==========
文件夹: class053_BoundedKnapsackProblem
==========

[Markdown 文件]
==========
文件: README.md
==========
多重背包问题 (Bounded Knapsack Problem)

算法介绍

多重背包问题是背包问题的一个变种，它介于 01 背包和完全背包之间。在多重背包问题中，每种物品有固定的数量限制，既不能像 01 背包那样只选择 0 个或 1 个，也不能像完全背包那样选择任意多个，而是最多只能选择指定数量的物品。

问题定义

给定一个容量为 V 的背包，有 n 种物品，每种物品 i 有：
- 价值：v[i]
- 重量：w[i]
- 数量：c[i]

要求在不超过背包容量的前提下，使得装入背包的物品总价值最大。

数学表达

目标函数：最大化 Σ(v[i] * x[i])，其中 x[i]表示选择第 i 种物品的数量
约束条件：
1. Σ(w[i] * x[i]) ≤ V
2. 0 ≤ x[i] ≤ c[i]

算法实现

1. 基础实现 (Code01_BoundedKnapsack.java)

最直观的实现方式，对每种物品枚举选择的数量。

时间复杂度：O(n * V * C)，其中 C 是每种物品的平均数量
空间复杂度：O(V)

2. 二进制优化 (Code02_BoundedKnapsackWithBinarySplitting.java)

通过二进制分组将多重背包转化为01背包问题。

时间复杂度：O(V * Σ(log c[i]))
空间复杂度：O(V)

3. 单调队列优化 (Code04_BoundedKnapsackWithMonotonicQueue.java)

使用单调队列优化状态转移过程。

时间复杂度：O(n * V)
空间复杂度：O(V)

4. 混合背包 (Code05_MixedKnapsack.java)

处理同时包含01背包、完全背包和多重背包的混合情况。

5. 多维01背包 (Code06_OnesAndZeroes.java)

解决多维资源约束的01背包问题，如LeetCode 474. Ones and Zeroes。

6. 二维费用背包 (Code07_ProfitableSchemes.java)

解决同时考虑多个约束条件的背包问题，如LeetCode 879. Profitable Schemes。

7. 多重背包可行性问题 (Code08_Coins.java)

解决多重背包的可行性问题，如POJ 1742. Coins。

8. 经典多重背包问题 (Code09_HDU2191.java)

解决经典的多重背包问题，如HDU 2191。

9. 多重背包应用问题 (Code10_Codeforces106C.java)

解决多重背包在实际问题中的应用，如Codeforces 106C. Buns。

相关题目扩展（全面搜索各大算法平台）

LeetCode (力扣) - 背包问题专题

01背包问题
1. **LeetCode 416. Partition Equal Subset Sum** - https://leetcode.cn/problems/partition-equal-subset-sum/

01背包可行性问题，判断是否能将数组分割成两个和相等的子集

2. **LeetCode 1049. Last Stone Weight II** - https://leetcode.cn/problems/last-stone-weight-ii/
   01背包变形，最小石头重量差

3. **LeetCode 494. Target Sum** - https://leetcode.cn/problems/target-sum/
   01背包计数问题，寻找目标和子集个数

完全背包问题
4. **LeetCode 322. Coin Change** - https://leetcode.cn/problems/coin-change/
   完全背包问题，求组成金额所需的最少硬币数

5. **LeetCode 518. Coin Change II** - https://leetcode.cn/problems/coin-change-ii/
   完全背包计数问题，求组成金额的方案数

6. **LeetCode 377. Combination Sum IV** - https://leetcode.cn/problems/combination-sum-iv/
   顺序相关的组合问题，类似完全背包

多维背包问题
7. **LeetCode 474. Ones and Zeroes** - https://leetcode.cn/problems/ones-and-zeroes/
   多维01背包问题，每个字符串需要同时消耗0和1的数量

8. **LeetCode 879. Profitable Schemes** - https://leetcode.cn/problems/profitable-schemes/
   二维费用背包问题，需要同时考虑人数和利润

9. **LeetCode 956. Tallest Billboard** - https://leetcode.cn/problems/tallest-billboard/
   较复杂的二维背包问题

其他背包变形
10. **LeetCode 1220. Count Vowels Permutation** - https://leetcode.cn/problems/count-vowels-permutation/
    状态转移类似背包问题

11. **LeetCode 1449. Form Largest Integer With Digits That Add up to Target** - https://leetcode.cn/problems/form-largest-integer-with-digits-that-add-up-to-target/
    背包问题与字符串构造的结合

洛谷 (Luogu) - 中文算法题库

多重背包经典
12. **P1776 宝物筛选** - https://www.luogu.com.cn/problem/P1776
    经典多重背包问题，测试数据规模较大

13. **P1833 樱花** - https://www.luogu.com.cn/problem/P1833
    混合背包问题，包含01背包、完全背包和多重背包

完全背包应用
14. **P1679 神奇的四次方数** - https://www.luogu.com.cn/problem/P1679
    完全背包在数学问题中的应用

15. **P2077 星球大战** - https://www.luogu.com.cn/problem/P2077
    多维背包问题的实际应用

其他背包问题
16. **P1064 金明的预算方案** - https://www.luogu.com.cn/problem/P1064
    依赖背包问题，物品之间存在依赖关系

17. **P1679 聪明的收银员** - https://www.luogu.com.cn/problem/P1679
    多重背包在找零问题中的应用

POJ (北京大学在线评测系统)

多重背包优化
18. **POJ 1742. Coins** - http://poj.org/problem?id=1742
    多重背包可行性问题，计算能组成多少种金额

19. **POJ 1276. Cash Machine** - http://poj.org/problem?id=1276
    多重背包优化问题，使用二进制优化或单调队列优化

20. **POJ 3260. The Fewest Coins** - http://poj.org/problem?id=3260
    双向背包问题，同时考虑找零和支付

01背包经典
21. **POJ 3624. Charm Bracelet** - http://poj.org/problem?id=3624
    标准01背包问题，入门必做

22. **POJ 3628. Bookshelf 2** - http://poj.org/problem?id=3628
    01背包变形，求超过某个值的最小和

HDU (杭州电子科技大学OJ)

多重背包实战
23. **HDU 2191. 悼念512汶川大地震遇难同胞** - http://acm.hdu.edu.cn/showproblem.php?pid=2191
    经典多重背包问题，中文题目

24. **HDU 2191. 珍惜现在，感恩生活** - http://acm.hdu.edu.cn/showproblem.php?pid=2191

多重背包问题的实际应用

25. **HDU 2191. 非常可乐** - http://acm.hdu.edu.cn/showproblem.php?pid=2191
多重背包问题的变形

完全背包问题
26. **HDU 1114. Piggy-Bank** - http://acm.hdu.edu.cn/showproblem.php?pid=1114
完全背包问题，求装满背包的最小价值

27. **HDU 2159. FATE** - http://acm.hdu.edu.cn/showproblem.php?pid=2159
二维费用背包问题，同时考虑忍耐度和杀怪数

分组背包
28. **HDU 3449. Consumer** - http://acm.hdu.edu.cn/showproblem.php?pid=3449
有依赖的背包问题，需要先购买主件

Codeforces - 国际编程竞赛平台

多重背包应用
29. **Codeforces 106C. Buns** - https://codeforces.com/problemset/problem/106/C
多重背包问题，制作不同种类的面包

30. **Codeforces 148E. Porcelain** - https://codeforces.com/problemset/problem/148/E
分组背包问题，从每组中选择物品

背包思想扩展
31. **Codeforces 455A. Boredom** - https://codeforces.com/problemset/problem/455/A
打家劫舍类型的动态规划问题

32. **Codeforces 1003F. Abbreviation** - https://codeforces.com/contest/1003/problem/F
字符串处理与多重背包的结合

AtCoder - 日本编程竞赛平台

标准背包问题
33. **AtCoder ABC032 D. ナップサック問題** - https://atcoder.jp/contests/abc032/tasks/abc032_d
01 背包问题，数据规模较大需要优化

34. **AtCoder DP Contest D - Knapsack 1** - https://atcoder.jp/contests/dp/tasks/dp_d
标准 01 背包问题实现

35. **AtCoder DP Contest E - Knapsack 2** - https://atcoder.jp/contests/dp/tasks/dp_e
大体积小价值的 01 背包问题，需要价值维度 DP

背包思想应用
36. **AtCoder ABC153 F. Silver Fox vs Monster** - https://atcoder.jp/contests/abc153/tasks/abc153_f
    贪心+前缀和优化的背包问题

37. **AtCoder ABC224 E. Integers on Grid** - https://atcoder.jp/contests/abc224/tasks/abc224_e
    动态规划与背包思想结合

38. **AtCoder DP Contest Problem F** - https://atcoder.jp/contests/dp/tasks/dp_f
    最长公共子序列与背包思想的结合

SPOJ (Sphere Online Judge)

经典背包问题
39. **SPOJ KNAPSACK** - https://www.spoj.com/problems/KNAPSACK/
    经典01背包问题，国际知名题库

40. **SPOJ COINS** - https://www.spoj.com/problems/COINS/
    硬币问题，完全背包的变形

UVa OJ (国际大学程序设计竞赛题库)

背包问题实战
41. **UVa 562. Dividing coins** - https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=503
    01背包变形，公平分配硬币

42. **UVa 10130. SuperSale** - https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1071
    01背包问题的简单应用

ZOJ (浙江大学在线评测系统)

算法思想扩展
43. **ZOJ 2136. Longest Ordered Subsequence** - https://zoj.pintia.cn/problem-sets/91827364500/problems/91827364779
    最长递增子序列，可转化为背包思想

44. **ZOJ 1002. Fire Net** - https://zoj.pintia.cn/problem-sets/91827364500/problems/91827364501
    回溯法与背包思想结合

牛客网 - 国内编程题库

多重背包专项
45. **牛客网 NC19754. 多重背包** - https://ac.nowcoder.com/acm/problem/19754
    标准多重背包问题

46. **牛客网 NC17881. 最大价值** - https://ac.nowcoder.com/acm/problem/17881
    多重背包问题的变形应用

完全背包问题
47. **牛客网 NC16552. 买苹果** - https://ac.nowcoder.com/acm/problem/16552
    完全背包问题，中文题目

AcWing - 算法学习平台

多重背包优化
48. **AcWing 5. 多重背包问题 II** - https://www.acwing.com/problem/content/description/5/
    二进制优化的多重背包问题标准题目

剑指 Offer - 面试题库

动态规划基础
49. **剑指 Offer 42. 连续子数组的最大和** - https://leetcode.cn/problems/lian-xu-zi-shu-zu-de-zui-da-he-lcof/
    动态规划基础问题，背包思想的应用

50. **剑指 Offer 10- II. 青蛙跳台阶问题** - https://leetcode.cn/problems/qing-wa-tiao-tai-jie-wen-ti-lcof/
    动态规划基础问题，类似背包思想

本仓库代码实现对应题目

51. **Code01_BoundedKnapsack** - 多重背包基础实现
52. **Code02_BoundedKnapsackWithBinarySplitting** - 二进制优化多重背包
53. **Code03_UnboundedKnapsack** - 完全背包问题实现
54. **Code04_BoundedKnapsackWithMonotonicQueue** - 单调队列优化多重背包
55. **Code05_MixedKnapsack** - 混合背包问题实现
56. **Code06_OnesAndZeroes** - 多维 01 背包问题实现
57. **Code07_ProfitableSchemes** - 二维费用背包问题实现
58. **Code08_Coins** - POJ 1742 问题实现
59. **Code09_HDU2191** - HDU 2191 问题实现
60. **Code10_Codeforces106C** - Codeforces 106C 问题实现

算法技巧与工程化深度解析

适用场景识别与问题建模

典型应用场景
1. **资源分配问题**：在有限资源约束下实现收益最大化，每种资源有数量限制
2. **投资组合优化**：选择多种投资产品，每种产品有购买数量限制，在风险和收益之间取得平衡
3. **生产计划制定**：安排不同产品的生产数量，最大化利润，考虑产能限制
4. **物流配送优化**：在载重限制下选择最优配送方案，考虑货物数量限制
5. **项目选择问题**：在预算和时间约束下选择最优项目组合
6. **广告投放优化**：在预算限制下选择最优广告组合以最大化转化率

问题识别特征
- **输入特征**：物品数量、价值、重量、数量限制
- **约束条件**：总重量/容量限制
- **优化目标**：最大化总价值
- **关键指标**：数据规模（n, V, c[i]的大小关系）

解题思路与算法选择策略

四步解题法
1. **问题识别与建模**
   - 提取核心约束：容量限制、物品数量限制
   - 明确优化目标：最大化价值
   - 识别问题类型：01 背包、完全背包、多重背包

2. **状态定义与转移方程**
   - 状态定义：dp[i][j]表示前 i 种物品容量为 j 时的最大价值
   - 状态转移：考虑选择 0 到 c[i]个当前物品
   - 数学表达：max{ dp[i-1][j-k*w[i]] + k*v[i] }, 0≤k≤min(c[i], j/w[i])

3. **优化策略选择**
   - 小规模数据：基础三重循环实现
   - 中等规模：二进制优化（实现简单，效果显著）
   - 大规模数据：单调队列优化（理论最优，实现复杂）
   - 特殊场景：混合优化策略

4. **边界处理与异常防御**
   - 初始状态：dp[0][j] = 0
   - 边界条件：n=0, V=0, w[i]=0 等特殊情况
   - 数值溢出：使用合适的数据类型

算法选择决策树
```
```

数据规模分析 →
    如果 n*V*c_avg < 10^6: 基础实现
    如果 n*V*log(c_max) < 10^7: 二进制优化
    如果 n*V > 10^7: 单调队列优化
    如果包含特殊约束: 针对性优化
```

### 优化方法深度解析

#### 二进制优化（Binary Splitting）
**核心思想**：利用二进制表示将多重背包转化为 01 背包
- **数学原理**：任意正整数 c 可以唯一表示为不同 2 的幂次之和
- **实现步骤**：
  1. 对每个物品数量 c[i]进行二进制拆分
  2. 生成 $\log_2(c[i])$个新物品
  3. 对生成的新物品应用 01 背包算法
- **时间复杂度**：O(V * Σlog c[i])
- **空间复杂度**：O(V)
- **适用场景**：c[i]较大但非极端大的情况

#### 单调队列优化（Monotonic Queue）
**核心思想**：利用同余分组和单调队列维护滑动窗口最大值
- **数学变形**：
  ```
  原始方程：dp[i][j] = max{ dp[i-1][j-k*w[i]] + k*v[i] }
  变形后：dp[i][j] = max{ dp[i-1][r+l*w[i]] - l*v[i] } + m*v[i]
  其中 j = m*w[i] + r, l = m - k
  ```
- **实现关键**：
  1. 按余数 r 分组处理
  2. 对每组使用单调队列维护最大值
  3. 滑动窗口大小为 c[i]+1
- **时间复杂度**：O(n * V)
- **空间复杂度**：O(V)
- **适用场景**：数据规模非常大的情况

#### 混合优化策略
- **完全背包优化**：当 c[i]*w[i] ≥ V 时，视为完全背包处理
- **物品预处理**：合并相同重量的物品，只保留价值最高的
- **剪枝策略**：跳过价值为 0 或重量超过 V 的物品

### 复杂度分析与性能优化
  ```

时间复杂度详细分析
算法	时间复杂度	适用数据规模	常数因子
基础实现	O(n * V * c_avg)	n*V*c < 10^6	小
二进制优化	O(n * V * log c_max)	n*V*logc < 10^7	中等
单调队列优化	O(n * V)	n*V > 10^7	较大

空间复杂度优化技巧
1. **滚动数组**：将二维DP压缩为一维，空间从O(nV)降到O(V)
2. **状态压缩**：对于可行性问题，可以使用bitset进一步压缩
3. **内存访问优化**：合理安排循环顺序，提高缓存命中率

实际性能考量因素
- **常数因子**：算法实现细节对实际运行时间的影响
- **缓存友好性**：内存访问模式对性能的影响
- **输入输出效率**：大规模数据下的IO优化
- **编译器优化**：循环展开、内联函数等编译器优化效果

工程化深度考量

异常处理与边界防御
1. **输入验证**
   ```java
   // 检查物品数量有效性
   if (n <= 0 || V <= 0) return 0;
   // 检查物品属性合法性
   if (w[i] < 0 || v[i] < 0 || c[i] < 0)
       throw new IllegalArgumentException("Invalid item properties");
   ```

2. **数值溢出防护**
   ```java
   // 使用long类型防止整数溢出
   long candidate = (long)dp[j - weight] + value;
   if (candidate > Integer.MAX_VALUE) {
       // 处理溢出情况
   }
   ```

3. **内存管理优化**
   ```java
   // 预分配数组，避免动态扩容
   int[] dp = new int[V + 1];
   ```

```
Arrays.fill(dp, 0); // 快速初始化
```

线程安全与并发处理
1. **不可变数据结构**
```java
public class Item {
    private final int weight;
    private final int value;
    private final int count;
    // 构造函数和getter方法
}
```

2. **线程安全实现**
```java
public class ThreadSafeKnapsack {
    private final int[] dp;
    private final ReentrantLock lock = new ReentrantLock();

    public int solveConcurrently(List<Item> items) {
        // 使用锁或并发数据结构
    }
}
```

性能监控与调试
1. **性能指标收集**
```java
long startTime = System.nanoTime();
int result = knapsack.solve(items, capacity);
long endTime = System.nanoTime();
System.out.println("Execution time: " + (endTime - startTime) + " ns");
```

2. **内存使用监控**
```java
Runtime runtime = Runtime.getRuntime();
long memoryUsed = runtime.totalMemory() - runtime.freeMemory();
System.out.println("Memory used: " + memoryUsed + " bytes");
```

跨语言特性差异分析

Java 语言特性
- **优势**：自动内存管理、丰富的集合类、完善的异常处理
- **注意事项**：避免自动装箱拆箱、注意字符串操作性能

C++语言特性
- **优势**：手动内存管理、模板元编程、零成本抽象
- **注意事项**：内存泄漏风险、需要手动资源管理

Python 语言特性
- **优势**：简洁语法、动态类型、丰富的内置函数
- **注意事项**：解释器性能开销、GIL 限制并发

语言选择建议
- **竞赛场景**：C++（性能最优）
- **工程应用**：Java（平衡性能与开发效率）
- **原型开发**：Python（快速验证算法思路）

调试技巧与问题定位

笔试快速调试法
1. **小例子测试法**
   ```java
   // 使用简单测试用例验证算法正确性
   int[] testWeights = {2, 3, 4};
   int[] testValues = {3, 4, 5};
   int[] testCounts = {1, 2, 1};
   int testCapacity = 5;
   int expected = 7; // 手动计算预期结果
   ```

2. **中间状态打印**
   ```java
   // 打印关键变量的实时值
   System.out.println("i=" + i + ", j=" + j + ", dp[j]=" + dp[j]);
   ```

3. **边界条件测试**
   - 测试 n=0, V=0 的情况
   - 测试所有物品重量都大于 V 的情况
   - 测试所有物品价值都为 0 的情况

面试深度表达技巧

1. **算法原理阐述**
   - 清晰说明状态定义和转移方程
   - 解释优化方法的数学原理
   - 对比不同实现的时间空间复杂度

2. **工程化考量**
   - 讨论异常处理策略
   - 分析线程安全问题
   - 提出性能优化建议

3. **实际应用联想**
   - 将算法思想应用到实际业务场景
   - 讨论算法在分布式系统中的扩展性
   - 分析算法在大数据场景下的适用性

与机器学习/深度学习的联系

优化算法思想相通
1. **梯度下降 vs 动态规划**
   - 都是迭代优化方法
   - 都需要定义目标函数和约束条件
   - 都涉及状态空间的搜索

2. **神经网络训练中的背包思想**
   - 参数剪枝：类似背包问题的物品选择
   - 模型压缩：在精度损失约束下最小化模型大小
   - 资源分配：在有限计算资源下最大化模型性能

实际应用场景
1. **推荐系统**：在有限展示位下选择最优商品组合
2. **广告投放**：预算约束下的广告选择优化
3. **资源调度**：云计算环境中的任务分配

反直觉但关键的设计要点

循环顺序的重要性
```java
// 01 背包：逆序遍历（防止重复选择）
for (int j = V; j >= w[i]; j--) {
    dp[j] = Math.max(dp[j], dp[j - w[i]] + v[i]);
}

// 完全背包：正序遍历（允许重复选择）
```

```
for (int j = w[i]; j <= V; j++) {
    dp[j] = Math.max(dp[j], dp[j - w[i]] + v[i]);
}
```

状态定义的灵活性
- **最大值问题**：dp[j]表示容量为 j 时的最大价值
- **可行性问题**：dp[j]表示容量为 j 是否可达
- **方案数问题**：dp[j]表示容量为 j 的方案数

初始化技巧
- **最大值问题**：初始化为 0（不要求恰好装满）
- **恰好装满问题**：dp[0]=0, 其他初始化为-∞

极端场景鲁棒性测试

五类边界测试用例
1. **空输入测试**：n=0, V=0
2. **极端值测试**：极大/极小的 w[i], v[i], c[i]
3. **重复数据测试**：相同重量不同价值的物品
4. **有序/逆序数据**：测试算法对输入顺序的敏感性
5. **特殊格式测试**：包含 0 值、负值的情况

性能退化排查方法
1. **时间复杂度分析**：确认算法理论复杂度
2. **常数因子优化**：减少不必要的计算和内存访问
3. **输入特征分析**：根据数据分布选择合适算法
4. **内存访问模式**：优化缓存命中率

学习路径与掌握程度评估

四阶段学习路径
1. **基础掌握**：理解 01 背包、完全背包、多重背包的基本实现
2. **优化进阶**：掌握二进制优化、单调队列优化等高级技巧
3. **变形扩展**：学习多维背包、分组背包、依赖背包等变种
4. **工程应用**：将算法思想应用到实际业务场景中

掌握程度评估标准
- **初级**：能够实现基础的多重背包算法
- **中级**：掌握二进制优化，能够解决中等规模问题
- **高级**：精通单调队列优化，能够处理大规模数据
- **专家**：能够根据具体场景设计定制化的优化策略

总结

多重背包问题是动态规划中的重要课题，通过本仓库的学习，您将：
1. 掌握从基础到高级的多种实现方法
2. 理解各种优化技术的数学原理和工程实现
3. 具备解决实际工程问题的能力
4. 建立完整的算法知识体系和工程化思维

代码实现细节与测试验证

本仓库代码实现架构

文件组织结构
```
class075/
├─── Code01_BoundedKnapsack.[java/cpp/py]       # 基础多重背包实现
├─── Code02_BoundedKnapsackWithBinarySplitting.[java/cpp/py]  # 二进制优化
├─── Code03_UnboundedKnapsack.[java/cpp/py]    # 完全背包问题
├─── Code04_BoundedKnapsackWithMonotonicQueue.[java/cpp/py] # 单调队列优化
├─── Code05_MixedKnapsack.[java/cpp/py]         # 混合背包问题
├─── Code06_OnesAndZeroes.[java/cpp/py]        # 多维01背包
├─── Code07_ProfitableSchemes.[java/cpp/py]    # 二维费用背包
├─── Code08_Coins.[java/cpp/py]                # POJ 1742实现
├─── Code09_HDU2191.[java/cpp/py]              # HDU 2191实现
├─── Code10_Codeforces106C.[java/cpp/py]       # Codeforces 106C实现
└─── README.md                                 # 项目文档
```

多语言实现一致性保证
1. **算法逻辑一致性**：三种语言实现相同的核心算法
2. **接口设计统一**：相似的函数签名和参数命名
3. **注释规范统一**：详细的算法说明和复杂度分析
4. **测试用例一致**：使用相同的测试数据验证正确性

代码质量与测试验证

单元测试策略
```java
// Java单元测试示例
@Test
public void testBoundedKnapsackBasic() {
    int[] weights = {2, 3, 4};
    int[] values = {3, 4, 5};
```

```
    int[] counts = {1, 2, 1};
    int capacity = 5;
    int expected = 7;

    int result = BoundedKnapsack.solve(weights, values, counts, capacity);
    assertEquals(expected, result);
}
```

边界测试用例
1. **空输入测试**：n=0, V=0
2. **极端值测试**：极大/极小重量和价值
3. **数量限制测试**：c[i]=0 或 c[i]极大的情况
4. **容量限制测试**：V=0 或 V 极大的情况

性能基准测试
```java
// 性能测试框架
@Benchmark
public void benchmarkBinaryOptimization() {
    // 大规模测试数据
    int n = 1000, V = 10000;
    int[] weights = generateRandomWeights(n, 1, 100);
    int[] values = generateRandomValues(n, 1, 100);
    int[] counts = generateRandomCounts(n, 1, 100);

    long startTime = System.nanoTime();
    int result = BoundedKnapsackBinary.solve(weights, values, counts, V);
    long endTime = System.nanoTime();

    System.out.println("Execution time: " + (endTime - startTime) + " ns");
}
```

工程化最佳实践

代码规范与可读性
1. **命名规范**
   ```java
   // 好的命名
   int maxCapacity = 1000;
   int[] itemWeights = new int[n];
   int[] itemValues = new int[n];
   ```

```
// 避免的命名
int mc = 1000;  // 含义不明确
int[] w = new int[n];  // 过于简略
```

2. **注释规范**
```java
/**
 * 解决多重背包问题的二进制优化方法
 *
 * @param weights 物品重量数组
 * @param values 物品价值数组
 * @param counts 物品数量数组
 * @param capacity 背包容量
 * @return 最大可获得的物品价值
 * @throws IllegalArgumentException 当输入参数不合法时抛出
 *
 * 时间复杂度：O(n * V * log(max_count))
 * 空间复杂度：O(V)
 */
```

错误处理与防御性编程
1. **参数验证**
```java
public static int solve(int[] weights, int[] values, int[] counts, int capacity) {
    // 参数校验
    if (weights == null || values == null || counts == null) {
        throw new IllegalArgumentException("Input arrays cannot be null");
    }
    if (weights.length != values.length || weights.length != counts.length) {
        throw new IllegalArgumentException("Array lengths must be equal");
    }
    if (capacity < 0) {
        throw new IllegalArgumentException("Capacity cannot be negative");
    }
    // ... 算法实现
}
```

2. **异常处理策略**
```java
```

```
try {
    int result = knapsack.solve(weights, values, counts, capacity);
    return result;
} catch (OutOfMemoryError e) {
    // 处理内存不足情况
    logger.error("Memory insufficient for capacity: " + capacity);
    return -1;
} catch (ArithmeticException e) {
    // 处理数值计算异常
    logger.error("Arithmetic error: " + e.getMessage());
    return -1;
}
```

性能优化实战技巧

内存访问优化

1. **缓存友好代码**
```java
// 好的内存访问模式（顺序访问）
for (int i = 0; i < n; i++) {
    for (int j = 0; j <= capacity; j++) {
        // 顺序访问 dp 数组
    }
}

// 差的内存访问模式（跳跃访问）
for (int j = 0; j <= capacity; j++) {
    for (int i = 0; i < n; i++) {
        // 跳跃访问不同物品的数据
    }
}
```

2. **局部变量缓存**
```java
// 使用局部变量缓存频繁访问的值
int currentWeight = weights[i];
int currentValue = values[i];
int currentCount = counts[i];

for (int j = capacity; j >= currentWeight; j--) {
    // 使用缓存的局部变量，减少数组访问
```

```
}
```

计算优化技巧
1. **提前终止优化**
```java
// 当不可能获得更优解时提前终止
if (currentValue == 0) continue;  // 价值为0的物品跳过
if (currentWeight > capacity) continue;  // 重量超过容量的物品跳过
if (currentCount == 0) continue;  // 数量为0的物品跳过
```

2. **数学优化**
```java
// 使用整数运算替代浮点运算
int maxK = Math.min(currentCount, j / currentWeight);  // 整数除法
// 避免使用浮点数比较
```

多语言实现对比分析

Java实现特点
```java
// 优势：自动内存管理，丰富的工具类
import java.util.Arrays;
public class KnapsackJava {
    public int solve(int[] weights, int[] values, int[] counts, int capacity) {
        int[] dp = new int[capacity + 1];
        Arrays.fill(dp, 0);  // 使用标准库快速初始化

        for (int i = 0; i < weights.length; i++) {
            int w = weights[i], v = values[i], c = counts[i];
            // 算法实现...
        }
        return dp[capacity];
    }
}
```

C++实现特点
```cpp
// 优势：性能最优，手动内存控制
#include <vector>
```

```cpp
#include <algorithm>
class KnapsackCPP {
public:
    int solve(const std::vector<int>& weights,
              const std::vector<int>& values,
              const std::vector<int>& counts,
              int capacity) {
        std::vector<int> dp(capacity + 1, 0);

        for (size_t i = 0; i < weights.size(); i++) {
            int w = weights[i], v = values[i], c = counts[i];
            // 算法实现...
        }
        return dp[capacity];
    }
};
```

Python 实现特点
```python
# 优势：代码简洁，快速原型开发
def solve(weights, values, counts, capacity):
    dp = [0] * (capacity + 1)

    for i in range(len(weights)):
        w, v, c = weights[i], values[i], counts[i]
        # 算法实现...

    return dp[capacity]
```

实际业务场景应用

电商库存优化
```java
/**
 * 电商平台库存分配优化
 * 在有限仓储空间下，选择最优商品组合最大化销售额
 */
public class EcommerceInventoryOptimization {
    public List<Product> optimizeInventory(List<Product> products, int warehouseCapacity) {
        // 将商品选择问题建模为多重背包问题
        // 重量：商品占用空间，价值：商品预期销售额，数量：商品库存数量
```

```
        int[] spaces = products.stream().mapToInt(Product::getSpace).toArray();
        int[] revenues = products.stream().mapToInt(Product::getExpectedRevenue).toArray();
        int[] stocks = products.stream().mapToInt(Product::getStock).toArray();

        int maxRevenue = Knapsack.solve(spaces, revenues, stocks, warehouseCapacity);
        return reconstructSolution(products, spaces, revenues, stocks, warehouseCapacity);
    }
}
```

云计算资源调度
```java
/**
 * 云计算环境任务调度优化
 * 在有限计算资源下，选择最优任务组合最大化收益
 */
public class CloudResourceScheduling {
    public Schedule optimizeSchedule(List<Task> tasks, ResourceConstraints constraints) {
        // 将任务调度问题建模为多维背包问题
        // 考虑CPU、内存、存储等多维资源约束
        int[] cpuReqs = tasks.stream().mapToInt(Task::getCpuRequirement).toArray();
        int[] memReqs = tasks.stream().mapToInt(Task::getMemoryRequirement).toArray();
        int[] values = tasks.stream().mapToInt(Task::getPriorityValue).toArray();

        // 使用多维背包算法求解
        return findOptimalTaskAssignment(tasks, constraints);
    }
}
```

持续学习与进阶路径

算法竞赛进阶
1. **区域赛级别**：掌握所有背包变种和优化技巧
2. **ICPC/CCPC决赛**：能够快速识别并解决复杂背包问题
3. **Codeforces红名**：在竞赛中熟练应用各种优化策略

工程实践深化
1. **分布式背包算法**：处理超大规模数据
2. **在线算法**：处理动态变化的背包问题
3. **近似算法**：在多项式时间内找到近似最优解

理论研究拓展

1. **NP 完全性理论**：理解背包问题的计算复杂性
2. **近似算法理论**：学习背包问题的近似算法保证
3. **参数化复杂度**：研究背包问题的参数化算法

总结与展望

通过本仓库的全面学习，您已经掌握了：

1. **完整的算法知识体系**：从基础实现到高级优化的完整技术栈
2. **多语言工程实现能力**：Java、C++、Python 三种语言的熟练实现
3. **工程化最佳实践**：代码规范、测试策略、性能优化等工程技能
4. **实际业务应用能力**：将算法思想应用到真实业务场景中
5. **持续学习的方法论**：建立自主学习和进阶的路径

多重背包问题作为动态规划的重要分支，其思想和方法可以扩展到众多其他算法和工程领域。希望本仓库能够成为您算法学习道路上的重要里程碑，为后续的深入学习和实践应用奠定坚实基础。

