实验五 0-1 背包问题的算法设计 实验报告

姓名：陈矛　　学号：SA22225116

一、实验原理

1. 算法流程图：

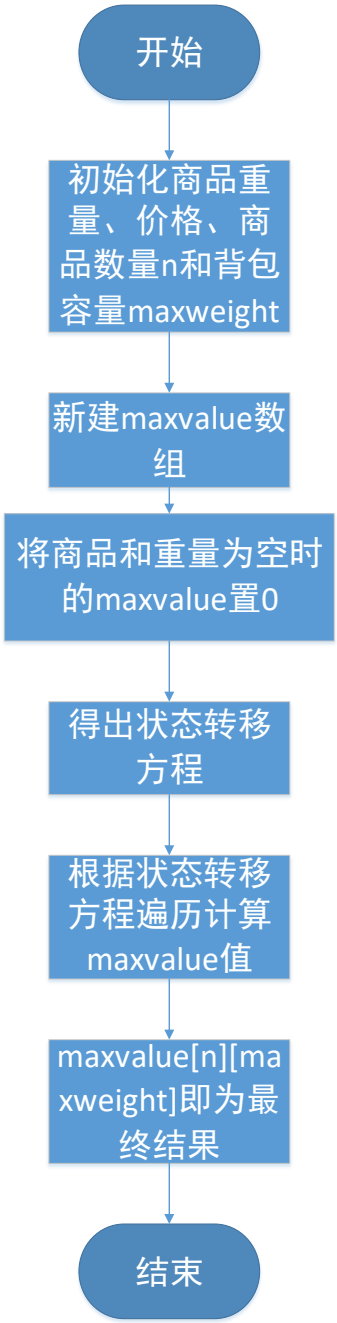

图 动态规划方法的算法流程图

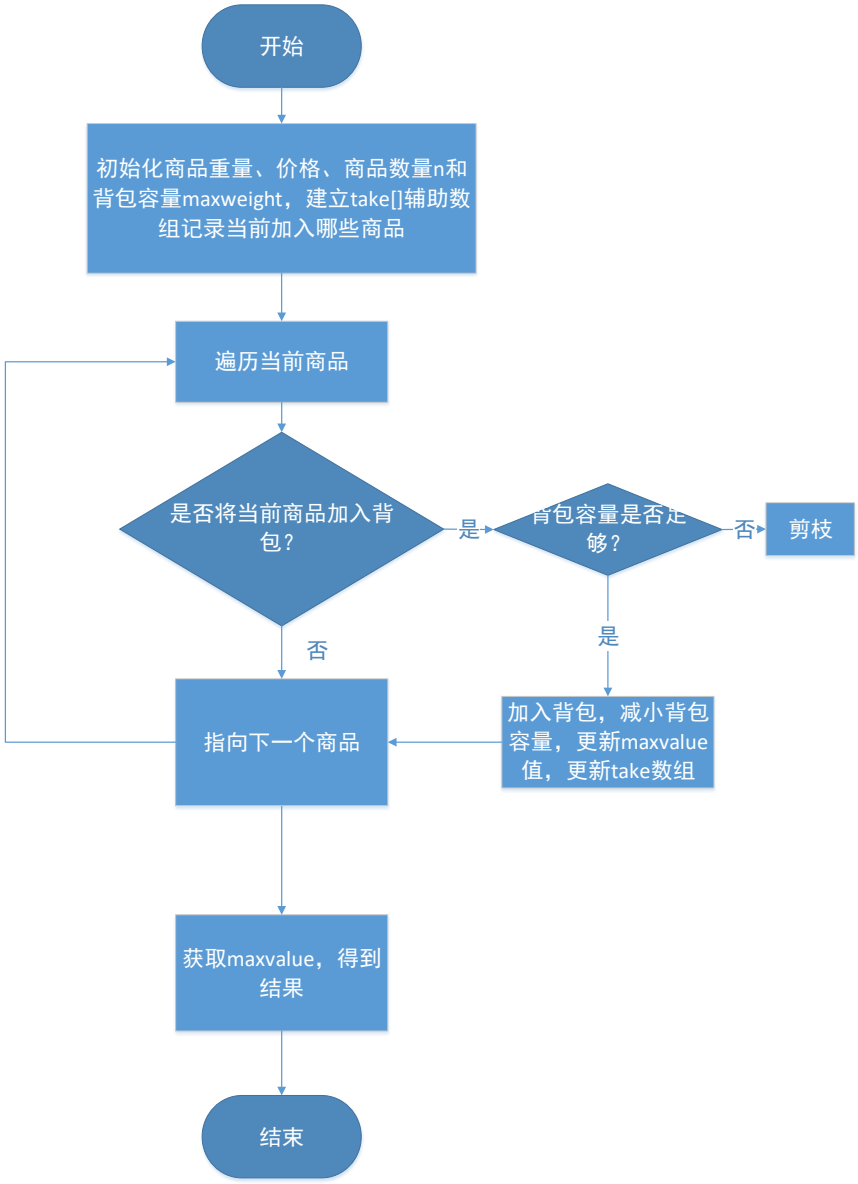

图 回溯方法的算法流程图

2. 背包问题

背包问题已经是一个很经典而且讨论很广泛的算法问题了。背包问题泛指这类种问题： 给定一组有固定价值和固定重量的物品，以及一个已知最大承重量的背包， 求在不超过背包最大承重量的前提下，能放进背包里面的物品的最大总价值。具体各类背包问题可以分成以下 3 种不同的子问题。

2.1 0-1 背包问题

问题描述：有编号分别为 a,b,c,d,e 的五件物品，它们的重量分别是 2,2,6,5,4，它们的价值分别是 6,3,5,4,6，每件物品数量只有一个，现在给你个承重为 10 的背包，如何让背包里装入的物品具有最大的价值总和？
特点:每个物品只有一件，选择放或者不放。

2.2 完全背包问题

问题描述:有编号分别为 a,b,c,d 的四件物品，它们的重量分别是 2,3,4,7，

它们的价值分别是 1,3,5,9，每件物品数量无限个，现在给你个承重为 10 的背包， 如何让背包里装入的物品具有最大的价值总和？
特点：每个物品可以无限选用。

2.3 多重背包问题

问题描述：有编号分别为 a,b,c 的三件物品，它们的重量分别是 1，2，2，它们的价值分别是 6，10，20， 他们的数目分别是 10，5，2，现在给你个承重为 8 的背包，如何让背包里装入的物品具有最大的价值总和？
特点：每个物品都有一定的数量。

3. 解决算法

3.1 动态规划算法

