# 背包九讲(超值得看的一大坨资料，附代码、视频、资料...)

原创 未名湖畔种千玺 已于 2022-04-20 22:18:36 修改 阅读量1.2k 收藏 11 点赞数 2 

文章标签： 算法 背包

## 背包九讲概述

### 1. 01背包问题

> 题目


- 01背包问题

每件物品只能选0次或者1次

有 $N$ 件物品和一个容量是 $V$ 的背包。每件物品只能使用一次。

第 $i$ 件物品的体积是 $v_i$，价值是 $w_i$。

求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总价值最大。
输出最大价值。

**输入格式**

第一行两个整数，$N$，$V$，用空格隔开，分别表示物品数量和背包容积。

接下来有 $N$ 行，每行两个整数 $v_i, w_i$，用空格隔开，分别表示第 $i$ 件物品的体积和价值。

**输出格式**

输出一个整数，表示最大价值。

**数据范围**

$0 < N, V \leq 1000$
$0 < v_i, w_i \leq 1000$



## 2. 完全背包问题

> 题目

- 完全背包问题

每件物品可以选无数次

有 $N$ 种物品和一个容量是 $V$ 的背包，每种物品都有无限件可用。

第 $i$ 种物品的体积是 $v_i$，价值是 $w_i$。

求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总价值最大。
输出最大价值。

**输入格式**

第一行两个整数，$N$，$V$，用空格隔开，分别表示物品种数和背包容积。

接下来有 $N$ 行，每行两个整数 $v_i, w_i$，用空格隔开，分别表示第 $i$ 种物品的体积和价值。

**输出格式**

输出一个整数，表示最大价值。

**数据范围**

$0 < N, V \leq 1000$
$0 < v_i, w_i \leq 1000$



## 3. 多重背包问题

> 题目

- 多重背包问题 I

限定每件物品的数量

有 $N$ 种物品和一个容量是 $V$ 的背包。

第 $i$ 种物品最多有 $s_i$ 件，每件体积是 $v_i$，价值是 $w_i$。

求解将哪些物品装入背包，可使物品体积总和不超过背包容量，且价值总和最大。
输出最大价值。

**输入格式**

第一行两个整数，$N$，$V$，用空格隔开，分别表示物品种数和背包容积。

接下来有 $N$ 行，每行三个整数 $v_i, w_i, s_i$，用空格隔开，分别表示第 $i$ 种物品的体积、价值和数量。

**输出格式**

输出一个整数，表示最大价值。

**数据范围**

$0 < N, V \leq 100$
$0 < v_i, w_i, s_i \leq 100$



## 4. 混合背包问题

> 题目

- 混合背包问题

有的物品只能选一次，有的可以选无限次，有的可以选n次

有 $N$ 种物品和一个容量是 $V$ 的背包。

物品一共有三类：

- 第一类物品只能用1次（01背包）；
- 第二类物品可以用无限次（完全背包）；
- 第三类物品最多只能用 $s_i$ 次（多重背包）；

每种体积是 $v_i$，价值是 $w_i$。

求解将哪些物品装入背包，可使物品体积总和不超过背包容量，且价值总和最大。
输出最大价值。

### 输入格式

第一行两个整数，$N$，$V$，用空格隔开，分别表示物品种数和背包容积。

接下来有 $N$ 行，每行三个整数 $v_i, w_i, s_i$，用空格隔开，分别表示第 $i$ 种物品的体积、价值和数量。

- $s_i = -1$ 表示第 $i$ 种物品只能用1次；
- $s_i = 0$ 表示第 $i$ 种物品可以用无限次；
- $s_i > 0$ 表示第 $i$ 种物品可以使用 $s_i$ 次；