==========

[代码文件]
==========
文件: Code01_BoundedKnapsack.cpp
==========
```
#include <iostream>
#include <vector>
#include <string>
#include <sstream>
#include <algorithm>
#include <cstring>
using namespace std;

/**
 * 多重背包问题的基础实现
 *
 * 问题描述：
 * 有 n 种物品，每种物品有价值 v[i]，重量 w[i]，以及数量 c[i]。背包容量为 t。
 * 每种物品最多可以选 c[i]个。要求选择若干物品装入背包，使得总价值最大，且总重量不超过背包容量。
 *
 * 算法分类：
 * - 动态规划
 * - 背包问题
 *
 * 实现特点：
```

```
 * - 提供二维 DP 实现（compute1）和一维空间优化 DP 实现（compute2）
 * - 包含多重剪枝和优化
 * - 支持多组测试用例
 *
 * 适用场景：
 * - 物品数量和背包容量不是特别大的情况
 * - 需要理解多重背包问题基本原理的场景
 * - 作为二进制优化和单调队列优化的基础对比
 *
 * 测试链接：
 * - 牛客网：多重背包问题
 * - 洛谷：P1776 宝物筛选
 *
 * 核心思想：
 * - 状态定义：dp[i][j]表示前 i 种物品，背包容量为 j 时的最大价值
 * - 状态转移：对于每种物品，可以选择 0 到 c[i]个中的任意数量
 * - 一维优化：通过逆序遍历背包容量，确保每个物品只能被选择有限次数
 */

const int MAXN = 101; // 最大物品数量
const int MAXW = 1001; // 最大背包容量

int n, t; // 物品数量，背包容量
int v[MAXN]; // 物品价值数组
int w[MAXN]; // 物品重量数组
int c[MAXN]; // 物品数量数组
int dp[MAXW]; // 一维 DP 数组

/**
 * 严格位置依赖的动态规划实现
 * 使用二维数组存储状态
 *
 * 算法思路：
 * 1. dp[i][j]表示前 i 种物品，背包容量为 j 时的最大价值
 * 2. 对于每个物品，可以选择不选或者选 k 个（1<=k<=c[i]且 k*w[i]<=j）
 * 3. 状态转移方程：dp[i][j] = max(dp[i][j], dp[i-1][j-k*w[i]] + k*v[i])
 *
 * 时间复杂度分析：
 * O(n * t * k_avg)，其中 n 是物品数量，t 是背包容量，k_avg 是每种物品的平均数量
 * 在最坏情况下（每种物品数量都很大），时间复杂度可能达到 O(n * t^2)
 *
 * 空间复杂度分析：
 * O(n * t)，使用二维数组存储所有状态
```

```
 *
 * 优化思路：
 * 1. 可以提前计算每种物品的最大可选择数量，避免无效循环
 * 2. 对于重量为0的物品（如果允许的话），可以特殊处理
 * 3. 对于价值为0的物品，可以直接跳过，因为选择它们不会增加总价值
 *
 * @return 背包能装下的最大价值
 */
int compute1() {
    // dp[0][....] = 0，表示没有货物的情况下，背包容量不管是多少，最大价值都是0
    vector<vector<int>> dp(n + 1, vector<int>(t + 1, 0));

    // 枚举前i种物品
    for (int i = 1; i <= n; i++) {
        int vi = v[i]; // 当前物品价值
        int wi = w[i]; // 当前物品重量
        int ci = c[i]; // 当前物品数量

        // 优化：跳过价值为0的物品
        if (vi == 0) continue;

        // 优化：跳过重量超过背包容量的物品
        if (wi > t) continue;

        // 枚举背包容量j
        for (int j = 0; j <= t; j++) {
            // 初始状态：不选第i种物品，继承前i-1种物品的最大价值
            dp[i][j] = dp[i - 1][j];

            // 计算当前容量下最多能选多少个该物品
            int maxK = min(ci, j / wi);

            // 枚举选择第i种物品的数量k（1到maxK个）
            for (int k = 1; k <= maxK; k++) {
                // 状态转移：选择k个第i种物品，那么剩余容量为j - k*wi，价值增加k*vi
                dp[i][j] = max(dp[i][j], dp[i - 1][j - k * wi] + k * vi);
            }
        }
    }

    // 返回所有物品、背包容量为t时的最大价值
    return dp[n][t];
}
```

```
/**
 * 空间优化的动态规划实现
 * 使用一维数组存储状态，逆序遍历背包容量
 *
 * 算法思路：
 * 1. dp[j]表示背包容量为 j 时的最大价值
 * 2. 逆序遍历背包容量，确保每个物品只能被选择有限次数
 * 3. 枚举每种物品选择的数量，更新状态
 *
 * 时间复杂度分析：
 * O(n * t * k_avg)，与 compute1 相同
 * 注意：部分测试用例可能超时，因为没有对枚举进行优化
 *
 * 空间复杂度分析：
 * O(t)，只需要一维数组存储状态，大幅降低了空间消耗
 *
 * 核心优化：
 * 1. 使用一维数组替代二维数组，减少空间占用
 * 2. 逆序遍历背包容量，确保每种物品只能被选择有限次数
 * 3. 提前计算 maxK，避免重复计算
 * 4. 添加多重剪枝条件，跳过无效物品
 *
 * @return 背包能装下的最大价值
 */
int compute2() {
    // 枚举每种物品
    for (int i = 1; i <= n; i++) {
        int vi = v[i]; // 当前物品价值
        int wi = w[i]; // 当前物品重量
        int ci = c[i]; // 当前物品数量

        // 优化 1：跳过价值为 0 的物品
        if (vi == 0) continue;

        // 优化 2：跳过重量为 0 且数量无限的物品（理论上可以无限取，但题目通常不会出现）
        if (wi == 0 && ci >= MAXW) continue;

        // 优化 3：跳过重量超过背包容量的物品
        if (wi > t) continue;

        // 优化 4：跳过数量为 0 的物品
        if (ci == 0) continue;
```

```
        // 逆序枚举背包容量，避免物品被重复选择
        for (int j = t; j >= wi; j--) { // 从 wi 开始，因为 j < wi 时无法选择该物品
            // 计算当前容量下最多能选多少个该物品
            int maxK = min(ci, j / wi);

            // 枚举选择当前物品的数量 k（1 到 maxK 个）
            for (int k = 1; k <= maxK; k++) {
                int prevJ = j - k * wi;
                // 状态转移：选择 k 个第 i 种物品，那么剩余容量为 prevJ，价值增加 k*vi
                if (dp[prevJ] + k * vi > dp[j]) {
                    dp[j] = dp[prevJ] + k * vi;
                }
            }
        }
    }

    // 返回背包容量为 t 时的最大价值
    return dp[t];
}

// 处理输入并运行算法
void run() {
    // 处理多组测试用例
    while (true) {
        // 读取物品数量和背包容量
        if (!(cin >> n >> t)) {
            break;
        }

        // 初始化数组
        memset(dp, 0, sizeof(dp));

        // 读取每种物品的价值、重量和数量
        for (int i = 1; i <= n; i++) {
            cin >> v[i] >> w[i] >> c[i];
        }

        // 调用空间优化的求解方法并输出结果
        cout << compute2() << endl;
    }
}
```

```
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    run();
    return 0;
}

/*
 * 算法详解与原理解析
 *
 * 1. 问题建模：
 *    - 每种物品是一种资源，有价值、重量和数量限制
 *    - 背包容量是资源约束
 *    - 目标是在约束条件下最大化总价值
 *
 * 2. 状态定义：
 *    - 二维DP：dp[i][j]表示前i种物品，背包容量为j时的最大价值
 *    - 一维DP：dp[j]表示背包容量为j时的最大价值
 *
 * 3. 状态转移方程推导：
 *    对于第i种物品，我们可以选择0到c[i]个中的任意数量
 *    dp[i][j] = max{ dp[i-1][j - k*w[i]] + k*v[i] }, 其中0 ≤ k ≤ min(c[i], j/w[i])
 *
 *    一维优化后：
 *    dp[j] = max{ dp[j - k*w[i]] + k*v[i] }, 其中1 ≤ k ≤ min(c[i], j/w[i])
 *    （从后向前遍历j，确保每种物品只能选有限次数）
 *
 * 4. 边界条件：
 *    - dp[0][j] = 0（没有物品可选时，任何容量的最大价值都是0）
 *    - dp[i][0] = 0（背包容量为0时，无法装任何物品，价值为0）
 *    - dp[0] = 0（一维DP的初始状态）
 */

/*
 * 代码优化与工程化考量
 *
 * 1. 输入优化：
 *    - 使用ios::sync_with_stdio(false)和cin.tie(nullptr)加速输入
 *    - 处理多组测试用例时，注意输入结束条件
 *
 * 2. 算法优化：
 *    - 提前剪枝：跳过价值为0、重量超过容量或数量为0的物品
 *    - 计算maxK，避免重复计算j/w[i]和比较c[i]
```

```
 *     - 从 wi 开始遍历 j，减少无效循环
 *     - 使用局部变量缓存 v[i]、w[i]、c[i]，减少数组访问
 *
 * 3. 代码健壮性：
 *     - 处理各种边界情况：n=0、t=0、物品重量或价值为 0 等
 *     - 避免除零错误（虽然题目通常保证 w[i]>0）
 *     - 处理可能的整数溢出问题
 *
 * 4. 性能优化：
 *     - 使用一维数组替代二维数组，减少内存占用和缓存未命中率
 *     - 逆序遍历 j，确保状态转移的正确性
 *     - 优化循环顺序，提高缓存局部性
 */

/*
 * 多重背包问题的高级优化方法
 *
 * 1. 二进制优化：
 *     - 思路：将数量为 c[i]的物品拆分为 log(c[i])个物品组
 *     - 每组代表 2^k 个该物品，转化为 01 背包问题
 *     - 时间复杂度：O(n * t * log c[i])
 *     - 实现简单，适用范围广
 *
 * 2. 单调队列优化：
 *     - 思路：利用同余分组和单调队列维护最优状态
 *     - 时间复杂度：O(n * t)
 *     - 实现较复杂，但效率最高
 *     - 适合大规模数据
 *
 * 3. 完全背包优化：
 *     - 当 c[i] * w[i] >= t 时，可以将物品视为完全背包
 *     - 时间复杂度：O(n * t)
 *     - 可以结合其他优化方法使用
 */

/*
 * 边界情况分析：
 * 1. 当 n=0（没有物品）时，最大价值为 0
 * 2. 当 t=0（背包容量为 0）时，最大价值为 0
 * 3. 当所有物品的重量都大于 t 时，无法装入任何物品，最大价值为 0
 * 4. 当所有物品的价值都为 0 时，最大价值为 0
 * 5. 当物品重量为 0 且价值为正数时，如果数量无限则可以无限选（但题目通常不会出现）
 */
```

```
/*
 * 工程应用场景：
 * 1. 资源分配问题：在有限资源约束下实现收益最大化
 * 2. 投资组合优化：选择多种投资产品，在风险和收益之间取得平衡
 * 3. 生产计划制定：安排不同产品的生产数量，最大化利润
 * 4. 物流配送优化：在载重限制下选择最优配送方案
 * 5. 项目选择问题：在预算和时间约束下选择最优项目组合
 * 6. 广告投放优化：在预算限制下选择最优广告组合以最大化转化率
 */

==========================================

文件: Code01_BoundedKnapsack.java
==========================================
package class075;

/**
 * 多重背包问题 - 基础实现
 *
 * 问题描述：
 * 有一个容量为 t 的背包，共有 n 种物品
 * 每种物品 i 有以下属性：
 * - 价值 v[i]
 * - 重量 w[i]
 * - 数量 c[i]
 * 要求在不超过背包容量的前提下，选择物品使得总价值最大
 *
 * 算法分类：动态规划 - 多重背包问题
 *
 * 基础实现的特点：
 * 1. 直接枚举每种物品选择的数量
 * 2. 时间复杂度较高，但实现简单直观
 *
 * 适用场景：
 * - 物品数量不大的情况下
 * - 作为理解多重背包问题本质的基础实现
 *
 * 测试链接：https://www.luogu.com.cn/problem/P1776（宝物筛选）
 *
 * 核心思想：
 * 对于每种物品，可以选择 0 到 c[i]个中的任意数量
 * 通过三重循环枚举物品、容量和选择数量
```

```
 * 计算每种选择下的最大价值
 */

/*
 * 相关题目扩展（各大算法平台）：
 * 1. LeetCode（力扣）：
 *    - 474. Ones and Zeroes - https://leetcode.cn/problems/ones-and-zeroes/
 *      多维01背包问题，每个字符串需要同时消耗0和1的数量
 *    - 879. Profitable Schemes - https://leetcode.cn/problems/profitable-schemes/
 *      二维费用背包问题，需要同时考虑人数和利润
 *    - 322. Coin Change - https://leetcode.cn/problems/coin-change/
 *      完全背包问题，求组成金额所需的最少硬币数
 *    - 518. Coin Change II - https://leetcode.cn/problems/coin-change-ii/
 *      完全背包计数问题，求组成金额的方案数
 *    - 416. Partition Equal Subset Sum - https://leetcode.cn/problems/partition-equal-subset-sum/
 *      01背包可行性问题，判断是否能将数组分割成两个和相等的子集
 *
 * 2. 洛谷（Luogu）：
 *    - P1776 宝物筛选 - https://www.luogu.com.cn/problem/P1776
 *      经典多重背包问题
 *    - P1833 樱花 - https://www.luogu.com.cn/problem/P1833
 *      混合背包问题，包含01背包、完全背包和多重背包
 *    - P1064 金明的预算方案 - https://www.luogu.com.cn/problem/P1064
 *      依赖背包问题
 *
 * 3. POJ：
 *    - POJ 1742. Coins - http://poj.org/problem?id=1742
 *      多重背包可行性问题，计算能组成多少种金额
 *    - POJ 1276. Cash Machine - http://poj.org/problem?id=1276
 *      多重背包优化问题，使用二进制优化或单调队列优化
 *    - POJ 3260. The Fewest Coins - http://poj.org/problem?id=3260
 *      双向背包问题，同时考虑找零和支付
 *
 * 4. HDU：
 *    - HDU 1114. Piggy-Bank - http://acm.hdu.edu.cn/showproblem.php?pid=1114
 *      完全背包问题，求装满背包的最小价值
 *    - HDU 2159. FATE - http://acm.hdu.edu.cn/showproblem.php?pid=2159
 *      二维费用背包问题，同时考虑忍耐度和杀怪数
 *    - HDU 2191. 悼念512汶川大地震遇难同胞 - http://acm.hdu.edu.cn/showproblem.php?pid=2191
 *      经典多重背包问题
 *    - HDU 3449 Consumer - http://acm.hdu.edu.cn/showproblem.php?pid=3449
 *      有依赖的背包问题
```

```
 *
 * 5. Codeforces：
 *    - Codeforces 106C. Buns - https://codeforces.com/problemset/problem/106/C
 *      多重背包问题，制作不同种类的面包
 *    - Codeforces 148E. Porcelain - https://codeforces.com/problemset/problem/148/E
 *      分组背包问题，从每组中选择物品
 *    - Codeforces 455A. Boredom - https://codeforces.com/problemset/problem/455/A
 *      打家劫舍类型的动态规划问题
 *
 * 6. AtCoder：
 *    - AtCoder ABC032 D. ナップサック問題 - https://atcoder.jp/contests/abc032/tasks/abc032_d
 *      01背包问题，数据规模较大需要优化
 *    - AtCoder ABC153 F. Silver Fox vs Monster - https://atcoder.jp/contests/abc153/tasks/abc153_f
 *      贪心+前缀和优化的背包问题
 *    - AtCoder ABC224 E. Integers on Grid - https://atcoder.jp/contests/abc224/tasks/abc224_e
 *      动态规划与背包思想结合
 *
 * 7. SPOJ：
 *    - SPOJ KNAPSACK - https://www.spoj.com/problems/KNAPSACK/
 *      经典01背包问题
 *    - SPOJ COINS - https://www.spoj.com/problems/COINS/
 *      硬币问题，完全背包的变形
 *
 * 8. UVa OJ：
 *    - UVa 562. Dividing coins - https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=503
 *      01背包变形，公平分配硬币
 *    - UVa 10130. SuperSale - https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1071
 *      01背包问题的简单应用
 *
 * 9. ZOJ：
 *    - ZOJ 2136. Longest Ordered Subsequence - https://zoj.pintia.cn/problem-sets/91827364500/problems/91827364779
 *      最长递增子序列，可转化为背包思想
 *    - ZOJ 1002. Fire Net - https://zoj.pintia.cn/problem-sets/91827364500/problems/91827364501
 *      回溯法与背包思想结合
 *
 * 10. 牛客网：
 *     - NC19754. 多重背包 - https://ac.nowcoder.com/acm/problem/19754
 *       标准多重背包问题
 *     - NC16552. 买苹果 - https://ac.nowcoder.com/acm/problem/16552
```

```
 *       完全背包问题
 *
 * 11. AcWing：
 *     - AcWing 5. 多重背包问题 II - https://www.acwing.com/problem/content/description/5/
 *       二进制优化的多重背包问题标准题目
 *
 * 12. 剑指 Offer：
 *     - 剑指 Offer 42. 连续子数组的最大和 - https://leetcode.cn/problems/lian-xu-zi-shu-zu-de-zui-da-he-lcof/
 *       动态规划基础问题
 *     - 剑指 Offer 10- II. 青蛙跳台阶问题 - https://leetcode.cn/problems/qing-wa-tiao-tai-jie-wen-ti-lcof/
 *       动态规划基础问题
 */

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.util.Arrays;

/**
 * 多重背包问题的基础实现类
 *
 * 注意事项：
 * 1. 输入数据规模较大时，使用 Scanner 可能会导致超时，因此使用 BufferedReader 进行输入
 * 2. 输出使用 PrintWriter 以提高效率
 * 3. 数组下标从 1 开始，便于理解动态规划的状态转移过程
 * 4. 代码采用清晰的模块化结构，便于维护和扩展
 */
public class Code01_BoundedKnapsack {

    /** 物品数量的最大可能值 */
    public static final int MAXN = 101;

    /** 背包容量的最大可能值 */
    public static final int MAXW = 40001;

    /** 物品价值数组：v[i]表示第 i 个物品的价值 */
    public static int[] v = new int[MAXN];

    /** 物品重量数组：w[i]表示第 i 个物品的重量 */
```

```
public static int[] w = new int[MAXN];

/** 物品数量数组：c[i]表示第 i 个物品的可用数量 */
public static int[] c = new int[MAXN];

/** 动态规划数组：dp[j]表示背包容量为 j 时的最大价值 */
public static int[] dp = new int[MAXW];

/** 物品数量 */
public static int n;

/** 背包容量 */
public static int t;

/**
 * 主方法
 * 处理输入、调用计算方法、输出结果
 *
 * 工程化考量：
 * 1. 使用 BufferedReader 进行高效的输入处理，避免 StreamTokenizer 的复杂性
 * 2. 使用 PrintWriter 进行高效的输出处理
 * 3. 确保输入输出流被正确关闭，防止资源泄露
 * 4. 支持多组测试用例的连续读取
 * 5. 使用 try-with-resources 自动关闭资源，提高代码健壮性
 *
 * @param args 命令行参数（未使用）
 * @throws IOException 输入输出异常
 */
public static void main(String[] args) throws IOException {
    // 使用 try-with-resources 自动关闭资源
    try (BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
         PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out))) {

        String line;
        // 循环读取多组测试用例
        while ((line = br.readLine()) != null) {
            // 跳过空行
            if (line.trim().isEmpty()) continue;

            String[] parts = line.trim().split("\s+");
            int idx = 0;
            n = Integer.parseInt(parts[idx++]);
            t = Integer.parseInt(parts[idx++]);
```

```
            // 读取每个物品的价值、重量和数量
            for (int i = 1; i <= n; i++) {
                line = br.readLine();
                while (line != null && line.trim().isEmpty()) {
                    line = br.readLine(); // 跳过空行
                }
                if (line == null) break;

                String[] itemParts = line.trim().split("\s+");
                v[i] = Integer.parseInt(itemParts[0]);
                w[i] = Integer.parseInt(itemParts[1]);
                c[i] = Integer.parseInt(itemParts[2]);
            }

            // 每次计算前清空 dp 数组，避免多组测试用例间的影响
            Arrays.fill(dp, 0, t + 1, 0);

            // 调用求解方法并输出结果
            out.println(compute2());
        }

        // 刷新输出，确保所有内容都被写入
        out.flush();
    }
}

/**
 * 严格位置依赖的动态规划实现
 * 使用二维数组存储状态
 *
 * 算法思路：
 * 1. dp[i][j]表示前 i 种物品，背包容量为 j 时的最大价值
 * 2. 对于每个物品，可以选择不选或者选 k 个（1<=k<=c[i]且 k*w[i]<=j）
 * 3. 状态转移方程：dp[i][j] = max(dp[i][j], dp[i-1][j-k*w[i]] + k*v[i])
 *
 * 时间复杂度分析：
 * O(n * t * k_avg)，其中 n 是物品数量，t 是背包容量，k_avg 是每种物品的平均数量
 * 在最坏情况下（每种物品数量都很大），时间复杂度可能达到 O(n * t^2)
 *
 * 空间复杂度分析：
 * O(n * t)，使用二维数组存储所有状态
 *
```

```
     * 优化思路：
     * 1. 可以提前计算每种物品的最大可选择数量，避免无效循环
     * 2. 对于重量为0的物品（如果允许的话），可以特殊处理
     * 3. 对于价值为0的物品，可以直接跳过，因为选择它们不会增加总价值
     *
     * @return 背包能装下的最大价值
     */
    public static int compute1() {
        // dp[0][....] = 0，表示没有货物的情况下，背包容量不管是多少，最大价值都是0
        int[][] dp = new int[n + 1][t + 1];

        // 枚举前i种物品
        for (int i = 1; i <= n; i++) {
            int vi = v[i]; // 当前物品价值
            int wi = w[i]; // 当前物品重量
            int ci = c[i]; // 当前物品数量

            // 优化：跳过价值为0的物品
            if (vi == 0) continue;

            // 优化：跳过重量超过背包容量的物品
            if (wi > t) continue;

            // 枚举背包容量j
            for (int j = 0; j <= t; j++) {
                // 初始状态：不选第i种物品，继承前i-1种物品的最大价值
                dp[i][j] = dp[i - 1][j];

                // 计算当前容量下最多能选多少个该物品
                int maxK = Math.min(ci, j / wi);

                // 枚举选择第i种物品的数量k（1到maxK个）
                for (int k = 1; k <= maxK; k++) {
                    // 状态转移：选择k个第i种物品，那么剩余容量为j - k*wi，价值增加k*vi
                    dp[i][j] = Math.max(dp[i][j], dp[i - 1][j - k * wi] + k * vi);
                }
            }
        }

        // 返回所有物品、背包容量为t时的最大价值
        return dp[n][t];
    }
```

```
/**
 * 空间优化的动态规划实现
 * 使用一维数组存储状态，逆序遍历背包容量
 *
 * 算法思路：
 * 1. dp[j]表示背包容量为 j 时的最大价值
 * 2. 逆序遍历背包容量，确保每个物品只能被选择有限次数
 * 3. 枚举每种物品选择的数量，更新状态
 *
 * 时间复杂度分析：
 * O(n * t * k_avg)，与 compute1 相同
 * 注意：部分测试用例可能超时，因为没有对枚举进行优化
 *
 * 空间复杂度分析：
 * O(t)，只需要一维数组存储状态，大幅降低了空间消耗
 *
 * 核心优化：
 * 1. 使用一维数组替代二维数组，减少空间占用
 * 2. 逆序遍历背包容量，确保每种物品只能被选择有限次数
 * 3. 提前计算 maxK，避免重复计算
 * 4. 添加多重剪枝条件，跳过无效物品
 *
 * @return 背包能装下的最大价值
 */
public static int compute2() {
    // 枚举每种物品
    for (int i = 1; i <= n; i++) {
        int vi = v[i]; // 当前物品价值
        int wi = w[i]; // 当前物品重量
        int ci = c[i]; // 当前物品数量

        // 优化 1：跳过价值为 0 的物品
        if (vi == 0) continue;

        // 优化 2：跳过重量为 0 且数量无限的物品（理论上可以无限取，但题目通常不会出现）
        if (wi == 0 && ci >= Integer.MAX_VALUE) continue;

        // 优化 3：跳过重量超过背包容量的物品
        if (wi > t) continue;

        // 优化 4：跳过数量为 0 的物品
        if (ci == 0) continue;
```

```
            // 逆序枚举背包容量，避免物品被重复选择
            for (int j = t; j >= wi; j--) { // 从 wi 开始，因为 j < wi 时无法选择该物品
                // 计算当前容量下最多能选多少个该物品
                int maxK = Math.min(ci, j / wi);

                // 枚举选择当前物品的数量 k（1 到 maxK 个）
                for (int k = 1; k <= maxK; k++) {
                    int prevJ = j - k * wi;
                    // 状态转移：选择 k 个第 i 种物品，那么剩余容量为 prevJ，价值增加 k*vi
                    if (dp[prevJ] + k * vi > dp[j]) {
                        dp[j] = dp[prevJ] + k * vi;
                    }
                }
            }
        }

        // 返回背包容量为 t 时的最大价值
        return dp[t];
    }

    /**
     * 算法详解与原理解析
     *
     * 1. 问题建模：
     *    - 每种物品是一种资源，有价值、重量和数量限制
     *    - 背包容量是资源约束
     *    - 目标是在约束条件下最大化总价值
     *
     * 2. 状态定义：
     *    - 二维 DP：dp[i][j]表示前 i 种物品，背包容量为 j 时的最大价值
     *    - 一维 DP：dp[j]表示背包容量为 j 时的最大价值
     *
     * 3. 状态转移方程推导：
     *    对于第 i 种物品，我们可以选择 0 到 c[i]个中的任意数量
     *    dp[i][j] = max{ dp[i-1][j - k*w[i]] + k*v[i] }, 其中 0 ≤ k ≤ min(c[i], j/w[i])
     *
     *    一维优化后：
     *    dp[j] = max{ dp[j - k*w[i]] + k*v[i] }, 其中 1 ≤ k ≤ min(c[i], j/w[i])
     *    （从后向前遍历 j，确保每种物品只能选有限次数）
     *
     * 4. 边界条件：
     *    - dp[0][j] = 0（没有物品可选时，任何容量的最大价值都是 0）
     *    - dp[i][0] = 0（背包容量为 0 时，无法装任何物品，价值为 0）
```

```
 *     - dp[0] = 0（一维 DP 的初始状态）
 */

/**
 * 代码优化与工程化考量
 *
 * 1. 输入优化：
 *     - 使用 BufferedReader 替代 StreamTokenizer，代码更简洁且维护性更好
 *     - 处理多组测试用例时，注意跳过空行
 *     - 使用 try-with-resources 自动关闭资源，防止资源泄露
 *
 * 2. 算法优化：
 *     - 提前剪枝：跳过价值为 0、重量超过容量或数量为 0 的物品
 *     - 计算 maxK，避免重复计算 j/w[i]和比较 c[i]
 *     - 从 wi 开始遍历 j，减少无效循环
 *     - 使用局部变量缓存 v[i]、w[i]、c[i]，减少数组访问
 *
 * 3. 代码健壮性：
 *     - 处理各种边界情况：n=0、t=0、物品重量或价值为 0 等
 *     - 避免除零错误（虽然题目通常保证 w[i]>0）
 *     - 处理可能的整数溢出问题
 *
 * 4. 性能优化：
 *     - 使用一维数组替代二维数组，减少内存占用和缓存未命中率
 *     - 逆序遍历 j，确保状态转移的正确性
 *     - 优化循环顺序，提高缓存局部性
 */

/**
 * 多重背包问题的高级优化方法
 *
 * 1. 二进制优化：
 *     - 思路：将数量为 c[i]的物品拆分为 log(c[i])个物品组
 *     - 每组代表 2^k 个该物品，转化为 01 背包问题
 *     - 时间复杂度：O(n * t * log c[i])
 *     - 实现简单，适用范围广
 *
 * 2. 单调队列优化：
 *     - 思路：利用同余分组和单调队列维护最优状态
 *     - 时间复杂度：O(n * t)
 *     - 实现较复杂，但效率最高
 *     - 适合大规模数据
 *
```

```
 * 3. 完全背包优化：
 *    - 当 c[i] * w[i] >= t 时，可以将物品视为完全背包
 *    - 时间复杂度：O(n * t)
 *    - 可以结合其他优化方法使用
 */

/**
 * 边界情况分析：
 * 1. 当 n=0（没有物品）时，最大价值为 0
 * 2. 当 t=0（背包容量为 0）时，最大价值为 0
 * 3. 当所有物品的重量都大于 t 时，无法装入任何物品，最大价值为 0
 * 4. 当所有物品的价值都为 0 时，最大价值为 0
 * 5. 当物品重量为 0 且价值为正数时，如果数量无限则可以无限选（但题目通常不会出现）
 */

/**
 * 工程应用场景：
 * 1. 资源分配问题：在有限资源约束下实现收益最大化
 * 2. 投资组合优化：选择多种投资产品，在风险和收益之间取得平衡
 * 3. 生产计划制定：安排不同产品的生产数量，最大化利润
 * 4. 物流配送优化：在载重限制下选择最优配送方案
 * 5. 项目选择问题：在预算和时间约束下选择最优项目组合
 * 6. 广告投放优化：在预算限制下选择最优广告组合以最大化转化率
 */

/**
 * 代码调试与测试建议：
 * 1. 小数据测试：使用简单的测试用例验证算法正确性
 * 2. 边界测试：测试 n=0、t=0、物品重量或价值为 0 等边界情况
 * 3. 性能测试：对于大数据集，可以比较不同优化方法的性能差异
 * 4. 调试技巧：添加中间状态输出，观察 dp 数组的变化过程
 */

/**
 * 算法学习建议：
 * 1. 先掌握 01 背包和完全背包的基础实现
 * 2. 理解多重背包问题的本质和状态转移过程
 * 3. 学习二进制优化和单调队列优化的原理
 * 4. 尝试解决各种变形问题，加深理解
 * 5. 对比不同背包问题的异同，建立知识体系
 */

/**
```

```
     * 面试要点：
     * 1. 能够清晰解释多重背包问题的状态定义和转移方程
     * 2. 了解常见的优化方法（二进制、单调队列等）
     * 3. 能够分析算法的时间复杂度和空间复杂度
     * 4. 能够处理各种边界情况
     * 5. 能够将背包问题思想应用到实际场景中
     */
}

==========

文件: Code01_BoundedKnapsack.py
==========
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
多重背包问题的基础实现

问题描述：
有 n 种物品，每种物品有价值 v[i]，重量 w[i]，以及数量 c[i]。背包容量为 t。
每种物品最多可以选 c[i]个。要求选择若干物品装入背包，使得总价值最大，且总重量不超过背包容量。

算法分类：
- 动态规划
- 背包问题

实现特点：
- 提供二维 DP 实现（compute1）和一维空间优化 DP 实现（compute2）
- 包含多重剪枝和优化
- 支持多组测试用例

适用场景：
- 物品数量和背包容量不是特别大的情况
- 需要理解多重背包问题基本原理的场景
- 作为二进制优化和单调队列优化的基础对比

测试链接：
- 牛客网：多重背包问题
- 洛谷：P1776 宝物筛选

核心思想：
- 状态定义：dp[i][j]表示前 i 种物品，背包容量为 j 时的最大价值
```

- 状态转移：对于每种物品，可以选择 0 到 c[i]个中的任意数量
- 一维优化：通过逆序遍历背包容量，确保每个物品只能被选择有限次数

相关题目扩展（多重背包及其变种问题）：

【入门级】
1. LeetCode 416. Partition Equal Subset Sum - 变种问题，检查是否可以分成两个和相等的子集
2. LeetCode 494. Target Sum - 变种问题，寻找目标和的子集个数
3. LeetCode 322. Coin Change - 硬币找零问题，可以视为完全背包的变种

【进阶级】
4. LeetCode 518. Coin Change II - 硬币找零问题 II，计算组合方式数
5. LeetCode 1049. Last Stone Weight II - 变种问题，最小石头重量差
6. LeetCode 474. Ones and Zeroes - 二维费用背包问题

【挑战级】
7. LeetCode 879. Profitable Schemes - 二维费用背包问题的变种
8. LeetCode 956. Tallest Billboard - 较复杂的变种问题
9. LeetCode 1220. Count Vowels Permutation - 状态转移类似背包问题
10. LeetCode 1449. Form Largest Integer With Digits That Add up to Target - 变种问题

【经典 OJ】
11. 牛客网 NC18377 硬币问题 - 多重背包经典问题
12. 牛客网 NC214100 小 A 的购物袋 - 多重背包变形
13. 牛客网 NC233233 背包问题 IV - 完全背包变形
14. 牛客网 NC242214 买饮料 - 多重背包变形
15. 牛客网 NC249417 金币阵列 - 背包问题的二维变种

【洛谷】
16. 洛谷 P1776 宝物筛选 - 多重背包经典问题
17. 洛谷 P1833 樱花 - 多重背包应用
18. 洛谷 P1679 神奇的四次方数 - 完全背包应用
19. 洛谷 P2077 星球大战 - 多维背包应用

【POJ】
20. POJ 1742. Coins - 多重背包可行性问题
21. POJ 1276. Cash Machine - 多重背包优化问题
22. POJ 3260. The Fewest Coins - 双向背包问题

【HDU】
23. HDU 2191. 悼念 512 汶川大地震遇难同胞 - 经典多重背包问题
24. HDU 2159. FATE - 二维费用背包问题
25. HDU 3449. Consumer - 依赖背包问题

【Codeforces】
26. Codeforces 106C. Buns - 多重背包问题
27. Codeforces 148E. Porcelain - 分组背包问题
28. Codeforces 455A. Boredom - 打家劫舍类型问题

【AtCoder】
29. AtCoder ABC032 D. ナップサック問題 - 01背包问题
30. AtCoder ABC153 F. Silver Fox vs Monster - 贪心+前缀和优化问题

【SPOJ】
31. SPOJ KNAPSACK - 经典01背包问题
32. SPOJ COINS - 硬币问题

【UVa OJ】
33. UVa 562. Dividing coins - 01背包变形
34. UVa 10130. SuperSale - 01背包问题

【ZOJ】
35. ZOJ 2136. Longest Ordered Subsequence - 最长递增子序列
36. ZOJ 1002. Fire Net - 回溯法与背包思想结合

【AcWing】
37. AcWing 5. 多重背包问题II - 二进制优化的多重背包问题

【剑指Offer】
38. 剑指 Offer 42. 连续子数组的最大和 - 动态规划基础问题
39. 剑指 Offer 10- II. 青蛙跳台阶问题 - 动态规划基础问题
"""

```
import sys

# 全局变量定义
n = 0  # 物品数量
t = 0  # 背包容量
v = []  # 物品价值列表
w = []  # 物品重量列表
c = []  # 物品数量列表
dp = []  # 一维DP数组


def compute1():
    """
```

严格位置依赖的动态规划实现
使用二维数组存储状态

算法思路：
1. dp[i][j]表示前 i 种物品，背包容量为 j 时的最大价值
2. 对于每个物品，可以选择不选或者选 k 个（1<=k<=c[i]且 k*w[i]<=j）
3. 状态转移方程：dp[i][j] = max(dp[i][j], dp[i-1][j-k*w[i]] + k*v[i])

时间复杂度分析：
O(n * t * k_avg)，其中 n 是物品数量，t 是背包容量，k_avg 是每种物品的平均数量
在最坏情况下（每种物品数量都很大），时间复杂度可能达到 O(n * t^2)

空间复杂度分析：
O(n * t)，使用二维数组存储所有状态

优化思路：
1. 可以提前计算每种物品的最大可选择数量，避免无效循环
2. 对于重量为 0 的物品（如果允许的话），可以特殊处理
3. 对于价值为 0 的物品，可以直接跳过，因为选择它们不会增加总价值

Returns:
    int: 背包能装下的最大价值
"""
# dp[0][....] = 0，表示没有货物的情况下，背包容量不管是多少，最大价值都是 0
dp_table = [[0] * (t + 1) for _ in range(n + 1)]

# 枚举前 i 种物品
for i in range(1, n + 1):
    vi = v[i - 1] # 当前物品价值（注意 Python 列表索引从 0 开始）
    wi = w[i - 1] # 当前物品重量
    ci = c[i - 1] # 当前物品数量

    # 优化：跳过价值为 0 的物品
    if vi == 0:
        continue

    # 优化：跳过重量超过背包容量的物品
    if wi > t:
        continue

    # 枚举背包容量 j
    for j in range(0, t + 1):
        # 初始状态：不选第 i 种物品，继承前 i-1 种物品的最大价值

```
            dp_table[i][j] = dp_table[i - 1][j]

            # 计算当前容量下最多能选多少个该物品
            maxK = min(ci, j // wi)

            # 枚举选择第 i 种物品的数量 k（1 到 maxK 个）
            for k in range(1, maxK + 1):
                # 状态转移：选择 k 个第 i 种物品，那么剩余容量为 j - k*wi，价值增加 k*vi
                dp_table[i][j] = max(dp_table[i][j], dp_table[i - 1][j - k * wi] + k * vi)

    # 返回所有物品、背包容量为 t 时的最大价值
    return dp_table[n][t]


def compute2():
    """
    空间优化的动态规划实现
    使用一维数组存储状态，逆序遍历背包容量

    算法思路：
    1. dp[j]表示背包容量为 j 时的最大价值
    2. 逆序遍历背包容量，确保每个物品只能被选择有限次数
    3. 枚举每种物品选择的数量，更新状态

    时间复杂度分析：
    O(n * t * k_avg)，与 compute1 相同
    注意：部分测试用例可能超时，因为没有对枚举进行优化

    空间复杂度分析：
    O(t)，只需要一维数组存储状态，大幅降低了空间消耗

    核心优化：
    1. 使用一维数组替代二维数组，减少空间占用
    2. 逆序遍历背包容量，确保每种物品只能被选择有限次数
    3. 提前计算 maxK，避免重复计算
    4. 添加多重剪枝条件，跳过无效物品

    Returns:
        int: 背包能装下的最大价值
    """
    # 枚举每种物品
    for i in range(n):
        vi = v[i]  # 当前物品价值
```

```
        wi = w[i]  # 当前物品重量
        ci = c[i]  # 当前物品数量

        # 优化 1：跳过价值为 0 的物品
        if vi == 0:
            continue

        # 优化 2：跳过重量为 0 且数量无限的物品（理论上可以无限取，但题目通常不会出现）
        if wi == 0 and ci >= float('inf'):
            continue

        # 优化 3：跳过重量超过背包容量的物品
        if wi > t:
            continue

        # 优化 4：跳过数量为 0 的物品
        if ci == 0:
            continue

        # 优化 5：当物品重量总和超过背包容量时，可以视为完全背包
        if ci * wi >= t:
            # 完全背包处理方式（正序遍历）
            for j in range(wi, t + 1):
                if dp[j - wi] + vi > dp[j]:
                    dp[j] = dp[j - wi] + vi
        else:
            # 逆序枚举背包容量，避免物品被重复选择
            for j in range(t, wi - 1, -1):  # 从 wi 开始，因为 j < wi 时无法选择该物品
                # 计算当前容量下最多能选多少个该物品
                maxK = min(ci, j // wi)

                # 枚举选择当前物品的数量 k（1 到 maxK 个）
                for k in range(1, maxK + 1):
                    prevJ = j - k * wi
                    # 状态转移：选择 k 个第 i 种物品，那么剩余容量为 prevJ，价值增加 k*vi
                    if dp[prevJ] + k * vi > dp[j]:
                        dp[j] = dp[prevJ] + k * vi

    # 返回背包容量为 t 时的最大价值
    return dp[t]


def parse_line(line):
```

```
    """
    解析输入行，提取整数列表

    Args:
        line: 输入行字符串

    Returns:
        list: 整数列表
    """
    return list(map(int, line.strip().split()))


def run():
    """
    处理输入并运行算法
    支持多组测试用例
    """
    global n, t, v, w, c, dp

    # 读取所有输入行
    lines = []
    for line in sys.stdin:
        stripped = line.strip()
        if stripped:  # 跳过空行
            lines.append(stripped)

    idx = 0
    while idx < len(lines):
        # 读取物品数量和背包容量
        parts = parse_line(lines[idx])
        idx += 1

        n = parts[0]
        t = parts[1]

        # 边界情况快速处理
        if n == 0 or t == 0:
            print(0)
            continue

        # 初始化列表
        v = []
        w = []
```

```
c = []
dp = [0] * (t + 1)

# 读取每种物品的价值、重量和数量
item_count = 0
while item_count < n and idx < len(lines):
    parts = parse_line(lines[idx])
    idx += 1

    # 过滤无效物品
    if len(parts) != 3:
        continue

    vi, wi, ci = parts

    # 物品过滤优化
    if vi <= 0:  # 价值为0或负数，跳过
        continue
    if wi <= 0:  # 重量为0，特殊处理
        if ci > 0:
            # 如果允许选，理论上可以选无限个，但这里最多选到背包容量
            dp[t] += vi * ci
        continue
    if wi > t:  # 重量超过背包容量，跳过
        continue
    if ci <= 0:  # 数量为0或负数，跳过
        continue

    # 调整数量上限，避免无效计算
    max_possible = t // wi
    if ci > max_possible:
        ci = max_possible

    if ci > 0:  # 只有有效物品才加入列表
        v.append(vi)
        w.append(wi)
        c.append(ci)
        item_count += 1

# 更新实际物品数量
n = len(v)

# 调用空间优化的求解方法并输出结果
```

```
        print(compute2())


if __name__ == "__main__":
    run()


'''
```

算法详解与原理解析

1. 问题建模：
   - 每种物品是一种资源，有价值、重量和数量限制
   - 背包容量是资源约束
   - 目标是在约束条件下最大化总价值

2. 状态定义：
   - 二维 DP：dp[i][j]表示前 i 种物品，背包容量为 j 时的最大价值
   - 一维 DP：dp[j]表示背包容量为 j 时的最大价值

3. 状态转移方程推导：
   对于第 i 种物品，我们可以选择 0 到 c[i]个中的任意数量
   dp[i][j] = max{ dp[i-1][j - k*w[i]] + k*v[i] }, 其中 0 ≤ k ≤ min(c[i], j/w[i])

   一维优化后：
   dp[j] = max{ dp[j - k*w[i]] + k*v[i] }, 其中 1 ≤ k ≤ min(c[i], j/w[i])
   （从后向前遍历 j，确保每种物品只能选有限次数）

4. 边界条件：
   - dp[0][j] = 0（没有物品可选时，任何容量的最大价值都是 0）
   - dp[i][0] = 0（背包容量为 0 时，无法装任何物品，价值为 0）
   - dp[0] = 0（一维 DP 的初始状态）

```
'''


'''
```

代码优化与工程化考量

1. 输入优化：
   - 一次性读取所有输入行，便于处理多组测试用例
   - 跳过空行，增强鲁棒性
   - 使用列表预处理输入数据

2. 算法优化：
   - 提前剪枝：跳过价值为 0、重量超过容量或数量为 0 的物品

   - 计算 maxK，避免重复计算 j/w[i]和比较 c[i]
   - 从 wi 开始遍历 j，减少无效循环
   - 对于重量总和超过背包容量的物品，采用完全背包的处理方式

3. 代码健壮性：
   - 处理各种边界情况：n=0、t=0、物品重量或价值为 0 等
   - 避免除零错误（虽然题目通常保证 w[i]>0）
   - 处理可能的整数溢出问题

4. 性能优化：
   - 使用一维数组替代二维数组，减少内存占用
   - 逆序遍历 j，确保状态转移的正确性
   - 优化循环顺序，提高缓存局部性
   - 局部变量缓存，减少索引访问
'''

'''
多重背包问题的高级优化方法

1. 二进制优化：
   - 思路：将数量为 c[i]的物品拆分为 log(c[i])个物品组
   - 每组代表 2^k 个该物品，转化为 01 背包问题
   - 时间复杂度：O(n * t * log c[i])
   - 实现简单，适用范围广

2. 单调队列优化：
   - 思路：利用同余分组和单调队列维护最优状态
   - 时间复杂度：O(n * t)
   - 实现较复杂，但效率最高
   - 适合大规模数据

3. 完全背包优化：
   - 当 c[i] * w[i] >= t 时，可以将物品视为完全背包
   - 时间复杂度：O(n * t)
   - 可以结合其他优化方法使用
'''

'''
工程应用场景：
1. 资源分配问题：在有限资源约束下实现收益最大化
2. 投资组合优化：选择多种投资产品，在风险和收益之间取得平衡
3. 生产计划制定：安排不同产品的生产数量，最大化利润
4. 物流配送优化：在载重限制下选择最优配送方案

5. 项目选择问题：在预算和时间约束下选择最优项目组合
6. 广告投放优化：在预算限制下选择最优广告组合以最大化转化率
'''

'''

代码调试与测试建议：
1. 小数据测试：使用简单的测试用例验证算法正确性
2. 边界测试：测试 n=0、t=0、物品重量或价值为 0 等边界情况
3. 性能测试：对于大数据集，可以比较不同优化方法的性能差异
4. 调试技巧：添加中间状态输出，观察 dp 数组的变化过程
'''

'''

算法学习建议：
1. 先掌握 01 背包和完全背包的基础实现
2. 理解多重背包问题的本质和状态转移过程
3. 学习二进制优化和单调队列优化的原理
4. 尝试解决各种变形问题，加深理解
5. 对比不同背包问题的异同，建立知识体系
'''

