Knapsack problem

1. 01Knapsack problem

1.1 Problem Description

There are N items and a backpack with a capacity of V; each item can only be used once. The size of the i item is vi and the value is wi. Solve which items are loaded into the backpack so that the total volume of these items does not exceed the backpack capacity and the total value is the largest. Output the maximum value.

1.2 Method

The number of items is N, the total capacity of the backpack is V, c[] represents the volume of each item, w[] represents the value of each item. Let f[i][j] be the maximum value of the backpack with the capacity of j in the first i item, so it is possible to traverse the item and calculate the maximum value when the assumed capacity is j.
There are two situations at this time:

- a. If the capacity is not enough (j < c[i]), the current item cannot be taken, directly f[i][j]=f[i-1][j]
- b. If the capacity is enough, judge the value of taking and not taking the current item.
- If the i-th piece is not taken, the problem is transformed into the problem that the former i-1 item is placed in the backpack of the capacity v, that is, f[i][j]=f[i-1][j]
- If the i-th piece is taken, the problem is transformed into the problem that the former i-1 item is placed in the backpack of the capacity jc[i], that is, f[i][j]=f[i-1][jc[i]] +w[i]

Finally, get f[N][V].

1.3 Code

```
1.  import java.util.Scanner;
2.
3.  public class Main {
4.      public static void main(String[] args) {
5.          Scanner sc = new Scanner(System.in);
6.          int N = sc.nextInt();
7.          int V = sc.nextInt();
8.          int[] v = new int[N + 1];
9.          int[] w = new int[N + 1];
10.         for(int i=1; i<=N; i++) {
11.             v[i] = sc.nextInt();
12.             w[i] = sc.nextInt();
13.         }
14.
15.         System.out.println(cal(N, V, w, v));
16.     }
17.
```

```
    /**
     * 求01背包的解
     * n 物品个数
     * v 背包总容量
     * w 每个物品的价值，注意下标从1开始
     * c 每个物品的体积，注意下标从1开始
     *
     */
    private static int cal(int n, int v, int[] w, int[] c) {
        /*

         */

        if (n == 0 || v == 0) return 0;
        if (w == null || w.length == 0) return 0;
        if (c == null || c.length == 0) return 0;


        int[][] f = new int[n + 1][v + 1];
        for(int i=1; i<=n; i++) {
            for(int j=1;j<=v;j++) {
                f[i][j] = j<c[i]? f[i-1][j] : Math.max(f[i-1][j], f[i
-1][j-c[i]] + w[i]);
            }
        }

        return f[n][v];
    }
}
```

2. Limit the number of items within m_i (0 < m_i ≤N)

2.1 Problem Description

The same as problem 1, but the number of items should be m_i.

2.2 Method

The differerence is that the state transmission function changes to this form:
F[i, v] = max{F[i-1, v], F[i-1, v – k*vi] + k*wi} 0 < mi≤N 0 < k≤mi
F[i, v] is when one puts the i items into the package in which the volume is v. i is not the total number of items in the package. The total number is $\sum_{k=1}^{i} m_k$.
Then we need to choice if we will put the i-th item into the package or not and the we select the maximal one.

2.3 Code

```
#include <stdio.h>
#include <stdlib.h>
```

```
typedef struct rice {
    int price, weight, num;
} rice;

void dynamic(rice *rices, int m, int n)
{
    int i, j, cur, k, **dp;

    dp = (int **)malloc(sizeof(int *) * (m + 1));
    for (i = 0; i <= m; i ++)
        dp[i] = (int *)malloc(sizeof(int) * (n + 1));

    // Inititation
    for (i = 0; i <= m; i ++)
        for (j = 0; j <= n; j ++)
            dp[i][j] = 0;

    // DP
    for (i = 1; i <= m; i ++) {
        for (j = 1; j <= n; j ++) {
            for (k = 0; k <= rices[i].num; k ++) {
                if (j - k * rices[i].price >= 0) {
                    cur = dp[i - 1][j - k * rices[i].price] + k * rices
[i].weight;
                    dp[i][j] = dp[i][j] > cur ? dp[i][j] : cur;
                } else {
                    break;
                }
            }
        }
    }

    printf("%d\n", dp[m][n]);

    for (i = 0; i <= m; i ++)
        free(dp[i]);
}


int main(void)
{
    int i, c, n, m;

    rice rices[2010];

    scanf("%d", &c);

    while (c --) {
        scanf("%d %d", &n, &m);

        for (i = 1; i <= m; i ++) {
```

```
                scanf("%d %d %d", &rices[i].price, &rices[i].weight, &rices
[i].num);
            }
            dynamic(rices, m, n);
        }
        return 0;
}
```

3. Complete Knapsack Problem

3.1 Problem Description

The same as problem 1, but $\sum_{i=1}^{n} w_i$ should be equal to V.

3.2 Method

The same as problem 1, except for matrix initialization. Set F[0][0]=0 and others are –inf.

1. When i=1, judge the loading of the first item
2. Since all f[i][j] are -inf at the beginning except f[0][0]=0, then f[i][j] can be made not equal to -inf only when j= w[1]. (Other cases f[j-w[1]]+c[1] = -inf+c[i] = -inf
3. When i=2, judge the loading of the second item
4. Now except for f[0][0]=0 and f[2,w[1]]=c[1], all f[i][j] are still -inf, then j=w[2], or j=w[1]+w[2] to make f[2,j] not equal to -inf.
5. And so on.
6. Finally, if f[i][j] is equal to -inf, then the capacity j cannot be filled. Otherwise, get the answer.

3.3 Code

```
1.  import java.util.Scanner;
2.
3.  public class Main {
4.      public static void main(String[] args) {
5.          Scanner sc = new Scanner(System.in);
6.          int N = sc.nextInt();
7.          int V = sc.nextInt();
8.          int[] v = new int[N + 1];
9.          int[] w = new int[N + 1];
10.         for(int i=1; i<=N; i++) {
11.             v[i] = sc.nextInt();
12.             w[i] = sc.nextInt();
13.         }
14.
15.         System.out.println(cal(N, V, w, v));
16.     }
17.         private static int cal(int n, int v, int[] w, int[] c)
18.         if (n == 0 || v == 0) return 0;
19.         if (w == null || w.length == 0) return 0;
20.         if (c == null || c.length == 0) return 0;
```

```
21.
22.         int[][] f = new int[n + 1][v + 1];
23.          for(int i=1; i<=n; i++){
24.              for(int j=1;j<=v;j++){
25.                   f[i][j]=-inf;}
26.          }
27.          f[1][1]=0;
28.         for(int i=1; i<=n; i++) {
29.             for(int j=1;j<=v;j++) {
30.                 f[i][j] = j<c[i]? f[i-1][j] : Math.max(f[i-1][j], f[i
-1][j-c[i]] + w[i]);
31.             }
32.         }
33.
34.         return f[n][v];
35.     }
36. }
```