### 输出格式

输出一个整数，表示最大价值。

### 数据范围

$0 < N, V \leq 1000$
$0 < v_i, w_i \leq 1000$
$-1 \leq s_i \leq 1000$

## 5. 二维费用的背包问题

> 题目

- 二维费用的背包问题

物品拥有体积质量和价值

有 $N$ 件物品和一个容量是 $V$ 的背包，背包能承受的最大重量是 $M$。

每件物品只能用一次。体积是 $v_i$，重量是 $m_i$，价值是 $w_i$。

求解将哪些物品装入背包，可使物品总体积不超过背包容量，总重量不超过背包可承受的最大重量，且价值总和最大。
输出最大价值。

**输入格式**

第一行三个整数，$N, V, M$，用空格隔开，分别表示物品件数、背包容积和背包可承受的最大重量。

接下来有 $N$ 行，每行三个整数 $v_i, m_i, w_i$，用空格隔开，分别表示第 $i$ 件物品的体积、重量和价值。

**输出格式**

输出一个整数，表示最大价值。

**数据范围**

$0 < N \leq 1000$
$0 < V, M \leq 100$
$0 < v_i, m_i \leq 100$
$0 < w_i \leq 1000$



## 6. 分组背包问题

> 题目

- 分组背包问题

若干组物品，每一组只能选一个

有 $N$ 组物品和一个容量是 $V$ 的背包。

每组物品有若干个，同一组内的物品最多只能选一个。
每件物品的体积是 $v_{ij}$，价值是 $w_{ij}$，其中 $i$ 是组号，$j$ 是组内编号。

求解将哪些物品装入背包，可使物品总体积不超过背包容量，且总价值最大。

输出最大价值。

**输入格式**

第一行有两个整数 $N$，$V$，用空格隔开，分别表示物品组数和背包容量。

接下来有 $N$ 组数据：

- 每组数据第一行有一个整数 $S_i$，表示第 $i$ 个物品组的物品数量；
- 每组数据接下来有 $S_i$ 行，每行有两个整数 $v_{ij}, w_{ij}$，用空格隔开，分别表示第 $i$ 个物品组的第 $j$ 个物品的体积和价值；

**输出格式**

输出一个整数，表示最大价值。

**数据范围**

$0 < N, V \leq 100$
$0 < S_i \leq 100$
$0 < v_{ij}, w_{ij} \leq 100$



## 7. 背包问题求方案数

> 题目

- 背包问题求方案数

最优解的数量

有 $N$ 件物品和一个容量是 $V$ 的背包。每件物品只能使用一次。

第 $i$ 件物品的体积是 $v_i$，价值是 $w_i$。

求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总价值最大。

输出 **最优选法的方案数**。注意答案可能很大，请输出答案模 $10^9+7$ 的结果。

**输入格式**

第一行两个整数，$N$，$V$，用空格隔开，分别表示物品数量和背包容积。

接下来有 $N$ 行，每行两个整数 $v_i, w_i$，用空格隔开，分别表示第 $i$ 件物品的体积和价值。

**输出格式**

输出一个整数，表示 **方案数** 模 $10^9+7$ 的结果。

**数据范围**

$0 < N, V \leq 1000$
$0 < v_i, w_i \leq 1000$



## 8. 求背包问题的方案

> 题目

- 背包问题求具体方案

输出最优方案

有 $N$ 件物品和一个容量是 $V$ 的背包。每件物品只能使用一次。

第 $i$ 件物品的体积是 $v_i$，价值是 $w_i$。

求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总价值最大。

输出 **字典序最小的方案**。这里的字典序是指：所选物品的编号所构成的序列。物品的编号范围是 $1 \dots N$。

**输入格式**

第一行两个整数，$N$，$V$，用空格隔开，分别表示物品数量和背包容积。

接下来有 $N$ 行，每行两个整数 $v_i, w_i$，用空格隔开，分别表示第 $i$ 件物品的体积和价值。

**输出格式**

输出一行，包含若干个用空格隔开的整数，表示最优解中所选物品的编号序列，且该编号序列的字典序最小。

物品编号范围是 $1 \dots N$。

**数据范围**

$0 < N, V \leq 1000$
$0 < v_i, w_i \leq 1000$



## 9. 有依赖的背包问题

如果选择某个物品，就必须先选择他的父节点物品

## 10、用最少的物品填满背包

> 题目

- 322. 零钱兑换

用最少的物品填满背包


# 1、01背包

动态规划DP0-1背包

## 理解

> 思路

首先要理解：

**f(k,w):当背包容量为w，现有k件物品可以偷，所能偷盗的最大价值**

**01背包是后面八种背包的基础，搞清楚01背包后面八种背包一看就会！！！！**

**谨记一句话：一句话先循环物品再循环体积再循环决策**

## 问题描述

现有四个物品，小偷背包总容量为8，怎么可以偷得价值最多的物品？

如：

| 物品编号 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 物品重量 | 2 | 3 | 4 | 5 |
| 物品价值 | 3 | 4 | 5 | 8 |