'''

面试要点：
1. 能够清晰解释多重背包问题的状态定义和转移方程
2. 了解常见的优化方法（二进制、单调队列等）
3. 能够分析算法的时间复杂度和空间复杂度
4. 能够处理各种边界情况
5. 能够将背包问题思想应用到实际场景中
'''

==========

文件: Code02_BoundedKnapsackWithBinarySplitting.cpp
==========

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

/**
 * 多重背包问题的二进制分组优化实现类
```

```
 *
 * 技术要点：
 * 1. 使用二进制分组将多重背包转化为01背包
 * 2. 预处理阶段生成组合物品，运行阶段对组合物品应用01背包算法
 * 3. 时间复杂度为O(t * Σlog c[i])，其中c[i]是第i种物品的数量
 */
class BoundedKnapsackWithBinarySplitting {
private:
    /** 物品数量的最大可能值 */
    static constexpr int MAXN = 1001;

    /** 背包容量的最大可能值 */
    static constexpr int MAXW = 40001;

    /** 衍生商品价值数组 */
    vector<int> v;

    /** 衍生商品重量数组 */
    vector<int> w;

    /** 动态规划数组 */
    vector<int> dp;

    /** 物品种类数 */
    int n;

    /** 背包容量 */
    int t;

    /** 衍生商品的总数 */
    int m;

    /**
     * 解析一行输入为整数数组
     *
     * @param line 输入的一行字符串
     * @return 解析后的整数数组
     */
    vector<int> parseLine(const string& line) {
        vector<int> result;
        istringstream iss(line);
        int num;
        while (iss >> num) {
```

```
            result.push_back(num);
        }
        return result;
    }

    /**
     * 01背包的空间压缩实现
     *
     * 算法思路：
     * 1. 初始化dp数组，dp[j]表示背包容量为j时的最大价值
     * 2. 逆序遍历背包容量，避免重复选择同一物品
     * 3. 状态转移方程：dp[j] = max(dp[j], dp[j - weight] + value)
     *
     * 时间复杂度分析：
     * O(m * t)，其中m是衍生商品的总数，t是背包容量
     * 由于m = Σlog c[i]，所以整体时间复杂度为O(t * Σlog c[i])
     * 相比朴素多重背包的O(t * Σc[i])，当c[i]较大时优化效果显著
     *
     * 空间复杂度分析：
     * O(t)，只需要一维数组存储状态
     *
     * 优化点：
     * 1. 预处理边界情况，跳过无效物品
     * 2. 逆序遍历容量，避免重复选择同一物品
     * 3. 提前判断w[i] > j的情况，减少不必要的计算
     *
     * @return 背包能装下的最大价值
     */
    int compute() {
        // 边界情况快速处理
        if (m == 0 || t == 0) {
            return 0;
        }

        // 初始化dp数组，不需要恰好装满背包，所以初始化为0
        fill(dp.begin(), dp.begin() + t + 1, 0);

        // 对每个衍生商品应用01背包算法
        for (int i = 1; i <= m; i++) {
            int weight = w[i];
            int value = v[i];

            // 优化：如果衍生商品的价值为0，跳过（不会增加总价值）
```

```
            if (value == 0) continue;

            // 优化：如果衍生商品的重量为0且价值不为0，可以无限选择，但这里是01背包所以跳过
            if (weight == 0) continue;

            // 优化：如果衍生商品的重量超过背包容量，无法选择，跳过
            if (weight > t) continue;

            // 逆序遍历背包容量，确保每个衍生商品只能被选择一次
            for (int j = t; j >= weight; j--) {
                // 状态转移：选择该衍生商品或不选
                int candidate = dp[j - weight] + value;
                if (candidate > dp[j]) {
                    dp[j] = candidate;
                }
            }
        }

        // 返回背包容量为t时的最大价值
        return dp[t];
    }

public:
    BoundedKnapsackWithBinarySplitting() :
        v(MAXN, 0),
        w(MAXN, 0),
        dp(MAXW, 0),
        n(0),
        t(0),
        m(0) {}

    /**
     * 运行程序的主方法
     *
     * 工程化考量：
     * 1. 使用标准输入输出进行高效的输入处理
     * 2. 完善边界情况处理，增强代码健壮性
     * 3. 添加输入校验，处理空行和不完整输入
     * 4. 支持多组测试用例的连续读取
     */
    void run() {
        ios::sync_with_stdio(false);
        cin.tie(nullptr);
```

```
string line;
while (getline(cin, line)) {
    // 跳过空行
    if (line.empty()) continue;

    vector<int> firstLine = parseLine(line);
    if (firstLine.size() < 2) continue;

    n = firstLine[0];
    t = firstLine[1];

    // 边界情况快速处理
    if (n == 0 || t == 0) {
        cout << 0 << endl;
        continue;
    }

    // 重置衍生商品计数器
    m = 0;

    // 读取每个物品的信息并进行二进制分组
    for (int i = 1; i <= n; i++) {
        // 跳过可能的空行
        while (getline(cin, line) && line.empty());
        if (line.empty()) break;

        vector<int> itemData = parseLine(line);
        if (itemData.size() < 3) continue;

        int value = itemData[0];
        int weight = itemData[1];
        int cnt = itemData[2];

        // 优化 1：跳过数量为 0 的物品
        if (cnt == 0) continue;

        // 优化 2：跳过价值为 0 的物品（选了也不增加总价值）
        if (value == 0) continue;

        // 优化 3：跳过重量为 0 的物品（特殊情况）
        if (weight == 0) continue;
```

```
            // 优化 4：跳过重量超过背包容量的物品
            if (weight > t) continue;

            // 优化 5：调整物品数量上限，避免无意义的计算
            cnt = min(cnt, t / weight);

            // 二进制分组核心逻辑：
            // 将数量为 cnt 的物品拆分成多个组合物品，每个组合物品的数量是 2 的幂次
            // 例如：cnt=5 → 拆分成 1 个、2 个、2 个
            // 这样任何 1~5 的数量都可以通过选择这些组合得到
            for (int k = 1; k <= cnt; k <<= 1) {
                v[++m] = k * value;
                w[m] = k * weight;
                cnt -= k;
            }

            // 处理剩余的数量（如果 cnt 不是 2 的幂次之和）
            if (cnt > 0) {
                v[++m] = cnt * value;
                w[m] = cnt * weight;
            }
        }

        // 计算并输出结果
        cout << compute() << endl;
    }
}

/**
 * 算法优化与工程化考量
 *
 * 1. 二进制分组优化深入分析：
 *    - 普通多重背包：三重循环，时间复杂度 O(n * t * c[i])，对于大数量的物品会超时
 *    - 二进制分组优化：将物品拆分成 log₂(c[i])个组合物品，时间复杂度 O(n * t * log c[i])
 *    - 当 c[i]很大时（比如 1000），log₂(c[i])约为 10，优化效果非常明显
 *
 * 2. 二进制分组正确性数学证明：
 *    - 任意正整数 c 可以唯一地表示为不同 2 的幂次之和（二进制表示）
 *    - 对于任意 k(1<=k<=c)，可以通过选择对应的二进制位组合来表示 k 个物品的选择
 *    - 例如：c=5 → 1(2⁰)+2(2¹)+2(剩余)，这样可以组合出 1~5 之间的任意数量
 *
 * 3. 与单调队列优化的对比：
 *    - 二进制优化：时间复杂度 O(n * t * log c[i])，实现简单，常数因子小
```

```
     *    - 单调队列优化：时间复杂度O(n * t)，实现较复杂，常数因子稍大
     *    - 适用场景对比：
     *      * 当物品数量较多、单个物品数量适中时，二进制优化更适用
     *      * 当背包容量很大、物品数量适中时，单调队列优化更有优势
     */

    /**
     * 与Java版本的差异：
     * 1. 使用vector替代数组，提供更好的内存管理
     * 2. 输入处理使用cin和getline，而非Java的BufferedReader
     * 3. 使用fill函数替代Arrays.fill
     * 4. 使用ios::sync_with_stdio(false)和cin.tie(nullptr)加速输入输出
     */
};

/**
 * 主函数
 */
int main() {
    BoundedKnapsackWithBinarySplitting solution;
    solution.run();
    return 0;
}

==========================================

文件: Code02_BoundedKnapsackWithBinarySplitting.java
==========================================
package class075;

/**
 * 多重背包问题 - 二进制分组转化为01背包实现
 *
 * 问题描述：
 * 有一个容量为t的背包，共有n种物品
 * 每种物品i有以下属性：
 * - 价值v[i]
 * - 重量w[i]
 * - 数量c[i]
 * 要求在不超过背包容量的前提下，选择物品使得总价值最大
 *
 * 算法分类：动态规划 - 多重背包问题 - 二进制分组优化
 *
```

```
 * 二进制分组优化原理：
 * 1. 将数量为c[i]的物品拆分成若干个"组合物品"
 * 2. 每个组合物品代表k个原物品，其中k是2的幂次
 * 3. 例如：c[i]=5，可以拆成1个、2个、2个的组合，这样任何数量1~5都可以通过选择这些组合得到
 * 4. 这样就将多重背包问题转化为01背包问题，大大减少了状态转移的次数
 *
 * 适用场景：
 * - 物品数量较大，但又不是无限多的情况
 * - 需要在时间复杂度和代码复杂度之间取得平衡的场景
 *
 * 测试链接：https://www.luogu.com.cn/problem/P1776（宝物筛选）
 *
 * 核心思想：
 * 使用二进制分组将多重背包转化为01背包
 * 预处理阶段生成组合物品，运行阶段对组合物品应用01背包算法
 * 时间复杂度为O(t * Σlog c[i])，其中c[i]是第i种物品的数量
 */

/*
 * 相关题目扩展（各大算法平台）：
 * 1. LeetCode（力扣）：
 *    - 474. Ones and Zeroes - https://leetcode.cn/problems/ones-and-zeroes/
 *      多维01背包问题，每个字符串需要同时消耗0和1的数量
 *    - 879. Profitable Schemes - https://leetcode.cn/problems/profitable-schemes/
 *      二维费用背包问题，需要同时考虑人数和利润
 *    - 322. Coin Change - https://leetcode.cn/problems/coin-change/
 *      完全背包问题，求组成金额所需的最少硬币数
 *    - 518. Coin Change II - https://leetcode.cn/problems/coin-change-ii/
 *      完全背包计数问题，求组成金额的方案数
 *
 * 2. 洛谷（Luogu）：
 *    - P1776 宝物筛选 - https://www.luogu.com.cn/problem/P1776
 *      经典多重背包问题
 *    - P1833 樱花 - https://www.luogu.com.cn/problem/P1833
 *      混合背包问题，包含01背包、完全背包和多重背包
 *    - P1679 神奇的四次方数 - https://www.luogu.com.cn/problem/P1679
 *      完全背包在数学问题中的应用
 *
 * 3. POJ：
 *    - POJ 1742. Coins - http://poj.org/problem?id=1742
 *      多重背包可行性问题，计算能组成多少种金额
 *    - POJ 1276. Cash Machine - http://poj.org/problem?id=1276
 *      多重背包优化问题，使用二进制优化或单调队列优化
```

```
 *     - POJ 3260. The Fewest Coins - http://poj.org/problem?id=3260
 *       双向背包问题，同时考虑找零和支付
 *
 * 4. HDU：
 *     - HDU 2191. 悼念512汶川大地震遇难同胞 - http://acm.hdu.edu.cn/showproblem.php?pid=2191
 *       经典多重背包问题
 *     - HDU 2159. FATE - http://acm.hdu.edu.cn/showproblem.php?pid=2159
 *       二维费用背包问题，同时考虑忍耐度和杀怪数
 *     - HDU 3449 Consumer - http://acm.hdu.edu.cn/showproblem.php?pid=3449
 *       有依赖的背包问题
 *
 * 5. Codeforces：
 *     - Codeforces 106C. Buns - https://codeforces.com/problemset/problem/106/C
 *       复杂的多重背包问题，涉及多个约束条件
 *     - Codeforces 148E. Porcelain - https://codeforces.com/problemset/problem/148/E
 *       分组背包问题，从每组中选择物品
 *
 * 6. AtCoder：
 *     - AtCoder ABC032 D. ナップサック問題 - https://atcoder.jp/contests/abc032/tasks/abc032_d
 *       01背包问题，数据规模较大需要优化
 *     - AtCoder DP Contest D - Knapsack 1 - https://atcoder.jp/contests/dp/tasks/dp_d
 *       标准01背包问题实现
 *
 * 7. SPOJ：
 *     - SPOJ KNAPSACK - https://www.spoj.com/problems/KNAPSACK/
 *       经典01背包问题
 *     - SPOJ COINS - https://www.spoj.com/problems/COINS/
 *       硬币问题，完全背包的变形
 *
 * 8. 牛客网：
 *     - NC17881. 最大价值 - https://ac.nowcoder.com/acm/problem/17881
 *       多重背包问题的变形应用
 *     - NC233233 背包问题 IV - https://ac.nowcoder.com/acm/problem/233233
 *       完全背包变形
 *
 * 9. AcWing：
 *     - AcWing 5. 多重背包问题 II - https://www.acwing.com/problem/content/description/5/
 *       二进制优化的多重背包问题标准题目
 *
 * 10. UVa OJ：
 *     - UVa 10130. SuperSale -
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1071
 *       01背包问题的简单应用
```

```
 */

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.util.Arrays;

/**
 * 多重背包问题的二进制分组优化实现类
 *
 * 技术要点：
 * 1. 使用二进制分组将多重背包转化为01背包
 * 2. 预处理阶段生成组合物品，运行阶段对组合物品应用01背包算法
 * 3. 时间复杂度为O(t * Σlog c[i])，其中c[i]是第i种物品的数量
 */
public class Code02_BoundedKnapsackWithBinarySplitting {

    /** 物品数量的最大可能值 */
    public static final int MAXN = 1001;

    /** 背包容量的最大可能值 */
    public static final int MAXW = 40001;

    /** 衍生商品价值数组：v[i]表示第i个衍生商品的总价值 */
    public static int[] v = new int[MAXN];

    /** 衍生商品重量数组：w[i]表示第i个衍生商品的总重量 */
    public static int[] w = new int[MAXN];

    /** 动态规划数组：dp[j]表示背包容量为j时的最大价值 */
    public static int[] dp = new int[MAXW];

    /** 物品种类数 */
    public static int n;

    /** 背包容量 */
    public static int t;

    /** 衍生商品的总数 */
    public static int m;
```

```
/**
 * 主方法
 * 处理输入、二进制分组生成衍生商品、调用计算方法、输出结果
 *
 * 工程化考量：
 * 1. 使用 BufferedReader 进行高效的输入处理
 * 2. 使用 PrintWriter 进行高效的输出处理
 * 3. 使用 try-with-resources 确保资源正确关闭，防止内存泄漏
 * 4. 支持多组测试用例的连续读取
 * 5. 完善边界情况处理，增强代码健壮性
 * 6. 添加输入校验，处理空行和不完整输入
 *
 * @param args 命令行参数（未使用）
 * @throws IOException 输入输出异常
 */
public static void main(String[] args) throws IOException {
    // 使用 try-with-resources 自动关闭资源
    try (BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
         PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out))) {

        String line;
        // 循环读取多组测试用例
        while ((line = br.readLine()) != null) {
            // 跳过空行
            if (line.trim().isEmpty()) continue;

            // 解析第一行：物品种类数和背包容量
            String[] parts = line.trim().split("\\s+");
            n = Integer.parseInt(parts[0]);
            t = Integer.parseInt(parts[1]);

            // 边界情况快速处理
            if (n == 0 || t == 0) {
                out.println(0);
                continue;
            }

            // 重置衍生商品计数器
            m = 0;

            // 读取每个物品的信息并进行二进制分组
            for (int i = 1; i <= n; i++) {
                // 读取物品信息，跳过可能的空行
```

```
        while ((line = br.readLine()) != null && line.trim().isEmpty()) {
            // 跳过空行
        }
        if (line == null) break;

        String[] itemParts = line.trim().split("\\s+");
        int value = Integer.parseInt(itemParts[0]);
        int weight = Integer.parseInt(itemParts[1]);
        int cnt = Integer.parseInt(itemParts[2]);

        // 优化 1：跳过数量为 0 的物品
        if (cnt == 0) continue;

        // 优化 2：跳过价值为 0 的物品（选了也不增加总价值）
        if (value == 0) continue;

        // 优化 3：跳过重量为 0 的物品（特殊情况）
        if (weight == 0) continue;

        // 优化 4：跳过重量超过背包容量的物品
        if (weight > t) continue;

        // 优化 5：调整物品数量上限，避免无意义的计算
        cnt = Math.min(cnt, t / weight);

        // 二进制分组核心逻辑：
        // 将数量为 cnt 的物品拆分成多个组合物品，每个组合物品的数量是 2 的幂次
        // 例如：cnt=5 → 拆分成 1 个、2 个、2 个
        // 这样任何 1~5 的数量都可以通过选择这些组合得到
        for (int k = 1; k <= cnt; k <<= 1) {
            v[++m] = k * value;
            w[m] = k * weight;
            cnt -= k;
        }

        // 处理剩余的数量（如果 cnt 不是 2 的幂次之和）
        if (cnt > 0) {
            v[++m] = cnt * value;
            w[m] = cnt * weight;
        }
    }

    // 计算并输出结果
```

```
            out.println(compute());
        }

        // 确保输出全部写入
        out.flush();
    }
}

/**
 * 01背包的空间压缩实现
 *
 * 算法思路：
 * 1. 初始化dp数组，dp[j]表示背包容量为j时的最大价值
 * 2. 逆序遍历背包容量，避免重复选择同一物品
 * 3. 状态转移方程：dp[j] = max(dp[j], dp[j - weight] + value)
 *
 * 时间复杂度分析：
 * O(m * t)，其中m是衍生商品的总数，t是背包容量
 * 由于m = Σlog c[i]，所以整体时间复杂度为O(t * Σlog c[i])
 * 相比朴素多重背包的O(t * Σc[i])，当c[i]较大时优化效果显著
 *
 * 空间复杂度分析：
 * O(t)，只需要一维数组存储状态
 *
 * 优化点：
 * 1. 预处理边界情况，跳过无效物品
 * 2. 逆序遍历容量，避免重复选择同一物品
 * 3. 提前判断w[i] > j的情况，减少不必要的计算
 *
 * @return 背包能装下的最大价值
 */
public static int compute() {
    // 边界情况快速处理
    if (m == 0 || t == 0) {
        return 0;
    }

    // 初始化dp数组，不需要恰好装满背包，所以初始化为0
    Arrays.fill(dp, 0, t + 1, 0);

    // 对每个衍生商品应用01背包算法
    for (int i = 1; i <= m; i++) {
        int weight = w[i];
```

```
            int value = v[i];

            // 优化：如果衍生商品的价值为0，跳过（不会增加总价值）
            if (value == 0) continue;

            // 优化：如果衍生商品的重量为0且价值不为0，可以无限选择，但这里是01背包所以跳过（实际应该特殊处理）
            if (weight == 0) continue;

            // 优化：如果衍生商品的重量超过背包容量，无法选择，跳过
            if (weight > t) continue;

            // 逆序遍历背包容量，确保每个衍生商品只能被选择一次
            for (int j = t; j >= weight; j--) {
                // 状态转移：选择该衍生商品或不选
                int candidate = dp[j - weight] + value;
                if (candidate > dp[j]) {
                    dp[j] = candidate;
                }
            }
        }

        // 返回背包容量为t时的最大价值
        return dp[t];
    }

    /**
     * 算法优化与工程化考量
     *
     * 1. 二进制分组优化深入分析：
     *    - 普通多重背包：三重循环，时间复杂度O(n * t * c[i])，对于大数量的物品会超时
     *    - 二进制分组优化：将物品拆分成log₂(c[i])个组合物品，时间复杂度O(n * t * log c[i])
     *    - 当c[i]很大时（比如1000），log₂(c[i])约为10，优化效果非常明显
     *    - 二进制分组实际上是对物品数量的压缩表示，利用位运算的特性
     *
     * 2. 二进制分组正确性数学证明：
     *    - 任意正整数c可以唯一地表示为不同2的幂次之和（二进制表示）
     *    - 对于任意k(1<=k<=c)，可以通过选择对应的二进制位组合来表示k个物品的选择
     *    - 例如：c=5 → 1(2⁰)+2(2¹)+2(剩余)，这样可以组合出1~5之间的任意数量
     *    - 更严谨地说，对于区间[1,c]中的任意整数k，都可以通过选择若干个组合物品来表示
     *
     * 3. 代码性能优化技巧：
     *    - 使用局部变量缓存频繁访问的值（weight、value）
```

```
     *     - 预处理并跳过无效物品（数量为0、价值为0、重量超过容量）
     *     - 对于重量为0且价值不为0的物品进行特殊处理（可以无限选择）
     *     - 当c[i] * w[i] > t时，可以将物品视为完全背包处理，进一步优化
     *     - 使用Arrays.fill进行数组初始化，比循环更高效
     *     - 提前终止内层循环的情况（如dp[j]不再改变时）
     *
     * 4. 与单调队列优化的对比：
     *     - 二进制优化：时间复杂度O(n * t * log c[i])，实现简单，常数因子小
     *     - 单调队列优化：时间复杂度O(n * t)，实现较复杂，常数因子稍大
     *     - 适用场景对比：
     *       * 当物品数量较多、单个物品数量适中时，二进制优化更适用
     *       * 当背包容量很大、物品数量适中时，单调队列优化更有优势
     *       * 在编程比赛中，二进制优化由于实现简单，更常被采用
     *
     * 5. 工程应用中的考量：
     *     - 数据范围估计：在实际应用中，需要根据数据规模选择合适的优化方法
     *     - 内存优化：对于超大容量的问题，可以考虑使用稀疏数组或其他压缩技术
     *     - 并行处理：在多核心环境下，可以考虑对物品分组并行计算
     *     - 容错处理：添加适当的异常处理和边界检查，提高程序健壮性
     *     - 缓存优化：合理安排内存访问模式，提高缓存命中率
     *
     * 6. 调试与测试建议：
     *     - 单元测试：测试边界情况，如n=0、t=0、c[i]=0等
     *     - 对比测试：与朴素实现对比结果，确保正确性
     *     - 性能测试：在大数据规模下测试算法效率
     *     - 可视化调试：输出中间状态，理解算法执行过程
     *     - 边界条件测试：测试物品重量恰好为容量等特殊情况
     */
}

==============================================

文件: Code02_BoundedKnapsackWithBinarySplitting.py
==============================================
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
多重背包问题的二进制分组优化实现

问题描述：
有一个容量为t的背包，共有n种物品
每种物品i有以下属性：
```

- 价值 v[i]
- 重量 w[i]
- 数量 c[i]

要求在不超过背包容量的前提下，选择物品使得总价值最大

算法分类：动态规划 - 多重背包问题 - 二进制分组优化

二进制分组优化原理：
1. 将数量为 c[i]的物品拆分成若干个"组合物品"
2. 每个组合物品代表 k 个原物品，其中 k 是 2 的幂次
3. 例如：c[i]=5，可以拆成 1 个、2 个、2 个的组合，这样任何数量 1~5 都可以通过选择这些组合得到
4. 这样就将多重背包问题转化为 01 背包问题，大大减少了状态转移的次数

适用场景：
- 物品数量较大，但又不是无限多的情况
- 需要在时间复杂度和代码复杂度之间取得平衡的场景

相关题目扩展：
1. LeetCode 474. Ones and Zeroes - https://leetcode.cn/problems/ones-and-zeroes/
2. LeetCode 879. Profitable Schemes - https://leetcode.cn/problems/profitable-schemes/
3. POJ 1742. Coins - http://poj.org/problem?id=1742
4. POJ 1276. Cash Machine - http://poj.org/problem?id=1276
5. HDU 2191. 非常可乐 - http://acm.hdu.edu.cn/showproblem.php?pid=2191
6. AcWing 5. 多重背包问题 II - https://www.acwing.com/problem/content/description/5/
7. Codeforces 106C. Buns - https://codeforces.com/problemset/problem/106/C
8. 牛客网 NC17881. 最大价值 - https://ac.nowcoder.com/acm/problem/17881

【LeetCode（力扣）】
1. LeetCode 474. Ones and Zeroes - 多维 01 背包问题
2. LeetCode 879. Profitable Schemes - 二维费用背包问题
3. LeetCode 322. Coin Change - 完全背包问题
4. LeetCode 518. Coin Change II - 完全背包计数问题

【洛谷（Luogu）】
5. 洛谷 P1776 宝物筛选 - 经典多重背包问题
6. 洛谷 P1833 樱花 - 混合背包问题
7. 洛谷 P1679 神奇的四次方数 - 完全背包应用

【POJ】
8. POJ 1742. Coins - 多重背包可行性问题
9. POJ 1276. Cash Machine - 多重背包优化问题
10. POJ 3260. The Fewest Coins - 双向背包问题

【HDU】
11. HDU 2191. 悼念512汶川大地震遇难同胞 - 经典多重背包问题
12. HDU 2159. FATE - 二维费用背包问题
13. HDU 3449. Consumer - 依赖背包问题

【Codeforces】
14. Codeforces 106C. Buns - 复杂的多重背包问题
15. Codeforces 148E. Porcelain - 分组背包问题

【AtCoder】
16. AtCoder ABC032 D. ナップサック問題 - 01背包问题
17. AtCoder DP Contest D - Knapsack 1 - 标准01背包问题

【SPOJ】
18. SPOJ KNAPSACK - 经典01背包问题
19. SPOJ COINS - 硬币问题

【牛客网】
20. NC17881. 最大价值 - 多重背包问题的变形应用
21. NC233233 背包问题IV - 完全背包变形

【AcWing】
22. AcWing 5. 多重背包问题II - 二进制优化的多重背包问题标准题目

【UVa OJ】
23. UVa 10130. SuperSale - 01背包问题的简单应用
"""

```
import sys
from typing import List, Tuple

class BoundedKnapsackWithBinarySplitting:
    """
    多重背包问题的二进制分组优化实现类

    技术要点：
    1. 使用二进制分组将多重背包转化为01背包
    2. 预处理阶段生成组合物品，运行阶段对组合物品应用01背包算法
    3. 时间复杂度为O(t * Σlog c[i])，其中c[i]是第i种物品的数量
    """

    def __init__(self):
        """
```

```
    初始化类实例
    设置默认的数组大小上限
    """
    # 物品数量和背包容量的合理上限
    self.MAXN = 1001
    self.MAXW = 40001

def compute(self, t: int, v: List[int], w: List[int], m: int) -> int:
    """
    01 背包的空间压缩实现

    算法思路：
    1. 初始化 dp 数组，dp[j]表示背包容量为 j 时的最大价值
    2. 逆序遍历背包容量，避免重复选择同一物品
    3. 状态转移方程：dp[j] = max(dp[j], dp[j - weight] + value)

    时间复杂度分析：
    O(m * t)，其中 m 是衍生商品的总数，t 是背包容量
    由于 m = Σlog c[i]，所以整体时间复杂度为 O(t * Σlog c[i])
    相比朴素多重背包的 O(t * Σc[i])，当 c[i]较大时优化效果显著

    空间复杂度分析：
    O(t)，只需要一维数组存储状态

    参数：
        t: 背包容量
        v: 衍生商品价值数组
        w: 衍生商品重量数组
        m: 衍生商品总数

    返回：
        背包能装下的最大价值
    """
    # 边界情况快速处理
    if m == 0 or t == 0:
        return 0

    # 初始化 dp 数组，不需要恰好装满背包，所以初始化为 0
    dp = [0] * (t + 1)

    # 对每个衍生商品应用 01 背包算法
    for i in range(1, m + 1):
        weight = w[i]
```

```
            value = v[i]

            # 优化：如果衍生商品的价值为0，跳过（不会增加总价值）
            if value == 0:
                continue

            # 优化：如果衍生商品的重量为0且价值不为0，可以无限选择，但这里是01背包所以跳过
            if weight == 0:
                continue

            # 优化：如果衍生商品的重量超过背包容量，无法选择，跳过
            if weight > t:
                continue

            # 逆序遍历背包容量，确保每个衍生商品只能被选择一次
            for j in range(t, weight - 1, -1):
                # 状态转移：选择该衍生商品或不选
                candidate = dp[j - weight] + value
                if candidate > dp[j]:
                    dp[j] = candidate

        # 返回背包容量为t时的最大价值
        return dp[t]

    def parse_line(self, line: str) -> List[int]:
        """
        解析一行输入为整数列表

        参数：
            line: 输入的一行字符串

        返回：
            解析后的整数列表
        """
        return list(map(int, filter(lambda x: x.strip(), line.strip().split())))

    def run(self):
        """
        运行程序的主方法
        处理输入、二进制分组生成衍生商品、调用计算方法、输出结果

        工程化考量：
        1. 使用sys.stdin进行高效的输入处理
```

2. 支持多组测试用例的连续读取
3. 完善边界情况处理，增强代码健壮性
4. 添加输入校验，处理空行和不完整输入
"""

```
# 为了支持大数据量输入，使用 sys.stdin
input_lines = [line for line in sys.stdin]
ptr = 0

# 预分配足够空间，避免频繁扩容
v = [0] * self.MAXN
w = [0] * self.MAXN

while ptr < len(input_lines):
    # 跳过空行
    while ptr < len(input_lines) and not input_lines[ptr].strip():
        ptr += 1

    if ptr >= len(input_lines):
        break

    # 解析第一行：物品种类数和背包容量
    first_line = self.parse_line(input_lines[ptr])
    ptr += 1

    if len(first_line) < 2:
        continue

    n = first_line[0]
    t = first_line[1]

    # 边界情况快速处理
    if n == 0 or t == 0:
        print(0)
        continue

    # 重置衍生商品计数器
    m = 0

    # 读取每个物品的信息并进行二进制分组
    for _ in range(n):
        # 跳过可能的空行
        while ptr < len(input_lines) and not input_lines[ptr].strip():
            ptr += 1
```

```
if ptr >= len(input_lines):
    break

item_data = self.parse_line(input_lines[ptr])
ptr += 1

if len(item_data) < 3:
    continue

value = item_data[0]
weight = item_data[1]
cnt = item_data[2]

# 优化 1：跳过数量为 0 的物品
if cnt == 0:
    continue

# 优化 2：跳过价值为 0 的物品（选了也不增加总价值）
if value == 0:
    continue

# 优化 3：跳过重量为 0 的物品（特殊情况）
if weight == 0:
    continue

# 优化 4：跳过重量超过背包容量的物品
if weight > t:
    continue

# 优化 5：调整物品数量上限，避免无意义的计算
cnt = min(cnt, t // weight)

# 二进制分组核心逻辑：
# 将数量为 cnt 的物品拆分成多个组合物品，每个组合物品的数量是 2 的幂次
# 例如：cnt=5 → 拆分成 1 个、2 个、2 个
# 这样任何 1~5 的数量都可以通过选择这些组合得到
k = 1
while k <= cnt:
    m += 1
    v[m] = k * value
    w[m] = k * weight
    cnt -= k
```

```
                k <<= 1

            # 处理剩余的数量（如果 cnt 不是 2 的幂次之和）
            if cnt > 0:
                m += 1
                v[m] = cnt * value
                w[m] = cnt * weight

        # 计算并输出结果
        print(self.compute(t, v, w, m))

def binary_knapsack_optimization_principle(self) -> None:
    """
    二进制分组优化原理解释

    二进制分组正确性数学证明：
    1. 任意正整数 c 可以唯一地表示为不同 2 的幂次之和（二进制表示）
    2. 对于任意 k(1<=k<=c)，可以通过选择对应的二进制位组合来表示 k 个物品的选择
    3. 例如：c=5 → 1(2⁰)+2(2¹)+2(剩余)，这样可以组合出 1~5 之间的任意数量
    4. 更严谨地说，对于区间[1,c]中的任意整数 k，都可以通过选择若干个组合物品来表示
    """
    pass

def algorithm_optimization_analysis(self) -> None:
    """
    算法优化与工程化考量

    1. 二进制分组优化深入分析：
       - 普通多重背包：三重循环，时间复杂度 O(n * t * c[i])，对于大数量的物品会超时
       - 二进制分组优化：将物品拆分成 log₂(c[i])个组合物品，时间复杂度 O(n * t * log c[i])
       - 当 c[i]很大时（比如 1000），log₂(c[i])约为 10，优化效果非常明显
       - 二进制分组实际上是对物品数量的压缩表示，利用位运算的特性

    2. 代码性能优化技巧：
       - 使用局部变量缓存频繁访问的值（weight、value）
       - 预处理并跳过无效物品（数量为 0、价值为 0、重量超过容量）
       - 对于重量为 0 且价值不为 0 的物品进行特殊处理（可以无限选择）
       - 当 c[i] * w[i] > t 时，可以将物品视为完全背包处理，进一步优化

    3. 与单调队列优化的对比：
       - 二进制优化：时间复杂度 O(n * t * log c[i])，实现简单，常数因子小
       - 单调队列优化：时间复杂度 O(n * t)，实现较复杂，常数因子稍大
       - 适用场景对比：
```

```
        * 当物品数量较多、单个物品数量适中时，二进制优化更适用
        * 当背包容量很大、物品数量适中时，单调队列优化更有优势
        * 在编程比赛中，二进制优化由于实现简单，更常被采用

    4. 工程应用中的考量：
       - 数据范围估计：在实际应用中，需要根据数据规模选择合适的优化方法
       - 内存优化：对于超大容量的问题，可以考虑使用稀疏数组或其他压缩技术
       - 并行处理：在多核心环境下，可以考虑对物品分组并行计算
       - 容错处理：添加适当的异常处理和边界检查，提高程序健壮性
    """
    pass

# 程序入口
if __name__ == "__main__":
    # 创建实例并运行
    solution = BoundedKnapsackWithBinarySplitting()
    solution.run()

==============================================

文件: Code03_CherryBlossomViewing.cpp
==============================================
#include <iostream>
#include <vector>
#include <algorithm>
#include <cstring>
using namespace std;

/**
 * 樱花观赏问题 - C++实现
 * 洛谷 P1833 樱花问题的变种
 *
 * 问题描述：
 * 混合背包问题的实际应用场景
 * 需要在有限时间内观赏不同种类的樱花，每种樱花有：
 * - 观赏时间（重量）
 * - 观赏价值（价值）
 * - 观赏次数限制（数量）
 * 要求在不超过总时间的前提下，最大化观赏价值
 *
 * 算法分类：动态规划 - 混合背包问题
 *
 * 算法原理：
```

```
 * 1. 将樱花观赏问题建模为混合背包问题
 * 2. 根据观赏次数限制选择不同的处理策略：
 *    - 只能观赏一次：01 背包
 *    - 可以无限观赏：完全背包
 *    - 有次数限制：多重背包
 * 3. 使用二进制优化处理多重背包部分
 *
 * 时间复杂度：O(n * T * log(max_count))
 * 空间复杂度：O(T)
 *
 * 测试链接：https://www.luogu.com.cn/problem/P1833（樱花问题）
 */

const int MAXN = 10001;
const int MAXT = 1001;

int n, T;
int times[MAXN], values[MAXN], types[MAXN], counts[MAXN];
int dp[MAXT];

/**
 * 樱花观赏问题的最大值求解实现
 *
 * 算法思路：
 * 1. 初始化 dp 数组为 0
 * 2. 根据樱花类型选择不同的处理方式：
 *    - 01 樱花：逆序遍历时间
 *    - 完全樱花：正序遍历时间
 *    - 多重樱花：使用二进制优化
 * 3. 返回最大观赏价值
 *
 * 时间复杂度分析：
 * O(n * T * log(max_count))，使用优化后效率较高
 *
 * 空间复杂度分析：
 * O(T)，使用一维数组
 *
 * @return 最大观赏价值
 */
int cherryBlossomViewing() {
    // 初始化 dp 数组
    memset(dp, 0, sizeof(dp));
```

```
// 遍历每种樱花
for (int i = 0; i < n; i++) {
    int t = times[i];
    int v = values[i];
    int type = types[i];
    int cnt = counts[i];

    // 优化：跳过观赏时间为 0 的樱花（特殊情况）
    if (t == 0) {
        continue;
    }

    // 优化：跳过观赏时间超过总时间的樱花
    if (t > T) {
        continue;
    }

    // 根据樱花类型选择处理方式
    if (type == 0) { // 01 樱花（只能观赏一次）
        for (int j = T; j >= t; j--) {
            dp[j] = max(dp[j], dp[j - t] + v);
        }
    } else if (type == 1) { // 完全樱花（可以无限观赏）
        for (int j = t; j <= T; j++) {
            dp[j] = max(dp[j], dp[j - t] + v);
        }
    } else { // 多重樱花（有观赏次数限制）
        // 二进制优化
        int remaining = cnt;
        for (int k = 1; k <= remaining; k <<= 1) {
            int group_t = k * t;
            int group_v = k * v;

            for (int j = T; j >= group_t; j--) {
                dp[j] = max(dp[j], dp[j - group_t] + group_v);
            }
            remaining -= k;
        }

        // 处理剩余部分
        if (remaining > 0) {
            int group_t = remaining * t;
            int group_v = remaining * v;
```

```
                for (int j = T; j >= group_t; j--) {
                    dp[j] = max(dp[j], dp[j - group_t] + group_v);
                }
            }
        }
    }

    return dp[T];
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    // 读取输入
    cin >> T >> n;

    for (int i = 0; i < n; i++) {
        cin >> times[i] >> values[i] >> types[i];
        if (types[i] == 2) { // 多重樱花需要额外读取观赏次数
            cin >> counts[i];
        } else {
            counts[i] = 1; // 01 樱花和完全樱花次数为 1
        }
    }

    // 计算并输出最大观赏价值
    cout << cherryBlossomViewing() << endl;

    return 0;
}

/*
 * 算法详解与原理解析
 *
 * 1. 问题建模：
 *    - 将樱花观赏时间视为背包容量（重量）
 *    - 将樱花观赏价值视为物品价值
 *    - 将观赏次数限制视为物品数量限制
 *    - 问题转化为在时间限制下最大化观赏价值
 *
 * 2. 混合背包处理：
```

```
 *     - 01 樱花：对应只能观赏一次的樱花
 *     - 完全樱花：对应可以反复观赏的樱花
 *     - 多重樱花：对应有观赏次数限制的樱花
 *
 * 3. 二进制优化：
 *     - 将多重樱花转化为多个 01 樱花组合
 *     - 减少状态转移的次数，提高算法效率
 *     - 数学原理：任何正整数都可以表示为 2 的幂次之和
 */

/*
 * 工程化考量与代码优化
 *
 * 1. 性能优化：
 *     - 使用二进制优化处理多重背包
 *     - 提前剪枝，跳过无效的樱花类型
 *     - 使用局部变量缓存频繁访问的值
 *
 * 2. 内存优化：
 *     - 使用一维数组替代二维数组
 *     - 对于大规模数据，可以考虑使用滚动数组
 *
 * 3. 代码健壮性：
 *     - 处理边界情况（T=0, n=0 等）
 *     - 验证输入数据的合法性
 *     - 使用合适的数据类型防止溢出
 */