动态规划原理：动态规划是一种将问题实例分解为更小的、相似的子问题，并存储子问题的解而避免计算重复的子问题，以解决最优化问题的算法策略。动态规划法所针对的问题有一个显著的特征，即它所对应的子问题树中的子问题呈现大量的重复。**动态规划法的关键就在于，对于重复出现的子问题，只在第一次遇到时加以求解，并把答案保存起来，让以后再遇到时直接引用，不必重新求解。**

用动态规划方法解决 0-1 背包问题：
步骤 1：找子问题。子问题必然是和物品有关的，对于每一个物品，有两种结果：能装下或者不能装下。第一，包的容量比物品体积小，装不下，这时的最大价值和前 i-1 个物品的最大价值是一样的。第二，还有足够的容量装下该物品，但是装了不一定大于当前相同体积的最优价值，所以要进行比较。
由上述分析，子问题中物品数和背包容量都应当作为变量。
因此子问题确定为背包容量为 j 时，求前 i 个物品所能达到最大价值。

步骤 2：确定状态：由上述分析，“状态”对应的“值”即为背包容量为 j 时，求前 i 个物品所能达到最大价值，设为 dp[i][j]。初始时，dp[0][j](0<=j<=V)为 0，没有物品也就没有价值。

步骤 3：确定状态转移方程：由上述分析，第 i 个物品的体积为 w,价值为 v，则状态转移方程为

- j<w,dp[i][j]=dp[i-1][j]//背包装不下该物品,最大价值不变
- j>=w,dp[i][j]=max{dp[i-1][j-list[i].w]+V,dp[i-1][j]}//和不放入该物品时同样达到该体积的最大价值比较

3.2 贪婪算法

贪心法把一个复杂问题分解为一系列较为简单的局部最优选择，每一步选择都是对当前的一个扩展，直到获得问题的完整解。

k-optimal 算法用来解决 0-1 背包问题：
步骤 1：计算每种物品单位重量的价值 Vi/Wi；

步骤 2：依贪心选择策略，将尽可能多的单位重量价值最高的物品装入背包。
步骤 3：若将这种物品全部装入背包后，背包内的物品总重量未超过 C，则选择单位重量价值次高的物品并尽可能多地装入背包。
依此策略一直地进行下去，直到背包装满为止。