状态转移方程：

$$f(k,w)=\begin{cases} f(k-1,w)，w_k>w（太重，放不下） \\ \max\{f(k-1,w),\ f(k-1,w-w_k)+v_k\},\ w_k<=w \end{cases}$$

记f(k,w)：当背包容量为w，现有k件物品可以偷，所能偷到最大价值

记f(k,w)：当背包容量为w，现有k件物品可以偷，所能偷到最大价值

| 背包容量 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1:w=2,v=3 | 0 | 0 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| 2:w=3,v=4 | 0 | 0 | 3 | 4 | 4 | 7 | 7 | 7 | 7 |
| 3:w=4,v=5 | 0 | 0 | 3 | 4 | 5 | 7 | 8 | 9 | 9 |
| 4:w=5,v=8 | 0 | 0 | 3 | 4 | 5 | 8 | 8 | 11 | 12 |

状态转移方程：

$$f(k,w)=\begin{cases} f(k-1,w)，w_k>w（太重，放不下） \\ \max\{f(k-1,w),\ f(k-1,w-w_k)+v_k\},\ w_k<=w \end{cases}$$




## 代码实现

> 代码

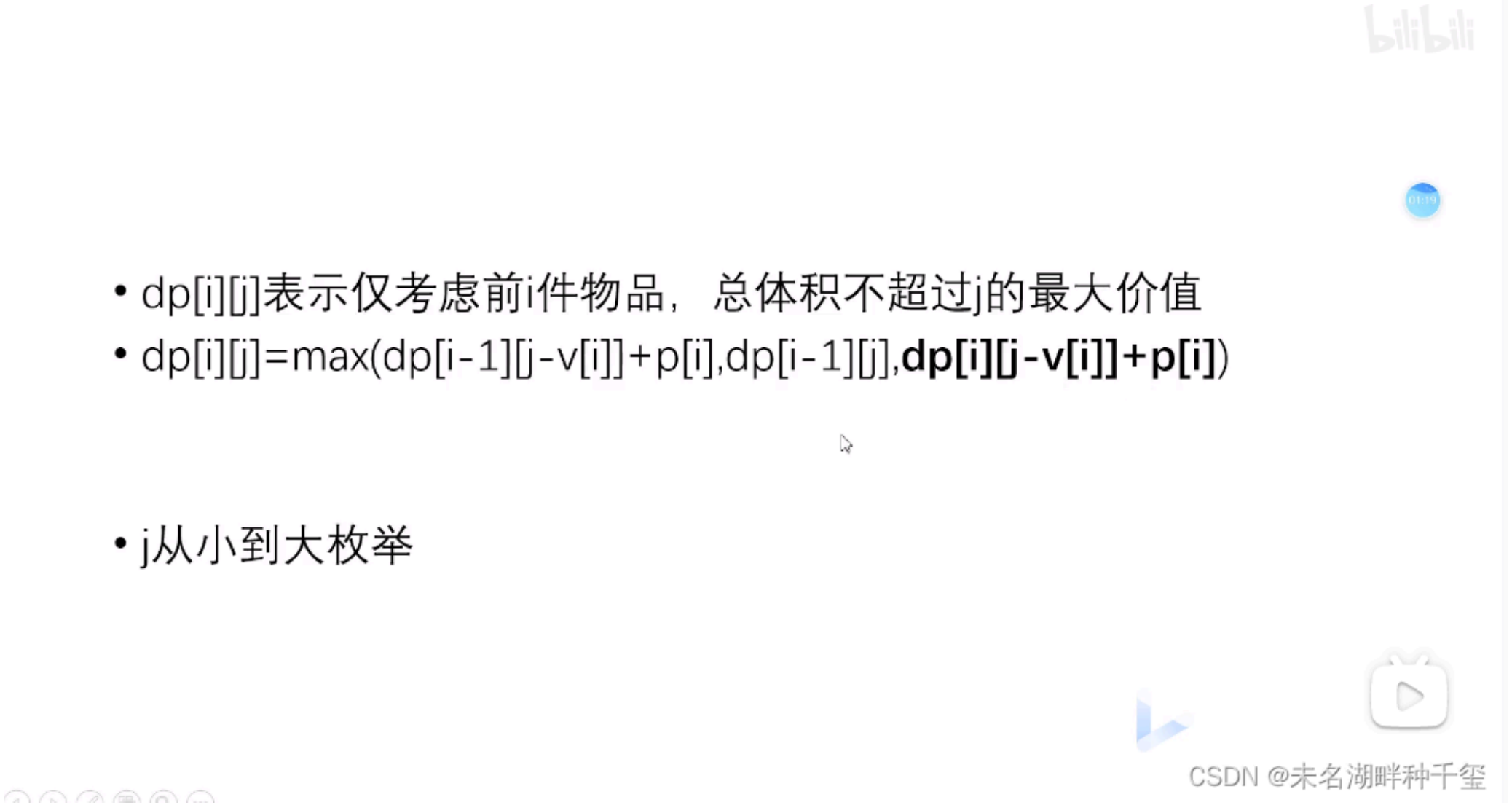
```java

package com.review.backpack_01;

public class BP01 {
    //01背包问题
    //每件物品只能选0次或者1次
    private int maxValue(int[] weight, int[] value, int bagWeight) {//常规做法
        // dp[i][j]的含义是：可选物品为i，背包容量是j时背包的最大价值
        // new int[weight.length][bagWeight + 1]为什么是bagWeight + 1呢?
        // 因为j代表的背包的真实容量，但是数组是从0开始的
        int[][] dp = new int[weight.length][bagWeight + 1];

        // 因为dp[0][j]中的0对应的是第一件物品，所以dp[0][j]的含义是可选物品为1，背包容量为j；
        // 当j大于等于weight[0]此时背包最大价值就是value[0]也就是第一件物品的价值
        for (int j = bagWeight; j >= weight[0]; j--) {
            dp[0][j] = value[0];
        }

        for (int i = 1; i < dp.length; i++) {
            for (int j = 1; j < dp[0].length; j++) {
                if (j < weight[i]) {
                    dp[i][j] = dp[i - 1][j];
                } else {
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i - 1][j - weight[i]] + value[i]);
                }
            }
        }
        return dp[dp.length - 1][dp[0].length - 1];
    }

    private int maxValue2(int[] weight, int[] value, int bagWeight) {//改进做法，使用一维数组