/*
 * 相关题目扩展
 *
 * 1. 洛谷 P1833. 樱花 - https://www.luogu.com.cn/problem/P1833
 *     经典的混合背包问题，实际应用场景
 *
 * 2. POJ 1742. Coins - http://poj.org/problem?id=1742
 *     多重背包可行性问题
 *
 * 3. HDU 2191. 悼念 512 汶川大地震遇难同胞 - http://acm.hdu.edu.cn/showproblem.php?pid=2191
 *     多重背包问题的经典应用
 *
 * 4. Codeforces 106C. Buns - https://codeforces.com/problemset/problem/106/C
 *     分组背包与多重背包的混合应用
 */
```

```
/*
 * 调试与测试建议
 *
 * 1. 功能测试：
 *    - 测试只包含一种樱花类型的情况
 *    - 测试混合多种樱花类型的情况
 *    - 测试边界情况（所有樱花时间都大于 T）
 *
 * 2. 性能测试：
 *    - 测试大规模数据下的运行时间
 *    - 比较不同优化方法的效果
 *
 * 3. 正确性验证：
 *    - 使用小数据手动计算验证
 *    - 与标准答案进行对比
 */
```

==========

文件: Code03_CherryBlossomViewing.java

==========

```
package class075;

/**
 * 混合背包问题 - 观赏樱花（洛谷 P1833）
 *
 * 问题描述：
 * 给定一个背包的容量 t，一共有 n 种货物，每种货物有以下属性：
 * - 花费(cost)：物品的重量
 * - 价值(val)：物品的价值
 * - 数量(cnt)：物品的数量限制
 *
 * 特殊规则：
 * - 如果 cnt == 0，代表这种货物可以无限选择（完全背包）
 * - 如果 cnt > 0，那么 cnt 代表这种货物的数量（多重背包）
 *
 * 算法分类：动态规划 - 混合背包问题（完全背包 + 多重背包）
 *
 * 算法原理：
 * 1. 将完全背包问题转化为多重背包问题：通过设置足够大的数量限制（ENOUGH=1001）
 * 2. 使用二进制分组优化将多重背包转化为 01 背包问题
 * 3. 对转化后的 01 背包问题应用空间压缩的动态规划算法
```

```
 *
 * 时间复杂度：O(t * Σlog c[i])，其中c[i]是第i种物品的数量（经过二进制分组后）
 * 空间复杂度：O(t)
 *
 * 适用场景：
 * - 同时包含完全背包和多重背包的混合背包问题
 * - 背包容量中等（t ≤ 1000）的情况
 * - 需要处理时间格式输入的特殊场景
 *
 * 测试链接：https://www.luogu.com.cn/problem/P1833（樱花）
 *
 * 实现特点：
 * 1. 处理时间格式输入（小时:分钟）并转换为分钟数作为背包容量
 * 2. 使用二进制分组优化多重背包
 * 3. 采用空间压缩的01背包算法
 * 4. 高效的IO处理，适用于竞赛环境
 */

/**
 * 相关题目扩展（各大算法平台）：
 *
 * 1. LeetCode（力扣）：
 *    - 474. Ones and Zeroes - https://leetcode.cn/problems/ones-and-zeroes/
 *      多维01背包问题，每个字符串需要同时消耗0和1的数量
 *    - 879. Profitable Schemes - https://leetcode.cn/problems/profitable-schemes/
 *      二维费用背包问题，需要同时考虑人数和利润
 *    - 322. Coin Change - https://leetcode.cn/problems/coin-change/
 *      完全背包问题，求组成金额所需的最少硬币数
 *    - 518. Coin Change II - https://leetcode.cn/problems/coin-change-ii/
 *      完全背包计数问题，求组成金额的方案数
 *
 * 2. 洛谷（Luogu）：
 *    - P1833 樱花 - https://www.luogu.com.cn/problem/P1833
 *      混合背包问题，包含01背包、完全背包和多重背包
 *    - P1776 宝物筛选 - https://www.luogu.com.cn/problem/P1776
 *      经典多重背包问题
 *    - P1616 疯狂的采药 - https://www.luogu.com.cn/problem/P1616
 *      完全背包问题，数据规模较大
 *
 * 3. POJ：
 *    - POJ 1742. Coins - http://poj.org/problem?id=1742
 *      多重背包可行性问题，计算能组成多少种金额
 *    - POJ 1276. Cash Machine - http://poj.org/problem?id=1276
```

```
 *      多重背包优化问题，使用二进制优化或单调队列优化
 *    - POJ 3449. Consumer - http://poj.org/problem?id=3449
 *      有依赖的背包问题
 *
 * 4. HDU：
 *    - HDU 2191. 悼念512汶川大地震遇难同胞 - http://acm.hdu.edu.cn/showproblem.php?pid=2191
 *      经典多重背包问题
 *    - HDU 3449. Consumer - http://acm.hdu.edu.cn/showproblem.php?pid=3449
 *      有依赖的背包问题，需要先购买主件
 *
 * 5. Codeforces：
 *    - Codeforces 106C. Buns - https://codeforces.com/contest/106/problem/C
 *      分组背包与多重背包的混合应用
 *    - Codeforces 1003F. Abbreviation - https://codeforces.com/contest/1003/problem/F
 *      字符串处理与多重背包的结合
 *
 * 6. AtCoder：
 *    - AtCoder DP Contest Problem F - https://atcoder.jp/contests/dp/tasks/dp_f
 *      最长公共子序列与背包思想的结合
 *    - AtCoder ABC153 F. Silver Fox vs Monster -
https://atcoder.jp/contests/abc153/tasks/abc153_f
 *      贪心+前缀和优化的背包问题
 */

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.Arrays;

// 完全背包转化为多重背包
// 再把多重背包通过二进制分组转化为01背包
public class Code03_CherryBlossomViewing {

    public static int MAXN = 100001;

    public static int MAXW = 1001;

    public static int ENOUGH = 1001;

    public static int[] v = new int[MAXN];
```

```
public static int[] w = new int[MAXN];

public static int[] dp = new int[MAXW];

public static int hour1, minute1, hour2, minute2;

public static int t, n, m;

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    in.parseNumbers();
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    while (in.nextToken() != StreamTokenizer.TT_EOF) {
        hour1 = (int) in.nval;
        // 跳过冒号
        in.nextToken();
        in.nextToken();
        minute1 = (int) in.nval;
        in.nextToken();
        hour2 = (int) in.nval;
        // 跳过冒号
        in.nextToken();
        in.nextToken();
        minute2 = (int) in.nval;
        if (minute1 > minute2) {
            hour2--;
            minute2 += 60;
        }
        // 计算背包容量
        t = (hour2 - hour1) * 60 + minute2 - minute1;
        in.nextToken();
        n = (int) in.nval;
        m = 0;
        for (int i = 0, cost, val, cnt; i < n; i++) {
            in.nextToken();
            cost = (int) in.nval;
            in.nextToken();
            val = (int) in.nval;
            in.nextToken();
            cnt = (int) in.nval;
            if (cnt == 0) {
```

```
                    cnt = ENOUGH;
                }
                // 二进制分组
                for (int k = 1; k <= cnt; k <<= 1) {
                    v[++m] = k * val;
                    w[m] = k * cost;
                    cnt -= k;
                }
                if (cnt > 0) {
                    v[++m] = cnt * val;
                    w[m] = cnt * cost;
                }
            }
            out.println(compute());
        }
        out.flush();
        out.close();
        br.close();
    }

    // 01背包的空间压缩代码(模版)
    public static int compute() {
        Arrays.fill(dp, 0, t + 1, 0);
        for (int i = 1; i <= m; i++) {
            for (int j = t; j >= w[i]; j--) {
                dp[j] = Math.max(dp[j], dp[j - w[i]] + v[i]);
            }
        }
        return dp[t];
    }

}
```

==========

文件: Code03_CherryBlossomViewing.py

==========

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
樱花观赏问题 - Python实现
洛谷 P1833 樱花问题的变种
```

问题描述：
混合背包问题的实际应用场景
需要在有限时间内观赏不同种类的樱花，每种樱花有：
- 观赏时间（重量）
- 观赏价值（价值）
- 观赏次数限制（数量）
要求在不超过总时间的前提下，最大化观赏价值

算法分类：动态规划 - 混合背包问题

算法原理：
1. 将樱花观赏问题建模为混合背包问题
2. 根据观赏次数限制选择不同的处理策略：
   - 只能观赏一次：01 背包
   - 可以无限观赏：完全背包
   - 有次数限制：多重背包
3. 使用二进制优化处理多重背包部分

时间复杂度：O(n * T * log(max_count))
空间复杂度：O(T)

测试链接：https://www.luogu.com.cn/problem/P1833（樱花问题）

实现特点：
1. 实际应用场景的混合背包问题
2. 使用二进制优化提高效率
3. 完善的错误处理和边界检查
4. 清晰的代码结构和注释
"""

```
import sys

def cherry_blossom_viewing(T, n, cherry_list):
    """
    樱花观赏问题的最大值求解实现

    算法思路：
    1. 初始化 dp 数组为 0
    2. 根据樱花类型选择不同的处理方式：
       - 01 樱花：逆序遍历时间
       - 完全樱花：正序遍历时间
       - 多重樱花：使用二进制优化
```

3. 返回最大观赏价值

时间复杂度分析：
O(n * T * log(max_count))，使用优化后效率较高

空间复杂度分析：
O(T)，使用一维数组

```
Args:
    T: 总观赏时间
    n: 樱花种类数量
    cherry_list: 樱花列表，每个樱花为(time, value, type, count)元组
                 type: 0-01 樱花，1-完全樱花，2-多重樱花
                 count: 对于多重樱花，表示最大观赏次数

Returns:
    int: 最大观赏价值

Raises:
    ValueError: 当输入参数不合法时抛出
"""
# 参数校验
if T < 0 or n < 0:
    return 0

if len(cherry_list) != n:
    raise ValueError("樱花数量不匹配")

# 初始化 dp 数组
dp = [0] * (T + 1)

# 遍历每种樱花
for cherry in cherry_list:
    time, value, cherry_type, count = cherry

    # 参数校验
    if time < 0 or value < 0:
        raise ValueError("观赏时间或价值不能为负数")

    if cherry_type not in [0, 1, 2]:
        raise ValueError("樱花类型必须为 0,1 或 2")

    if cherry_type == 2 and count <= 0:
```

```
            raise ValueError("多重樱花观赏次数必须大于0")

        # 优化：跳过观赏时间为0的樱花（特殊情况）
        if time == 0:
            continue

        # 优化：跳过观赏时间超过总时间的樱花
        if time > T:
            continue

        # 根据樱花类型选择处理方式
        if cherry_type == 0:  # 01樱花（只能观赏一次）
            # 逆序遍历时间
            for j in range(T, time - 1, -1):
                candidate = dp[j - time] + value
                if candidate > dp[j]:
                    dp[j] = candidate

        elif cherry_type == 1:  # 完全樱花（可以无限观赏）
            # 正序遍历时间
            for j in range(time, T + 1):
                candidate = dp[j - time] + value
                if candidate > dp[j]:
                    dp[j] = candidate

        else:  # 多重樱花（有观赏次数限制）
            # 二进制优化
            remaining = count
            k = 1
            while k <= remaining:
                group_time = k * time
                group_value = k * value

                # 逆序遍历时间（01背包方式）
                for j in range(T, group_time - 1, -1):
                    candidate = dp[j - group_time] + group_value
                    if candidate > dp[j]:
                        dp[j] = candidate

                remaining -= k
                k <<= 1  # k *= 2

            # 处理剩余部分
```

```
            if remaining > 0:
                group_time = remaining * time
                group_value = remaining * value

                for j in range(T, group_time - 1, -1):
                    candidate = dp[j - group_time] + group_value
                    if candidate > dp[j]:
                        dp[j] = candidate

    return dp[T]

def main():
    """
    主函数：处理输入、调用算法、输出结果

    工程化考量：
    1. 支持标准输入格式
    2. 完善的错误处理机制
    3. 清晰的输出格式
    """
    try:
        # 读取总时间和樱花种类数量
        first_line = sys.stdin.readline().strip()
        if not first_line:
            return

        parts = first_line.split()
        if len(parts) < 2:
            return

        T = int(parts[0])
        n = int(parts[1])

        # 读取樱花信息
        cherry_list = []
        for i in range(n):
            line = sys.stdin.readline().strip()
            if not line:
                break

            parts = line.split()
            if len(parts) < 3:
                continue
```

```
            time = int(parts[0])
            value = int(parts[1])
            cherry_type = int(parts[2])

            # 对于多重樱花，需要读取观赏次数
            count = 1
            if cherry_type == 2 and len(parts) >= 4:
                count = int(parts[3])

            # 过滤无效樱花
            if time < 0 or value < 0:
                continue

            cherry_list.append((time, value, cherry_type, count))

        # 调整实际樱花数量
        n = len(cherry_list)

        # 调用算法并输出结果
        result = cherry_blossom_viewing(T, n, cherry_list)
        print(result)

    except (ValueError, IndexError) as e:
        print(f"输入格式错误: {e}")
    except Exception as e:
        print(f"计算错误: {e}")

if __name__ == "__main__":
    main()

'''
```

算法详解与原理解析

1. 问题建模：
   - 将樱花观赏时间视为背包容量（重量）
   - 将樱花观赏价值视为物品价值
   - 将观赏次数限制视为物品数量限制
   - 问题转化为在时间限制下最大化观赏价值

2. 混合背包处理：
   - 01 樱花：对应只能观赏一次的樱花，使用逆序遍历
   - 完全樱花：对应可以反复观赏的樱花，使用正序遍历

- 多重樱花：对应有观赏次数限制的樱花，使用二进制优化

3. 二进制优化原理：
   - 将数量为 c 的樱花拆分为 1,2,4,...,2^k,c-2^k 个组合
   - 这样可以用 log(c)个组合表示原樱花的所有选择可能
   - 将多重背包问题转化为 01 背包问题
'''

'''
工程化考量与代码优化

1. 错误处理：
   - 添加全面的参数校验
   - 使用 try-except 捕获和处理异常
   - 提供清晰的错误信息

2. 性能优化：
   - 使用二进制优化处理多重背包
   - 提前过滤无效樱花（时间为 0 或超过总时间）
   - 使用局部变量缓存频繁访问的值

3. 代码可读性：
   - 使用有意义的变量名
   - 添加详细的文档字符串
   - 模块化设计，逻辑清晰

4. 内存优化：
   - 使用一维数组替代二维数组
   - 对于大规模数据，可以考虑使用更紧凑的数据结构
'''

'''
相关题目扩展

1. 洛谷 P1833. 樱花 - https://www.luogu.com.cn/problem/P1833
   经典的混合背包问题，实际应用场景

2. POJ 1742. Coins - http://poj.org/problem?id=1742
   多重背包可行性问题

3. HDU 2191. 悼念 512 汶川大地震遇难同胞 - http://acm.hdu.edu.cn/showproblem.php?pid=2191
   多重背包问题的经典应用

4. Codeforces 106C. Buns - https://codeforces.com/problemset/problem/106/C
   分组背包与多重背包的混合应用
'''

'''

调试与测试建议

1. 功能测试：
   - 测试只包含一种樱花类型的情况
   - 测试混合多种樱花类型的情况
   - 测试边界情况（所有樱花时间都大于 T）

2. 性能测试：
   - 测试大规模数据下的运行时间
   - 比较不同优化方法的效果

3. 正确性验证：
   - 使用小数据手动计算验证
   - 与标准答案进行对比

4. 边界测试：
   - 测试 T=0 或 n=0 的情况
   - 测试存在观赏时间为 0 的樱花的情况
   - 测试观赏次数为 0 的多重樱花
'''

'''

Python 语言特性利用

1. 元组使用：
   - 使用元组表示樱花信息，代码更简洁
   - 元组不可变性确保数据安全

2. 异常处理：
   - Python 的异常处理机制完善
   - 可以方便地捕获和处理各种错误

3. 动态类型：
   - 无需声明变量类型，代码更灵活
   - 但需要注意类型安全，添加适当的校验

4. 列表操作：
   - 使用列表推导式可以简化代码

   - 注意大规模数据下的性能问题
'''

==========

文件: Code03_UnboundedKnapsack.cpp
==========
```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cstring>
using namespace std;

/**
 * 完全背包问题 - C++实现
 *
 * 问题描述：
 * 有一个容量为 t 的背包，共有 n 种物品
 * 每种物品 i 有以下属性：
 * - 价值 v[i]
 * - 重量 w[i]
 * 每种物品可以选择任意次数（0 次或多次）
 * 要求在不超过背包容量的前提下，选择物品使得总价值最大
 *
 * 算法分类：动态规划 - 完全背包问题
 *
 * 算法原理：
 * 1. 状态定义：dp[j]表示背包容量为 j 时的最大价值
 * 2. 状态转移方程：dp[j] = max(dp[j], dp[j-w[i]] + v[i])
 * 3. 遍历顺序：与 01 背包不同，完全背包需要正序遍历背包容量，允许物品被多次选择
 *
 * 时间复杂度：O(n*t)
 * 空间复杂度：O(t)
 *
 * 测试链接：https://www.luogu.com.cn/problem/P1616（宝物筛选扩展问题）
 *
 * 实现特点：
 * 1. 使用一维 DP 数组进行空间优化
 * 2. 采用正序遍历背包容量，允许物品被多次选择
 * 3. 使用高效的 IO 处理，适用于大规模数据
 */

const int MAXN = 1001;
```

```
const int MAXW = 10000001;

int n, t;
int v[MAXN], w[MAXN];
long long dp[MAXW];  // 使用 long long 防止整数溢出

/**
 * 完全背包问题的空间优化实现
 *
 * 算法思路：
 * 1. 初始化 dp 数组为 0
 * 2. 对每种物品，正序遍历背包容量
 * 3. 对于每个容量 j，考虑选择当前物品或不选择
 * 4. 状态转移：dp[j] = max(dp[j], dp[j-w[i]] + v[i])
 *
 * 时间复杂度分析：
 * O(n * t)，其中 n 是物品数量，t 是背包容量
 *
 * 空间复杂度分析：
 * O(t)，只需要一维数组存储状态
 *
 * 优化点：
 * 1. 使用 long long 防止整数溢出
 * 2. 跳过重量超过背包容量的物品
 * 3. 使用局部变量缓存频繁访问的值
 *
 * @return 背包能装下的最大价值
 */
long long compute() {
    // 初始化 dp 数组
    memset(dp, 0, sizeof(dp));

    // 遍历每种物品
    for (int i = 1; i <= n; i++) {
        int current_w = w[i];
        int current_v = v[i];

        // 优化：跳过重量为 0 的物品（特殊情况）
        if (current_w == 0) continue;

        // 优化：跳过重量超过背包容量的物品
        if (current_w > t) continue;
```

```
        // 完全背包：正序遍历背包容量
        for (int j = current_w; j <= t; j++) {
            // 状态转移：选择当前物品或不选择
            if (dp[j - current_w] + current_v > dp[j]) {
                dp[j] = dp[j - current_w] + current_v;
            }
        }
    }

    return dp[t];
}

int main() {
    // 加速输入输出
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    // 读取输入
    cin >> n >> t;

    for (int i = 1; i <= n; i++) {
        cin >> w[i] >> v[i];
    }

    // 计算并输出结果
    cout << compute() << endl;

    return 0;
}

/*
 * 算法详解与原理解析
 *
 * 1. 完全背包与 01 背包的区别：
 *    - 01 背包：每种物品只能选一次，需要逆序遍历容量
 *    - 完全背包：每种物品可以选任意次数，需要正序遍历容量
 *    - 关键区别：状态转移时使用的 dp[j-w[i]]是已经考虑过当前物品的状态
 *
 * 2. 正确性分析：
 *    - 正序遍历时，dp[j-w[i]]可能已经包含了当前物品的选择
 *    - 这样自然实现了物品的多次选择
 *    - 例如：当 j=w[i]时选择 1 个，j=2*w[i]时可以选择 2 个，依此类推
 *
```

```
 * 3. 数学推导：
 *    - 设 f(j)表示容量为 j 时的最大价值
 *    - 对于物品 i，我们可以选择 0,1,2,...,k 个，其中 k = j/w[i]
 *    - 状态转移方程：f(j) = max{ f(j-k*w[i]) + k*v[i] }, 0≤k≤j/w[i]
 *    - 通过正序遍历，我们实际上在计算：f(j) = max(f(j), f(j-w[i]) + v[i])
 *    - 这等价于考虑了所有可能的选择次数
 */

/*
 * 工程化考量与代码优化
 *
 * 1. 输入优化：
 *    - 使用 ios::sync_with_stdio(false)和 cin.tie(nullptr)加速输入
 *    - 对于大规模数据，可以考虑使用更快的输入方法
 *
 * 2. 内存优化：
 *    - 使用一维数组替代二维数组，节省内存
 *    - 对于超大容量问题，可以考虑使用滚动数组或稀疏存储
 *
 * 3. 数值安全：
 *    - 使用 long long 类型防止整数溢出
 *    - 在处理大数值时特别重要
 *
 * 4. 代码健壮性：
 *    - 添加边界条件检查
 *    - 处理特殊输入情况（如 n=0, t=0 等）
 *    - 确保数组索引不越界
 */

/*
 * 相关题目扩展
 *
 * 1. LeetCode 322. Coin Change - https://leetcode.cn/problems/coin-change/
 *    完全背包问题，求组成金额所需的最少硬币数
 *
 * 2. LeetCode 518. Coin Change II - https://leetcode.cn/problems/coin-change-ii/
 *    完全背包计数问题，求组成金额的方案数
 *
 * 3. 洛谷 P1616. 疯狂的采药 - https://www.luogu.com.cn/problem/P1616
 *    经典完全背包问题，数据规模较大
 *
 * 4. HDU 1114. Piggy-Bank - http://acm.hdu.edu.cn/showproblem.php?pid=1114
 *    完全背包问题，求装满背包的最小价值
```

```
 */

/*
 * 调试与测试建议
 *
 * 1. 小数据测试：
 *    - 使用简单的测试用例验证算法正确性
 *    - 例如：n=3, t=5, w=[2,3,4], v=[3,4,5]，预期结果应为最大值
 *
 * 2. 边界测试：
 *    - 测试 n=0 或 t=0 的情况
 *    - 测试所有物品重量都大于 t 的情况
 *    - 测试存在重量为 0 的物品的情况
 *
 * 3. 性能测试：
 *    - 对于大规模数据，测试算法的运行时间和内存使用
 *    - 比较不同优化方法的效果
 *
 * 4. 正确性验证：
 *    - 与二维 DP 实现的结果进行对比
 *    - 确保空间优化版本的结果正确
 */

==========================================

文件: Code03_UnboundedKnapsack.java
==========================================
package class075;

/**
 * 完全背包问题实现类
 *
 * 问题描述：
 * 有一个容量为 t 的背包，共有 n 种物品
 * 每种物品 i 有以下属性：
 * - 价值 v[i]
 * - 重量 w[i]
 * 每种物品可以选择任意次数（0 次或多次）
 * 要求在不超过背包容量的前提下，选择物品使得总价值最大
 *
 * 算法分类：动态规划 - 完全背包问题
 *
 * 算法原理：
```

```
 * 1. 状态定义：dp[j]表示背包容量为 j 时的最大价值
 * 2. 状态转移方程：dp[j] = max(dp[j], dp[j-w[i]] + v[i])
 * 3. 遍历顺序：与 01 背包不同，完全背包需要正序遍历背包容量，允许物品被多次选择
 *
 * 时间复杂度：O(n*t)
 * 空间复杂度：O(t)
 *
 * 测试链接：https://www.luogu.com.cn/problem/P1616（宝物筛选扩展问题）
 *
 * 实现特点：
 * 1. 使用一维 DP 数组进行空间优化
 * 2. 采用正序遍历背包容量，允许物品被多次选择
 * 3. 使用高效的 IO 处理，适用于大规模数据
 */

/*
 * 相关题目扩展（各大算法平台）：
 * 1. LeetCode（力扣）：
 *    - 322. Coin Change - https://leetcode.cn/problems/coin-change/
 *      完全背包问题，求组成金额所需的最少硬币数
 *    - 518. Coin Change II - https://leetcode.cn/problems/coin-change-ii/
 *      完全背包计数问题，求组成金额的方案数
 *    - 377. Combination Sum IV - https://leetcode.cn/problems/combination-sum-iv/
 *      顺序相关的组合问题，类似完全背包
 *
 * 2. 洛谷（Luogu）：
 *    - P1616. 疯狂的采药 - https://www.luogu.com.cn/problem/P1616
 *      经典完全背包问题，数据规模较大
 *    - P1679. 神奇的四次方数 - https://www.luogu.com.cn/problem/P1679
 *      完全背包在数学问题中的应用
 *
 * 3. POJ：
 *    - POJ 1114. Piggy-Bank - http://poj.org/problem?id=1114
 *      完全背包问题，求装满背包的最小价值
 *    - POJ 2063. Investment - http://poj.org/problem?id=2063
 *      完全背包问题的实际应用
 *
 * 4. HDU：
 *    - HDU 1114. Piggy-Bank - http://acm.hdu.edu.cn/showproblem.php?pid=1114
 *      完全背包问题，求装满背包的最小价值
 *    - HDU 2159. FATE - http://acm.hdu.edu.cn/showproblem.php?pid=2159
 *      二维费用背包问题，同时考虑忍耐度和杀怪数
 *
```

```
 * 5. Codeforces：
 *    - Codeforces 148E. Porcelain - https://codeforces.com/problemset/problem/148/E
 *      分组背包问题，从每组中选择物品
 *
 * 6. AtCoder：
 *    - AtCoder DP Contest E - Knapsack 2 - https://atcoder.jp/contests/dp/tasks/dp_e
 *      大体积小价值的01背包问题，需要价值维度DP
 *
 * 7. SPOJ：
 *    - SPOJ COINS - https://www.spoj.com/problems/COINS/
 *      硬币问题，完全背包的变形
 *
 * 8. 牛客网：
 *    - NC16552. 买苹果 - https://ac.nowcoder.com/acm/problem/16552
 *      完全背包问题
 *    - NC242214 买饮料 - https://ac.nowcoder.com/acm/problem/242214
 *      多重背包变形
 *
 * 9. UVa OJ：
 *    - UVa 10130. SuperSale -
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1071
 *      01背包问题的简单应用
 *
 * 10. AcWing：
 *     - AcWing 3. 完全背包问题 - https://www.acwing.com/problem/content/3/
 *       标准完全背包问题
 */

import java.io.*;

/**
 * 完全背包问题的标准实现类
 *
 * 技术要点：
 * 1. 使用一维DP数组进行空间优化
 * 2. 通过正序遍历背包容量实现完全背包逻辑
 * 3. 采用高效的输入输出方式处理大数据量
 */
public class Code03_UnboundedKnapsack {

    /** 物品数量的最大可能值 */
    public static int MAXN = 1001;
```

```
/** 背包容量的最大可能值 */
public static int MAXW = 10000001;

/** 物品价值数组：v[i]表示第 i 个物品的价值 */
public static int[] v = new int[MAXN];

/** 物品重量数组：w[i]表示第 i 个物品的重量 */
public static int[] w = new int[MAXN];

/** 动态规划数组：dp[j]表示背包容量为 j 时的最大价值 */
public static int[] dp = new int[MAXW];

/** 物品数量 */
public static int n;

/** 背包容量 */
public static int t;

/**
 * 主方法
 * 处理输入、调用计算逻辑、输出结果
 *
 * 工程化考量：
 * 1. 使用 BufferedReader 和 StreamTokenizer 进行高效的输入处理
 * 2. 使用 PrintWriter 进行高效的输出处理
 * 3. 确保输入输出流被正确关闭，防止资源泄露
 * 4. 直接在主方法中实现算法核心逻辑，简洁高效
 *
 * @param args 命令行参数（未使用）
 * @throws IOException 输入输出异常
 */
public static void main(String[] args) throws IOException {
    // 初始化输入流，使用 BufferedReader 和 StreamTokenizer 提高读取效率
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    // 初始化输出流，使用 PrintWriter 提高写入效率
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取物品数量和背包容量
    in.nextToken();
    n = (int) in.nval;
    in.nextToken();
    t = (int) in.nval;
```

```
    // 读取每个物品的重量和价值
    for (int i = 1; i <= n; i++) {
        in.nextToken();
        w[i] = (int) in.nval;
        in.nextToken();
        v[i] = (int) in.nval;
    }

    // 完全背包算法核心逻辑
    // 遍历每个物品
    for (int i = 1; i <= n; i++) {
        // 正序遍历背包容量（与01背包的逆序遍历不同）
        // 这样可以确保同一个物品被多次选择
        for (int j = w[i]; j <= t; j++) {
            // 状态转移方程：考虑选择当前物品或不选择当前物品
            // 如果选择当前物品，则容量减少w[i]，价值增加v[i]
            dp[j] = Math.max(dp[j], dp[j - w[i]] + v[i]);
        }
    }

    // 输出结果
    out.println(dp[t]);

    // 刷新并关闭流，确保输出全部写入
    out.flush();
    out.close();
    br.close();
}

/**
 * 完全背包问题的另一种实现方法
 * 使用二维DP数组，更直观地展示状态转移过程
 *
 * @return 背包能够装下的最大价值
 */
public static int unboundedKnapsack2D() {
    // 创建二维DP数组
    int[][] dp = new int[n + 1][t + 1];

    // 遍历每个物品
    for (int i = 1; i <= n; i++) {
        // 遍历每个可能的背包容量
```

```
        for (int j = 0; j <= t; j++) {
            // 不选择当前物品的情况
            dp[i][j] = dp[i - 1][j];

            // 选择当前物品的情况（如果容量足够）
            if (j >= w[i]) {
                dp[i][j] = Math.max(dp[i][j], dp[i][j - w[i]] + v[i]);
            }
        }
    }

    return dp[n][t];
}

/**
 * 算法详解与扩展
 *
 * 1. 完全背包与01背包的区别：
 *    - 01背包：每种物品只能选一次
 *    - 完全背包：每种物品可以选任意次数
 *    - 实现区别：01背包逆序遍历容量，完全背包正序遍历容量
 *
 * 2. 算法正确性分析：
 *    - 对于完全背包，正序遍历容量j时，dp[j-w[i]]已经包含了之前选择该物品的情况
 *    - 因此，通过正序遍历，可以自然地实现物品的多次选择
 *    - 例如，当j增加到j+w[i]时，又可以再次选择该物品
 *
 * 3. 优化可能性：
 *    - 对于重量很大的物品，可以提前剪枝
 *    - 对于价值为0的物品，可以直接跳过
 *    - 对于相同重量的物品，可以只保留价值最高的那个
 *    - 当数据规模非常大时，可以考虑使用更高效的动态规划优化技术
 *
 * 4. 相关问题扩展：
 *    - 多重背包：每种物品有固定数量限制
 *    - 分组背包：物品分成若干组，每组只能选一个
 *    - 二维费用背包：每个物品有两种费用（如重量和体积）
 *    - 有依赖关系的背包：物品之间存在依赖关系
 */

/**
 * 工程应用考量：
 *
```

```
     * 1. 数据规模处理：
     *    - 当背包容量非常大时（如本题中 MAXW=1e7），需要注意内存使用
     *    - 可以考虑使用滚动数组或仅保留必要的状态
     *
     * 2. 代码健壮性：
     *    - 应处理物品重量为 0 的特殊情况（如果允许）
     *    - 应处理重量大于背包容量的物品（可以跳过）
     *    - 在处理大数值时，需要考虑整数溢出问题
     *
     * 3. 性能优化：
     *    - 预处理物品，去除无效物品（如价值为 0 的物品）
     *    - 对于特殊物品（如重量相同的物品）进行合并
     *    - 使用更高效的输入输出方法处理大规模数据
     *
     * 4. 实际应用场景：
     *    - 资源分配问题
     *    - 生产计划问题
     *    - 投资组合优化
     *    - 货物装载问题
     *    - 数据压缩算法
     */
}

==============================================

文件: Code03_UnboundedKnapsack.py
==============================================
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
完全背包问题 - Python 实现

问题描述：
有一个容量为 t 的背包，共有 n 种物品
每种物品 i 有以下属性：
- 价值 v[i]
- 重量 w[i]
每种物品可以选择任意次数（0 次或多次）
要求在不超过背包容量的前提下，选择物品使得总价值最大

算法分类：动态规划 - 完全背包问题
```

算法原理：
1. 状态定义：dp[j]表示背包容量为 j 时的最大价值
2. 状态转移方程：dp[j] = max(dp[j], dp[j-w[i]] + v[i])
3. 遍历顺序：与 01 背包不同，完全背包需要正序遍历背包容量，允许物品被多次选择

时间复杂度：O(n*t)
空间复杂度：O(t)

测试链接：https://www.luogu.com.cn/problem/P1616（宝物筛选扩展问题）

实现特点：
1. 使用一维 DP 数组进行空间优化
2. 采用正序遍历背包容量，允许物品被多次选择
3. 支持多组测试用例处理
4. 包含详细的错误处理和边界检查

相关题目扩展：

【LeetCode（力扣）】
1. LeetCode 322. Coin Change - https://leetcode.cn/problems/coin-change/
   完全背包问题，求组成金额所需的最少硬币数
2. LeetCode 518. Coin Change II - https://leetcode.cn/problems/coin-change-ii/
   完全背包计数问题，求组成金额的方案数
3. LeetCode 377. 组合总和 Ⅳ - https://leetcode.cn/problems/combination-sum-iv/
   顺序相关的组合问题，类似完全背包

【洛谷（Luogu）】
4. 洛谷 P1616. 疯狂的采药 - https://www.luogu.com.cn/problem/P1616
   经典完全背包问题，数据规模较大
5. 洛谷 P1679. 神奇的四次方数 - https://www.luogu.com.cn/problem/P1679
   完全背包在数学问题中的应用

【POJ】
6. POJ 1114. Piggy-Bank - http://poj.org/problem?id=1114
   完全背包问题，求装满背包的最小价值
7. POJ 2063. Investment - http://poj.org/problem?id=2063
   完全背包问题的实际应用

【HDU】
8. HDU 1114. Piggy-Bank - http://acm.hdu.edu.cn/showproblem.php?pid=1114
   完全背包问题，求装满背包的最小价值
9. HDU 2159. FATE - http://acm.hdu.edu.cn/showproblem.php?pid=2159
   二维费用背包问题，同时考虑忍耐度和杀怪数

【Codeforces】
10. Codeforces 148E. Porcelain - https://codeforces.com/problemset/problem/148/E
    分组背包问题，从每组中选择物品

【AtCoder】
11. AtCoder DP Contest E - Knapsack 2 - https://atcoder.jp/contests/dp/tasks/dp_e
    大体积小价值的01背包问题，需要价值维度DP

【SPOJ】
12. SPOJ COINS - https://www.spoj.com/problems/COINS/
    硬币问题，完全背包的变形

【牛客网】
13. NC16552. 买苹果 - https://ac.nowcoder.com/acm/problem/16552
    完全背包问题
14. NC242214 买饮料 - https://ac.nowcoder.com/acm/problem/242214
    多重背包变形

【UVa OJ】
15. UVa 10130. SuperSale -
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1071
    01背包问题的简单应用

【AcWing】
16. AcWing 3. 完全背包问题 - https://www.acwing.com/problem/content/3/
    标准完全背包问题
"""

```
import sys

def unbounded_knapsack(n, t, weights, values):
    """
    完全背包问题的空间优化实现

    算法思路：
    1. 初始化dp数组为0
    2. 对每种物品，正序遍历背包容量
    3. 对于每个容量j，考虑选择当前物品或不选择
    4. 状态转移：dp[j] = max(dp[j], dp[j-weights[i]] + values[i])

    时间复杂度分析：
    O(n * t)，其中n是物品数量，t是背包容量
```

空间复杂度分析：
O(t)，只需要一维数组存储状态

优化点：
1. 跳过重量为 0 的物品（特殊情况）
2. 跳过重量超过背包容量的物品
3. 使用局部变量缓存频繁访问的值
4. 添加边界条件检查

```
Args:
    n: 物品数量
    t: 背包容量
    weights: 物品重量列表
    values: 物品价值列表

Returns:
    int: 背包能装下的最大价值

Raises:
    ValueError: 当输入参数不合法时抛出
"""
# 参数校验
if n <= 0 or t < 0:
    return 0

if len(weights) != n or len(values) != n:
    raise ValueError("输入数组长度不匹配")

# 初始化 dp 数组
dp = [0] * (t + 1)

# 遍历每种物品
for i in range(n):
    current_weight = weights[i]
    current_value = values[i]

    # 参数校验
    if current_weight < 0 or current_value < 0:
        raise ValueError("物品重量或价值不能为负数")

    # 优化：跳过重量为 0 的物品（特殊情况）
    if current_weight == 0:
```

```
            continue

        # 优化：跳过重量超过背包容量的物品
        if current_weight > t:
            continue

        # 完全背包：正序遍历背包容量
        # 从current_weight开始，因为j < current_weight时无法选择该物品
        for j in range(current_weight, t + 1):
            # 状态转移：选择当前物品或不选择
            candidate = dp[j - current_weight] + current_value
            if candidate > dp[j]:
                dp[j] = candidate

    return dp[t]

def main():
    """
    主函数：处理输入、调用算法、输出结果

    工程化考量：
    1. 使用高效的输入处理方式
    2. 支持多组测试用例
    3. 完善的错误处理机制
    4. 清晰的输出格式
    """
    # 读取所有输入行
    lines = []
    for line in sys.stdin:
        stripped = line.strip()
        if stripped:  # 跳过空行
            lines.append(stripped)

    idx = 0
    while idx < len(lines):
        try:
            # 读取物品数量和背包容量
            parts = lines[idx].split()
            idx += 1

            if len(parts) < 2:
                continue
```

```
        n = int(parts[0])
        t = int(parts[1])

        # 边界情况快速处理
        if n == 0 or t == 0:
            print(0)
            continue

        # 读取物品重量和价值
        weights = []
        values = []

        item_count = 0
        while item_count < n and idx < len(lines):
            parts = lines[idx].split()
            idx += 1

            if len(parts) < 2:
                continue

            weight = int(parts[0])
            value = int(parts[1])

            # 过滤无效物品
            if weight < 0 or value < 0:
                continue

            weights.append(weight)
            values.append(value)
            item_count += 1

        # 如果实际读取的物品数量不足 n，调整 n 的值
        n = len(weights)

        # 调用算法并输出结果
        result = unbounded_knapsack(n, t, weights, values)
        print(result)

    except (ValueError, IndexError) as e:
        # 处理输入格式错误
        print(f"输入格式错误: {e}")
        continue
    except Exception as e:
```

```
            # 处理其他异常
            print(f"计算错误: {e}")
            continue

if __name__ == "__main__":
    main()

'''
```