3.3 回溯法

回溯法先确定解空间的结构，使用深度优先搜索，搜索路径一般沿树形结构进行，在搜索过程中，首先会判断所搜索的树结点是否包含问题的解，如果肯定不包含，则不再搜索以该结点为根的树结点，而向其祖先结点回溯；否则进入该子树，继续按深度优先策略搜索。
运用回溯法解题通常包含以下三个步骤：
a.针对所给问题，定义问题的解空间；
b.确定易于搜索的解空间结构；
c.以深度优先的方式搜索解空间，并且在搜索过程中用剪枝函数避免无效搜索；

3.4 分支定界法

分支限界法类似于回溯法，也是在问题的解空间上搜索问题解的算法。
分支限界法首先要确定一个合理的限界函数（boundfunciton），并根据限界函数确定目标函数的界[down,up]，按照广度优先策略或以最小耗费优先搜索问题的解空间树，在分直结点上依次扩展该结点的孩子结点，分别估算孩子结点的目标函数可能值，如果某孩子结点的目标函数可能超出目标函数的界，则将其丢弃；否则将其加入待处理结点表（简称 PT 表），依次从表 PT 中选取使目标函数取得极值的结点成为当前扩展结点，重复上述过程，直到得到最优解。
常见的两种分枝限界法包含队列式(FIFO)分支限界法和优先队列式分支限界法。

4. 时间复杂度和空间复杂度

1) 动态规划方法
   a) 该算法的时间复杂度为 O(MN)。其中 N 为商品的总数，M 为背包的容量（maxweight）。根据后续的代码，可以看出，使用动态规划方法时，该算法的主体部分就是一个嵌套循环求 dp 数组的过程。算法会逐一算出 dp[i][j] 中的每个元素，所以总的时间复杂度就是这个循环嵌套的时间复杂度 O(MN)。
   b) 该算法的空间复杂度为 O(MN)。需要建立一个 dp 数组，dp 数组的大小就是 M×N，所以空间复杂度为 O(MN)。
2) 回溯方法
   a) 该算法的时间复杂度为 O(N*2^N)。其中 N 为商品的总数。使用回溯算法时，可以把整个遍历过程看成对一个深度为 N 的二叉树的遍历过程，这个树的节点总数为 2^N。而对每一个节点遍历时，都需要判断该节点是否可以继续往下遍历，继续往下遍历是否还有意义。这个判断函数的时间复杂度为 O(N)。所以算法的总时间复杂度为 O(N*2^N)。

b) 该算法的空间复杂度为 O(N)。该算法本质上可以理解成是对一个二叉树的遍历过程，检索深度最多为 N，所以空间复杂度为 O(N)。

二、运行结果及分析

算法设计：输入物品数 n， 背包容量 c， 输入 n 个物品的重量、价值，在以上算法中**任选两个**实现最优解决 0-1 背包问题。

算法选择：选择使用**动态规划**方法和**回溯算法**解决 0-1 背包问题。

问题描述：有编号分别为 a, b, c, d, e 的五件物品，它们的重量分别是 2, 2, 6, 5, 4，它们的价值分别是 6, 3, 5, 4, 6，每件物品数量只有一个，现在给你个承重为 10 的背包，如何让背包里装入的物品具有最大的价值总和？