```

```
        int[] dp = new int[bagWeight + 1];

        for (int i = 0; i < weight.length; i++) {
            for (int j = bagWeight; j >= weight[i]; j--) {
                dp[j] = Math.max(dp[j], dp[j - weight[i]] + value[i]);
            }
        }
        return dp[dp.length - 1];
    }

    public static void main(String[] args) {
        System.out.println("常规做法，使用二维数组：" + new BP01().maxValue(new int[]{1, 3, 4}, new int[]{15, 20, 30}
        System.out.println("改进做法，使用一维数组：" + new BP01().maxValue2(new int[]{1, 3, 4}, new int[]{15, 20, 30
    }
}
```

运行结果

```
常规做法，使用二维数组：35
改进做法，使用一维数组：35

Process finished with exit code 0
```

## 2、完全背包




## 代码实现

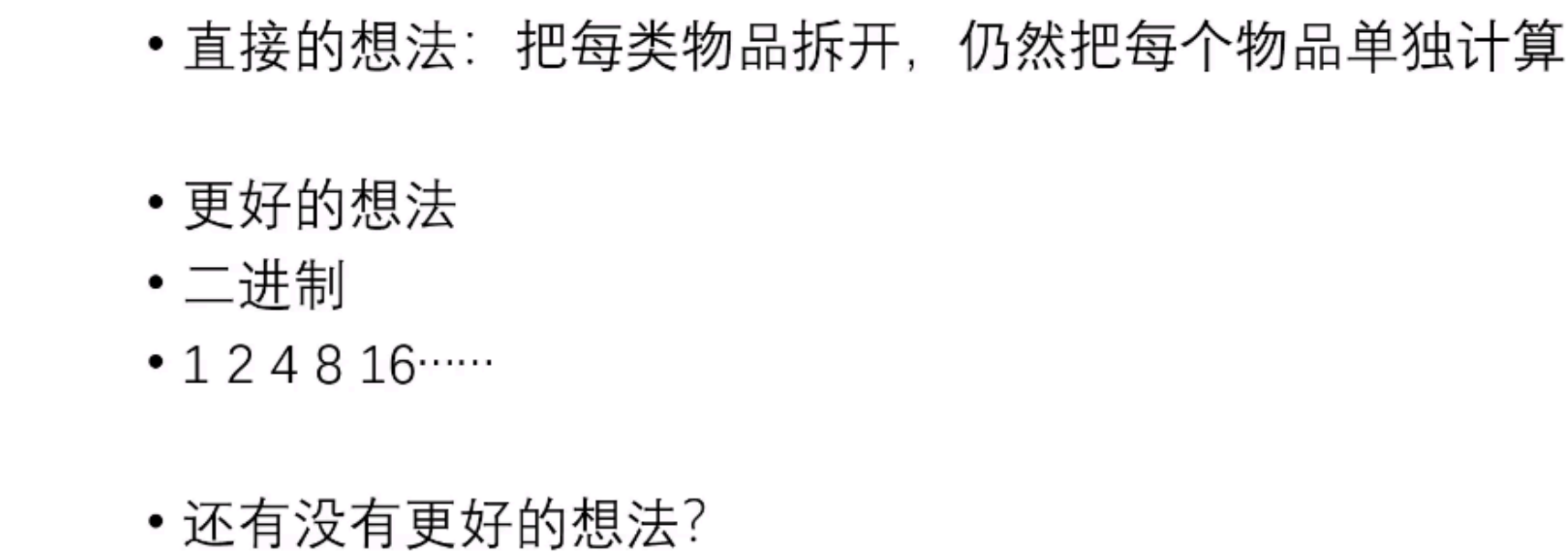
```
package com.review.backpack_01;

public class BP02 {

```

```java
    //2. 完全背包问题
    //每件物品可以选无数次
    private int maxValue(int[] weight, int[] value, int bagWeight) {//改进做法
        int[] dp = new int[bagWeight + 1];
        for (int i = 0; i < weight.length; i++) {
            for (int j = weight[i]; j <= bagWeight; j++) {
                dp[j] = Math.max(dp[j], dp[j - weight[i]] + value[i]);
            }
        }
        return dp[dp.length - 1];
    }

    private int maxValue2(int[] weight, int[] value, int bagWeight) {//常规做法
        int[] dp = new int[bagWeight + 1];
        for (int i = 0; i < weight.length; i++) {
            for (int j = bagWeight; j >= weight[i]; j--) {
                for (int k = 0; k * weight[i] <= j; k++) {
                    dp[j] = Math.max(dp[j], dp[j - k * weight[i]] + k * value[i]);
                }
            }
        }
        return dp[dp.length - 1];
    }

    public static void main(String[] args) {
        System.out.println("常规做法: " + new BP02().maxValue2(new int[]{1, 3, 4}, new int[]{15, 20, 30}, 4));
        System.out.println("改进做法: " + new BP02().maxValue(new int[]{1, 3, 4}, new int[]{15, 20, 30}, 4));
    }
}
```