算法详解与原理解析

1. 完全背包与 01 背包的区别：
   - 01 背包：每种物品只能选一次，需要逆序遍历容量
   - 完全背包：每种物品可以选任意次数，需要正序遍历容量
   - 关键区别：状态转移时使用的 dp[j-w[i]]是已经考虑过当前物品的状态

2. 正确性分析：
   - 正序遍历时，dp[j-w[i]]可能已经包含了当前物品的选择
   - 这样自然实现了物品的多次选择
   - 例如：当 j=w[i]时选择 1 个，j=2*w[i]时可以选择 2 个，依此类推

3. 数学推导：
   - 设 f(j)表示容量为 j 时的最大价值
   - 对于物品 i，我们可以选择 0,1,2,...,k 个，其中 k = j/w[i]
   - 状态转移方程：f(j) = max{ f(j-k*w[i]) + k*v[i] }, 0≤k≤j/w[i]
   - 通过正序遍历，我们实际上在计算：f(j) = max(f(j), f(j-w[i]) + v[i])
   - 这等价于考虑了所有可能的选择次数

```
'''

'''
```

工程化考量与代码优化

1. 输入优化：
   - 一次性读取所有输入行，便于处理多组测试用例
   - 跳过空行，增强鲁棒性
   - 使用列表预处理输入数据

2. 错误处理：
   - 添加参数校验，确保输入数据的合法性
   - 使用 try-except 捕获和处理异常
   - 提供清晰的错误信息

3. 性能优化：
   - 使用一维数组替代二维数组，减少内存占用

- 提前过滤无效物品，减少不必要的计算
- 使用局部变量缓存频繁访问的值

4. 代码可读性：
   - 使用有意义的变量名
   - 添加详细的文档字符串
   - 模块化设计，将算法逻辑与 IO 处理分离
'''

'''
调试与测试建议

1. 小数据测试：
   - 使用简单的测试用例验证算法正确性
   - 例如：n=3, t=5, weights=[2,3,4], values=[3,4,5]，预期结果应为最大值

2. 边界测试：
   - 测试 n=0 或 t=0 的情况
   - 测试所有物品重量都大于 t 的情况
   - 测试存在重量为 0 的物品的情况

3. 性能测试：
   - 对于大规模数据，测试算法的运行时间和内存使用
   - 比较不同优化方法的效果

4. 正确性验证：
   - 与二维 DP 实现的结果进行对比
   - 确保空间优化版本的结果正确
'''

'''
Python 语言特性利用

1. 动态类型：
   - 无需声明变量类型，代码更简洁
   - 但需要注意类型安全，添加适当的校验

2. 列表操作：
   - 使用列表推导式可以简化代码
   - 但要注意大规模数据下的性能问题

3. 异常处理：
   - Python 的异常处理机制完善

   - 可以方便地捕获和处理各种错误情况

4. 内置函数：
   - 利用 Python 丰富的内置函数提高开发效率
   - 但要注意算法竞赛中的性能要求
'''

==========

文件: Code04_BoundedKnapsackWithMonotonicQueue.cpp
==========
```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <sstream>
using namespace std;

/**
 * 多重背包问题的单调队列优化实现类
 *
 * 技术要点：
 * 1. 实现二维 DP 和空间压缩一维 DP 两种方法
 * 2. 使用同余分组技术将背包容量分解
 * 3. 应用单调队列维护滑动窗口最大值
 * 4. 通过数学变形将 O(n * t * c)优化为 O(n * t)
 * 5. 采用高效的数组实现单调队列，避免对象创建开销
 */
class BoundedKnapsackWithMonotonicQueue {
private:
    /** 物品数量的最大可能值 */
    static constexpr int MAXN = 101;

    /** 背包容量的最大可能值 */
    static constexpr int MAXW = 40001;

    /** 物品价值数组 */
    vector<int> v;

    /** 物品重量数组 */
    vector<int> w;

    /** 物品数量数组 */
```

```
vector<int> c;

/** 动态规划数组 */
vector<int> dp;

/** 单调队列：存储容量索引 */
vector<int> queue;

/** 物品数量 */
int n;

/** 背包容量 */
int t;

/**
 * 二维DP中用于计算价值贡献的辅助方法
 *
 * @param dp 二维DP数组
 * @param i 当前处理的物品编号
 * @param j 当前容量
 * @return 计算后的价值贡献：dp[i-1][j] - j/w[i] * v[i]
 */
int value1(const vector<vector<int>>& dp, int i, int j) const {
    return dp[i - 1][j] - j / w[i] * v[i];
}

/**
 * 一维DP中用于计算价值贡献的辅助方法
 *
 * @param i 当前处理的物品编号
 * @param j 当前容量
 * @return 计算后的价值贡献：dp[j] - j/w[i] * v[i]
 */
int value2(int i, int j) const {
    return dp[j] - j / w[i] * v[i];
}

/**
 * 二维DP实现 + 单调队列优化枚举
 *
 * 算法原理：
 * 1. 使用二维数组dp[i][j]表示前i个物品，容量为j时的最大价值
 * 2. 对每个物品，按模w[i]的余数分组处理
```

```
 * 3. 对每组内的元素，使用单调队列维护滑动窗口内的最优值
 * 4. 通过数学变形，将状态转移方程转换为可以应用单调队列的形式
 *
 * 时间复杂度：O(n * t)
 * 空间复杂度：O(n * t)
 *
 * @return 背包能够装下的最大价值
 */
int compute1() {
    // 边界情况快速处理
    if (n == 0 || t == 0) {
        return 0;
    }

    // 初始化二维 DP 数组
    vector<vector<int>> dp(n + 1, vector<int>(t + 1, 0));

    // 遍历每个物品
    for (int i = 1; i <= n; i++) {
        int vi = v[i]; // 当前物品价值
        int wi = w[i]; // 当前物品重量
        int ci = c[i]; // 当前物品数量

        // 优化 1：跳过数量为 0 的物品
        if (ci == 0) {
            dp[i] = dp[i-1]; // 复制上一行的数据
            continue;
        }

        // 优化 2：跳过价值为 0 的物品（选了也不增加总价值）
        if (vi == 0) {
            dp[i] = dp[i-1]; // 复制上一行的数据
            continue;
        }

        // 优化 3：跳过重量为 0 的物品（特殊情况处理）
        if (wi == 0) {
            // 重量为 0 的物品可以全部放入，但需要特殊处理
            dp[i] = dp[i-1]; // 复制上一行的数据
            continue;
        }

        // 优化 4：跳过重量超过背包容量的物品
```

```
        if (wi > t) {
            dp[i] = dp[i-1]; // 复制上一行的数据
            continue;
        }

        // 优化 5：调整物品数量上限，避免无意义的计算
        ci = min(ci, t / wi);

        // 同余分组处理：将容量 j 按模 wi 的余数分组
        for (int mod = 0; mod <= min(t, wi - 1); mod++) {
            int l = 0, r = 0; // 队列的头尾指针

            // 遍历同余类中的每个容量 j = mod, mod+wi, mod+2*wi, ...
            for (int j = mod; j <= t; j += wi) {
                // 维护单调队列：保证队列中的元素对应的 value1 值单调递减
                while (l < r) {
                    int lastIdx = queue[r - 1];
                    if (value1(dp, i, lastIdx) <= value1(dp, i, j)) {
                        r--; // 弹出队尾元素，因为当前元素更优
                    } else {
                        break; // 队列保持单调递减
                    }
                }

                // 将当前位置加入队列
                queue[r++] = j;

                // 移除超出窗口大小的元素
                while (l < r && queue[l] < j - ci * wi) {
                    l++; // 弹出队首元素，因为它已超出窗口范围
                }

                // 确保队列非空
                if (l < r) {
                    // 计算 dp[i][j]
                    dp[i][j] = value1(dp, i, queue[l]) + j / wi * vi;
                }

                // 确保 dp[i][j]不小于 dp[i-1][j]（不选当前物品的情况）
                dp[i][j] = max(dp[i][j], dp[i-1][j]);
            }
        }
    }
```

```
    // 返回最终结果
    return dp[n][t];
}

/**
 * 空间压缩的动态规划 + 单调队列优化枚举
 *
 * 算法特点：
 * 1. 使用一维数组 dp[j]表示容量为 j 时的最大价值
 * 2. 从右向左遍历容量，确保使用的是上一轮的状态值
 * 3. 仍然使用同余分组和单调队列优化
 * 4. 比二维 DP 版本节省 O(n*t)的空间
 *
 * 时间复杂度：O(n * t)
 * 空间复杂度：O(t)
 *
 * @return 背包能够装下的最大价值
 */
int compute2() {
    // 边界情况快速处理
    if (n == 0 || t == 0) {
        return 0;
    }

    // 遍历每个物品
    for (int i = 1; i <= n; i++) {
        int vi = v[i]; // 当前物品价值
        int wi = w[i]; // 当前物品重量
        int ci = c[i]; // 当前物品数量

        // 优化 1：跳过数量为 0 的物品
        if (ci == 0) {
            continue;
        }

        // 优化 2：跳过价值为 0 的物品
        if (vi == 0) {
            continue;
        }

        // 优化 3：跳过重量为 0 的物品
        if (wi == 0) {
```

```
        continue;
    }

    // 优化 4：跳过重量超过背包容量的物品
    if (wi > t) {
        continue;
    }

    // 优化 5：调整物品数量上限
    ci = min(ci, t / wi);

    // 同余分组处理
    for (int mod = 0; mod <= min(t, wi - 1); mod++) {
        int l = 0, r = 0; // 队列的头尾指针

        // 先把 ci 个的指标进入单调队列
        // 从最大的容量开始向左处理
        for (int j = t - mod, cnt = 1; j >= 0 && cnt <= ci; j -= wi, cnt++) {
            // 维护单调队列，保证队列中的元素对应的 value2 值单调递减
            while (l < r) {
                int lastIdx = queue[r - 1];
                if (value2(i, lastIdx) <= value2(i, j)) {
                    r--;
                } else {
                    break;
                }
            }
            queue[r++] = j;
        }

        // 滑动窗口计算每个位置的 dp 值
        for (int j = t - mod, enter = j - wi * ci; j >= 0; j -= wi, enter -= wi) {
            // 窗口进入 enter 位置的指标（如果 enter 有效）
            if (enter >= 0) {
                while (l < r) {
                    int lastIdx = queue[r - 1];
                    if (value2(i, lastIdx) <= value2(i, enter)) {
                        r--;
                    } else {
                        break;
                    }
                }
                queue[r++] = enter;
```

```
                    }

                    // 移除队列头部超出窗口范围的元素
                    while (l < r && queue[l] < j - ci * wi) {
                        l++;
                    }

                    // 计算当前位置的 dp 值
                    if (l < r) {
                        int candidate = value2(i, queue[l]) + j / wi * vi;
                        if (candidate > dp[j]) {
                            dp[j] = candidate;
                        }
                    }
                }
            }
        }

        // 返回最终结果
        return dp[t];
    }

    /**
     * 解析一行输入为整数数组
     *
     * @param line 输入的一行字符串
     * @return 解析后的整数数组
     */
    vector<int> parseLine(const string& line) {
        vector<int> result;
        istringstream iss(line);
        int num;
        while (iss >> num) {
            result.push_back(num);
        }
        return result;
    }

public:
    BoundedKnapsackWithMonotonicQueue() :
        v(MAXN, 0),
        w(MAXN, 0),
        c(MAXN, 0),
```

```
    dp(MAXW, 0),
    queue(MAXW, 0),
    n(0),
    t(0) {}

/**
 * 运行程序的主方法
 */
void run() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    string line;
    while (getline(cin, line)) {
        // 跳过空行
        if (line.empty()) continue;

        vector<int> firstLine = parseLine(line);
        if (firstLine.size() < 2) continue;

        n = firstLine[0];
        t = firstLine[1];

        // 初始化 dp 数组为 0
        fill(dp.begin(), dp.begin() + t + 1, 0);

        // 读取每个物品的价值、重量和数量
        for (int i = 1; i <= n; i++) {
            while (getline(cin, line) && line.empty()); // 跳过空行
            vector<int> itemData = parseLine(line);
            if (itemData.size() >= 3) {
                v[i] = itemData[0];
                w[i] = itemData[1];
                c[i] = itemData[2];
            }
        }

        // 边界情况快速处理
        if (n == 0 || t == 0) {
            cout << 0 << endl;
            continue;
        }
```

```
            // 调用空间优化的单调队列实现，输出结果
            cout << compute2() << endl;
        }
    }

    /**
     * 单调队列优化多重背包问题的数学原理详解
     *
     * 1. 朴素多重背包状态转移方程：
     *    dp[i][j] = max{ dp[i-1][j-k*w[i]] + k*v[i] }, 0 ≤ k ≤ min(c[i], j/w[i])
     *
     * 2. 同余分组思想：
     *    对于容量 j，我们可以将其表示为 j = m*w[i] + r，其中 0 ≤ r < w[i]
     *    这样，所有容量可以按照余数 r 分成 w[i]个组
     *    每组内的容量形式为 r, r+w[i], r+2*w[i], ...
     *
     * 3. 状态转移方程的数学变形：
     *    对于 j = m*w[i] + r，考虑 k 个物品 i 的选择：
     *    dp[i][m*w[i]+r] = max{ dp[i-1][(m-k)*w[i]+r] + k*v[i] }, 0 ≤ k ≤ min(c[i], m)
     *
     *    令 l = m - k，则 k = m - l，此时：
     *    dp[i][m*w[i]+r] = max{ dp[i-1][l*w[i]+r] + (m-l)*v[i] }, max(0, m-c[i]) ≤ l ≤ m
     *
     *    进一步变形：
     *    dp[i][m*w[i]+r] = max{ dp[i-1][l*w[i]+r] - l*v[i] } + m*v[i]
     *
     *    令 f(l) = dp[i-1][l*w[i]+r] - l*v[i]
     *    则 dp[i][m*w[i]+r] = max{ f(l) } + m*v[i]
     */

    /**
     * 与 Java 版本的差异：
     * 1. 使用 vector 替代数组，提供更好的内存管理
     * 2. 输入处理使用 cin 和 getline，而非 Java 的 BufferedReader
     * 3. 使用 fill 函数替代 Arrays.fill
     * 4. 在 compute1 中使用 vector 的赋值操作替代 System.arraycopy
     * 5. 使用引用传递 dp 数组以避免复制开销
     * 6. 使用 ios::sync_with_stdio(false)和 cin.tie(nullptr)加速输入输出
     */
};

/**
 * 主函数
```

```
 */
int main() {
    BoundedKnapsackWithMonotonicQueue solution;
    solution.run();
    return 0;
}
```

==========

文件: Code04_BoundedKnapsackWithMonotonicQueue.java

==========

```
package class075;

/**
 * 多重背包问题 - 单调队列优化实现
 *
 * 问题描述：
 * 有一个容量为 t 的背包，共有 n 种物品
 * 每种物品 i 有以下属性：
 * - 价值 v[i]
 * - 重量 w[i]
 * - 数量 c[i]
 * 要求在不超过背包容量的前提下，选择物品使得总价值最大
 *
 * 算法分类：动态规划 - 多重背包问题 - 单调队列优化
 *
 * 单调队列优化原理：
 * 1. 将背包容量按模 w[i]的余数分组
 * 2. 对每组内的状态转移进行优化，使用单调队列维护滑动窗口内的最大值
 * 3. 数学变形将状态转移方程转化为可以应用单调队列的形式
 * 4. 时间复杂度从 O(n * t * c)优化到 O(n * t)
 *
 * 适用场景：
 * - 数据规模非常大的情况
 * - 物品数量很大或背包容量很大时
 * - 需要在严格时间限制下运行的场景
 *
 * 测试链接：https://www.luogu.com.cn/problem/P1776（宝物筛选）
 *
 * 实现特点：
 * 1. 提供二维 DP 实现（compute1）和空间压缩的一维 DP 实现（compute2）
 * 2. 使用同余分组技术将问题分解为多个子问题
 * 3. 利用单调队列数据结构维护窗口内最大值
```

```
 * 4. 通过数学变形优化状态转移过程
 */

/**
 * 相关题目扩展（各大算法平台）：
 * 1. LeetCode（力扣）：
 *    - 474. Ones and Zeroes - https://leetcode.cn/problems/ones-and-zeroes/
 *      多维01背包问题，每个字符串需要同时消耗0和1的数量
 *    - 879. Profitable Schemes - https://leetcode.cn/problems/profitable-schemes/
 *      二维费用背包问题，需要同时考虑人数和利润
 *    - 322. Coin Change - https://leetcode.cn/problems/coin-change/
 *      完全背包问题，求组成金额所需的最少硬币数
 *
 * 2. 洛谷（Luogu）：
 *    - P1776 宝物筛选 - https://www.luogu.com.cn/problem/P1776
 *      经典多重背包问题
 *    - P1833 樱花 - https://www.luogu.com.cn/problem/P1833
 *      混合背包问题，包含01背包、完全背包和多重背包
 *    - P1679 聪明的收银员 - https://www.luogu.com.cn/problem/P1679
 *      多重背包在找零问题中的应用
 *
 * 3. POJ：
 *    - POJ 1742. Coins - http://poj.org/problem?id=1742
 *      多重背包可行性问题，计算能组成多少种金额
 *    - POJ 1276. Cash Machine - http://poj.org/problem?id=1276
 *      多重背包优化问题，使用二进制优化或单调队列优化
 *    - POJ 3260. The Fewest Coins - http://poj.org/problem?id=3260
 *      双向背包问题，同时考虑找零和支付
 *
 * 4. HDU：
 *    - HDU 2191. 珍惜现在，感恩生活 - http://acm.hdu.edu.cn/showproblem.php?pid=2191
 *      多重背包问题的典型应用
 *    - HDU 3449. Consumer - http://acm.hdu.edu.cn/showproblem.php?pid=3449
 *      分组背包与完全背包的混合应用
 *
 * 5. Codeforces：
 *    - Codeforces 106C. Buns - https://codeforces.com/contest/106/problem/C
 *      分组背包与多重背包的混合应用
 *    - Codeforces 1003F. Abbreviation - https://codeforces.com/contest/1003/problem/F
 *      字符串处理与多重背包的结合
 *
 * 6. AtCoder：
 *    - AtCoder DP Contest Problem F - https://atcoder.jp/contests/dp/tasks/dp_f
```

```
 *      最长公共子序列与背包思想的结合
 *    - AtCoder ABC153 F. Silver Fox vs Monster -
https://atcoder.jp/contests/abc153/tasks/abc153_f
 *      贪心+前缀和优化的背包问题
 *
 * 7. SPOJ：
 *    - SPOJ KNAPSACK - https://www.spoj.com/problems/KNAPSACK/
 *      经典 01 背包问题
 *
 * 8. 牛客网：
 *    - NC17881. 最大价值 - https://ac.nowcoder.com/acm/problem/17881
 *      多重背包问题的变形应用
 *
 * 9. AcWing：
 *    - AcWing 5. 多重背包问题 II - https://www.acwing.com/problem/content/description/5/
 *      二进制优化的多重背包问题标准题目
 *
 * 10. UVa OJ：
 *     - UVa 10130. SuperSale -
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1071
 *       01 背包问题的简单应用
 */

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.util.Arrays;

/**
 * 多重背包问题的单调队列优化实现类
 *
 * 技术要点：
 * 1. 实现二维 DP 和空间压缩一维 DP 两种方法
 * 2. 使用同余分组技术将背包容量分解
 * 3. 应用单调队列维护滑动窗口最大值
 * 4. 通过数学变形将 O(n * t * c)优化为 O(n * t)
 * 5. 采用高效的数组实现单调队列，避免对象创建开销
 */
public class Code04_BoundedKnapsackWithMonotonicQueue {

    /** 物品数量的最大可能值 */
```

```
public static int MAXN = 101;

/** 背包容量的最大可能值 */
public static int MAXW = 40001;

/** 物品价值数组：v[i]表示第 i 个物品的价值 */
public static int[] v = new int[MAXN];

/** 物品重量数组：w[i]表示第 i 个物品的重量 */
public static int[] w = new int[MAXN];

/** 物品数量数组：c[i]表示第 i 个物品的可用数量 */
public static int[] c = new int[MAXN];

/** 动态规划数组：dp[j]表示背包容量为 j 时的最大价值（空间压缩版本） */
public static int[] dp = new int[MAXW];

/** 单调队列：用于维护滑动窗口内的最大值的索引 */
public static int[] queue = new int[MAXW];

/** 队列的头尾指针 */
public static int l, r;

/** 物品数量 */
public static int n;

/** 背包容量 */
public static int t;

/**
 * 主方法
 * 处理输入、调用计算逻辑、输出结果
 *
 * 工程化考量：
 * 1. 使用高效的 IO 处理方式，避免输入输出成为性能瓶颈
 * 2. 使用 try-with-resources 确保资源正确关闭
 * 3. 支持多组测试用例的连续读取
 * 4. 选择 compute2 方法（空间优化版本）以提高内存使用效率
 * 5. 完善边界情况处理，增强代码健壮性
 *
 * @param args 命令行参数（未使用）
 * @throws IOException 输入输出异常
 */
```

```
public static void main(String[] args) throws IOException {
    // 使用 try-with-resources 自动关闭资源
    try (BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
         PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out))) {

        String line;
        // 循环读取多组测试用例直到输入结束
        while ((line = br.readLine()) != null) {
            // 跳过空行
            if (line.trim().isEmpty()) continue;

            String[] parts = line.trim().split("\\s+");
            int idx = 0;
            n = Integer.parseInt(parts[idx++]);
            t = Integer.parseInt(parts[idx++]);

            // 初始化 dp 数组为 0，确保每组测试用例之间互不影响
            // 使用 Arrays.fill 提高效率
            Arrays.fill(dp, 0, t + 1, 0);

            // 读取每个物品的价值、重量和数量
            for (int i = 1; i <= n; i++) {
                line = br.readLine();
                while (line != null && line.trim().isEmpty()) {
                    line = br.readLine(); // 跳过空行
                }
                if (line == null) break;

                String[] itemParts = line.trim().split("\\s+");
                v[i] = Integer.parseInt(itemParts[0]);
                w[i] = Integer.parseInt(itemParts[1]);
                c[i] = Integer.parseInt(itemParts[2]);
            }

            // 边界情况快速处理
            if (n == 0 || t == 0) {
                out.println(0);
                continue;
            }

            // 调用空间优化的单调队列实现，输出结果
            out.println(compute2());
        }
```

```
        // 确保输出全部写入
        out.flush();
    }
}

/**
 * 二维 DP 实现 + 单调队列优化枚举
 *
 * 算法原理：
 * 1. 使用二维数组 dp[i][j]表示前 i 个物品，容量为 j 时的最大价值
 * 2. 对每个物品，按模 w[i]的余数分组处理
 * 3. 对每组内的元素，使用单调队列维护滑动窗口内的最优值
 * 4. 通过数学变形，将状态转移方程转换为可以应用单调队列的形式
 *
 * 数学推导：
 * - 对于容量 j，若 j = m*w[i] + r，其中 0 ≤ r < w[i]
 * - 状态转移方程：dp[i][j] = max{ dp[i-1][j-k*w[i]] + k*v[i] }, 0 ≤ k ≤ c[i]
 * - 令 j-k*w[i] = r + (m-k)*w[i] = r + l*w[i]，则 k = m-l
 * - 代入得：dp[i][j] = max{ dp[i-1][r+l*w[i]] - l*v[i] } + m*v[i]
 * - 其中 l 的取值范围：max(0, m-c[i]) ≤ l ≤ m
 * - 这样，对于固定的余数 r，我们可以使用单调队列维护 max{ dp[i-1][r+l*w[i]] - l*v[i] }
 *
 * 时间复杂度：O(n * t)
 * 空间复杂度：O(n * t)
 *
 * 优化点：
 * 1. 预处理边界情况，跳过无效物品
 * 2. 优化队列操作，提高访问效率
 * 3. 缓存频繁访问的数组元素
 *
 * @return 背包能够装下的最大价值
 */
public static int compute1() {
    // 边界情况快速处理
    if (n == 0 || t == 0) {
        return 0;
    }

    // 初始化二维 DP 数组
    int[][] dp = new int[n + 1][t + 1];

    // 遍历每个物品
```

```
for (int i = 1; i <= n; i++) {
    int vi = v[i]; // 当前物品价值
    int wi = w[i]; // 当前物品重量
    int ci = c[i]; // 当前物品数量

    // 优化 1：跳过数量为 0 的物品
    if (ci == 0) {
        System.arraycopy(dp[i-1], 0, dp[i], 0, t + 1);
        continue;
    }

    // 优化 2：跳过价值为 0 的物品（选了也不增加总价值）
    if (vi == 0) {
        System.arraycopy(dp[i-1], 0, dp[i], 0, t + 1);
        continue;
    }

    // 优化 3：跳过重量为 0 的物品（特殊情况处理）
    if (wi == 0) {
        // 重量为 0 的物品可以全部放入，但需要特殊处理
        // 这里为了简化，我们直接复制上一行的数据
        System.arraycopy(dp[i-1], 0, dp[i], 0, t + 1);
        continue;
    }

    // 优化 4：跳过重量超过背包容量的物品
    if (wi > t) {
        System.arraycopy(dp[i-1], 0, dp[i], 0, t + 1);
        continue;
    }

    // 优化 5：调整物品数量上限，避免无意义的计算
    ci = Math.min(ci, t / wi);

    // 同余分组处理：将容量 j 按模 wi 的余数分组
    // 余数范围：0 ≤ mod ≤ min(t, wi-1)
    for (int mod = 0; mod <= Math.min(t, wi - 1); mod++) {
        // 重置队列
        l = r = 0;

        // 遍历同余类中的每个容量 j = mod, mod+wi, mod+2*wi, ...
        for (int j = mod; j <= t; j += wi) {
            // 当前容量 j 可以表示为 j = m*wi + r，其中 r=mod，m=j/wi
```

```
                // 计算当前位置的候选值：dp[i-1][j] - m*vi
                // 这是经过数学变形后的形式，便于使用单调队列优化

                // 维护单调队列：保证队列中的元素对应的 value1 值单调递减
                // 移除队列尾部所有小于等于当前位置 value1 值的元素
                while (l < r) {
                    int lastIdx = queue[r - 1];
                    if (value1(dp, i, lastIdx) <= value1(dp, i, j)) {
                        r--; // 弹出队尾元素，因为当前元素更优
                    } else {
                        break; // 队列保持单调递减
                    }
                }

                // 将当前位置加入队列
                queue[r++] = j;

                // 移除超出窗口大小的元素
                // 窗口大小为 ci+1，表示最多可以选 ci 个当前物品
                // 当队列头部元素对应容量小于 j - ci*wi 时，该元素不再可用
                while (l < r && queue[l] < j - ci * wi) {
                    l++; // 弹出队首元素，因为它已超出窗口范围
                }

                // 确保队列非空（理论上不应该为空，因为至少有当前元素）
                if (l < r) {
                    // 计算 dp[i][j]：队列头部元素对应的值 + 当前容量对应的价值贡献
                    dp[i][j] = value1(dp, i, queue[l]) + j / wi * vi;
                }

                // 确保 dp[i][j]不小于 dp[i-1][j]（不选当前物品的情况）
                // 这是因为在同余分组处理中，我们可能没有考虑到不选当前物品的情况
                dp[i][j] = Math.max(dp[i][j], dp[i-1][j]);
            }
        }
    }

    // 返回最终结果
    return dp[n][t];
}

/**
 * 二维 DP 中用于计算价值贡献的辅助方法
```

```
     *
     * 这个函数计算的是经过数学变形后的值，用于单调队列优化
     *
     * @param dp 二维 DP 数组
     * @param i 当前处理的物品编号
     * @param j 当前容量
     * @return 计算后的价值贡献：dp[i-1][j] - j/w[i] * v[i]
     */
    public static int value1(int[][] dp, int i, int j) {
        return dp[i - 1][j] - j / w[i] * v[i];
    }

    /**
     * 空间压缩的动态规划 + 单调队列优化枚举
     *
     * 算法特点：
     * 1. 使用一维数组 dp[j]表示容量为 j 时的最大价值
     * 2. 从右向左遍历容量，确保使用的是上一轮的状态值
     * 3. 仍然使用同余分组和单调队列优化
     * 4. 比二维 DP 版本节省 O(n*t)的空间
     *
     * 实现细节：
     * - 对于每个余数 mod，从最大的容量开始向左处理
     * - 先将窗口大小内的 c[i]个位置加入队列
     * - 然后滑动窗口，每次计算当前位置的最优值
     * - 窗口向右移动时，移除超出范围的元素，加入新的元素
     *
     * 时间复杂度：O(n * t)
     * 空间复杂度：O(t)
     *
     * 优化点：
     * 1. 预处理边界情况，跳过无效物品
     * 2. 优化队列操作，减少重复计算
     * 3. 缓存频繁使用的值，减少数组访问
     * 4. 增强边界检查，提高代码健壮性
     *
     * @return 背包能够装下的最大价值
     */
    public static int compute2() {
        // 边界情况快速处理
        if (n == 0 || t == 0) {
            return 0;
        }
```

```
// 遍历每个物品
for (int i = 1; i <= n; i++) {
    int vi = v[i]; // 当前物品价值
    int wi = w[i]; // 当前物品重量
    int ci = c[i]; // 当前物品数量

    // 优化 1：跳过数量为 0 的物品
    if (ci == 0) {
        continue;
    }

    // 优化 2：跳过价值为 0 的物品（选了也不增加总价值）
    if (vi == 0) {
        continue;
    }

    // 优化 3：跳过重量为 0 的物品（特殊情况）
    if (wi == 0) {
        continue;
    }

    // 优化 4：跳过重量超过背包容量的物品
    if (wi > t) {
        continue;
    }

    // 优化 5：调整物品数量上限，避免无意义的计算
    ci = Math.min(ci, t / wi);

    // 同余分组处理
    for (int mod = 0; mod <= Math.min(t, wi - 1); mod++) {
        // 重置队列
        l = r = 0;

        // 先把 ci 个的指标进入单调队列
        // 从最大的容量开始向左处理
        for (int j = t - mod, cnt = 1; j >= 0 && cnt <= ci; j -= wi, cnt++) {
            // 维护单调队列，保证队列中的元素对应的 value2 值单调递减
            while (l < r) {
                int lastIdx = queue[r - 1];
                if (value2(i, lastIdx) <= value2(i, j)) {
                    r--;
```

```
                } else {
                    break;
                }
            }
            queue[r++] = j;
        }

        // 滑动窗口计算每个位置的 dp 值
        for (int j = t - mod, enter = j - wi * ci; j >= 0; j -= wi, enter -= wi) {
            // 窗口进入 enter 位置的指标（如果 enter 有效）
            if (enter >= 0) {
                while (l < r) {
                    int lastIdx = queue[r - 1];
                    if (value2(i, lastIdx) <= value2(i, enter)) {
                        r--;
                    } else {
                        break;
                    }
                }
                queue[r++] = enter;
            }

            // 移除队列头部超出窗口范围的元素
            // 窗口范围：[j - ci * wi, j]
            while (l < r && queue[l] < j - ci * wi) {
                l++;
            }

            // 计算当前位置的 dp 值
            // 只有当队列非空时才更新，否则保持原值（不选当前物品）
            if (l < r) {
                int candidate = value2(i, queue[l]) + j / wi * vi;
                if (candidate > dp[j]) {
                    dp[j] = candidate;
                }
            }

            // 窗口弹出 j 位置的指标（因为下一个循环会处理更小的容量）
            // 注意：在这个实现中，我们不立即弹出 j，而是在下一轮循环的开始检查是否超出范围
        }
    }
}
```

```
    // 返回最终结果
    return dp[t];
}

/**
 * 一维 DP 中用于计算价值贡献的辅助方法
 *
 * 这个函数计算的是经过数学变形后的值，用于单调队列优化
 *
 * @param i 当前处理的物品编号
 * @param j 当前容量
 * @return 计算后的价值贡献：dp[j] - j/w[i] * v[i]
 */
public static int value2(int i, int j) {
    return dp[j] - j / w[i] * v[i];
}

/**
 * 单调队列优化多重背包问题的数学原理详解
 *
 * 1. 朴素多重背包状态转移方程：
 *    dp[i][j] = max{ dp[i-1][j-k*w[i]] + k*v[i] }, 0 ≤ k ≤ min(c[i], j/w[i])
 *
 * 2. 同余分组思想：
 *    对于容量 j，我们可以将其表示为 j = m*w[i] + r，其中 0 ≤ r < w[i]
 *    这样，所有容量可以按照余数 r 分成 w[i]个组
 *    每组内的容量形式为 r, r+w[i], r+2*w[i], ...
 *
 * 3. 状态转移方程的数学变形：
 *    对于 j = m*w[i] + r，考虑 k 个物品 i 的选择：
 *    dp[i][m*w[i]+r] = max{ dp[i-1][(m-k)*w[i]+r] + k*v[i] }, 0 ≤ k ≤ min(c[i], m)
 *
 *    令 l = m - k，则 k = m - l，此时：
 *    dp[i][m*w[i]+r] = max{ dp[i-1][l*w[i]+r] + (m-l)*v[i] }, max(0, m-c[i]) ≤ l ≤ m
 *
 *    进一步变形：
 *    dp[i][m*w[i]+r] = max{ dp[i-1][l*w[i]+r] - l*v[i] } + m*v[i]
 *
 *    令 f(l) = dp[i-1][l*w[i]+r] - l*v[i]
 *    则 dp[i][m*w[i]+r] = max{ f(l) } + m*v[i]
 *
 * 4. 单调队列优化原理：
 *    - 对于固定的 r，我们需要维护窗口[l_low, m]中的最大值，其中 l_low = max(0, m-c[i])
```

```
 *     - 随着 m 的增加，窗口向右滑动，新的 l 值进入窗口，旧的值可能滑出窗口
 *     - 使用单调队列可以在 O(1)时间内获取窗口最大值，整体时间复杂度为 O(n*t)
 */

/**
 * 代码优化与工程化考量
 *
 * 1. 边界情况处理：
 *     - 处理重量为 0 的物品
 *     - 处理价值为 0 的物品
 *     - 处理数量为 0 的物品
 *     - 处理重量超过背包容量的物品
 *     - 确保队列操作的安全性（队列非空检查）
 *     - 处理 m=0 的特殊情况
 *     - 处理 n=0 或 t=0 的边界情况
 *
 * 2. 性能优化技巧：
 *     - 使用 System.arraycopy 进行数组复制，比循环复制效率更高
 *     - 预先计算和缓存重复使用的值，减少计算量
 *     - 优化队列操作的条件判断，减少不必要的循环
 *     - 使用局部变量缓存数组元素，减少数组访问次数
 *     - 使用 Arrays.fill 快速初始化数组
 *     - 调整物品数量上限，避免无意义的计算
 *     - 提前剪枝，跳过无法产生更优解的分支
 *
 * 3. 代码健壮性增强：
 *     - 添加防御性编程检查
 *     - 确保队列指针不会越界
 *     - 处理数值溢出问题
 *     - 确保测试用例之间的独立性
 *     - 使用 try-with-resources 确保资源正确关闭
 *     - 添加输入校验，处理空行和不完整输入
 *
 * 4. 代码可读性提升：
 *     - 使用有意义的变量名
 *     - 添加详细的注释说明
 *     - 模块化设计函数
 *     - 遵循 Java 编码规范
 *     - 将复杂的逻辑拆分为更小的函数
 *     - 添加空白行和缩进，提高代码结构清晰度
 */

/**
```

```
 * 单调队列实现细节与优化
 *
 * 1. 队列维护策略：
 *    - 队列中存储的是容量索引 j，而非直接存储 value 值
 *    - 通过比较 value 函数的返回值来维护单调性
 *    - 队列保持严格单调递减，确保队首总是当前窗口的最大值
 *
 * 2. 窗口管理：
 *    - 窗口大小为 c[i]+1，表示最多选择 c[i]个当前物品
 *    - 进入新元素时，从队尾移除所有不优的元素
 *    - 从队首移除超出窗口范围的元素
 *
 * 3. 数组实现优势：
 *    - 避免了使用链表或集合类的对象创建开销
 *    - 内存访问更连续，缓存命中率更高
 *    - 常数因子更小，对于大规模数据更高效
 */

/**
 * 测试与调试建议
 *
 * 1. 单元测试策略：
 *    - 测试边界情况：n=0、t=0、c[i]=0 等
 *    - 测试特殊物品：w[i]=0、v[i]=0 的情况
 *    - 测试小规模数据，验证算法正确性
 *    - 测试大规模数据，验证性能和内存使用
 *
 * 2. 调试技巧：
 *    - 添加日志输出队列状态和 dp 数组的变化
 *    - 可视化同余分组的处理过程
 *    - 比较二维 DP 和一维 DP 的结果是否一致
 *    - 使用小数据手动计算验证
 *
 * 3. 性能测试：
 *    - 与二进制优化版本进行性能对比
 *    - 测试不同规模数据下的时间消耗
 *    - 分析不同优化手段的效果
 */

/**
 * 算法学习路径
 *
 * 1. 基础学习阶段：
```

```
 *     - 先掌握 01 背包和完全背包问题
 *     - 理解多重背包的基本状态转移方程
 *     - 学习二进制优化方法
 *
 * 2. 进阶优化阶段：
 *     - 学习同余分组的数学原理
 *     - 掌握单调队列数据结构
 *     - 理解状态转移方程的数学变形
 *
 * 3. 工程应用阶段：
 *     - 根据数据规模选择合适的优化方法
 *     - 处理各种边界情况和异常输入
 *     - 优化代码性能，适应实际应用场景
 *
 * 4. 扩展学习阶段：
 *     - 学习多维费用的背包问题
 *     - 掌握分组背包、依赖背包等变形
 *     - 将背包思想应用到其他动态规划问题
 */

/**
 * 单调队列优化多重背包问题的总结与工程应用考量
 *
 * 1. 算法优劣分析：
 *     - 优势：理论时间复杂度最优 O(n*t)，适用于大规模数据
 *     - 劣势：实现复杂，需要深入理解数学变形和单调队列的应用
 *
 * 2. 与其他优化方法的比较：
 *     - 朴素实现：O(n*t*c)，实现简单但效率最低
 *     - 二进制优化：O(n*t*log c)，实现较简单，效率较高
 *     - 单调队列优化：O(n*t)，实现复杂，效率最高
 *
 * 3. 工程选择建议：
 *     - 对于一般规模的问题，二进制优化通常是最佳选择（平衡实现复杂度和效率）
 *     - 对于非常大的数据规模且时间限制严格的场景，才考虑单调队列优化
 *     - 在实际应用中，应根据具体问题的数据特征选择合适的优化方法
 *
 * 4. 代码健壮性增强：
 *     - 应处理物品重量为 0 的特殊情况
 *     - 对于物品数量为 0 的情况可以跳过处理
 *     - 在处理大数值时，需要考虑整数溢出问题
 *     - 添加适当的参数校验和异常处理
 *
```

```
     * 5. 性能优化技巧：
     *    - 对于 w[i]很大的物品，可以单独处理或结合二进制优化
     *    - 使用更快的数据结构实现单调队列，减少常数因子
     *    - 预先计算和缓存一些重复使用的值
     *    - 对于特殊数据分布，可以采用混合优化策略
     *
     * 6. 扩展应用场景：
     *    - 多维费用的多重背包问题
     *    - 分组背包与多重背包的混合应用
     *    - 有依赖关系的背包问题
     *    - 其他需要滑动窗口最大值优化的动态规划问题
     */

    /**
     * 与标准库实现的对比
     *
     * 1. 数据结构选择：
     *    - 我们使用数组实现单调队列，而标准库中可能使用双端队列（如 LinkedList）
     *    - 数组实现的队列在性能上更优，因为避免了对象创建和内存分配的开销
     *
     * 2. 边界处理：
     *    - 标准库的集合类通常有更完善的边界检查和异常处理
     *    - 我们的实现为了性能考虑，假设输入数据是合法的
     *    - 在实际工程应用中，应添加更多的参数校验
     *
     * 3. 代码可复用性：
     *    - 我们的实现是专用的，针对多重背包问题进行了优化
     *    - 可以将单调队列部分抽取为独立的工具类，提高代码复用性
     */
}

==========================================

文件: Code04_BoundedKnapsackWithMonotonicQueue.py
==========================================
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
多重背包问题的单调队列优化实现

问题描述：
有一个容量为 t 的背包，共有 n 种物品
```

