商
    {
        // 该供应商的价格小于总价格限制
        if (currentState.cost + cost[p][i] <= d)
        {
            currentState.weight += weight[p][i];
            currentState.cost += cost[p][i];
            tempResult[p] = i;                    // 记录结果
            backtrack(p + 1);                     // 继续
            currentState.weight -= weight[p][i]; // 回溯上一状态
            currentState.cost -= cost[p][i];     // 回溯上一状态
        }
    }
}

int main()
{
    input();
    backtrack(1); // 回溯
    cout << minWeight << endl;
    for (int i = 1; i <= n; ++i) // 输出结果
        cout << result[i] << " ";
    return 0;
}
```

```
8 18 14
18 15 20 5 15 10 16 6 1 6 17 6 1 2 17 15 13 17
16 6 7 4 7 2 11 6 18 4 13 12 8 5 2 8 15 14
12 6 19 10 13 8 2 10 16 4 15 15 16 13 17 12 14 4
18 18 2 13 15 19 5 12 18 7 13 9 8 17 10 13 15 11
8 5 14 11 18 20 17 3 11 17 13 11 4 9 17 14 19 1
10 7 8 11 13 3 19 3 12 11 12 14 4 2 12 10 14 15
12 9 13 9 16 17 12 15 6 3 11 17 13 17 14 13 4 4
19 12 3 19 3 20 19 12 8 19 8 10 19 20 3 1 7 1
16 12 4 16 2 6 15 1 13 3 7 16 5 3 16 16 14 19
12 14 6 2 11 15 9 17 15 16 19 20 14 14 20 9 4 4
6 13 16 6 3 12 12 19 11 20 4 13 9 18 7 17 8 1
4 17 3 20 3 8 12 7 4 12 6 12 1 18 13 20 20 8
4 15 1 10 2 12 8 11 5 4 20 13 12 20 1 3 3 11
1 9 2 1 16 1 12 4 5 2 7 15 12 3 9 4 13 6
13 1 10 8 5 13 20 10 6 4 8 15 8 8 20 11 9 9
2 10 11 1 18 8 20 11 18 2 3 6 14 16 19 4 3 15
57
13 6 7 3 18 14 10 16
请按任意键继续. . .
```

2.2.5. 调试分析

这道题除了有重量限制外，还有价格限制，一般这种多策略问题可以用贪心算法，动态规划和回溯算法，这道题给出了三者的区别，我也明白了回溯算法的适用情形。我认为这道题的关键在于剪枝函数的确定，回溯算法需要遍历整个解空间，效率较低，所以需要找到更好的剪枝策略，这里参考网上思路，使用了代价函数来进行比较，代价函数不仅包括当前代价还需要预知未来的最小重量。

2.3. 工作分配问题（5-13）

2.3.1. 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 cij。试设计一个算法，为每一个人都分配 1 件不同的工作，并使总费用达到最小。

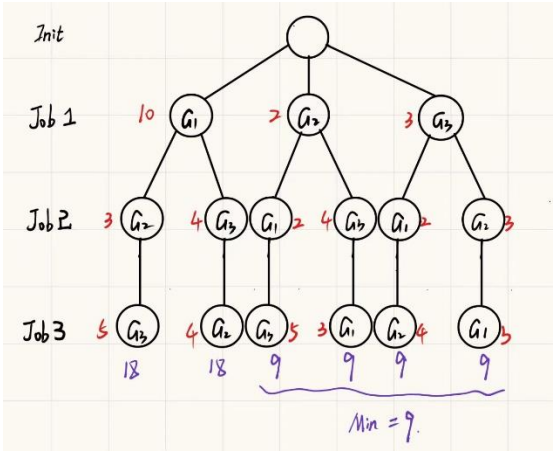
2.3.2. 问题分析

首先，我们需要将问题建模为一个二分图，其中一个集合代表工作，另一个集合代表人员，费 cij 表示将工作 i 分配给人员 j 的费用。让一方选另一方，这样就可以构成一棵排列树。我们让工作“选”人，那排列树的结点代表人，而层就代表工作。例如用 G1 表示工人 1，且在第一层，表示为工作 1 安排第 1 个人，即将工作 1 分配给工人 1。

解空间：解空间由所有可能的工作分配方案组成。由于有 n 个人和 n 件工作，解空间的大小是 $n!$，因为每个工作都有 n 种分配给不同人的可能性。

约束条件：每个人只能分配到一项工作，同理，每项工作只能分给一个人。分配方案的总费用必须不超过一个预设的阈值，如果问题中没有给出预算限制，则默认为最小化费用。

限界函数：最小费用限界 (Min)：用于记录迄今为止找到的最小总分配费用。在搜索过程中，如果当前分配方案的累积费用，sumCost 已经超过 Min，则可以提前终止该路径的搜索，因为继续搜索不会得到更优的解。



数据结构设计：