请问：所选算法的实现流程图或者伪代码是什么？比较时间复杂度和空间复杂度，得出什么结论？

1. 解决 0-1 背包问题的**动态规划**方法，并写出流程图或者伪代码。

1) 算法流程图：

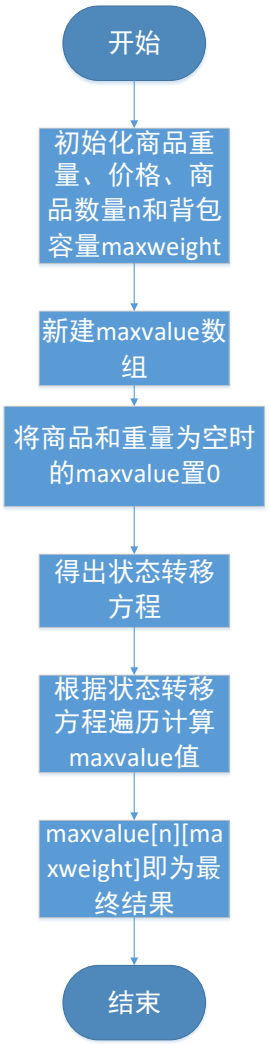

图 动态规划算法流程图

2) 关键代码：

```
public int knapsack_dp(int[] weight, int[] value, int maxweight){

    int n = weight.length;
    //最大价值数组为maxvalue[N+1][maxWeight+1]，因为我们要从0开始保存
    int[][] maxvalue = new int[n+1][maxweight + 1];

    //重量和物品为0时，价值为0
    for (int i = 0; i < maxweight + 1; i++) {
        maxvalue[0][i] = 0;

    }
    for (int i = 0; i < n + 1; i++) {
        maxvalue[i][0] = 0;

    }

    //i：只拿前i件物品（这里的i因为取了0，所以对应到weight和value里面都是i-1号位置）
    //j：假设能取的总重量为j
    //n是物品件数
    for (int i = 1; i <= n ; i++) {
        for (int j = 1; j <= maxweight; j++) {
            //当前最大价值等于放上一件的最大价值
            maxvalue[i][j] = maxvalue[i-1][j];
            //如果当前件的重量小于总重量，可以放进去或者拿出别的东西再放进去
            if (weight[i-1] <= j) {
                //比较（不放这个物品的价值）和
                //（这个物品的价值 加上 当前能放的总重量减去当前物品重量时取前i-1个物品时的对应重量时候的最高价值）
                if(maxvalue[i-1][j - weight[i-1]] + value[i-1]>maxvalue[i-1][j]) {
                    maxvalue[i][j] = maxvalue[i-1][j - weight[i-1]] + value[i-1];
                }
            }
        }
    }

    System.out.println();
    System.out.println("动态规划表如下：");
```

```
    for(int i = 1; i <= n ; i++) {
        for(int j = 1; j <= maxweight; j++) {
            System.out.print(maxvalue[i][j] + "\t");
        }
        System.out.println();
    }

    return maxvalue[n][maxweight];
}
```

3) 算法实现流程：

a) 找子问题。子问题必然是和物品有关的，对于每一个物品，有两种结果：**能装下或者不能装下**。第一，包的容量比物品体积小，装不下，这时的最大价值和前 i-1 个物品的最大价值是一样的。第二，还有足够的容量装下该物品，但是装了不一定大于当前相同体积的最优价值，所以要进行比较。由上述分析，子问题中物品数和背包容量都应当作为变量。因此子问题确定为背包容量为 j 时，求前 i 个物品所能达到最大价值。

b) 确定状态：由上述分析，“状态”对应的“值”即为背包容量为 j 时，求前 i 个物品所能达到最大价值，设为 maxvalue[i][j]。maxvalue[i][j]代表当背包容量为 j 时，前 i 个物品所能达到的最大价值。当 i=n,j=maxweight 时，就可以代表背包容量为 maxweight 时，总共 n 个物品所能达到的最大价值，也就是我们所求问题的解。初始时，当 i=0 或 j=0 时，maxvalue[i][j]为 0，代表没有物品或者背包容量为 0 时，所能获取的最大价值为 0。

c) 确定状态转移方程：由上述分析，第 i 个物品的重量为 weight,价值为 value，则状态转移方程为

- i=0 或 j=0 时， maxvalue[i][j] = 0。
- j < $weight_i$,maxvalue[i][j]=maxvalue[i-1][j]。表示当前背包容量已经不足以放下这个编号为 i 的物品。
- j >= $weight_i$,maxvalue[i][j]=max{maxvalue[i-1][j],maxvalue[i-1][j - $weight_i$] + $value_i$}。表示放入该物品或者不放入该物品的物品价值比较。

$$dp(i,j)=\begin{cases} 0, & i=0 \text{ 或 } j=0 \\ dp(i-1,j), & j<w_i \\ \max\{dp(i-1,j),\ dp(i-1,j-w_i)+v_i\}, & j\geq w_i \end{cases}$$

图 动态规划状态方程

4) 算法运行结果：

```
动态规划表如下：
0   6   6   6   6   6   6   6   6   6
0   6   6   9   9   9   9   9   9   9
0   6   6   9   9   9   9   11  11  14
0   6   6   9   9   9   10  11  13  14
0   6   6   9   9   12  12  15  15  15
背包内最大的物品价值总和为：15
```