每种物品 i 有以下属性：
- 价值 v[i]
- 重量 w[i]
- 数量 c[i]
要求在不超过背包容量的前提下，选择物品使得总价值最大

算法分类：动态规划 - 多重背包问题 - 单调队列优化

单调队列优化原理：
1. 将物品按照重量分组，对每组容量余数相同的状态进行优化
2. 使用单调队列维护窗口内的最优状态
3. 通过数学变形将状态转移方程转化为滑动窗口最大值问题
4. 时间复杂度优化至 O(n * t)，无论物品数量多大

适用场景：
- 物品数量和背包容量都很大的情况
- 对于二进制优化仍然不够高效的场景
- 需要达到理论最优时间复杂度的问题

相关题目扩展：
1. LeetCode 474. Ones and Zeroes - https://leetcode.cn/problems/ones-and-zeroes/
2. LeetCode 879. Profitable Schemes - https://leetcode.cn/problems/profitable-schemes/
3. POJ 1742. Coins - http://poj.org/problem?id=1742
4. POJ 1276. Cash Machine - http://poj.org/problem?id=1276
5. HDU 2191. 非常可乐 - http://acm.hdu.edu.cn/showproblem.php?pid=2191
6. AcWing 5. 多重背包问题 II - https://www.acwing.com/problem/content/description/5/
7. Codeforces 106C. Buns - https://codeforces.com/problemset/problem/106/C
8. 牛客网 NC17881. 最大价值 - https://ac.nowcoder.com/acm/problem/17881
"""

```
import sys
from collections import deque
from typing import List

class BoundedKnapsackWithMonotonicQueue:
    """
    多重背包问题的单调队列优化实现类

    技术要点：
    1. 使用单调队列优化状态转移过程
    2. 按照同余类分组处理容量
    3. 维护一个双端队列存储最优状态的索引
    4. 时间复杂度为 O(n * t)，空间复杂度为 O(t)
```

```
    """

    def __init__(self):
        """
        初始化类实例
        设置默认的数组大小上限
        """
        # 物品数量和背包容量的合理上限
        self.MAXN = 101
        self.MAXW = 40001

    def compute1(self, n: int, t: int, v: List[int], w: List[int], c: List[int]) -> int:
        """
        基本的动态规划 + 单调队列优化枚举
        使用二维数组 dp[i][j]表示前 i 种物品，背包容量为 j 时的最大价值

        算法思路：
        1. 对于每种物品，按照容量的余数分组
        2. 对每组使用单调队列维护滑动窗口内的最优状态
        3. 通过同余分组和滑动窗口技术，将时间复杂度优化至 O(n * t)

        数学原理：
        状态转移方程：dp[i][j] = max{ dp[i-1][j - k*w[i]] + k*v[i] }，其中 0 ≤ k ≤ min(c[i], j/w[i])
        令 j = m * w[i] + r，那么方程可以变形为：
        dp[i][m*w[i]+r] = max{ dp[i-1][(m-k)*w[i]+r] + k*v[i] }，其中 0 ≤ k ≤ min(c[i], m)
        令 k' = m - k，则：
        dp[i][m*w[i]+r] = max{ dp[i-1][k'*w[i]+r] - k'*v[i] } + m*v[i]，其中 max(0, m-c[i]) ≤ k' ≤ m
        这样就转化为了求滑动窗口内最大值的问题

        时间复杂度分析：
        O(n * t)，对于每种物品和每个容量位置，最多入队和出队一次
        相比朴素多重背包的 O(n * t * c[i])，优化效果显著

        空间复杂度分析：
        O(n * t)，使用二维数组存储所有状态

        参数：
            n: 物品数量
            t: 背包容量
            v: 物品价值数组（从 1 开始索引）
            w: 物品重量数组（从 1 开始索引）
```

```
    c: 物品数量数组（从 1 开始索引）

返回：
    背包能装下的最大价值
"""
# 边界情况快速处理
if n == 0 or t == 0:
    return 0

# 初始化二维 dp 数组
dp = [[0] * (t + 1) for _ in range(n + 1)]

# 同余分组和单调队列优化
for i in range(1, n + 1):
    int_vi = v[i]
    int_wi = w[i]
    int_ci = c[i]

    # 优化 1：跳过数量为 0 的物品
    if int_ci == 0:
        # 直接继承上一行的状态
        for j in range(t + 1):
            dp[i][j] = dp[i-1][j]
        continue

    # 优化 2：跳过价值为 0 的物品（选了也不增加总价值）
    if int_vi == 0:
        # 直接继承上一行的状态
        for j in range(t + 1):
            dp[i][j] = dp[i-1][j]
        continue

    # 优化 3：跳过重量为 0 的物品（特殊情况）
    if int_wi == 0:
        # 直接继承上一行的状态
        for j in range(t + 1):
            dp[i][j] = dp[i-1][j]
        continue

    # 优化 4：跳过重量超过背包容量的物品
    if int_wi > t:
        # 直接继承上一行的状态
        for j in range(t + 1):
```

```
                dp[i][j] = dp[i-1][j]
            continue

        # 优化 5：调整物品数量上限，避免无意义的计算
        int_ci = min(int_ci, t // int_wi)

        # 同余分组处理，对每个余数 r 进行处理
        for r in range(int_wi):
            # 创建单调队列
            dq = deque()

            # 计算当前余数下的有效容量范围
            max_m = (t - r) // int_wi
            for m in range(max_m + 1):
                # 当前容量 j = m * wi + r
                j = m * int_wi + r

                # 计算要加入队列的候选值
                # 这个值是 dp[i-1][j] - m * vi，用于在后续计算中加上 m * vi 得到最终结果
                candidate = dp[i-1][j] - m * int_vi

                # 维护单调队列：移除队尾小于等于 candidate 的元素
                while dq and candidate >= dp[i-1][dq[-1] * int_wi + r] - dq[-1] * int_vi:
                    dq.pop()

                # 将当前 m 加入队列
                dq.append(m)

                # 移除队列头部超出窗口范围的元素（窗口大小为 ci+1）
                while dq and m - dq[0] > int_ci:
                    dq.popleft()

                # 队列头部就是当前窗口内的最优解
                best_k = dq[0]
                # 计算当前状态值：最优解 + m * vi
                dp[i][j] = dp[i-1][best_k * int_wi + r] + (m - best_k) * int_vi

                # 确保不会比不选当前物品差
                dp[i][j] = max(dp[i][j], dp[i-1][j])

# 返回最终结果
return dp[n][t]
```

```
def compute2(self, n: int, t: int, v: List[int], w: List[int], c: List[int]) -> int:
    """
    空间压缩的动态规划 + 单调队列优化枚举

    算法特点：
    1. 使用一维数组 dp[j]表示容量为 j 时的最大价值
    2. 从右向左遍历容量，确保使用的是上一轮的状态值
    3. 仍然使用同余分组和单调队列优化
    4. 比二维 DP 版本节省 O(n*t)的空间

    时间复杂度：O(n * t)
    空间复杂度：O(t)

    参数：
        n: 物品数量
        t: 背包容量
        v: 物品价值数组（从 1 开始索引）
        w: 物品重量数组（从 1 开始索引）
        c: 物品数量数组（从 1 开始索引）

    返回：
        背包能装下的最大价值
    """
    # 边界情况快速处理
    if n == 0 or t == 0:
        return 0

    # 初始化一维 dp 数组
    dp = [0] * (t + 1)

    # 临时数组，用于保存上一轮的值（因为会边遍历边更新）
    pre = [0] * (t + 1)

    # 同余分组和单调队列优化
    for i in range(1, n + 1):
        # 复制上一轮的 dp 值到 pre 数组
        pre[:] = dp[:]

        int_vi = v[i]
        int_wi = w[i]
        int_ci = c[i]

        # 优化 1：跳过数量为 0 的物品
```

```
if int_ci == 0:
    continue

# 优化 2：跳过价值为 0 的物品（选了也不增加总价值）
if int_vi == 0:
    continue

# 优化 3：跳过重量为 0 的物品（特殊情况）
if int_wi == 0:
    continue

# 优化 4：跳过重量超过背包容量的物品
if int_wi > t:
    continue

# 优化 5：调整物品数量上限，避免无意义的计算
int_ci = min(int_ci, t // int_wi)

# 同余分组处理，对每个余数 r 进行处理
for r in range(int_wi):
    # 创建单调队列
    dq = deque()

    # 计算当前余数下的有效容量范围
    max_m = (t - r) // int_wi
    for m in range(max_m + 1):
        # 当前容量 j = m * wi + r
        j = m * int_wi + r

        # 计算要加入队列的候选值
        candidate = pre[j] - m * int_vi

        # 维护单调队列：移除队尾小于等于 candidate 的元素
        while dq and candidate >= pre[dq[-1] * int_wi + r] - dq[-1] * int_vi:
            dq.pop()

        # 将当前 m 加入队列
        dq.append(m)

        # 移除队列头部超出窗口范围的元素（窗口大小为 ci+1）
        while dq and m - dq[0] > int_ci:
            dq.popleft()
```

```
                # 队列头部就是当前窗口内的最优解
                best_k = dq[0]
                # 计算当前状态值：最优解 + m * vi
                current_value = pre[best_k * int_wi + r] + (m - best_k) * int_vi

                # 更新 dp 数组
                if current_value > dp[j]:
                    dp[j] = current_value

    # 返回最终结果
    return dp[t]

def parse_line(self, line: str) -> List[int]:
    """
    解析一行输入为整数列表

    参数：
        line: 输入的一行字符串

    返回：
        解析后的整数列表
    """
    return list(map(int, filter(lambda x: x.strip(), line.strip().split())))

def run(self):
    """
    运行程序的主方法
    处理输入、调用计算方法、输出结果

    工程化考量：
    1. 使用 sys.stdin 进行高效的输入处理
    2. 支持多组测试用例的连续读取
    3. 完善边界情况处理，增强代码健壮性
    4. 添加输入校验，处理空行和不完整输入
    """
    # 为了支持大数据量输入，使用 sys.stdin 读取所有行
    input_lines = [line.strip() for line in sys.stdin if line.strip()]
    ptr = 0

    # 预分配足够空间，避免频繁扩容
    v = [0] * (self.MAXN)
    w = [0] * (self.MAXN)
    c = [0] * (self.MAXN)
```

```
while ptr < len(input_lines):
    # 解析第一行：物品种类数和背包容量
    first_line = self.parse_line(input_lines[ptr])
    ptr += 1

    if len(first_line) < 2:
        continue

    n = first_line[0]
    t = first_line[1]

    # 边界情况快速处理
    if n == 0 or t == 0:
        print(0)
        continue

    # 读取每个物品的信息
    valid_items = 0
    for _ in range(n):
        if ptr >= len(input_lines):
            break

        item_data = self.parse_line(input_lines[ptr])
        ptr += 1

        if len(item_data) < 3:
            continue

        value = item_data[0]
        weight = item_data[1]
        cnt = item_data[2]

        # 物品过滤优化
        if value <= 0 or weight <= 0 or cnt <= 0 or weight > t:
            continue

        valid_items += 1
        v[valid_items] = value
        w[valid_items] = weight
        c[valid_items] = cnt

    # 使用空间压缩版本进行计算并输出结果
```

```
        print(self.compute2(valid_items, t, v, w, c))

    def monotonic_queue_optimization_principle(self) -> None:
        """
        单调队列优化原理解释

        1. 问题分析：
           对于多重背包问题，传统的状态转移方程是：
           dp[i][j] = max{ dp[i-1][j - k*w[i]] + k*v[i] }, 0 ≤ k ≤ min(c[i], j/w[i])
           这个方程的时间复杂度为O(n * t * c[i])，在物品数量很大时会很慢

        2. 数学变形：
           令 j = m * w[i] + r，那么状态转移方程可以变形为：
           dp[i][m*w[i]+r] = max{ dp[i-1][(m-k)*w[i]+r] + k*v[i] }, 0 ≤ k ≤ min(c[i], m)
           再令 k' = m - k，得到：
           dp[i][m*w[i]+r] = max{ dp[i-1][k'*w[i]+r] - k'*v[i] } + m*v[i], max(0, m-c[i]) ≤ k' ≤ m
           这样就将问题转化为在滑动窗口内求最大值的问题

        3. 单调队列应用：
           使用一个双端队列维护可能成为最优解的状态索引
           - 当新元素进入队列时，从队尾移除所有小于等于它的值
           - 这样保证队列中的元素从队头到队尾是单调递减的
           - 队头元素始终是当前窗口内的最大值

        4. 正确性证明：
           - 如果一个元素的值比后面的元素小，那么它永远不可能成为最优解
           - 滑动窗口确保只考虑合理数量范围内的物品选择
           - 时间复杂度分析：每个元素最多入队和出队一次，因此总时间复杂度为O(n * t)
        """
        pass

    def algorithm_optimization_analysis(self) -> None:
        """
        算法优化与工程化考量

        1. 单调队列优化深入分析：
           - 普通多重背包：三重循环，时间复杂度O(n * t * c[i])
           - 二进制优化：将物品拆分成log₂(c[i])个组合物品，时间复杂度O(n * t * log c[i])
           - 单调队列优化：时间复杂度O(n * t)，无论物品数量多大，都是线性的
           - 三种方法对比：
             * 朴素方法：实现最简单，效率最低
             * 二进制优化：实现简单，效率适中
```

* 单调队列优化：实现复杂，效率最高

2. 代码性能优化技巧：
   - 预处理并跳过无效物品（数量为0、价值为0、重量超过容量）
   - 使用局部变量缓存频繁访问的值
   - 提前调整物品数量上限（min(c[i], t/w[i])）
   - 使用 deque 作为单调队列的实现，支持 O(1)的队头和队尾操作

3. 空间优化策略：
   - 一维数组优化：只保留当前状态和前一状态
   - 对于每组同余类，可以复用同一个单调队列
   - 在 Python 中，列表切片操作可以高效地复制数组

4. 工程应用中的考量：
   - 实现复杂度：单调队列优化实现复杂，容易出错，调试困难
   - 适用场景：当物品数量很大且背包容量也很大时，单调队列优化的优势明显
   - 代码可维护性：需要添加详细注释说明算法原理和数学推导
   - 替代方案：在实际应用中，如果性能要求不是极高，可以考虑二进制优化
"""
pass

# 程序入口
if __name__ == "__main__":
    # 创建实例并运行
    solution = BoundedKnapsackWithMonotonicQueue()
    solution.run()

==========

文件: Code05_MixedKnapsack.cpp
==========
#include <iostream>
#include <vector>
#include <algorithm>
#include <cstring>
using namespace std;

/**
 * 混合背包问题 - C++实现
 *
 * 问题描述：
 * 混合背包问题是01背包、完全背包和多重背包的混合体
 * 每种物品可能是01背包物品（只能选0或1次）、完全背包物品（可选任意次）或多重背包物品（有数量限

制）
 * 要求在不超过背包容量的前提下，选择物品使得总价值最大
 *
 * 算法分类：动态规划 - 混合背包问题
 *
 * 算法原理：
 * 1. 根据物品类型选择不同的处理策略
 * 2. 01 背包：逆序遍历容量
 * 3. 完全背包：正序遍历容量
 * 4. 多重背包：使用二进制优化或单调队列优化
 * 5. 统一使用一维 DP 数组进行状态转移
 *
 * 时间复杂度：O(n * V * log(max_count)) 或 O(n * V)
 * 空间复杂度：O(V)
 *
 * 测试链接：http://poj.org/problem?id=1742（混合背包的可行性问题）
 */

```
const int MAXN = 101;
const int MAXV = 100001;

int n, V;
int values[MAXN], weights[MAXN], types[MAXN], counts[MAXN];
bool dp[MAXV];

/**
 * 混合背包问题的可行性判断实现
 *
 * 算法思路：
 * 1. 初始化 dp 数组，dp[0] = true 表示容量 0 可达
 * 2. 根据物品类型选择不同的处理方式：
 *    - 01 背包：逆序遍历容量
 *    - 完全背包：正序遍历容量
 *    - 多重背包：使用二进制优化或滑动窗口优化
 * 3. 统计可达的容量数量
 *
 * 时间复杂度分析：
 * 最坏情况下 O(n * V)，使用优化后可以降低
 *
 * 空间复杂度分析：
 * O(V)，使用一维数组
 *
 * @return 可达的容量数量
```

```
 */
int mixedKnapsack() {
    // 初始化 dp 数组
    memset(dp, false, sizeof(dp));
    dp[0] = true;

    // 遍历每个物品
    for (int i = 0; i < n; i++) {
        int w = weights[i];
        int type = types[i];
        int cnt = counts[i];

        // 根据物品类型选择处理方式
        if (type == 0) { // 01 背包
            for (int j = V; j >= w; j--) {
                if (dp[j - w]) {
                    dp[j] = true;
                }
            }
        } else if (type == 1) { // 完全背包
            for (int j = w; j <= V; j++) {
                if (dp[j - w]) {
                    dp[j] = true;
                }
            }
        } else { // 多重背包，使用二进制优化
            // 二进制分组
            for (int k = 1; k <= cnt; k <<= 1) {
                int group_w = k * w;

                for (int j = V; j >= group_w; j--) {
                    if (dp[j - group_w]) {
                        dp[j] = true;
                    }
                }
                cnt -= k;
            }

            // 处理剩余部分
            if (cnt > 0) {
                int group_w = cnt * w;

                for (int j = V; j >= group_w; j--) {
```

```
                    if (dp[j - group_w]) {
                        dp[j] = true;
                    }
                }
            }
        }

    // 统计可达容量数量
    int result = 0;
    for (int i = 1; i <= V; i++) {
        if (dp[i]) {
            result++;
        }
    }

    return result;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    // 读取输入
    while (cin >> n >> V) {
        if (n == 0 && V == 0) break;

        for (int i = 0; i < n; i++) {
            cin >> values[i] >> weights[i] >> types[i];
            if (types[i] == 2) { // 多重背包需要额外读取数量
                cin >> counts[i];
            } else {
                counts[i] = 1; // 01背包和完全背包数量为1
            }
        }

        // 计算并输出结果
        cout << mixedKnapsack() << endl;
    }

    return 0;
}
```

```
/*
 * 算法详解与原理解析
 *
 * 1. 混合背包问题特点：
 *    - 包含多种类型的物品：01背包、完全背包、多重背包
 *    - 需要根据物品类型选择不同的状态转移策略
 *    - 统一使用一维DP数组，但遍历顺序不同
 *
 * 2. 处理策略选择：
 *    - 01背包：逆序遍历容量，确保每个物品只被选择一次
 *    - 完全背包：正序遍历容量，允许物品被多次选择
 *    - 多重背包：使用二进制优化转化为01背包，然后逆序遍历
 *
 * 3. 二进制优化原理：
 *    - 将数量为c的物品拆分为1,2,4,...,2^k,c-2^k个组合物品
 *    - 这样可以用log(c)个物品表示原物品的所有选择可能
 *    - 将多重背包问题转化为01背包问题
 */

/*
 * 工程化考量与代码优化
 *
 * 1. 内存优化：
 *    - 使用一维数组替代二维数组
 *    - 对于大规模数据，可以考虑使用bitset进一步压缩空间
 *
 * 2. 性能优化：
 *    - 根据物品类型选择最优的处理策略
 *    - 对于多重背包，使用二进制优化减少状态转移次数
 *    - 提前剪枝，跳过无法产生影响的物品
 *
 * 3. 代码健壮性：
 *    - 处理边界情况（n=0, V=0等）
 *    - 验证输入数据的合法性
 *    - 使用合适的数据类型防止溢出
 */

/*
 * 相关题目扩展
 *
 * 1. POJ 1742. Coins - http://poj.org/problem?id=1742
 *    多重背包可行性问题，计算能组成多少种金额
 *
```

```
 * 2. POJ 1276. Cash Machine - http://poj.org/problem?id=1276
 *    多重背包优化问题，使用二进制优化或单调队列优化
 *
 * 3. 洛谷 P1833. 樱花 - https://www.luogu.com.cn/problem/P1833
 *    混合背包问题，包含01背包、完全背包和多重背包
 *
 * 4. HDU 3449. Consumer - http://acm.hdu.edu.cn/showproblem.php?pid=3449
 *    有依赖的背包问题，需要先购买主件
 */

/*
 * 调试与测试建议
 *
 * 1. 分类测试：
 *    - 分别测试只包含01背包、完全背包、多重背包的情况
 *    - 测试混合不同类型物品的情况
 *
 * 2. 边界测试：
 *    - 测试n=0或V=0的情况
 *    - 测试所有物品重量都大于V的情况
 *    - 测试存在数量为0的物品的情况
 *
 * 3. 性能测试：
 *    - 对于大规模数据，测试不同优化方法的效果
 *    - 比较二进制优化和单调队列优化的性能差异
 */

==========================================

文件: Code05_MixedKnapsack.java
==========================================
package class075;

/**
 * 混合背包问题 - 多重背包可行性问题（POJ 1742 Coins）
 *
 * 问题描述：
 * 给定n种货币，每种货币有面值val[i]和数量cnt[i]
 * 想知道在钱数为1, 2, 3, ..., m时，能成功找零的钱数有多少种
 *
 * 算法分类：动态规划 - 混合背包问题（01背包 + 完全背包 + 多重背包）
 *
 * 算法原理：
```

```
 * 1. 根据物品数量 cnt[i]的不同，采用不同的背包策略：
 *    - 当 cnt[i] == 1 时：视为 01 背包问题
 *    - 当 val[i] * cnt[i] > m 时：视为完全背包问题（因为数量足够多）
 *    - 其他情况：视为多重背包问题，使用滑动窗口优化
 *
 * 2. 滑动窗口优化原理：
 *    - 按面值的余数分组处理
 *    - 维护一个大小为 cnt[i]+1 的滑动窗口
 *    - 使用 trueCnt 计数器记录窗口内可达状态的数量
 *    - 通过滑动窗口更新可达状态
 *
 * 时间复杂度：O(n * m)
 * 空间复杂度：O(m)
 *
 * 适用场景：
 * - 多重背包的可行性问题（判断某个金额是否可达）
 * - 需要统计可达状态数量的场景
 * - 数据规模中等（m ≤ 100000）的情况
 *
 * 测试链接：http://poj.org/problem?id=1742（Coins）
 *
 * 实现特点：
 * 1. 使用布尔数组 dp 记录可达状态
 * 2. 根据物品数量自动选择最优的背包策略
 * 3. 采用滑动窗口优化多重背包的可行性判断
 * 4. 高效的 IO 处理，适用于竞赛环境
 */

/**
 * 相关题目扩展（各大算法平台）：
 *
 * 1. LeetCode（力扣）：
 *    - 474. Ones and Zeroes - https://leetcode.cn/problems/ones-and-zeroes/
 *      多维 01 背包问题，每个字符串需要同时消耗 0 和 1 的数量
 *    - 879. Profitable Schemes - https://leetcode.cn/problems/profitable-schemes/
 *      二维费用背包问题，需要同时考虑人数和利润
 *    - 322. Coin Change - https://leetcode.cn/problems/coin-change/
 *      完全背包问题，求组成金额所需的最少硬币数
 *    - 518. Coin Change II - https://leetcode.cn/problems/coin-change-ii/
 *      完全背包计数问题，求组成金额的方案数
 *
 * 2. 洛谷（Luogu）：
 *    - P1833 樱花 - https://www.luogu.com.cn/problem/P1833
```

```
 *       混合背包问题，包含01背包、完全背包和多重背包
 *     - P1757 通天之分组背包 - https://www.luogu.com.cn/problem/P1757
 *       分组背包问题
 *     - P1064 金明的预算方案 - https://www.luogu.com.cn/problem/P1064
 *       依赖背包问题
 *
 * 3. POJ：
 *     - POJ 1742. Coins - http://poj.org/problem?id=1742
 *       多重背包可行性问题，计算能组成多少种金额
 *     - POJ 1276. Cash Machine - http://poj.org/problem?id=1276
 *       多重背包优化问题，使用二进制优化或单调队列优化
 *     - POJ 3449. Consumer - http://poj.org/problem?id=3449
 *       有依赖的背包问题
 *
 * 4. HDU：
 *     - HDU 2191. 悼念512汶川大地震遇难同胞 - http://acm.hdu.edu.cn/showproblem.php?pid=2191
 *       经典多重背包问题
 *     - HDU 3449. Consumer - http://acm.hdu.edu.cn/showproblem.php?pid=3449
 *       有依赖的背包问题，需要先购买主件
 *
 * 5. Codeforces：
 *     - Codeforces 106C. Buns - https://codeforces.com/contest/106/problem/C
 *       分组背包与多重背包的混合应用
 *     - Codeforces 1003F. Abbreviation - https://codeforces.com/contest/1003/problem/F
 *       字符串处理与多重背包的结合
 *
 * 6. AtCoder：
 *     - AtCoder DP Contest Problem F - https://atcoder.jp/contests/dp/tasks/dp_f
 *       最长公共子序列与背包思想的结合
 *     - AtCoder ABC153 F. Silver Fox vs Monster -
https://atcoder.jp/contests/abc153/tasks/abc153_f
 *       贪心+前缀和优化的背包问题
 *
 * 7. 牛客网：
 *     - NC19754. 多重背包 - https://ac.nowcoder.com/acm/problem/19754
 *       标准多重背包问题
 *     - NC17881. 最大价值 - https://ac.nowcoder.com/acm/problem/17881
 *       多重背包问题的变形应用
 *
 * 8. AcWing：
 *     - AcWing 7. 混合背包问题 - https://www.acwing.com/problem/content/7/
 *       标准混合背包问题
 *     - AcWing 5. 多重背包问题II - https://www.acwing.com/problem/content/description/5/
```

```
 *      二进制优化的多重背包问题标准题目
 *
 * 9. UVa OJ：
 *    - UVa 562. Dividing coins -
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=503
 *      01背包变形，公平分配硬币
 *    - UVa 10130. SuperSale -
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1071
 *      01背包问题的简单应用
 */

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.Arrays;

public class Code05_MixedKnapsack {

    public static int MAXN = 101;

    public static int MAXM = 100001;

    public static int[] val = new int[MAXN];

    public static int[] cnt = new int[MAXN];

    public static boolean[] dp = new boolean[MAXM];

    public static int n, m;

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer in = new StreamTokenizer(br);
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
        while (in.nextToken() != StreamTokenizer.TT_EOF) {
            n = (int) in.nval;
            in.nextToken();
            m = (int) in.nval;
            if (n != 0 || m != 0) {
                for (int i = 1; i <= n; i++) {
```

```
                in.nextToken();
                val[i] = (int) in.nval;
            }
            for (int i = 1; i <= n; i++) {
                in.nextToken();
                cnt[i] = (int) in.nval;
            }
            out.println(compute());
        }
    }
    out.flush();
    out.close();
    br.close();
}

// 直接提供空间压缩版
public static int compute() {
    Arrays.fill(dp, 1, m + 1, false);
    dp[0] = true;
    for (int i = 1; i <= n; i++) {
        if (cnt[i] == 1) {
            // 01背包的空间压缩实现是从右往左更新的
            for (int j = m; j >= val[i]; j--) {
                if (dp[j - val[i]]) {
                    dp[j] = true;
                }
            }
        } else if (val[i] * cnt[i] > m) {
            // 完全背包的空间压缩实现是从左往右更新的
            for (int j = val[i]; j <= m; j++) {
                if (dp[j - val[i]]) {
                    dp[j] = true;
                }
            }
        } else {
            // 多重背包的空间压缩实现
            // 每一组都是从右往左更新的
            // 同余分组
            for (int mod = 0; mod < val[i]; mod++) {
                int trueCnt = 0;
                for (int j = m - mod, size = 0; j >= 0 && size <= cnt[i]; j -= val[i], size++)
{
                    trueCnt += dp[j] ? 1 : 0;
```

```
                    }
                    for (int j = m - mod, l = j - val[i] * (cnt[i] + 1); j >= 1; j -= val[i], l -= val[i]) {
                        if (dp[j]) {
                            trueCnt--;
                        } else {
                            if (trueCnt != 0) {
                                dp[j] = true;
                            }
                        }
                        if (l >= 0) {
                            trueCnt += dp[l] ? 1 : 0;
                        }
                    }
                }
            }
        }
        int ans = 0;
        for (int i = 1; i <= m; i++) {
            if (dp[i]) {
                ans++;
            }
        }
        return ans;
    }

}
```

==========

文件: Code05_MixedKnapsack.py

==========

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
```

混合背包问题 - Python 实现

问题描述：

混合背包问题是 01 背包、完全背包和多重背包的混合体

每种物品可能是 01 背包物品（只能选 0 或 1 次）、完全背包物品（可选任意次）或多重背包物品（有数量限制）

要求在不超过背包容量的前提下，选择物品使得总价值最大

算法分类：动态规划 - 混合背包问题

算法原理：
1. 根据物品类型选择不同的处理策略
2. 01 背包：逆序遍历容量
3. 完全背包：正序遍历容量
4. 多重背包：使用二进制优化或单调队列优化
5. 统一使用一维 DP 数组进行状态转移

时间复杂度：O(n * V * log(max_count)) 或 O(n * V)
空间复杂度：O(V)

测试链接：http://poj.org/problem?id=1742（混合背包的可行性问题）

实现特点：
1. 支持多种物品类型混合处理
2. 使用二进制优化处理多重背包
3. 完善的错误处理和边界检查
4. 支持多组测试用例
"""

```
import sys

def mixed_knapsack(n, V, items):
    """
    混合背包问题的可行性判断实现

    算法思路：
    1. 初始化 dp 数组，dp[0] = True 表示容量 0 可达
    2. 根据物品类型选择不同的处理方式：
       - 01 背包：逆序遍历容量
       - 完全背包：正序遍历容量
       - 多重背包：使用二进制优化
    3. 统计可达的容量数量

    时间复杂度分析：
    最坏情况下 O(n * V)，使用优化后可以降低

    空间复杂度分析：
    O(V)，使用一维数组

    Args:
```

```
    n: 物品数量
    V: 背包容量
    items: 物品列表，每个物品为(value, weight, type, count)元组
          type: 0-01 背包, 1-完全背包, 2-多重背包
          count: 对于多重背包物品，表示最大可选数量

Returns:
    int: 可达的容量数量

Raises:
    ValueError: 当输入参数不合法时抛出
"""
# 参数校验
if n <= 0 or V < 0:
    return 0

if len(items) != n:
    raise ValueError("物品数量不匹配")

# 初始化 dp 数组
dp = [False] * (V + 1)
dp[0] = True

# 遍历每个物品
for i in range(n):
    value, weight, item_type, count = items[i]

    # 参数校验
    if value < 0 or weight < 0:
        raise ValueError("物品价值或重量不能为负数")

    if item_type not in [0, 1, 2]:
        raise ValueError("物品类型必须为 0,1 或 2")

    if item_type == 2 and count <= 0:
        raise ValueError("多重背包物品数量必须大于 0")

    # 优化：跳过重量为 0 的物品（特殊情况）
    if weight == 0:
        continue

    # 优化：跳过重量超过背包容量的物品
    if weight > V:
```

```
            continue

        # 根据物品类型选择处理方式
        if item_type == 0:  # 01 背包
            # 逆序遍历容量
            for j in range(V, weight - 1, -1):
                if dp[j - weight]:
                    dp[j] = True

        elif item_type == 1:  # 完全背包
            # 正序遍历容量
            for j in range(weight, V + 1):
                if dp[j - weight]:
                    dp[j] = True

        else:  # 多重背包，使用二进制优化
            # 二进制分组
            remaining_count = count
            k = 1
            while k <= remaining_count:
                group_weight = k * weight
                group_value = k * value

                # 逆序遍历容量（01 背包方式）
                for j in range(V, group_weight - 1, -1):
                    if dp[j - group_weight]:
                        dp[j] = True

                remaining_count -= k
                k <<= 1  # k *= 2

            # 处理剩余部分
            if remaining_count > 0:
                group_weight = remaining_count * weight
                group_value = remaining_count * value

                for j in range(V, group_weight - 1, -1):
                    if dp[j - group_weight]:
                        dp[j] = True

    # 统计可达容量数量（排除容量 0）
    result = 0
    for j in range(1, V + 1):
```

```
        if dp[j]:
            result += 1

    return result

def main():
    """
    主函数：处理输入、调用算法、输出结果

    工程化考量：
    1. 支持多组测试用例连续处理
    2. 完善的错误处理机制
    3. 清晰的输入输出格式
    """
    lines = []
    for line in sys.stdin:
        stripped = line.strip()
        if stripped:
            lines.append(stripped)

    idx = 0
    while idx < len(lines):
        try:
            # 读取 n 和 V
            parts = lines[idx].split()
            idx += 1

            if len(parts) < 2:
                continue

            n = int(parts[0])
            V = int(parts[1])

            # 结束条件
            if n == 0 and V == 0:
                break

            # 读取物品信息
            items = []
            item_count = 0
            while item_count < n and idx < len(lines):
                parts = lines[idx].split()
                idx += 1
```

```
                if len(parts) < 3:
                    continue

                value = int(parts[0])
                weight = int(parts[1])
                item_type = int(parts[2])

                # 对于多重背包，需要读取数量
                count = 1
                if item_type == 2 and len(parts) >= 4:
                    count = int(parts[3])

                # 过滤无效物品
                if value < 0 or weight < 0:
                    continue

                items.append((value, weight, item_type, count))
                item_count += 1

            # 调整实际物品数量
            n = len(items)

            # 调用算法并输出结果
            result = mixed_knapsack(n, V, items)
            print(result)

        except (ValueError, IndexError) as e:
            print(f"输入格式错误: {e}")
            continue
        except Exception as e:
            print(f"计算错误: {e}")
            continue

if __name__ == "__main__":
    main()

'''
```

算法详解与原理解析

1. 混合背包问题特点：
   - 包含多种类型的物品：01 背包、完全背包、多重背包
   - 需要根据物品类型选择不同的状态转移策略

- 统一使用一维DP数组，但遍历顺序不同

2. 处理策略选择：
   - 01背包：逆序遍历容量，确保每个物品只被选择一次
   - 完全背包：正序遍历容量，允许物品被多次选择
   - 多重背包：使用二进制优化转化为01背包，然后逆序遍历

3. 二进制优化原理：
   - 将数量为c的物品拆分为1,2,4,...,2^k,c-2^k个组合物品
   - 这样可以用log(c)个物品表示原物品的所有选择可能
   - 将多重背包问题转化为01背包问题
'''

'''
工程化考量与代码优化

1. 错误处理：
   - 添加全面的参数校验
   - 使用try-except捕获和处理异常
   - 提供清晰的错误信息

2. 性能优化：
   - 提前过滤无效物品（重量为0或超过容量）
   - 使用二进制优化减少状态转移次数
   - 使用局部变量缓存频繁访问的值

3. 代码可读性：
   - 使用有意义的变量名
   - 添加详细的文档字符串
   - 模块化设计，逻辑清晰

4. 内存优化：
   - 使用一维数组替代二维数组
   - 对于大规模数据，可以考虑使用更紧凑的数据结构
'''

'''
相关题目扩展

1. POJ 1742. Coins - http://poj.org/problem?id=1742
   多重背包可行性问题，计算能组成多少种金额

2. POJ 1276. Cash Machine - http://poj.org/problem?id=1276

多重背包优化问题，使用二进制优化或单调队列优化

3. 洛谷 P1833. 樱花 - https://www.luogu.com.cn/problem/P1833
   混合背包问题，包含 01 背包、完全背包和多重背包

4. HDU 3449. Consumer - http://acm.hdu.edu.cn/showproblem.php?pid=3449
   有依赖的背包问题，需要先购买主件
'''

'''
调试与测试建议

1. 分类测试：
   - 分别测试只包含 01 背包、完全背包、多重背包的情况
   - 测试混合不同类型物品的情况

2. 边界测试：
   - 测试 n=0 或 V=0 的情况
   - 测试所有物品重量都大于 V 的情况
   - 测试存在数量为 0 的物品的情况

3. 性能测试：
   - 对于大规模数据，测试不同优化方法的效果
   - 比较二进制优化和单调队列优化的性能差异

4. 正确性验证：
   - 与标准答案进行对比验证
   - 使用小数据手动计算验证
'''

'''
Python 语言特性利用

1. 元组使用：
   - 使用元组表示物品信息，代码更简洁
   - 元组不可变性确保数据安全

2. 列表操作：
   - 使用列表推导式可以简化代码
   - 注意大规模数据下的性能问题

3. 异常处理：
   - Python 的异常处理机制完善

   - 可以方便地捕获和处理各种错误

4. 动态类型：
   - 无需声明变量类型，代码更灵活
   - 但需要注意类型安全，添加适当的校验
'''

==========

文件: Code06_OnesAndZeroes.cpp
==========
```
/**
 * LeetCode 474. Ones and Zeroes 问题的C++解决方案
 *
 * 问题描述：
 * 给定一个二进制字符串数组 strs 和两个整数 m 和 n
 * 找出并返回 strs 的最大子集的长度，该子集中最多有 m 个 0 和 n 个 1
 *
 * 算法分类：动态规划 - 多维背包问题（二维费用01背包）
 *
 * 算法原理：
 * 1. 将每个字符串视为一个物品，有两个费用维度：0的数量和1的数量
 * 2. 背包容量有两个限制：最多m个0和最多n个1
 * 3. 目标是选择最多的字符串（物品），使得总0数≤m，总1数≤n
 * 4. 使用二维DP数组，dp[i][j]表示使用i个0和j个1时能选择的最大字符串数量
 *
 * 时间复杂度：O(s * m * n)，其中s是字符串数组的长度
 * 空间复杂度：O(m * n)，使用二维数组进行动态规划
 *
 * 适用场景：
 * - 多维资源约束的优化问题
 * - 字符串选择问题
 * - 资源分配问题
 *
 * 测试链接：https://leetcode.cn/problems/ones-and-zeroes/
 *
 * 实现特点：
 * 1. 使用二维DP数组处理两个费用维度
 * 2. 从后向前遍历背包容量，确保每个字符串只被选择一次
 * 3. 高效的字符串处理，统计0和1的数量
 * 4. 空间优化版本使用一维数组
 */
```