> 运行结果

```
常规做法：60
改进做法：60

Process finished with exit code 0
```

## 3、多重背包






## 代码实现

```
1
2
3
```

```java
package com.review.backpack_01;
```

```java
public class BP03 {
    //3. 多重背包问题
    //限定每件物品的数量
    private int maxValue(int[] weight, int[] value, int[] num, int bagWeight) {

        int[] dp = new int[bagWeight + 1];
        for (int i = 0; i < weight.length; i++) {
            for (int j = bagWeight; j >= weight[i]; j--) {//前半部分和01背包相同
                for (int k = 1; k <= num[i] && k * weight[i] <= j; k++) {
                    dp[j] = Math.max(dp[j], dp[j - k * weight[i]] + k * value[i]);
                }
            }
        }
        return dp[dp.length - 1];
    }

    //优化
    private int maxValue2(int[] weight, int[] value, int[] num, int bagWeight) {

        int[] dp = new int[bagWeight + 1];
        for (int i = 0; i < weight.length; i++) {
            int s = num[i];
            for (int j = 1; j <= s; s -= j, j <<= 1) {
                for (int k = bagWeight; k >= 0 && k >= weight[i] * j; k--) {
                    dp[k] = Math.max(dp[k], dp[k - j * weight[i]] + j * value[i]);
                }
            }
            if (s > 0) {
                for (int j = bagWeight; j >= s * weight[i]; j--) {
                    dp[j] = Math.max(dp[j], dp[j - s * weight[i]] + s * value[i]);
                }
            }

```

```
        }
        return dp[dp.length - 1];
    }

    public static void main(String[] args) {
        System.out.println(new BP03().maxValue(new int[]{1, 2, 3, 4}, new int[]{2, 4, 4, 5}, new int[]{3, 1, 3, 2
        System.out.println(new BP03().maxValue2(new int[]{1, 2, 3, 4}, new int[]{2, 4, 4, 5}, new int[]{3, 1, 3,

    }
}
```

运行结果

```
10
10

Process finished with exit code 0
```

## 4、混合背包

### 代码实现

```
package com.some;

import com.review.backpack_01.BP04;

```

```java
public class BP04_test {
    /*
    混合背包
    p=-1  表示第 i 种物品只能用1次;
    p=0 表示第 i 种物品可以用无限次;
    p>0 表示第 i 种物品可以使用 p 次;
     */
    public static int knapsackProblem(int[] w, int[] v, int[] p, int cap) {
        int[] dp = new int[cap + 1];

        for (int i = 0; i < w.length; i++) {

            if (p[i] == -1) {
                // 01背包
                for (int j = cap; j >= w[i]; j--) {
                    dp[j] = Math.max(dp[j], dp[j - w[i]] + v[i]);
                }
            } else if (p[i] == 0) {
                // 完全背包
                for (int j = w[i]; j <= cap; j++) {
                    dp[j] = Math.max(dp[j], dp[j - w[i]] + v[i]);
                }
            } else {
                // 多重背包
                for (int j = cap; j >= w[i]; j--) {
                    for (int k = 0; (k <= p[i]) && (k * w[i] <= j); k++) {
                        dp[j] = Math.max(dp[j], dp[j - k * w[i]] + k * v[i]);
                    }
                }
            }
        }
        return dp[cap];
    }
```

```
39
40      public static void main(String[] args) {
41          System.out.println(
42                  knapsackProblem(
43                          new int[]{2, 3, 4, 5, 6, 7, 8},
44                          new int[]{3, 4, 5, 6, 7, 8, 9},
45                          new int[]{1, 1, 0, 0, 1, 3, 4}, 27
46                  )
47          );
48          System.out.println(
49                  knapsackProblem(
50                          new int[]{1, 2, 3, 4},
51                          new int[]{2, 4, 4, 5},
52                          new int[]{-1, 1, 0, 2}, 5
53                  )
54          );
55
56      }
57  }
```

运行结果

```
1  34
2  8
3
4  Process finished with exit code 0
```

## 5、二维费用的背包问题

### 代码实现

```
package com.review.backpack_01;

public class BP05 {
    //5. 二维费用的背包问题
    //物品拥有体积质量和价值
    public int knapsackProblem(int[] volume, int maxV, int[] weight, int maxW, int[] value) {
        int[][] dp = new int[maxV + 1][maxW + 1];
        for (int i = 0; i < volume.length; i++) {
            for (int j = maxV; j >= volume[i]; j--) {
                for (int k = maxW; k >= weight[i]; k--) {
                    dp[j][k] = Math.max(dp[j][k], dp[j - volume[i]][k - weight[i]] + value[i]);
                }
            }
        }
        return dp[maxV][maxW];
    }

    public static void main(String[] args) {
        System.out.println(
                new BP05().knapsackProblem(
                        new int[]{1, 2, 3, 4},
                        5,
                        new int[]{2, 4, 4, 5},
                        6,
                        new int[]{3, 4, 5, 6}
                )
        );