5) 得到这样结果的原因（以该示例为例）：

a) 初始化所有物品的重量数组、价值数组和背包最大承重，weight = {2,2,6,5,4}; value = {6,3,5,4,6};maxweight = 10。

b) 建立 maxvalue 数组并对 i=0 和 j=0 的情况进行初始化（实现了状态转移方程中的第一行）

c) 开始进行动态规划。对于 maxvalue[i][j]而言，每一个 maxvalue[i][j]最先都被初始化为 maxvalue[i-1][j]，因为对于新增了一件物品 i（暂时未加入背包）而言，只是多了一种选择而已，maxvalue[i][j]肯定不会变小，最差也是 maxvalue[i-1][j]。以 j=1 为例：j=1 时，即背包最大容量为 1，新增的所有物品的 weight 都大于 1，所以背包放不下任何物品，所以所有的 maxvalue[i][1]都为 0。如图：

```
动态规划表如下：
0   6   6   6   6   6   6   6   6   6
0   6   6   9   9   9   9   9   9   9
0   6   6   9   9   9   9   11  11  14
0   6   6   9   9   9   10  11  13  14
0   6   6   9   9   12  12  15  15  15
背包内最大的物品价值总和为：15
```

另外，以 j=2 为例：j=2 时，maxvalue[1][2]=6，因为背包容量为 2 时，可以放入物品 1，得到最大的物品价值 6；对于后续新增的所有物品选择，都保底有“拿取物品 1”这样的选择，所以 maxvalue[i][2]一定大于等于 maxvalue[1][2]。如图：

```
动态规划表如下：
0   6   6   6   6   6   6   6   6   6
0   6   6   9   9   9   9   9   9   9
0   6   6   9   9   9   9   11  11  14
0   6   6   9   9   9   10  11  13  14
0   6   6   9   9   12  12  15  15  15
背包内最大的物品价值总和为：15
```

d) 当 j >= weight[i-1]时，代表此时的背包可以放下当前这个物品。但是可以放

下该物品并不代表就一定要放入该物品，此时 maxvalue[i][j]=max{maxvalue[i-1][j],maxvalue[i-1][j - $weight_i$] + $value_i$}，表示不加入该物品或者加入该物品并减少背包容量这两种情况。以 j=10 为例。maxvalue[1][10]=6,代表此时只有物品 1 可供选择，那么 maxvalue[1][10]=max{maxvalue[0][10],maxvalue[0][8] + 6}=6。同理，maxvalue[2][10]=max{maxvalue[1][10],maxvalue[1][8] + 3}=9。依次可以得到 maxvalue[3][10]=14, maxvalue[4][10]=14, maxvalue[5][10]=15。