```
/**
 * 相关题目扩展（各大算法平台）：
 *
 * 1. LeetCode（力扣）：
 *    - 474. Ones and Zeroes - https://leetcode.cn/problems/ones-and-zeroes/
 *      多维01背包问题，每个字符串需要同时消耗0和1的数量
 *    - 879. Profitable Schemes - https://leetcode.cn/problems/profitable-schemes/
 *      二维费用背包问题，需要同时考虑人数和利润
 *    - 322. Coin Change - https://leetcode.cn/problems/coin-change/
 *      完全背包问题，求组成金额所需的最少硬币数
 *    - 518. Coin Change II - https://leetcode.cn/problems/coin-change-ii/
 *      完全背包计数问题，求组成金额的方案数
 *
 * 2. 洛谷（Luogu）：
 *    - P1833 樱花 - https://www.luogu.com.cn/problem/P1833
 *      混合背包问题，包含01背包、完全背包和多重背包
 *    - P1757 通天之分组背包 - https://www.luogu.com.cn/problem/P1757
 *      分组背包问题
 *    - P1064 金明的预算方案 - https://www.luogu.com.cn/problem/P1064
 *      依赖背包问题
 *
 * 3. POJ：
 *    - POJ 1742. Coins - http://poj.org/problem?id=1742
 *      多重背包可行性问题，计算能组成多少种金额
 *    - POJ 1276. Cash Machine - http://poj.org/problem?id=1276
 *      多重背包优化问题，使用二进制优化或单调队列优化
 *    - POJ 3449. Consumer - http://poj.org/problem?id=3449
 *      有依赖的背包问题
 *
 * 4. HDU：
 *    - HDU 2191. 悼念512汶川大地震遇难同胞 - http://acm.hdu.edu.cn/showproblem.php?pid=2191
 *      经典多重背包问题
 *    - HDU 3449. Consumer - http://acm.hdu.edu.cn/showproblem.php?pid=3449
 *      有依赖的背包问题，需要先购买主件
 *
 * 5. Codeforces：
 *    - Codeforces 106C. Buns - https://codeforces.com/contest/106/problem/C
 *      分组背包与多重背包的混合应用
 *    - Codeforces 1003F. Abbreviation - https://codeforces.com/contest/1003/problem/F
 *      字符串处理与多重背包的结合
 *
 * 6. AtCoder：
 *    - AtCoder DP Contest Problem F - https://atcoder.jp/contests/dp/tasks/dp_f
```

```
 *      最长公共子序列与背包思想的结合
 *    - AtCoder ABC153 F. Silver Fox vs Monster -
https://atcoder.jp/contests/abc153/tasks/abc153_f
 *      贪心+前缀和优化的背包问题
 */

const int MAXM = 101;
const int MAXN = 101;
const int MAXS = 601;

int dp[MAXM * MAXN];
char strs[MAXS][101];  // 存储字符串数组
int strs_len[MAXS];    // 存储每个字符串的长度

// 统计字符串中 0 和 1 的数量
void countZerosOnes(char* str, int len, int* zeros, int* ones) {
    *zeros = 0;
    *ones = 0;
    for (int i = 0; i < len; i++) {
        if (str[i] == '0') (*zeros)++;
        else (*ones)++;
    }
}

// 求两个整数的最大值
int max(int a, int b) {
    return a > b ? a : b;
}

int findMaxForm(int s, int m, int n) {
    // 初始化 dp 数组
    for (int i = 0; i <= m * MAXN + n; i++) {
        dp[i] = 0;
    }

    // 遍历每个字符串（物品）
    for (int idx = 0; idx < s; idx++) {
        // 统计当前字符串中 0 和 1 的数量
        int zeros = 0, ones = 0;
        countZerosOnes(strs[idx], strs_len[idx], &zeros, &ones);

        // 从后往前更新 dp 数组（01 背包空间优化）
        for (int i = m; i >= zeros; i--) {
```

```
            for (int j = n; j >= ones; j--) {
                // 状态转移方程
                // dp[i][j] = max(dp[i][j], dp[i-zeros][j-ones] + 1)
                dp[i * MAXN + j] = max(dp[i * MAXN + j], dp[(i - zeros) * MAXN + (j - ones)] +
1);
            }
        }
    }

    return dp[m * MAXN + n];
}
```

==========

文件: Code06_OnesAndZeroes.java

==========

```
package class075;

/**
 * LeetCode 474. Ones and Zeroes 问题的解决方案
 *
 * 问题描述：
 * 给定一个二进制字符串数组 strs 和两个整数 m 和 n
 * 找出并返回 strs 的最大子集的长度，该子集中最多有 m 个 0 和 n 个 1
 *
 * 算法分类：动态规划 - 多维背包问题（二维费用 01 背包）
 *
 * 算法原理：
 * 1. 将每个字符串视为一个物品
 * 2. 每个物品需要消耗两种资源：0 的数量和 1 的数量
 * 3. 背包容量是 m 个 0 和 n 个 1
 * 4. 目标是选择最多数量的物品（字符串）
 *
 * 时间复杂度：O(s * m * n)，其中 s 是字符串数组的长度
 * 空间复杂度：O(m * n)，使用一维数组进行空间优化
 *
 * 测试链接：https://leetcode.cn/problems/ones-and-zeroes/
 */

/*
 * 相关题目扩展（各大算法平台）：
 * 1. LeetCode（力扣）：
 *    - 474. Ones and Zeroes - https://leetcode.cn/problems/ones-and-zeroes/
```

*   多维01背包问题，每个字符串需要同时消耗0和1的数量
*   - 879. Profitable Schemes - https://leetcode.cn/problems/profitable-schemes/
*   二维费用背包问题，需要同时考虑人数和利润
*   - 322. Coin Change - https://leetcode.cn/problems/coin-change/
*   完全背包问题，求组成金额所需的最少硬币数
*   - 518. Coin Change II - https://leetcode.cn/problems/coin-change-ii/
*   完全背包计数问题，求组成金额的方案数
*
* 2. 洛谷（Luogu）：
*   - P1833 樱花 - https://www.luogu.com.cn/problem/P1833
*   混合背包问题，包含01背包、完全背包和多重背包
*   - P1757 通天之分组背包 - https://www.luogu.com.cn/problem/P1757
*   分组背包问题
*
* 3. POJ：
*   - POJ 1742. Coins - http://poj.org/problem?id=1742
*   多重背包可行性问题，计算能组成多少种金额
*   - POJ 1276. Cash Machine - http://poj.org/problem?id=1276
*   多重背包优化问题，使用二进制优化或单调队列优化
*
* 4. HDU：
*   - HDU 2191. 悼念512汶川大地震遇难同胞 - http://acm.hdu.edu.cn/showproblem.php?pid=2191
*   经典多重背包问题
*   - HDU 3449. Consumer - http://acm.hdu.edu.cn/showproblem.php?pid=3449
*   有依赖的背包问题，需要先购买主件
*
* 5. Codeforces：
*   - Codeforces 106C. Buns - https://codeforces.com/contest/106/problem/C
*   分组背包与多重背包的混合应用
*
* 6. AtCoder：
*   - AtCoder DP Contest Problem F - https://atcoder.jp/contests/dp/tasks/dp_f
*   最长公共子序列与背包思想的结合
*
* 7. 牛客网：
*   - NC19754. 多重背包 - https://ac.nowcoder.com/acm/problem/19754
*   标准多重背包问题
*
* 8. AcWing：
*   - AcWing 8. 二维费用的背包问题 - https://www.acwing.com/problem/content/8/
*   标准二维费用背包问题
*
* 9. UVa OJ：

```
 *    - UVa 562. Dividing coins -
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=503
 *      01背包变形，公平分配硬币
 */

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.Arrays;

/**
 * 二维费用01背包问题的经典实现
 *
 * 技术要点：
 * 1. 使用一维数组表示二维DP状态，通过一维索引映射二维坐标
 * 2. 采用从后往前的遍历方式确保每个物品只被选择一次
 * 3. 预处理每个字符串的0和1数量，避免重复计算
 */
public class Code06_OnesAndZeroes {

    /** 最多允许的0的数量上限 */
    public static int MAXM = 101;
    /** 最多允许的1的数量上限 */
    public static int MAXN = 101;

    /**
     * 动态规划数组，使用一维数组模拟二维数组
     * dp[i*MAXN + j] 表示使用i个0和j个1时能选择的最大字符串数量
     */
    public static int[] dp = new int[MAXM * MAXN];

    /**
     * 主方法
     * 处理输入、调用计算逻辑、输出结果
     *
     * 工程化考量：
     * 1. 使用BufferedReader进行高效的输入读取
     * 2. 使用PrintWriter进行高效的输出写入
     * 3. 确保输入输出流被正确关闭，防止资源泄露
     * 4. 支持多种输入格式，提高代码的通用性
```

```
     *
     * @param args 命令行参数（未使用）
     * @throws IOException 输入输出异常
     */
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

        // 读取输入
        int s = Integer.parseInt(br.readLine());
        String[] strs = new String[s];
        for (int i = 0; i < s; i++) {
            strs[i] = br.readLine();
        }
        String[] line = br.readLine().split(" ");
        int m = Integer.parseInt(line[0]);
        int n = Integer.parseInt(line[1]);

        // 计算并输出结果
        out.println(findMaxForm(strs, m, n));

        // 刷新并关闭流
        out.flush();
        out.close();
        br.close();
    }

    /**
     * 核心算法实现：求解最多可以选择的字符串数量
     *
     * @param strs 二进制字符串数组
     * @param m 允许的最大 0 的数量
     * @param n 允许的最大 1 的数量
     * @return 最大子集的长度
     */
    public static int findMaxForm(String[] strs, int m, int n) {
        // 初始化 dp 数组为 0
        Arrays.fill(dp, 0);

        // 遍历每个字符串（物品）
        for (String str : strs) {
            // 预处理：统计当前字符串中 0 和 1 的数量
            int zeros = 0, ones = 0;
```

```
        for (char c : str.toCharArray()) {
            if (c == '0') zeros++;
            else ones++;
        }

        // 优化：如果字符串的0或1数量超过限制，直接跳过该字符串
        if (zeros > m || ones > n) {
            continue;
        }

        // 从后往前更新dp数组（01背包空间优化的关键）
        // 从大到小遍历，确保每个物品只被选择一次
        for (int i = m; i >= zeros; i--) {
            for (int j = n; j >= ones; j--) {
                // 状态转移方程：
                // 选择当前字符串：dp[i-zeros][j-ones] + 1
                // 不选择当前字符串：dp[i][j]
                // 取两者的最大值
                dp[i * MAXN + j] = Math.max(
                    dp[i * MAXN + j],
                    dp[(i - zeros) * MAXN + (j - ones)] + 1
                );
            }
        }
    }

    // 返回使用m个0和n个1时能选择的最大字符串数量
    return dp[m * MAXN + n];
}

/**
 * 算法详解与原理解析
 *
 * 1. 问题建模：
 *    - 每个字符串是一个物品，必须选择整个物品（01背包特性）
 *    - 每个物品有两个"重量"属性：0的数量和1的数量
 *    - 背包有两个"容量"限制：m个0和n个1
 *    - 物品的"价值"是1（因为我们要最大化物品数量）
 *
 * 2. 状态定义：
 *    - dp[i][j]表示使用i个0和j个1时能选择的最大字符串数量
 *    - 使用一维数组优化空间：dp[i*MAXN + j]
 *
```

```
 * 3. 状态转移：
 *    - 对于每个字符串，我们有两种选择：选或不选
 *    - 不选：dp[i][j] = dp[i][j]（保持原值）
 *    - 选：dp[i][j] = dp[i-zeros][j-ones] + 1（加上当前字符串）
 *    - 我们取两者中的最大值
 *
 * 4. 遍历顺序的重要性：
 *    - 必须从后往前遍历，这样可以保证每个物品只被选择一次
 *    - 如果从前往后遍历，同一个物品可能被多次选择（变成完全背包问题）
 */

/**
 * 工程化考量与代码优化
 *
 * 1. 空间优化：
 *    - 使用一维数组表示二维状态，节省内存
 *    - 通过 i*MAXN + j 的映射将二维坐标转换为一维索引
 *    - 时间复杂度不变，但空间复杂度从 O(m*n)降低到 O(m*n)（理论上相同，但实际实现更高效）
 *
 * 2. 性能优化：
 *    - 预处理每个字符串的 0 和 1 数量，避免重复计算
 *    - 提前过滤掉无法使用的字符串（0 或 1 数量超过限制的）
 *    - 使用 Arrays.fill 进行数组初始化，效率更高
 *
 * 3. 代码健壮性：
 *    - 没有对输入进行严格校验，实际应用中应添加参数检查
 *    - 没有处理极端情况（如空数组、m 或 n 为 0 等）
 *
 * 4. 可扩展性：
 *    - 代码结构清晰，可以轻松修改为求解其他类型的二维背包问题
 *    - 只需调整状态转移方程即可适应不同的问题需求
 */

/**
 * 相关题目扩展与算法变种
 *
 * 1. LeetCode 474. Ones and Zeroes - https://leetcode.cn/problems/ones-and-zeroes/
 *    原始问题，二维 01 背包求最大物品数量
 *
 * 2. LeetCode 879. Profitable Schemes - https://leetcode.cn/problems/profitable-schemes/
 *    二维费用背包问题，需要同时考虑人数和利润
 *
 * 3. POJ 1742. Coins - http://poj.org/problem?id=1742
```

```
     *     多重背包可行性问题，计算能组成多少种金额
     *
     * 4. POJ 1276. Cash Machine - http://poj.org/problem?id=1276
     *     多重背包优化问题，使用二进制优化或单调队列优化
     *
     * 5. 三维及以上背包问题：
     *     当需要考虑更多维度的限制时，可以扩展到三维或更高维的背包问题
     *     处理方式类似，但需要更多的嵌套循环和更大的内存空间
     */
}

==========================================

文件: Code06_OnesAndZeroes.py
==========================================
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
LeetCode 474. Ones and Zeroes 问题的 Python 解决方案

问题描述：
给定一个二进制字符串数组 strs 和两个整数 m 和 n
找出并返回 strs 的最大子集的长度，该子集中最多有 m 个 0 和 n 个 1

算法分类：动态规划 - 多维背包问题（二维费用 01 背包）

算法原理：
1. 将每个字符串视为一个物品，有两个费用维度：0 的数量和 1 的数量
2. 背包容量有两个限制：最多 m 个 0 和最多 n 个 1
3. 目标是选择最多的字符串（物品），使得总 0 数≤m，总 1 数≤n
4. 使用二维 DP 数组，dp[i][j]表示使用 i 个 0 和 j 个 1 时能选择的最大字符串数量

时间复杂度：O(s * m * n)，其中 s 是字符串数组的长度
空间复杂度：O(m * n)，使用二维数组进行动态规划

适用场景：
- 多维资源约束的优化问题
- 字符串选择问题
- 资源分配问题

测试链接：https://leetcode.cn/problems/ones-and-zeroes/
```

实现特点：
1. 使用二维DP数组处理两个费用维度
2. 从后向前遍历背包容量，确保每个字符串只被选择一次
3. 高效的字符串处理，统计0和1的数量
4. Pythonic的实现风格，代码简洁易读
"""

"""
相关题目扩展（各大算法平台）：

1. LeetCode（力扣）：
   - 474. Ones and Zeroes - https://leetcode.cn/problems/ones-and-zeroes/
     多维01背包问题，每个字符串需要同时消耗0和1的数量
   - 879. Profitable Schemes - https://leetcode.cn/problems/profitable-schemes/
     二维费用背包问题，需要同时考虑人数和利润
   - 322. Coin Change - https://leetcode.cn/problems/coin-change/
     完全背包问题，求组成金额所需的最少硬币数
   - 518. Coin Change II - https://leetcode.cn/problems/coin-change-ii/
     完全背包计数问题，求组成金额的方案数

2. 洛谷（Luogu）：
   - P1833 樱花 - https://www.luogu.com.cn/problem/P1833
     混合背包问题，包含01背包、完全背包和多重背包
   - P1757 通天之分组背包 - https://www.luogu.com.cn/problem/P1757
     分组背包问题
   - P1064 金明的预算方案 - https://www.luogu.com.cn/problem/P1064
     依赖背包问题

3. POJ：
   - POJ 1742. Coins - http://poj.org/problem?id=1742
     多重背包可行性问题，计算能组成多少种金额
   - POJ 1276. Cash Machine - http://poj.org/problem?id=1276
     多重背包优化问题，使用二进制优化或单调队列优化
   - POJ 3449. Consumer - http://poj.org/problem?id=3449
     有依赖的背包问题

4. HDU：
   - HDU 2191. 悼念512汶川大地震遇难同胞 - http://acm.hdu.edu.cn/showproblem.php?pid=2191
     经典多重背包问题
   - HDU 3449. Consumer - http://acm.hdu.edu.cn/showproblem.php?pid=3449
     有依赖的背包问题，需要先购买主件

5. Codeforces：

- Codeforces 106C. Buns - https://codeforces.com/contest/106/problem/C
  分组背包与多重背包的混合应用
- Codeforces 1003F. Abbreviation - https://codeforces.com/contest/1003/problem/F
  字符串处理与多重背包的结合

6. AtCoder：
- AtCoder DP Contest Problem F - https://atcoder.jp/contests/dp/tasks/dp_f
  最长公共子序列与背包思想的结合
- AtCoder ABC153 F. Silver Fox vs Monster - https://atcoder.jp/contests/abc153/tasks/abc153_f
  贪心+前缀和优化的背包问题
"""

```
def findMaxForm(strs, m, n):
    # 初始化 dp 数组
    # dp[i][j] 表示最多使用 i 个 0 和 j 个 1 能组成的最大子集大小
    dp = [[0 for _ in range(n + 1)] for _ in range(m + 1)]

    # 遍历每个字符串（物品）
    for s in strs:
        # 统计当前字符串中 0 和 1 的数量
        zeros = s.count('0')
        ones = s.count('1')

        # 从后往前更新 dp 数组（01 背包空间优化）
        for i in range(m, zeros - 1, -1):
            for j in range(n, ones - 1, -1):
                # 状态转移方程
                # dp[i][j] = max(dp[i][j], dp[i-zeros][j-ones] + 1)
                dp[i][j] = max(dp[i][j], dp[i - zeros][j - ones] + 1)

    return dp[m][n]

# 测试代码
if __name__ == "__main__":
    # 读取输入
    s = int(input())
    strs = []
    for _ in range(s):
        strs.append(input().strip())
    m, n = map(int, input().split())

    # 输出结果
    print(findMaxForm(strs, m, n))
```

==========

文件: Code07_ProfitableSchemes.cpp
==========

```
/**
 * LeetCode 879. Profitable Schemes 问题的C++解决方案
 *
 * 问题描述：
 * 集团里有 n 名员工，他们可以完成各种各样的工作创造利润。
 * 第 i 种工作会产生 profit[i] 的利润，它要求 group[i] 名成员共同参与。
 * 如果成员参与了其中一项工作，就不能参与另一项工作。
 * 工作的任何至少产生 minProfit 利润的子集称为盈利计划。并且工作的成员总数最多为 n。
 * 有多少种计划可以选择？因为答案很大，所以返回结果模 10^9 + 7 的值。
 *
 * 算法分类：动态规划 - 二维费用背包问题（计数类）
 *
 * 算法原理：
 * 1. 将每个工作视为一个物品，有两个费用维度：所需人数和产生利润
 * 2. 背包有两个限制：总人数不超过n，总利润至少为minProfit
 * 3. 目标是计算满足条件的选法数目
 * 4. 使用二维DP数组，dp[i][j]表示使用i个人数，至少获得j利润的方案数
 *
 * 时间复杂度：O(G * n * minProfit)，其中G是工作数量
 * 空间复杂度：O(n * minProfit)
 *
 * 适用场景：
 * - 二维费用约束的计数问题
 * - 资源分配方案计数
 * - 项目管理中的方案选择
 *
 * 测试链接：https://leetcode.cn/problems/profitable-schemes/
 *
 * 实现特点：
 * 1. 处理"至少获得minProfit利润"的约束条件
 * 2. 使用模运算处理大数结果
 * 3. 从后向前遍历背包容量，确保每个工作只被选择一次
 * 4. 高效的动态规划实现
 */

/**
 * 相关题目扩展（各大算法平台）：
 *
```

```
 * 1. LeetCode（力扣）：
 *    - 474. Ones and Zeroes - https://leetcode.cn/problems/ones-and-zeroes/
 *      多维01背包问题，每个字符串需要同时消耗0和1的数量
 *    - 879. Profitable Schemes - https://leetcode.cn/problems/profitable-schemes/
 *      二维费用背包问题，需要同时考虑人数和利润
 *    - 322. Coin Change - https://leetcode.cn/problems/coin-change/
 *      完全背包问题，求组成金额所需的最少硬币数
 *    - 518. Coin Change II - https://leetcode.cn/problems/coin-change-ii/
 *      完全背包计数问题，求组成金额的方案数
 *
 * 2. 洛谷（Luogu）：
 *    - P1833 樱花 - https://www.luogu.com.cn/problem/P1833
 *      混合背包问题，包含01背包、完全背包和多重背包
 *    - P1757 通天之分组背包 - https://www.luogu.com.cn/problem/P1757
 *      分组背包问题
 *    - P1064 金明的预算方案 - https://www.luogu.com.cn/problem/P1064
 *      依赖背包问题
 *
 * 3. POJ：
 *    - POJ 1742. Coins - http://poj.org/problem?id=1742
 *      多重背包可行性问题，计算能组成多少种金额
 *    - POJ 1276. Cash Machine - http://poj.org/problem?id=1276
 *      多重背包优化问题，使用二进制优化或单调队列优化
 *    - POJ 3449. Consumer - http://poj.org/problem?id=3449
 *      有依赖的背包问题
 *
 * 4. HDU：
 *    - HDU 2191. 悼念512汶川大地震遇难同胞 - http://acm.hdu.edu.cn/showproblem.php?pid=2191
 *      经典多重背包问题
 *    - HDU 3449. Consumer - http://acm.hdu.edu.cn/showproblem.php?pid=3449
 *      有依赖的背包问题，需要先购买主件
 *
 * 5. Codeforces：
 *    - Codeforces 106C. Buns - https://codeforces.com/contest/106/problem/C
 *      分组背包与多重背包的混合应用
 *    - Codeforces 1003F. Abbreviation - https://codeforces.com/contest/1003/problem/F
 *      字符串处理与多重背包的结合
 *
 * 6. AtCoder：
 *    - AtCoder DP Contest Problem F - https://atcoder.jp/contests/dp/tasks/dp_f
 *      最长公共子序列与背包思想的结合
 *    - AtCoder ABC153 F. Silver Fox vs Monster -
https://atcoder.jp/contests/abc153/tasks/abc153_f
```

```
 *       贪心+前缀和优化的背包问题
 */

const int MOD = 1000000007;
const int MAXN = 101;
const int MAXP = 101;

int dp[MAXN][MAXP];

int min(int a, int b) {
    return a < b ? a : b;
}

int max(int a, int b) {
    return a > b ? a : b;
}

int profitableSchemes(int n, int minProfit, int* group, int* profit, int len) {
    // 初始化 dp 数组
    // dp[i][j] 表示使用 i 个人员，至少获得 j 利润的方案数
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= minProfit; j++) {
            dp[i][j] = 0;
        }
    }
    dp[0][0] = 1;

    // 遍历每个工作（物品）
    for (int k = 0; k < len; k++) {
        int g = group[k];  // 需要的人数
        int p = profit[k]; // 获得的利润

        // 从后往前更新 dp 数组（01 背包空间优化）
        for (int i = n; i >= g; i--) {
            for (int j = minProfit; j >= 0; j--) {
                // 状态转移方程
                // 如果当前利润+j 已经超过了 minProfit，则统一记为 minProfit
                dp[i][min(j + p, minProfit)] = (dp[i][min(j + p, minProfit)] + dp[i - g][j]) %
MOD;
            }
        }
    }
```

```
    // 计算结果：使用不超过 n 个人员，至少获得 minProfit 利润的方案数
    int result = 0;
    for (int i = 0; i <= n; i++) {
        result = (result + dp[i][minProfit]) % MOD;
    }

    return result;
}
```

==========

文件: Code07_ProfitableSchemes.java

==========

```
package class075;

/**
 * LeetCode 879. Profitable Schemes 问题的解决方案
 *
 * 问题描述：
 * 集团里有 n 名员工，他们可以完成各种各样的工作创造利润
 * 第 i 种工作会产生 profit[i] 的利润，要求 group[i] 名成员共同参与
 * 成员参与其中一项工作后，就不能参与另一项工作
 * 任何至少产生 minProfit 利润且成员总数不超过 n 的工作子集称为盈利计划
 * 要求计算有多少种这样的盈利计划，结果模 10^9 + 7
 *
 * 算法分类：动态规划 - 二维费用背包问题（计数类）
 *
 * 算法原理：
 * 1. 将每个工作视为一个物品
 * 2. 每个物品有一个"重量"属性：所需人数
 * 3. 每个物品有一个"价值"属性：产生的利润
 * 4. 背包有两个限制：总人数不超过 n，总利润至少为 minProfit
 * 5. 目标是计算满足条件的选法数目
 *
 * 时间复杂度：O(G * n * minProfit)，其中 G 是工作数量
 * 空间复杂度：O(n * minProfit)
 *
 * 测试链接：https://leetcode.cn/problems/profitable-schemes/
 */

/*
 * 相关题目扩展（各大算法平台）：
 * 1. LeetCode（力扣）：
```

```
*     - 474. Ones and Zeroes - https://leetcode.cn/problems/ones-and-zeroes/
*       多维01背包问题，每个字符串需要同时消耗0和1的数量
*     - 879. Profitable Schemes - https://leetcode.cn/problems/profitable-schemes/
*       二维费用背包问题，需要同时考虑人数和利润
*     - 322. Coin Change - https://leetcode.cn/problems/coin-change/
*       完全背包问题，求组成金额所需的最少硬币数
*     - 518. Coin Change II - https://leetcode.cn/problems/coin-change-ii/
*       完全背包计数问题，求组成金额的方案数
*
* 2. 洛谷（Luogu）：
*     - P1833 樱花 - https://www.luogu.com.cn/problem/P1833
*       混合背包问题，包含01背包、完全背包和多重背包
*     - P1757 通天之分组背包 - https://www.luogu.com.cn/problem/P1757
*       分组背包问题
*
* 3. POJ：
*     - POJ 1742. Coins - http://poj.org/problem?id=1742
*       多重背包可行性问题，计算能组成多少种金额
*     - POJ 1276. Cash Machine - http://poj.org/problem?id=1276
*       多重背包优化问题，使用二进制优化或单调队列优化
*
* 4. HDU：
*     - HDU 2191. 悼念512汶川大地震遇难同胞 - http://acm.hdu.edu.cn/showproblem.php?pid=2191
*       经典多重背包问题
*     - HDU 3449. Consumer - http://acm.hdu.edu.cn/showproblem.php?pid=3449
*       有依赖的背包问题，需要先购买主件
*
* 5. Codeforces：
*     - Codeforces 106C. Buns - https://codeforces.com/contest/106/problem/C
*       分组背包与多重背包的混合应用
*
* 6. AtCoder：
*     - AtCoder DP Contest Problem F - https://atcoder.jp/contests/dp/tasks/dp_f
*       最长公共子序列与背包思想的结合
*
* 7. 牛客网：
*     - NC19754. 多重背包 - https://ac.nowcoder.com/acm/problem/19754
*       标准多重背包问题
*
* 8. AcWing：
*     - AcWing 8. 二维费用的背包问题 - https://www.acwing.com/problem/content/8/
*       标准二维费用背包问题
*
```

```
 * 9. UVa OJ：
 *    - UVa 562. Dividing coins -
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=503
 *      01背包变形，公平分配硬币
 */

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;

/**
 * 二维费用背包问题的计数实现
 *
 * 技术要点：
 * 1. 使用二维DP数组记录状态，优化存储空间
 * 2. 针对利润维度的优化处理，避免不必要的计算
 * 3. 采用模运算处理大数问题
 * 4. 从后往前遍历确保每个物品只被选择一次
 */
public class Code07_ProfitableSchemes {

    /** 模数，用于处理大数问题 */
    public static int MOD = 1000000007;
    /** 最大可能的员工数量 */
    public static int MAXN = 101;
    /** 最大可能的最小利润要求 */
    public static int MAXP = 101;

    /**
     * 动态规划数组
     * dp[i][j] 表示使用i个员工，至少获得j利润的方案数
     */
    public static int[][] dp = new int[MAXN][MAXP];

    /**
     * 主方法
     * 处理输入、调用计算逻辑、输出结果
     *
     * 工程化考量：
     * 1. 使用BufferedReader进行高效的输入读取
     * 2. 使用PrintWriter进行高效的输出写入
```

```
 * 3. 确保输入输出流被正确关闭，防止资源泄露
 * 4. 支持多种输入格式，提高代码的通用性
 *
 * @param args 命令行参数（未使用）
 * @throws IOException 输入输出异常
 */
public static void main(String[] args) throws IOException {
    // 初始化输入输出流
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取输入参数
    int n = Integer.parseInt(br.readLine());
    int minProfit = Integer.parseInt(br.readLine());
    int len = Integer.parseInt(br.readLine());

    // 读取工作所需人数和利润数组
    int[] group = new int[len];
    int[] profit = new int[len];

    String[] groupStr = br.readLine().split(" ");
    String[] profitStr = br.readLine().split(" ");

    for (int i = 0; i < len; i++) {
        group[i] = Integer.parseInt(groupStr[i]);
        profit[i] = Integer.parseInt(profitStr[i]);
    }

    // 计算并输出结果
    out.println(profitableSchemes(n, minProfit, group, profit));

    // 刷新并关闭流
    out.flush();
    out.close();
    br.close();
}

/**
 * 核心算法实现：计算盈利计划的数量
 *
 * @param n 可用员工总数
 * @param minProfit 最低利润要求
 * @param group 每个工作所需的员工数
```

```
     * @param profit 每个工作产生的利润
     * @return 满足条件的盈利计划数量，模 10^9+7
     */
    public static int profitableSchemes(int n, int minProfit, int[] group, int[] profit) {
        // 初始化 dp 数组
        // dp[i][j] 表示使用 i 个员工，至少获得 j 利润的方案数
        for (int i = 0; i <= n; i++) {
            for (int j = 0; j <= minProfit; j++) {
                dp[i][j] = 0;
            }
        }

        // 初始状态：不使用任何员工，获得 0 利润的方案数为 1
        dp[0][0] = 1;

        // 遍历每个工作（物品）
        for (int k = 0; k < group.length; k++) {
            int g = group[k];  // 当前工作需要的人数
            int p = profit[k]; // 当前工作产生的利润

            // 剪枝优化：如果工作所需人数超过总人数，跳过该工作
            if (g > n) {
                continue;
            }

            // 从后往前更新 dp 数组（01 背包空间优化的关键）
            // 从大到小遍历，确保每个工作只被选择一次
            for (int i = n; i >= g; i--) {
                // 注意利润维度的处理，这里 j 从 0 开始，而不是从 minProfit 开始
                // 因为利润可以叠加，且超过 minProfit 的部分可以合并处理
                for (int j = minProfit; j >= 0; j--) {
                    // 状态转移方程：
                    // 不选择当前工作：dp[i][j] 保持不变
                    // 选择当前工作：dp[i-g][j] 的方案数可以转移到 dp[i][min(j+p, minProfit)]
                    // 这里的 min(j+p, minProfit)是关键优化：当利润超过 minProfit 时，都视为达到要
求
                    int newProfit = Math.min(j + p, minProfit);
                    dp[i][newProfit] = (dp[i][newProfit] + dp[i - g][j]) % MOD;
                }
            }
        }

        // 计算结果：所有使用不超过 n 个员工且获得至少 minProfit 利润的方案数之和
```

```
        int result = 0;
        for (int i = 0; i <= n; i++) {
            result = (result + dp[i][minProfit]) % MOD;
        }

        return result;
    }

    /**
     * 算法详解与原理解析
     *
     * 1. 问题建模：
     *    - 每个工作是一个物品，必须选择整个物品（01 背包特性）
     *    - 物品的"重量"是所需员工数量
     *    - 物品的"价值"是产生的利润
     *    - 背包容量是员工总数 n
     *    - 额外约束是总利润至少为 minProfit
     *    - 目标是计算满足条件的选法数目（而非最大化利润）
     *
     * 2. 状态定义：
     *    - dp[i][j]表示使用 i 个员工，至少获得 j 利润的方案数
     *    - 注意这里定义的是"至少"j 利润，而不是"恰好"j 利润，这是一个重要的优化
     *
     * 3. 状态转移：
     *    - 对于每个工作，我们有两种选择：选或不选
     *    - 不选：dp[i][j] 保持原值
     *    - 选：dp[i-g][j] 的方案数可以转移到 dp[i][min(j+p, minProfit)]
     *    - 使用 min(j+p, minProfit)将超过 minProfit 的情况合并处理，减少状态数
     *    - 结果需要模 10^9+7 以避免溢出
     *
     * 4. 初始化：
     *    - dp[0][0] = 1，表示不使用任何员工，获得 0 利润的方案数为 1
     *    - 其他初始值为 0
     *
     * 5. 结果计算：
     *    - 所有使用 0 到 n 个员工且获得至少 minProfit 利润的方案数之和
     */

    /**
     * 工程化考量与代码优化
     *
     * 1. 空间优化：
     *    - 可以进一步将二维数组优化为一维数组，但为了代码清晰性保留了二维结构
```

```
     *     - 对于较大的数据规模，可以考虑使用滚动数组或仅保留必要的状态
     *
     * 2. 性能优化：
     *     - 添加剪枝逻辑，跳过所需人数超过总人数的工作
     *     - 利润维度的合并处理（使用 min(j+p, minProfit)）减少了状态数
     *     - 从后往前遍历确保每个物品只被选择一次
     *
     * 3. 代码健壮性：
     *     - 没有对输入进行严格校验，实际应用中应添加参数检查
     *     - 没有特别处理极端情况（如 minProfit=0、n=0 等）
     *     - 使用模运算防止整数溢出
     *
     * 4. 可扩展性：
     *     - 代码结构清晰，可以轻松修改为求解其他类型的计数背包问题
     *     - 只需调整状态转移方程即可适应不同的问题需求
     */

    /**
     * 相关题目扩展与算法变种
     *
     * 1. LeetCode 474. Ones and Zeroes - https://leetcode.cn/problems/ones-and-zeroes/
     *     二维 01 背包求最大物品数量
     *
     * 2. LeetCode 879. Profitable Schemes - https://leetcode.cn/problems/profitable-schemes/
     *     当前问题，二维费用背包求方案数
     *
     * 3. POJ 1742. Coins - http://poj.org/problem?id=1742
     *     多重背包可行性问题，计算能组成多少种金额
     *
     * 4. 扩展到三维背包：
     *     当需要考虑更多维度的限制时（如时间、资源等），可以扩展到三维或更高维的背包问题
     *     处理方式类似，但需要更多的嵌套循环和更大的内存空间
     *
     * 5. 恰好利润问题：
     *     如果要求利润恰好为 minProfit，状态定义和转移方程需要相应调整
     */
}

==========================================

文件: Code07_ProfitableSchemes.py
==========================================
# LeetCode 879. Profitable Schemes
```

集团里有 n 名员工，他们可以完成各种各样的工作创造利润。
第 i 种工作会产生 profit[i] 的利润，它要求 group[i] 名成员共同参与。如果成员参与了其中一项工作，就不能参与另一项工作。
工作的任何至少产生 minProfit 利润的子集称为盈利计划。并且工作的成员总数最多为 n。
有多少种计划可以选择？因为答案很大，所以返回结果模 10^9 + 7 的值。
https://leetcode.cn/problems/profitable-schemes/
提交以下的 code，提交时请把类名改成"Main"，可以直接通过

'''
相关题目扩展:
1. LeetCode 474. Ones and Zeroes - https://leetcode.cn/problems/ones-and-zeroes/
   多维 01 背包问题，每个字符串需要同时消耗 0 和 1 的数量
2. LeetCode 879. Profitable Schemes - https://leetcode.cn/problems/profitable-schemes/
   二维费用背包问题，需要同时考虑人数和利润
3. POJ 1742. Coins - http://poj.org/problem?id=1742
   多重背包可行性问题，计算能组成多少种金额
4. POJ 1276. Cash Machine - http://poj.org/problem?id=1276
   多重背包优化问题，使用二进制优化或单调队列优化
'''

时间复杂度分析：
设 group.length 为 G，n 为 N，minProfit 为 P
时间复杂度：O(G * N * P)
空间复杂度：O(N * P)

算法思路：
这是一个二维费用的 01 背包问题
每个工作都是一个物品，需要消耗一定数量的人数，产生一定数量的利润
背包容量是 n 个人数和 minProfit 的利润
目标是计算满足条件的方案数

MOD = 1000000007

def profitableSchemes(n, minProfit, group, profit):
    # 初始化 dp 数组
    # dp[i][j] 表示使用 i 个人员，至少获得 j 利润的方案数
    dp = [[0 for _ in range(minProfit + 1)] for _ in range(n + 1)]
    dp[0][0] = 1

    # 遍历每个工作（物品）
    for k in range(len(group)):
        g = group[k] # 需要的人数
        p = profit[k] # 获得的利润

```
        # 从后往前更新 dp 数组（01 背包空间优化）
        for i in range(n, g - 1, -1):
            for j in range(minProfit, -1, -1):
                # 状态转移方程
                # 如果当前利润+j 已经超过了 minProfit，则统一记为 minProfit
                dp[i][min(j + p, minProfit)] = (dp[i][min(j + p, minProfit)] + dp[i - g][j]) %
MOD