```

```
    }
}
```

运行结果

```
8

Process finished with exit code 0
```

## 6、 分组背包问题

### 代码实现

```
package com.review.backpack_01;

public class BP06 {
    //6. 分组背包问题
    //若干组物品，每一组只能选一个
    public int knapsackProblem(int[][] c, int[][] v, int cap) {
        int[] dp = new int[cap + 1];

        for (int i = 0; i < c.length; i++) {
            for (int j = cap; j >= 0; j--) {// 确保每个组中至多只有一个物品被选择
                for (int k = 0; k < c[i].length; k++) {
                    if (j >= c[i][k]) {
                        dp[j] = Math.max(dp[j], dp[j - c[i][k]] + v[i][k]);
                    }
```

```
                }
            }
        }
        return dp[cap];
    }

    public static void main(String[] args) {
        System.out.println(
                new BP06().knapsackProblem(
                        new int[][]{{1, 2, 3}, {2, 3, 4}, {5, 6, 7, 8}, {7, 9}},
                        new int[][]{{2, 3, 4}, {1, 2, 3}, {4, 5, 6, 7}, {8, 9}},
                        11
                )
        );
    }
}
```

运行结果

```
12

Process finished with exit code 0
```

## 7、背包问题求方案数

- AcWing 11. Java-背包问题求方案数—注释版代码

## 代码实现

```
package com.review.backpack_01;

import java.util.Arrays;

public class BP07_2 {
    //7. 背包问题求方案数
    //最优解的数量
    //在01背包里加入判断

    public static int knapsackProblem(int[] c, int[] v, int V) {
        int[] dp = new int[V + 1];
        int[] count = new int[V + 1];

        // 初始化，此时的count代表i=0，即没有物品可以选择时
        // 什么都不选，也是一种方案
        Arrays.fill(count, 1);

        for (int i = 0; i < c.length; i++) {
            for (int j = V; j >= c[i]; j--) {
                if (dp[j] < dp[j - c[i]] + v[i]) {
                    // 选择当前物品的收益要高，因此这里dp的选择必然是选择当前物品，
                    // 因为是必然，所以这样的方案数量不变
                    count[j] = count[j - c[i]];
                } else if (dp[j] == dp[j - c[i]] + v[i]) {
                    // 二者的收益相等，这里可以选，也可以不选
                    // 因此这里是加上count[j-v[i]]
                    count[j] += count[j - c[i]];
                } else {
                    // 不选的收益更高，那么这里dp的选择必然是不选
                    // 因此保持count[n-1][j]即可

```

```
                }
                dp[j] = Math.max(dp[j], dp[j - c[i]] + v[i]);
            }
        }
        return count[V];
    }

    public static int knapsackProblem_2(int[] c, int[] v, int V) {
        int[][] dp = new int[c.length + 1][V + 1];
        int[][] count = new int[c.length + 1][V + 1];

        // 初始化，此时的count代表i=0，即没有物品可以选择时
        // 什么都不选，也是一种方案
        for (int[] ints : count) {
            Arrays.fill(ints, 1);
        }


        for (int i = 1; i < c.length; i++) {
            for (int j = 0; j <= V; j++) {
                if (j >= c[i]) {
                    // 背包容量大于当前物品容量
                    if (dp[i - 1][j] < dp[i - 1][j - c[i]] + v[i]) {
                        // 选择当前物品的收益更高
                        // 更新当前收益
                        dp[i][j] = dp[i - 1][j - c[i]] + v[i];
                        // 当前方案只是在之前方案的基础上加了当前的物品，因此方案数量不变
                        count[i][j] = count[i - 1][j - c[i]];

                    } else if (dp[i - 1][j] == dp[i - 1][j - c[i]] + v[i]) {
                        // 两种办法收益相等
                        // 更新当前收益
```

```
                    dp[i][j] = dp[i - 1][j];
                    // 当前方案有两种选择，一是不增加当前物品，二是增加当前物品，所以方案数量等于二者之和
                    count[i][j] = count[i - 1][j] + count[i - 1][j - c[i]];
                } else {
                    // 选择当前物品的收益更高
                    // 更新当前收益
                    dp[i][j] = dp[i - 1][j];
                    // 当前方案不增加物品，与之前的方案等同
                    count[i][j] = count[i - 1][j];
                }
            } else {
                // 背包容量下雨当前物品容量
                dp[i][j] = dp[i - 1][j];
                count[i][j] = count[i - 1][j];
            }

        }
    }
    return count[c.length][V];
}

public static void main(String[] args) {
    System.out.println(knapsackProblem(new int[]{1, 3, 4}, new int[]{15, 20, 30}, 4));
    System.out.println(knapsackProblem_2(new int[]{1, 3, 4}, new int[]{15, 20, 30}, 4));
}
}
```