e) 最终的 maxvalue[5][10]就是我们所求的结果。

2. 解决 0-1 背包问题的**贪心**方法，并写出流程图或者伪代码。

1) 算法流程图：

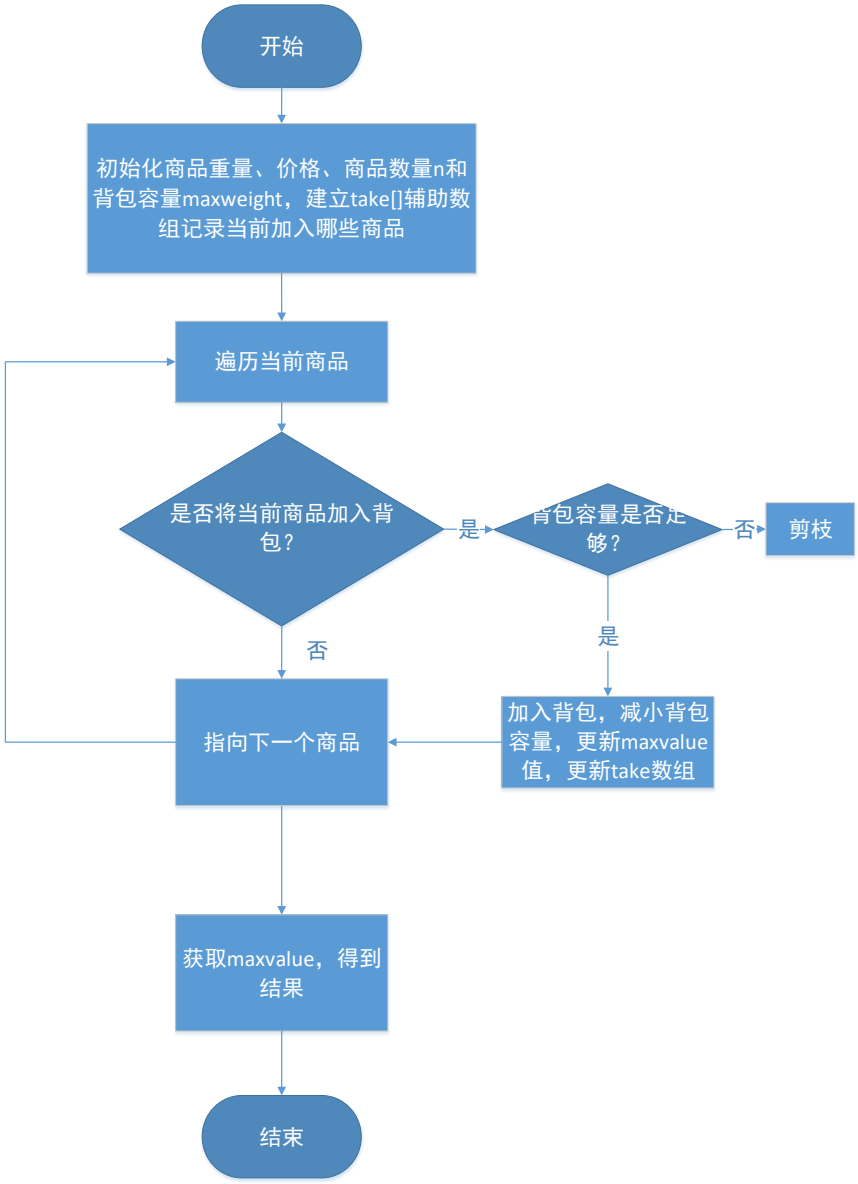

图 回溯方法的算法流程图

2) 关键代码：

```
public int[] backtrack(int x) {
    //终止条件
    if (x > count - 1) {
        //更新最优解
        if (curValue > bestValue) {
            bestValue = curValue;
            for (int i = 0; i < take.length; i++) {
                bestChoice[i] = take[i];
            }
        }
    } else { //遍历当前节点（物品）的子节点：0 不放入背包 1：放入背包
        for (int i = 0; i < 2; i++) {
            take[x] = i;
            if (i == 0) {
                //不放入背包，接着往下走
                backtrack(x + 1);//递归下一个物品
            } else {
                //约束条件，如果小于背包容量
                if (curWeight + weight[x] <= maxWeight) {
                    //更新当前重量和价值
                    curWeight += weight[x];
                    curValue += value[x];
                    //继续向下深入
                    backtrack(x + 1);
                    //回溯
                    curWeight -= weight[x];
                    curValue -= value[x];
                }
            }
        }
    }
    return bestChoice;
}
```

3) 算法实现流程：

对于每一个商品，都有将其加入背包和不将其加入背包两种可能性。所以可以想到使用构造二叉树的方式来进行回溯算法。这样的方式，相当于就是将所有可能的结果都遍历了一遍，每次加入物品时，就和当前的 maxvalue 作比较，如果当前价值更高，则该方案更合理，更新 maxvalue，并更新 take 数组（take 数组用于记录物品选取方案）。为了降低时间复杂度，使用剪枝的方法，如果当前的背包已经满了，那么继续往下遍历肯定永远都不可能加入新物品，所以进行剪枝操作。

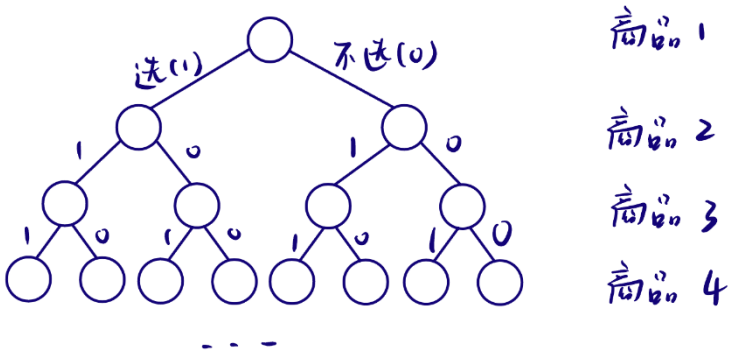

图 回溯树

4) 算法运行结果：

```
E:\University\java\develop_tools\jdk1.8.0_131\bin\java ...
最佳选择为：[1,1,0,0,1]
此时价值最大，即15

Process finished with exit code 0
```