```
int n;                // 工作数量
int cost[21][21];     // 第 i 个人做第 j 个工作的花费
int Min = 0x7fffffff; // 最小花费,先初始化为最大值
int sumCost = 0;      // 当前花费
int flag[21];         // 标记是否已经被分配工作了
```

2.3.3. 算法设计

采用回溯搜索算法来探索所有可能的工作分配方案。算法从第一个工作开始，为每个人尝试分配工作，直到找到最小化总费用的分配方案。在算法设计的过程中，需要引入剪枝策略来保证效率，当前总费用 sumCost 与已知的最小费用 Min 进行比较。如果 sumCost 已经超过 Min，则剪枝，回溯以探索其他可能的分配方案。

同时，为了保证满足约束条件，使用一个标记数组 flag 来记录每个人员是否已经分配到工作。在算法执行前需要进行初始化，由于本题要求最小花费，可以将这个变量先存一个很大的数，到时候慢慢减小即可。

从工作 1 开始，使用 backtrack 函数为每个人尝试分配工作。回溯搜索：在 backtrack 函数中，如果当前工作编号 p 等于 n，检查并更新最小费用 Min。剪枝：在分配工作前，如果 sumCost 已经超过 Min，则跳过当前分配，直接回溯。分配与回溯：为当前工作 p 分配一个未分配工作的人员 i，更新 sumCost，递归调用 backtrack(p + 1) 继续分配下一项工作。之后，取消分配，回溯以探索其他分配方案。

回溯函数实现：

```
/**
 * @brief 回溯法求解最小花费
 * @param p 当前前考虑的人
 */
void backtrack(int p)
{
    if (p >= n) // 如果已经考虑完所有人，就返回
    {
        if (Min > sumCost) // 如果当前花费小于最小花费，就更新最小花费
        {
            Min = sumCost;
            return;
        }
    }
    // 如果当前花费已经大于最小花费，就不用再继续分配工作了
    for (int i = 0; i < n; ++i)
    {
        if (flag[i] == 0)// 如果第 i 个工作还没有被分配
        {
            flag[i] = 1;
            sumCost += cost[p][i];
            if (sumCost < Min)// 如果当前花费小于最小花费，就继续分配工作
                backtrack(p + 1);
            flag[i] = 0;
            sumCost -= cost[p][i];// 回溯
        }
    }
}
```

剪枝函数：

- 在 backtrack 函数中，当所有工作都已经被分配（即 p >= n），算法进入基本情况。此时，算法会比较当前解的总费用 sumCost 与已知的最小费用 Min。如果当前解更优（即 sumCost < Min），则更新 Min。
- 如果在分配过程中，当前总费用 sumCost 已经超过了当前已知的最小费用 Min，那么继续分配将不会得到一个更优的解。因此，算法会提前终止当前分支的搜索，这个直接体现了剪枝策略。
- 在为工作 p 分配人员 i 的循环中，算法首先检查人员 i 是否已经被分配了工作（flag[i] == 0）。这是一种隐含的剪枝，因为它避免了重复分配。
- 在分配工作后，算法不是立即递归调用 backtrack(p + 1)，而是先检查更新后的 sumCost 是否小于 Min。只有当 sumCost < Min 时，才会继续搜索子问题。

2.3.4. 在分配完工作并递归调用 backtrack(p + 1) 之后，算法通过将 flag[i] 重置为 0 和从 sumCost 中减去相应的费用 cost[p][i] 来进行回溯。

2.3.5. 算法代码

```
#include <iostream>
using namespace std;
int n;                // 工作数量
int cost[21][21];     // 第 i 个人做第 j 个工作的花费
```

```cpp
int Min = 0x7fffffff; // 最小花费,先初始化为最大值
int sumCost = 0;      // 当前花费
int flag[21];         // 标记是否已经被分配工作了
/**
 * @brief 回溯法求解最小花费
 * @param p 当前前考虑的人
 */
void backtrack(int p)
{
    if (p >= n) // 如果已经考虑完所有人，就返回
    {
        if (Min > sumCost) // 如果当前花费小于最小花费，就更新最小花费
        {
            Min = sumCost;
            return;
        }
    }
    // 如果当前花费已经大于最小花费，就不用再继续分配工作了
    for (int i = 0; i < n; ++i)
    {
        if (flag[i] == 0)// 如果第 i 个工作还没有被分配
        {
            flag[i] = 1;
            sumCost += cost[p][i];
            if (sumCost < Min)// 如果当前花费小于最小花费，就继续分配工作
                backtrack(p + 1);
            flag[i] = 0;
            sumCost -= cost[p][i];// 回溯
        }
    }
}
int main()
{
    cin >> n;
    for (int i = 0; i < n; ++i)
    {
        for (int j = 0; j < n; ++j)
        {
            cin >> cost[i][j];
        }
        flag[i] = 0;
    }
    backtrack(0);
    cout << Min << endl;
    system("pause");
    return 0;
}
```

```
3
4  2  5
2  3  6
3  4  5
9
请按任意键继续. . .
```

2.3.6. 调试分析

这里面约束条件除了每个人只能分配一个工作外，还要求每个工作只能分配一个人。引入 flag 标志，flag 数组的状态如果没有正确更新或检查，会导致同一人员被多次分配工作。

3. 实验总结

本次作业涉及了回溯法的相关知识。在做题的时候对回溯法还不是很了解，对剪枝函数或约束函数也不是很明白，也对动态规划，贪心，回溯的区别理解不深刻。但通过这次实验，我对回溯法的理解更加深刻，也明白了不同的算法有不同的适用情形。