运行结果

```
1
1

Process finished with exit code 0
```

## 8、求背包问题的方案

### 代码实现

```
package com.review.backpack_01;

import java.util.ArrayList;
import java.util.Collections;
import java.util.LinkedList;
import java.util.List;

public class BP08_2 {
    //8. 求背包问题的方案
    //输出最优方案
    public static int knapsackProblem(int[] c, int[] v, int cap) {
        // 记录最大价值
        int[][] dp = new int[c.length][cap + 1];
        // 记录当前物品是否被选择
        int[][] choose = new int[c.length][cap + 1];

        for (int i = cap; i >= c[0]; i--) {
            dp[0][i] = v[0];
            choose[0][i] = 1;
        }
        for (int i = 1; i < c.length; i++) {
            for (int j = 1; j <= cap; j++) {
```

```
                if (j >= c[i]) {
                    if (dp[i - 1][j] <= dp[i - 1][j - c[i]] + v[i]) {
                        // 选择当前物品为最优解
                        dp[i][j] = dp[i - 1][j - c[i]] + v[i];
                        choose[i][j] = 1;
                    } else {
                        dp[i][j] = dp[i - 1][j];
                    }
                } else {
                    dp[i][j] = dp[i - 1][j];
                }
            }
        }
        // 反推出物品编号
        List<Integer> list = new LinkedList<>();
        int i = c.length - 1, V = cap;
        while (i >= 0) {
            if (choose[i][V] != 0) {
                list.add(i);
                V -= c[i];
            }
            i--;
        }
        Collections.reverse(list);
        System.out.println("最优方案选择: " + list);

        return dp[dp.length - 1][dp[0].length - 1];
    }

    public static void main(String[] args) {
        System.out.println(knapsackProblem(new int[]{1, 3, 4}, new int[]{15, 20, 30}, 4));
    }

```

```
}
```

运行结果

```
最优方案选择：[0, 1]
35

Process finished with exit code 0
```

## 10、用最少的物品填满背包

### 代码实现

```
import java.util.Arrays;

public class BP10 {
    public static int coinChange(int[] coins, int amount) {
        int[] dp = new int[amount + 1];
        Arrays.fill(dp, amount + 1);
        dp[0] = 0;
        for (int i = 0; i < coins.length; i++) {
            for (int j = coins[i]; j <= amount; j++) {
                dp[j] = Math.min(dp[j], dp[j - coins[i]] + 1);
            }
        }
        return dp[amount] == (amount + 1) ? -1 : dp[amount];
```

```
    }

    public static void main(String[] args) {
        System.out.println(coinChange(new int[]{1, 2, 5}, 11));
    }
}
```

运行结果

```
3

Process finished with exit code 0
```

## 资料引用

### 视频链接(B站)

https://www.bilibili.com/video/BV1qt411Z7nE?from=search&seid=1595971988062633820

https://www.bilibili.com/video/av34467850/?p=2&spm_id_from=333.788.b_636f6d6d656e74.18

### 题库链接

https://www.acwing.com/problem/

### 优秀博客(借鉴)

https://blog.csdn.net/qq_14873461/category_9800545.html

## 网上资料

> 背包问题九讲

https://ishare.iask.sina.com.cn/f/17570695.html

> 背包九讲 2.0.pdf

https://max.book118.com/html/2017/0615/115637327.shtm

## 代码

> 码云

https://gitee.com/huangjialin3344/leet-code-test/tree/master/LeetCode_time_1022/src/main/java/com/review/backpack_01

## 总结

一句话先循环物品再循环体积再循环决策