3. 比较两种方法的时间复杂度和空间复杂度，得出什么结论？

1) 动态规划方法

a) 该算法的时间复杂度为 O(MN)。其中 N 为商品的总数，M 为背包的容量（maxweight）。根据后续的代码，可以看出，使用动态规划方法时，该算法的主体部分就是一个嵌套循环求 dp 数组的过程。算法会逐一算出 dp[i][j]中的每个元素，所以总的时间复杂度就是这个循环嵌套的时间复杂度 O(MN)。

b) 该算法的空间复杂度为 O(MN)。需要建立一个 dp 数组，dp 数组的大小就是 M×N，所以空间复杂度为 O(MN)。

2) 回溯方法

a) 该算法的时间复杂度为 O(N*2^N)。其中 N 为商品的总数。使用回溯算法时，可以把整个遍历过程看成对一个深度为 N 的二叉树的遍历过程，这个树的节点总数为 2^N。而对每一个节点遍历时，都需要判断该节点是否可以继续往下遍历，继续往下遍历是否还有意义。这个判断函数的时间复杂度为 O(N)。所以算法的总时间复杂度为 O(N*2^N)。

b) 该算法的空间复杂度为 O(N)。该算法本质上可以理解成是对一个二叉树的遍历过程，检索深度最多为 N，所以空间复杂度为 O(N)。

可以得出结论：在这个问题中，动态规划方法可以大大的减少问题的时间复杂度。而回溯算法的时间复杂度则有一个指数级的上界；在数据量很大的时候，动态规划算法可以大大减少计算所需要的时间。这是因为动态规划算法对于重复出现的子问题，只在第一次遇到时加以求解，并把答案保存起来，让以后再遇到时直接引用，不必重新求解，大大减少了计算量；而回溯算法本质上是一个暴力穷举的过程（但是使用剪枝，减少了实际遍历的节点的数

量，比暴力穷举算法优越，但时间复杂度的上界仍然很高，因为可能存在完全不剪枝的情况）。

修改数据集如下所示

```
int weight[] = {2,2,6,5,4,7,8,9,3,4,18,20,80,2,3,43,49,2,90,218};
int value[] = {6,3,5,4,6,1,9,7,6,7,20,20,80,2,3,43,49,2,90,218};
int maxweight = 500;
```

并记录两种算法的运行时间：

```
E:\University\java\develop_tools\jdk1.8.0_131\bin\java
背包内最大的物品价值总和为：516
Run Time:0(ms)
```

图 动态规划算法的运行时间

```
E:\University\java\develop_tools\jdk1.8.0_131\bin\java ...
最佳选择为：[1,1,0,0,1,0,1,0,1,1,1,1,1,0,0,0,1,1,1,1]
此时价值最大，即516
Run Time:15(ms)
```

图 回溯算法的运行时间

很明显能看出两种算法运行所需的时间不在一个数量级上。对于大规模的 0-1 背包问题，动态规划算法的时间复杂度更好。

三、实验总结

在这次实验中，我设计了一种算法来解决 0-1 背包问题。这是一类经典的 NP 问题，目标是在限定的背包容量内，最大化物品的价值。

在这次实验中，我们分别设计了动态规划算法和回溯算法来解决这个问题，首先来介绍动态规划算法。在动态规划算法中，我们通过递推来解决这个问题。首先，我们建立一个二维数组，其中第一维表示物品数量，第二维表示背包容量。接下来，我们遍历每一个物品，并根据其体积和价值，更新数组中的值。关键点在于，我们要明确状态转移方程。我们可以用下面的公式来表示：f[i][j] = max(f[i-1][j], f[i-1][j-v[i]] + w[i])。其中，f[i][j] 表示前 i 个物品，容量为 j 时的最大价值。v[i] 和 w[i] 分别表示第 i 个物品的体积和价值。需要注意在进行状态转移时，需要注意是否满足背包的容量限制。在代码的设计中，关键点有两点：一个是在设计状态转移方程时，需要考虑到当前物品是否可以放入背包中，以及选择不放入和放入这两种情况的最大收益。另一个是，在实现代码时，应该使用循环遍历的方式来枚举状态，并在每次遍历时进行状态转移。

而回溯算法中，易错点在于回溯算法求解背包问题时，会产生很多的分支，需要设置剪枝条件来减少不必要的分支。代码的关键点在于设计好递归结束条件，并且清楚地定义选和不选的状态转移方程。

通过这次实验，我对于 0-1 背包问题的动态规划算法和回溯算法有了更加深入的理解，并且对于动态规划算法中状态转移方程的设计和实现有了更加细致的掌握。我能够更加深入地理解 0-1 背包问题的解决方案，并能够根据实际情况选

择合适的算法来解决问题。

实验日期：2022 年 12 月 10 日

附录

实验源码如下：

Knapsack.java

```
package SolutionForLab5;

/**
 * @author ChenMao
 * @create 2022-12-10 14:21
 */

//0-1 背包问题
public class Knapsack {


    public static void main(String[] args) {
        Knapsack test = new Knapsack();
        int weight[] = {2,2,6,5,4};
        int value[] = {6,3,5,4,6};
        int maxweight = 10;
        System.out.println("背包内最大的物品价值总和为: " +
test.knapsack_dp(weight, value, maxweight));
    }

    public int knapsack_dp(int[] weight, int[] value, int maxweight){

        int n = weight.length;
        //最大价值数组为maxvalue[N+1][maxWeight+1]，因为我们要从0开始保
存
        int[][] maxvalue = new int[n+1][maxweight + 1];

        //重量和物品为0时，价值为0
        for (int i = 0; i < maxweight + 1; i++) {
            maxvalue[0][i] = 0;
```

```
        }
        for (int i = 0; i < n + 1; i++) {
            maxvalue[i][0] = 0;

        }

        //i：只拿前 i 件物品（这里的 i 因为取了 0，所以对应到 weight 和 value
里面都是 i-1 号位置）
        //j：假设能取的总重量为 j
        //n 是物品件数
        for (int i = 1; i <= n ; i++) {
            for (int j = 1; j <= maxweight; j++) {
                //当前最大价值等于放上一件的最大价值
                maxvalue[i][j] = maxvalue[i-1][j];
                //如果当前件的重量小于总重量，可以放进去或者拿出别的东西再放
进去
                if (weight[i-1] <= j) {
                    //比较（不放这个物品的价值）和
                    //（这个物品的价值 加上 当前能放的总重量减去当前物品重量
时取前 i-1 个物品时的对应重量时候的最高价值）
                    if(maxvalue[i-1][j - weight[i-1]] +
value[i-1]>maxvalue[i-1][j]) {
                        maxvalue[i][j] = maxvalue[i-1][j - weight[i-1]]
+ value[i-1];
                    }
                }
            }
        }

        System.out.println();
        System.out.println("动态规划表如下：");

        for(int i = 1; i <= n ; i++) {
            for(int j = 1; j <= maxweight; j++) {
                System.out.print(maxvalue[i][j] + "\t");
            }
            System.out.println();
        }

        return maxvalue[n][maxweight];
    }

}
```

Knapsack_backtrace.java

```java
package SolutionForLab5;

/**
 * @author ChenMao
 * @create 2022-12-10 16:14
 */

//回溯算法
public class Knapsack_backtrace {
    public int[] weight;
    public int[] value;
    public int[] take;
    int curWeight = 0;
    int curValue = 0;
    int bestValue = 0;
    int[] bestChoice;
    int maxWeight = 0;
    int count;

    public Knapsack_backtrace(int[] weight, int[] value, int maxWeight,
int n) {
        this.value = value;
        this.weight = weight;
        this.maxWeight = maxWeight;
        this.count = n;
        take = new int[n];
        bestChoice = new int[n];
    }

    public int[] backtrack(int x) {
        //终止条件
        if (x > count - 1) {
            //更新最优解
            if (curValue > bestValue) {
                bestValue = curValue;
                for (int i = 0; i < take.length; i++) {
                    bestChoice[i] = take[i];
                }
            }
        } else { //遍历当前节点（物品）的子节点：0 不放入背包 1：放入背包
            for (int i = 0; i < 2; i++) {
                take[x] = i;
                if (i == 0) {
                    //不放入背包，接着往下走
```

```
                    backtrack(x + 1);//递归下一个物品
                } else {
                    //约束条件，如果小于背包容量
                    if (curWeight + weight[x] <= maxWeight) {
                        //更新当前重量和价值
                        curWeight += weight[x];
                        curValue += value[x];
                        //继续向下深入
                        backtrack(x + 1);
                        //回溯
                        curWeight -= weight[x];
                        curValue -= value[x];
                    }
                }
            }
        }
        return bestChoice;
    }


    public static void backtrace_method(int[] w, int[] v, int m, int n)
{

        Knapsack_backtrace bt = new Knapsack_backtrace(w, v, m, n);
        int[] result = bt.backtrack(0);
        System.out.print("最佳选择为: [");
        for (int i = 0; i < bt.bestChoice.length; i++) {
            if (i == bt.bestChoice.length - 1) {
                System.out.print(bt.bestChoice[i] + "]");
            } else {
                System.out.print(bt.bestChoice[i] + ",");
            }
        }
        System.out.print("\n此时价值最大，即" + bt.bestValue + "\n");

    }

    public static void main(String[] args) {
        int[] weight = {2,2,6,5,4};
        int[] value = {6,3,5,4,6};
        int n = weight.length;
        int maxweight = 10;

        backtrace_method(weight,value,maxweight,n);
```

```
    }
}
```