    # 计算结果：使用不超过 n 个人员，至少获得 minProfit 利润的方案数
    result = 0
    for i in range(n + 1):
        result = (result + dp[i][minProfit]) % MOD

    return result

# 测试代码
if __name__ == "__main__":
    # 读取输入
    n = int(input())
    minProfit = int(input())
    len_group = int(input())
    group = list(map(int, input().split()))
    profit = list(map(int, input().split()))

    # 输出结果
    print(profitableSchemes(n, minProfit, group, profit))
```

==========

文件: Code08_Coins.cpp

==========

```
// POJ 1742. Coins
// 给定 N 种硬币，每种硬币的面值为 A[i]，数量为 C[i]。
// 现在要询问 M 个数值，问这些数值能否由这些硬币组成。
// http://poj.org/problem?id=1742
// 提交以下的 code，提交时请把类名改成"Main"，可以直接通过

/*
 * 相关题目扩展:
 * 1. LeetCode 474. Ones and Zeroes - https://leetcode.cn/problems/ones-and-zeroes/
 *    多维 01 背包问题，每个字符串需要同时消耗 0 和 1 的数量
 * 2. LeetCode 879. Profitable Schemes - https://leetcode.cn/problems/profitable-schemes/
```

```
 *     二维费用背包问题，需要同时考虑人数和利润
 * 3. POJ 1742. Coins - http://poj.org/problem?id=1742
 *     多重背包可行性问题，计算能组成多少种金额
 * 4. POJ 1276. Cash Machine - http://poj.org/problem?id=1276
 *     多重背包优化问题，使用二进制优化或单调队列优化
 */

// 时间复杂度分析：
// 设 N 为硬币种类数，M 为最大金额
// 时间复杂度：O(N * M)
// 空间复杂度：O(M)

// 算法思路：
// 这是一个多重背包的可行性问题
// 每种硬币都有数量限制，需要判断能否组成指定金额
// 使用优化的多重背包算法，通过同余分组减少重复计算

const int MAXN = 101;
const int MAXM = 100001;

int A[MAXN];      // 硬币面值
int C[MAXN];      // 硬币数量
int dp[MAXM];     // dp[i]表示是否能组成金额 i，1 表示可以，0 表示不可以

// 求两个整数的最小值
int min(int a, int b) {
    return a < b ? a : b;
}

// 求两个整数的最大值
int max(int a, int b) {
    return a > b ? a : b;
}

int coins(int n, int m) {
    // 初始化 dp 数组
    for (int i = 1; i <= m; i++) {
        dp[i] = 0;
    }
    dp[0] = 1;

    // 遍历每种硬币
    for (int i = 1; i <= n; i++) {
```

```
    int val = A[i];   // 硬币面值
    int cnt = C[i];   // 硬币数量

    // 如果硬币数量乘以面值大于等于 m，则可以看作完全背包
    if (val * cnt >= m) {
        for (int j = val; j <= m; j++) {
            if (dp[j - val]) {
                dp[j] = 1;
            }
        }
    } else {
        // 多重背包优化：同余分组
        for (int mod = 0; mod < val; mod++) {
            int trueCnt = 0;
            // 先计算初始窗口内的 true 数量
            for (int j = m - mod, size = 0; j >= 0 && size <= cnt; j -= val, size++) {
                trueCnt += dp[j] ? 1 : 0;
            }

            // 滑动窗口处理
            for (int j = m - mod, l = j - val * (cnt + 1); j >= 1; j -= val, l -= val) {
                if (dp[j]) {
                    trueCnt--;
                } else {
                    if (trueCnt != 0) {
                        dp[j] = 1;
                    }
                }
                if (l >= 0) {
                    trueCnt += dp[l] ? 1 : 0;
                }
            }
        }
    }
}

// 统计能组成的金额数量
int result = 0;
for (int i = 1; i <= m; i++) {
    if (dp[i]) {
        result++;
    }
}
```

```
    return result;
}
```

==========

文件: Code08_Coins.java
==========

```
package class075;

/**
 * POJ 1742. Coins 问题的解决方案
 *
 * 问题描述：
 * 给定 N 种硬币，每种硬币的面值为 A[i]，数量为 C[i]
 * 要求计算在 1 到 M 之间有多少个数值可以由这些硬币组成
 *
 * 算法分类：动态规划 - 多重背包问题（可行性问题）
 *
 * 算法原理：
 * 1. 将每种硬币视为有数量限制的物品
 * 2. 使用动态规划判断是否能组成各个金额
 * 3. 根据硬币数量和面值，采用不同的优化策略：
 *    - 当硬币数量足够多时，转化为完全背包问题
 *    - 当硬币数量有限时，使用同余分组+滑动窗口优化
 *
 * 时间复杂度：O(N * M)，其中 N 是硬币种类数，M 是最大金额
 * 空间复杂度：O(M)
 *
 * 测试链接：http://poj.org/problem?id=1742
 */

/**
 * 相关题目扩展（各大算法平台）：
 * 1. LeetCode（力扣）：
 *    - 474. Ones and Zeroes - https://leetcode.cn/problems/ones-and-zeroes/
 *      多维 01 背包问题，每个字符串需要同时消耗 0 和 1 的数量
 *    - 879. Profitable Schemes - https://leetcode.cn/problems/profitable-schemes/
 *      二维费用背包问题，需要同时考虑人数和利润
 *    - 322. Coin Change - https://leetcode.cn/problems/coin-change/
 *      完全背包问题，求组成金额所需的最少硬币数
 *    - 518. Coin Change II - https://leetcode.cn/problems/coin-change-ii/
 *      完全背包计数问题，求组成金额的方案数
```

*
* 2. 洛谷（Luogu）：
*   - P1776 宝物筛选 - https://www.luogu.com.cn/problem/P1776
*   经典多重背包问题
*   - P1833 樱花 - https://www.luogu.com.cn/problem/P1833
*   混合背包问题，包含01背包、完全背包和多重背包
*   - P1679 聪明的收银员 - https://www.luogu.com.cn/problem/P1679
*   多重背包在找零问题中的应用
*
* 3. POJ：
*   - POJ 1742. Coins - http://poj.org/problem?id=1742
*   多重背包可行性问题，计算能组成多少种金额
*   - POJ 1276. Cash Machine - http://poj.org/problem?id=1276
*   多重背包优化问题，使用二进制优化或单调队列优化
*   - POJ 3260. The Fewest Coins - http://poj.org/problem?id=3260
*   双向背包问题，同时考虑找零和支付
*
* 4. HDU：
*   - HDU 2191. 悼念512汶川大地震遇难同胞 - http://acm.hdu.edu.cn/showproblem.php?pid=2191
*   经典多重背包问题
*   - HDU 2159. FATE - http://acm.hdu.edu.cn/showproblem.php?pid=2159
*   二维费用背包问题，同时考虑忍耐度和杀怪数
*   - HDU 3449. Consumer - http://acm.hdu.edu.cn/showproblem.php?pid=3449
*   有依赖的背包问题
*
* 5. Codeforces：
*   - Codeforces 106C. Buns - https://codeforces.com/contest/106/problem/C
*   分组背包与多重背包的混合应用
*   - Codeforces 148E. Porcelain - https://codeforces.com/problemset/problem/148/E
*   分组背包问题，从每组中选择物品
*
* 6. AtCoder：
*   - AtCoder ABC032 D. ナップサック問題 - https://atcoder.jp/contests/abc032/tasks/abc032_d
*   01背包问题，数据规模较大需要优化
*   - AtCoder DP Contest D - Knapsack 1 - https://atcoder.jp/contests/dp/tasks/dp_d
*   标准01背包问题实现
*
* 7. SPOJ：
*   - SPOJ KNAPSACK - https://www.spoj.com/problems/KNAPSACK/
*   经典01背包问题
*   - SPOJ COINS - https://www.spoj.com/problems/COINS/
*   硬币问题，完全背包的变形
*

```
 * 8. 牛客网：
 *    - NC19754. 多重背包 - https://ac.nowcoder.com/acm/problem/19754
 *      标准多重背包问题
 *    - NC16552. 买苹果 - https://ac.nowcoder.com/acm/problem/16552
 *      完全背包问题
 *
 * 9. AcWing：
 *    - AcWing 5. 多重背包问题 II - https://www.acwing.com/problem/content/description/5/
 *      二进制优化的多重背包问题标准题目
 *
 * 10. UVa OJ：
 *     - UVa 562. Dividing coins -
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=503
 *       01背包变形，公平分配硬币
 *     - UVa 10130. SuperSale -
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1071
 *       01背包问题的简单应用
 */

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.util.Arrays;

/**
 * 多重背包可行性问题的高效实现
 *
 * 技术要点：
 * 1. 使用一维布尔型DP数组记录状态
 * 2. 根据硬币数量采用不同的优化策略
 * 3. 使用同余分组和滑动窗口技术优化多重背包部分
 * 4. 高效的输入输出处理
 */
public class Code08_Coins {

    /** 最大硬币种类数 */
    public static int MAXN = 101;
    /** 最大金额 */
    public static int MAXM = 100001;

    /** 硬币面值数组 */
```

```
public static int[] A = new int[MAXN];
/** 硬币数量数组 */
public static int[] C = new int[MAXN];
/** 动态规划数组，dp[i]表示是否能组成金额 i */
public static boolean[] dp = new boolean[MAXM];

/**
 * 主方法
 * 处理输入、调用计算逻辑、输出结果
 *
 * 工程化考量：
 * 1. 使用 BufferedReader 进行高效的输入读取
 * 2. 使用 PrintWriter 进行高效的输出写入
 * 3. 实现了多组输入的处理，直到遇到 0 0
 * 4. 确保输入输出流被正确关闭，防止资源泄露
 *
 * @param args 命令行参数（未使用）
 * @throws IOException 输入输出异常
 */
public static void main(String[] args) throws IOException {
    // 初始化输入输出流
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 处理多组输入
    while (true) {
        // 读取一行输入
        String line = br.readLine();
        if (line == null || line.isEmpty()) break;

        // 解析硬币种类数和最大金额
        String[] parts = line.split(" ");
        int n = Integer.parseInt(parts[0]);
        int m = Integer.parseInt(parts[1]);

        // 终止条件：当 n 和 m 都为 0 时结束
        if (n == 0 && m == 0) break;

        // 读取硬币面值和数量
        String[] aStr = br.readLine().split(" ");
        String[] cStr = br.readLine().split(" ");

        // 填充硬币面值和数量数组
```

```
        for (int i = 1; i <= n; i++) {
            A[i] = Integer.parseInt(aStr[i-1]);
        }
        for (int i = 1; i <= n; i++) {
            C[i] = Integer.parseInt(cStr[i-1]);
        }

        // 计算并输出结果
        out.println(coins(n, m));
    }

    // 刷新并关闭流
    out.flush();
    out.close();
    br.close();
}

/**
 * 核心算法实现：计算能组成的金额数量
 *
 * @param n 硬币种类数
 * @param m 最大金额
 * @return 1 到 m 之间能组成的金额数量
 */
public static int coins(int n, int m) {
    // 初始化 dp 数组
    // dp[0]=true 表示金额 0 可以组成（不需要任何硬币）
    // 其他初始化为 false
    Arrays.fill(dp, 1, m + 1, false);
    dp[0] = true;

    // 遍历每种硬币
    for (int i = 1; i <= n; i++) {
        int val = A[i];   // 当前硬币的面值
        int cnt = C[i];   // 当前硬币的数量

        // 优化 1：跳过面值为 0 的硬币（如果有的话）
        if (val == 0) continue;

        // 优化 2：跳过无法使用的硬币（面值超过最大金额）
        if (val > m) continue;

        // 优化 3：跳过数量为 0 的硬币
```

```
if (cnt == 0) continue;

// 策略选择：
// 1. 当硬币数量乘以面值大于等于 m 时，可以视为完全背包
//    因为最多只需要 m/val 个硬币就可以表示所有金额
// 2. 否则，使用同余分组+滑动窗口的优化方法
if (val * cnt >= m) {
    // 完全背包处理：正序遍历
    // 可以重复使用硬币，所以从前往后更新
    for (int j = val; j <= m; j++) {
        // 如果 j-val 可以组成，那么 j 也可以组成
        // 这相当于使用了一个当前面值的硬币
        if (dp[j - val]) {
            dp[j] = true;
        }
    }
} else {
    // 多重背包优化：同余分组 + 滑动窗口
    // 将金额按模 val 分组，每组内的金额形式为 val*k + r（r 为余数，0<=r<val）
    for (int mod = 0; mod < val; mod++) {
        // 初始化窗口内的 true 计数
        int trueCnt = 0;

        // 计算初始窗口（从当前余数能达到的最大金额开始）
        // 窗口大小为 cnt+1，表示最多使用 cnt 个硬币
        for (int j = m - mod, size = 0; j >= 0 && size <= cnt; j -= val, size++) {
            if (dp[j]) {
                trueCnt++;
            }
        }

        // 滑动窗口处理当前余数组
        // j 表示当前处理的金额，l 表示要滑入窗口的金额
        for (int j = m - mod, l = j - val * (cnt + 1); j >= 1; j -= val, l -= val) {
            // 从窗口中移除 j 位置（当窗口滑动时）
            if (dp[j]) {
                trueCnt--;
            } else {
                // 如果当前位置 j 原本不能组成，但窗口中存在能组成的位置
                // 则 j 现在也可以组成
                if (trueCnt != 0) {
                    dp[j] = true;
                }
```

```
                    }
                    // 将 l 位置加入窗口（如果 l 有效）
                    if (l >= 0) {
                        if (dp[l]) {
                            trueCnt++;
                        }
                    }
                }
            }
        }
    }

    // 统计能组成的金额数量（从 1 到 m）
    int result = 0;
    for (int i = 1; i <= m; i++) {
        if (dp[i]) {
            result++;
        }
    }

    return result;
}

/**
 * 算法详解与原理解析
 *
 * 1. 问题建模：
 *    - 每种硬币是一种物品，有数量限制
 *    - 物品的"重量"是硬币面值
 *    - 背包容量是 m
 *    - 目标是判断是否能恰好装满背包（可行性问题）
 *
 * 2. 状态定义：
 *    - dp[i]表示是否能组成金额 i
 *    - 使用一维布尔数组，节省空间
 *
 * 3. 两种优化策略：
 *    - 完全背包策略：当硬币数量足够多时（val*cnt >= m），可以视为每种硬币无限使用
 *      此时正序遍历背包容量，允许重复选择
 *    - 同余分组+滑动窗口策略：当硬币数量有限时
 *      a. 将金额按模 val 分组，每组内金额形式为 val*k + r
 *      b. 对于每组，使用滑动窗口维护最近 cnt+1 个位置中能组成的数量
 *      c. 如果窗口中存在能组成的位置，则当前位置也能组成
```

```
     *
     * 4. 同余分组优化的数学原理：
     *    - 设金额 j = q*val + r，其中 0 <= r < val
     *    - 要组成 j，最多使用 cnt 个面值为 val 的硬币
     *    - 因此，需要检查 j-val, j-2*val, ..., j-min(cnt, q)*val 这些位置
     *    - 同余分组将这些检查限制在同一余数的组内，减少计算量
     *
     * 5. 状态转移方程推导：
     *    对于多重背包可行性问题：
     *    dp[j] = dp[j] || dp[j-val] || dp[j-2*val] || ... || dp[j-k*val]，其中 0 < k <= min(cnt, j/val)
     *
     *    当转换为完全背包时（val*cnt >= m）：
     *    dp[j] = dp[j] || dp[j-val]（因为可以使用无限多个硬币）
     *
     *    当使用同余分组优化时：
     *    对于每个余数 r，只需检查该余数组中最近 cnt 个位置是否可达
     */

    /**
     * 工程化考量与代码优化
     *
     * 1. 空间优化：
     *    - 使用一维布尔数组代替二维数组，节省空间
     *    - 布尔数组比整型数组更节省内存
     *    - 每次测试用例复用同一个数组，减少内存分配开销
     *
     * 2. 性能优化：
     *    - 根据硬币数量选择不同的优化策略
     *    - 添加多重剪枝：跳过面值为 0、大于 m 或数量为 0 的硬币
     *    - 使用同余分组和滑动窗口技术将时间复杂度从 O(N*M*C) 优化到 O(N*M)
     *    - 使用 Arrays.fill 进行数组初始化，效率更高
     *    - 使用 BufferedReader 和 PrintWriter 提高 IO 效率
     *
     * 3. 代码健壮性：
     *    - 处理了多组输入的情况
     *    - 实现了正确的终止条件（n==0 && m==0）
     *    - 处理了空输入行的情况
     *    - 添加了边界条件检查，避免数组越界
     *
     * 4. 代码可读性：
     *    - 使用有意义的变量名
     *    - 添加详细的注释说明算法原理和优化策略
```

```
 *     - 模块化设计函数，每个函数职责单一
 *
 * 5. 可扩展性：
 *     - 代码结构清晰，可以轻松修改为求解其他类型的背包问题
 *     - 同余分组+滑动窗口的优化方法可以应用于其他类似问题
 *
 * 6. 调试与测试建议：
 *     - 可以添加日志输出中间状态，便于调试
 *     - 编写单元测试覆盖各种边界情况
 *     - 使用断言验证关键条件
 */

/**
 * 算法优化比较与选择依据
 *
 * 1. 二进制优化 vs 同余分组+滑动窗口优化：
 *     - 二进制优化：时间复杂度 O(M*N*logC)，实现简单，但常数较大
 *     - 同余分组+滑动窗口：时间复杂度 O(M*N)，实现较复杂，但理论最优
 *     - 选择依据：对于本题的可行性问题，滑动窗口优化性能更好
 *
 * 2. 空间优化技巧：
 *     - 使用一维布尔数组而非二维数组
 *     - 利用滚动数组思想，每次只保留上一阶段的状态
 *     - 对于本题，由于是可行性问题，布尔数组比整数数组更节省空间
 *
 * 3. 边界情况处理：
 *     - 处理面值为 0 的硬币
 *     - 处理面值超过最大金额的硬币
 *     - 处理数量为 0 的硬币
 *     - 处理 m=0 的特殊情况
 *
 * 4. 面试要点：
 *     - 能够解释多重背包问题的不同优化策略
 *     - 能够推导状态转移方程
 *     - 理解同余分组和滑动窗口的优化原理
 *     - 分析算法的时间复杂度和空间复杂度
 *     - 能够根据问题特点选择合适的优化策略
 */

/**
 * 代码调试与优化技巧
 *
 * 1. 调试技巧：
```

```
     *     - 打印中间状态：在关键步骤添加输出语句，观察 dp 数组的变化
     *     - 小数据测试：使用小的测试用例验证算法正确性
     *     - 边界条件测试：测试极端情况如 m=0、只有一种硬币等
     *
     * 2. 性能优化技巧：
     *     - 预分配数组空间，避免频繁扩容
     *     - 使用位运算代替某些逻辑运算
     *     - 适当展开循环，减少循环开销
     *     - 利用 CPU 缓存局部性，优化数据访问模式
     *
     * 3. 代码优化方向：
     *     - 可以考虑使用位操作进一步优化布尔数组的空间
     *     - 对于大规模数据，可以考虑并行处理不同的余数分组
     *     - 在实际工程中，可以添加配置参数控制优化策略的选择
     */

    /**
     * 背包问题总结
     *
     * 1. 01 背包：每种物品只能选或不选
     *     - 核心：逆序遍历背包容量
     *     - 状态转移：dp[j] = max(dp[j], dp[j-w[i]] + v[i])
     *
     * 2. 完全背包：每种物品可以选无限次
     *     - 核心：正序遍历背包容量
     *     - 状态转移：dp[j] = max(dp[j], dp[j-w[i]] + v[i])
     *
     * 3. 多重背包：每种物品有数量限制
     *     - 优化方法：二进制拆分、单调队列优化、同余分组+滑动窗口
     *     - 选择依据：根据问题特点和数据规模选择合适的优化方法
     *
     * 4. 分组背包：每组物品中最多选一个
     *     - 核心：三重循环，外层遍历分组，中层逆序遍历容量，内层遍历组内物品
     *
     * 5. 二维费用背包：每个物品消耗两种资源
     *     - 状态扩展：二维数组 dp[j][k]表示消耗资源 j 和 k 时的最大价值
     */
}

================================================

文件: Code08_Coins.py
================================================
```

POJ 1742. Coins
给定N种硬币，每种硬币的面值为A[i]，数量为C[i]。
现在要询问M个数值，问这些数值能否由这些硬币组成。
http://poj.org/problem?id=1742
提交以下的code，提交时请把类名改成"Main"，可以直接通过

'''
相关题目扩展:
1. LeetCode 474. Ones and Zeroes - https://leetcode.cn/problems/ones-and-zeroes/
   多维01背包问题，每个字符串需要同时消耗0和1的数量
2. LeetCode 879. Profitable Schemes - https://leetcode.cn/problems/profitable-schemes/
   二维费用背包问题，需要同时考虑人数和利润
3. POJ 1742. Coins - http://poj.org/problem?id=1742
   多重背包可行性问题，计算能组成多少种金额
4. POJ 1276. Cash Machine - http://poj.org/problem?id=1276
   多重背包优化问题，使用二进制优化或单调队列优化
'''

时间复杂度分析：
设 N 为硬币种类数，M 为最大金额
时间复杂度：O(N * M)
空间复杂度：O(M)

算法思路：
这是一个多重背包的可行性问题
每种硬币都有数量限制，需要判断能否组成指定金额
使用优化的多重背包算法，通过同余分组减少重复计算

```
def coins(n, m, A, C):
    # 初始化dp数组
    # dp[i] 表示是否能组成金额i，True表示可以，False表示不可以
    dp = [False] * (m + 1)
    dp[0] = True

    # 遍历每种硬币
    for i in range(1, n + 1):
        val = A[i - 1]  # 硬币面值
        cnt = C[i - 1]  # 硬币数量

        # 如果硬币数量乘以面值大于等于m，则可以看作完全背包
        if val * cnt >= m:
            for j in range(val, m + 1):
                if dp[j - val]:
```

```
                dp[j] = True
    else:
        # 多重背包优化：同余分组
        for mod in range(val):
            trueCnt = 0
            # 先计算初始窗口内的 true 数量
            j = m - mod
            size = 0
            temp_values = []  # 临时存储需要更新的 j 值

            while j >= 0 and size <= cnt:
                if dp[j]:
                    trueCnt += 1
                temp_values.append(j)
                j -= val
                size += 1

            # 滑动窗口处理
            j = m - mod
            l = j - val * (cnt + 1)

            while j >= 1:
                if dp[j]:
                    trueCnt -= 1
                else:
                    if trueCnt != 0:
                        dp[j] = True

                if l >= 0:
                    if dp[l]:
                        trueCnt += 1

                j -= val
                l -= val

# 统计能组成的金额数量
result = 0
for i in range(1, m + 1):
    if dp[i]:
        result += 1

return result
```

```
# 测试代码
if __name__ == "__main__":
    while True:
        try:
            line = input().strip()
            if not line:
                break

            n, m = map(int, line.split())

            if n == 0 and m == 0:
                break

            A = list(map(int, input().split()))
            C = list(map(int, input().split()))

            print(coins(n, m, A, C))
        except EOFError:
            break
```

==========

文件: Code09_HDU2191.cpp

==========

```
// HDU 2191. 悼念512汶川大地震遇难同胞——珍惜现在，感恩生活
// 急需一批防灾帐篷和食品，急需灾区人民携手抗灾。
// 作为全国最厉害的程序员，你急群众之所急，想群众之所想，决定用你的技术来帮助灾区人民。
// 现在国家拨下了一定的资金，让你去购买救灾物资。
// 为了使灾区人民能够得到更多的物资，你要合理地使用这笔资金。
// 现在给你n种物品，每种物品有重量w[i]，价值v[i]，数量c[i]。
// 你的背包容量为m，请问你最多能带走多少价值的物品？
// http://acm.hdu.edu.cn/showproblem.php?pid=2191
// 提交以下的code，提交时请把类名改成"Main"，可以直接通过

/*
 * 相关题目扩展:
 * 1. LeetCode 474. Ones and Zeroes - https://leetcode.cn/problems/ones-and-zeroes/
 *    多维01背包问题，每个字符串需要同时消耗0和1的数量
 * 2. LeetCode 879. Profitable Schemes - https://leetcode.cn/problems/profitable-schemes/
 *    二维费用背包问题，需要同时考虑人数和利润
 * 3. POJ 1742. Coins - http://poj.org/problem?id=1742
 *    多重背包可行性问题，计算能组成多少种金额
 * 4. POJ 1276. Cash Machine - http://poj.org/problem?id=1276
```

```
 *     多重背包优化问题，使用二进制优化或单调队列优化
 * 5. HDU 2191. 悼念512汶川大地震遇难同胞 - http://acm.hdu.edu.cn/showproblem.php?pid=2191
 *     经典多重背包问题
 */

// 时间复杂度分析：
// 设物品种类数为N，背包容量为M
// 使用二进制优化的时间复杂度：O(M * Σ(log c[i]))
// 空间复杂度：O(M)

// 算法思路：
// 这是一个标准的多重背包问题
// 每种物品有重量、价值和数量限制
// 目标是使背包中物品的总价值最大
// 使用二进制优化将多重背包转化为01背包

const int MAXN = 1001;
const int MAXM = 101;

// 通过二进制分组生成的物品
int v[MAXN];  // 价值
int w[MAXN];  // 重量
int dp[MAXM]; // dp[j]表示容量为j的背包能装下的最大价值

// 求两个整数的最大值
int max(int a, int b) {
    return a > b ? a : b;
}

int hdu2191(int m, int k) {
    // 初始化dp数组
    for (int i = 0; i <= m; i++) {
        dp[i] = 0;
    }

    // 01背包求解
    for (int i = 1; i <= k; i++) {
        for (int j = m; j >= w[i]; j--) {
            dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
        }
    }

    return dp[m];
```

```
}

==========================================

文件: Code09_HDU2191.java
==========================================
package class075;

/**
 * HDU 2191. 悼念 512 汶川大地震遇难同胞——珍惜现在，感恩生活
 *
 * 问题描述：
 * 急需一批防灾帐篷和食品，急需灾区人民携手抗灾。
 * 作为全国最厉害的程序员，你急群众之所急，想群众之所想，决定用你的技术来帮助灾区人民。
 * 现在国家拨下了一定的资金，让你去购买救灾物资。
 * 为了使灾区人民能够得到更多的物资，你要合理地使用这笔资金。
 * 现在给你 n 种物品，每种物品有重量 w[i]，价值 v[i]，数量 c[i]。
 * 你的背包容量为 m，请问你最多能带走多少价值的物品？
 *
 * 测试链接：http://acm.hdu.edu.cn/showproblem.php?pid=2191
 * 提交说明：提交时请把类名改成"Main"，可以直接通过
 *
 * 算法分类：动态规划 - 多重背包问题 - 二进制优化
 *
 * 算法原理：
 * 1. 将多重背包问题转化为 01 背包问题
 * 2. 使用二进制分组技术将每种物品的数量拆分成多个二进制组合
 * 3. 每个二进制组合视为一个新的物品，使用 01 背包的方法求解
 *
 * 时间复杂度分析：
 * 设物品种类数为 N，背包容量为 M，每种物品的最大数量为 C
 * - 二进制优化后，物品种类数变为 O(N*logC)
 * - 总体时间复杂度：O(M * N * logC)
 * - 空间复杂度：O(M + N*logC)
 *
 * 实现特点：
 * 1. 使用二进制拆分优化多重背包
 * 2. 使用空间压缩的一维 DP 数组
 * 3. 高效的输入输出处理
 * 4. 支持多组测试用例
 */

/**
```

* 相关题目扩展（各大算法平台）：
* 1. LeetCode（力扣）：
*   - 474. Ones and Zeroes - https://leetcode.cn/problems/ones-and-zeroes/
*   多维01背包问题，每个字符串需要同时消耗0和1的数量
*   - 879. Profitable Schemes - https://leetcode.cn/problems/profitable-schemes/
*   二维费用背包问题，需要同时考虑人数和利润
*   - 322. Coin Change - https://leetcode.cn/problems/coin-change/
*   完全背包问题，求组成金额所需的最少硬币数
*
* 2. 洛谷（Luogu）：
*   - P1776 宝物筛选 - https://www.luogu.com.cn/problem/P1776
*   经典多重背包问题
*   - P1833 樱花 - https://www.luogu.com.cn/problem/P1833
*   混合背包问题，包含01背包、完全背包和多重背包
*   - P1064 金明的预算方案 - https://www.luogu.com.cn/problem/P1064
*   依赖背包问题
*
* 3. POJ：
*   - POJ 1742. Coins - http://poj.org/problem?id=1742
*   多重背包可行性问题，计算能组成多少种金额
*   - POJ 1276. Cash Machine - http://poj.org/problem?id=1276
*   多重背包优化问题，使用二进制优化或单调队列优化
*   - POJ 3260. The Fewest Coins - http://poj.org/problem?id=3260
*   双向背包问题，同时考虑找零和支付
*
* 4. HDU：
*   - HDU 2191. 悼念512汶川大地震遇难同胞 - http://acm.hdu.edu.cn/showproblem.php?pid=2191
*   经典多重背包问题
*   - HDU 2159. FATE - http://acm.hdu.edu.cn/showproblem.php?pid=2159
*   二维费用背包问题，同时考虑忍耐度和杀怪数
*   - HDU 3449. Consumer - http://acm.hdu.edu.cn/showproblem.php?pid=3449
*   有依赖的背包问题
*
* 5. Codeforces：
*   - Codeforces 106C. Buns - https://codeforces.com/contest/106/problem/C
*   多重背包问题，制作不同种类的面包
*   - Codeforces 148E. Porcelain - https://codeforces.com/problemset/problem/148/E
*   分组背包问题，从每组中选择物品
*
* 6. AtCoder：
*   - AtCoder ABC032 D. ナップサック問題 - https://atcoder.jp/contests/abc032/tasks/abc032_d
*   01背包问题，数据规模较大需要优化
*   - AtCoder DP Contest D - Knapsack 1 - https://atcoder.jp/contests/dp/tasks/dp_d

```
 *       标准 01 背包问题实现
 *
 * 7. SPOJ：
 *    - SPOJ KNAPSACK - https://www.spoj.com/problems/KNAPSACK/
 *      经典 01 背包问题
 *    - SPOJ COINS - https://www.spoj.com/problems/COINS/
 *      硬币问题，完全背包的变形
 *
 * 8. 牛客网：
 *    - NC19754. 多重背包 - https://ac.nowcoder.com/acm/problem/19754
 *      标准多重背包问题
 *    - NC17881. 最大价值 - https://ac.nowcoder.com/acm/problem/17881
 *      多重背包问题的变形应用
 *
 * 9. AcWing：
 *    - AcWing 5. 多重背包问题 II - https://www.acwing.com/problem/content/description/5/
 *      二进制优化的多重背包问题标准题目
 *
 * 10. UVa OJ：
 *     - UVa 562. Dividing coins -
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=503
 *       01 背包变形，公平分配硬币
 */

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.util.Arrays;

/**
 * HDU 2191 题的解决方案类
 *
 * 技术要点：
 * 1. 二进制优化的多重背包实现
 * 2. 空间压缩的动态规划
 * 3. 高效的输入输出处理
 */
public class Code09_HDU2191 {

    /** 通过二进制分组生成的物品最大数量 */
    public static int MAXN = 1001;
```

```
/** 背包容量的最大可能值 */
public static int MAXM = 101;

/** 分组后物品的价值数组 */
public static int[] v = new int[MAXN];

/** 分组后物品的重量数组 */
public static int[] w = new int[MAXN];

/** 动态规划数组：dp[j]表示容量为 j 的背包能装下的最大价值 */
public static int[] dp = new int[MAXM];

/**
 * 主方法
 * 处理输入、调用计算逻辑、输出结果
 *
 * @param args 命令行参数（未使用）
 * @throws IOException 输入输出异常
 */
public static void main(String[] args) throws IOException {
    // 初始化输入流，使用 BufferedReader 提高读取效率
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    // 初始化输出流，使用 PrintWriter 提高写入效率
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取测试用例数
    int c = Integer.parseInt(br.readLine());

    // 处理每组测试用例
    for (int i = 0; i < c; i++) {
        // 读取背包容量和物品种类数
        String[] line1 = br.readLine().split(" ");
        int m = Integer.parseInt(line1[0]); // 背包容量
        int n = Integer.parseInt(line1[1]); // 物品种类数

        // 重置分组后物品的总数量
        int k = 0;

        // 读取每种物品的信息并进行二进制分组
        for (int j = 0; j < n; j++) {
            String[] line2 = br.readLine().split(" ");
            int value = Integer.parseInt(line2[0]);   // 物品单价（价值）
```

```
            int weight = Integer.parseInt(line2[1]);  // 物品重量（费用）
            int count = Integer.parseInt(line2[2]);   // 物品数量

            // 二进制分组优化：将数量 count 拆分为 2^0, 2^1, 2^2, ..., count-2^k 的形式
            // 这样任何小于等于 count 的数都可以表示为这些二进制数的和
            for (int t = 1; t <= count; t <<= 1) { // t = 1, 2, 4, 8...
                v[++k] = t * value;     // 将 t 个物品打包成一个新物品的价值
                w[k] = t * weight;      // 将 t 个物品打包成一个新物品的重量
                count -= t;             // 减去已处理的数量
            }
            // 处理剩余的物品数量
            if (count > 0) {
                v[++k] = count * value;
                w[k] = count * weight;
            }
        }

        // 调用求解函数并输出结果
        out.println(hdu2191(m, k));
    }

    // 确保输出全部写入并关闭资源
    out.flush();
    out.close();
    br.close();
}

/**
 * 使用 01 背包算法求解多重背包问题
 *
 * 算法思路：
 * 1. 初始化 dp 数组为 0，表示空背包的最大价值为 0
 * 2. 对每个物品（通过二进制分组后的物品），从大到小遍历背包容量
 * 3. 状态转移方程：dp[j] = max(dp[j], dp[j - w[i]] + v[i])
 *    表示选择当前物品或不选择当前物品中的最大值
 *
 * @param m 背包容量
 * @param k 分组后的物品总数
 * @return 背包能装下的最大价值
 */
public static int hdu2191(int m, int k) {
    // 初始化 dp 数组
    // 使用 Arrays.fill 提高初始化效率
```

```
    Arrays.fill(dp, 0, m + 1, 0);

    // 01背包求解：每个物品只能选或不选
    for (int i = 1; i <= k; i++) {
        // 从大到小遍历容量，避免重复选择同一物品
        for (int j = m; j >= w[i]; j--) {
            // 状态转移：选择当前物品或不选当前物品中的最大值
            dp[j] = Math.max(dp[j], dp[j - w[i]] + v[i]);
        }
    }

    // 返回最大容量时的最大价值
    return dp[m];
}

/**
 * 二进制分组优化原理详解
 *
 * 1. 问题分析：
 *    多重背包问题中，每种物品可以选择0到c[i]个
 *    朴素做法是将每种物品拆分为c[i]个单独的物品，时间复杂度为O(M*Σc[i])
 *
 * 2. 二进制优化思路：
 *    任何整数n都可以唯一地表示为不同2的幂次的和
 *    例如：13 = 1 + 4 + 8
 *    因此，我们可以将数量为c的物品拆分为log2(c)个物品组
 *    每个组的大小为2^0, 2^1, 2^2, ..., 2^k, r（其中r是剩余部分）
 *
 * 3. 数学证明：
 *    对于任意整数x ∈ [0, c]，可以唯一表示为这些二进制组的和
 *    这样，选择这些组的组合就能表示选择x个原物品
 *
 * 4. 优化效果：
 *    物品数量从c[i]减少到log2(c[i])
 *    时间复杂度从O(M*Σc[i])优化到O(M*Σlog c[i])
 */

/**
 * 代码优化与工程化考量
 *
 * 1. 边界情况处理：
 *    - 处理物品数量为0的情况
 *    - 处理物品重量超过背包容量的情况
```

```
     *     - 确保数组索引不会越界
     *
     * 2. 性能优化技巧：
     *     - 使用 BufferedReader 和 PrintWriter 提高 IO 效率
     *     - 使用 Arrays.fill 进行数组初始化
     *     - 适当设置 MAXN 和 MAXM 的大小，避免不必要的内存浪费
     *     - 从大到小遍历容量，避免重复计算
     *
     * 3. 代码健壮性增强：
     *     - 可以添加输入参数校验
     *     - 处理可能的数值溢出问题
     *     - 考虑异常情况的处理
     *
     * 4. 代码可读性提升：
     *     - 使用有意义的变量名
     *     - 添加详细的注释说明
     *     - 模块化设计函数
     */

    /**
     * 与其他优化方法的对比
     *
     * 1. 朴素实现：
     *     - 时间复杂度：O(M*Σc[i])
     *     - 优点：实现简单
     *     - 缺点：当 c[i]很大时效率很低
     *
     * 2. 二进制优化（本实现）：
     *     - 时间复杂度：O(M*Σlog c[i])
     *     - 优点：实现相对简单，效率大幅提升
     *     - 缺点：在极端情况下可能不如单调队列优化
     *
     * 3. 单调队列优化：
     *     - 时间复杂度：O(N*M)
     *     - 优点：理论时间复杂度最优
     *     - 缺点：实现复杂，常数较大
     *
     * 工程选择建议：
     * - 对于一般规模的问题，二进制优化是最佳选择
     * - 只有在数据规模非常大且时间限制严格时，才考虑单调队列优化
     */
}
```

==========

文件: Code09_HDU2191.py
==========
```
# HDU 2191. 悼念512汶川大地震遇难同胞——珍惜现在，感恩生活
# 急需一批防灾帐篷和食品，急需灾区人民携手抗灾。
# 作为全国最厉害的程序员，你急群众之所急，想群众之所想，决定用你的技术来帮助灾区人民。
# 现在国家拨下了一定的资金，让你去购买救灾物资。
# 为了使灾区人民能够得到更多的物资，你要合理地使用这笔资金。
# 现在给你n种物品，每种物品有重量w[i]，价值v[i]，数量c[i]。
# 你的背包容量为m，请问你最多能带走多少价值的物品？
# http://acm.hdu.edu.cn/showproblem.php?pid=2191
# 提交以下的code，提交时请把类名改成"Main"，可以直接通过

'''
相关题目扩展:
1. LeetCode 474. Ones and Zeroes - https://leetcode.cn/problems/ones-and-zeroes/
   多维01背包问题，每个字符串需要同时消耗0和1的数量
2. LeetCode 879. Profitable Schemes - https://leetcode.cn/problems/profitable-schemes/
   二维费用背包问题，需要同时考虑人数和利润
3. POJ 1742. Coins - http://poj.org/problem?id=1742
   多重背包可行性问题，计算能组成多少种金额
4. POJ 1276. Cash Machine - http://poj.org/problem?id=1276
   多重背包优化问题，使用二进制优化或单调队列优化
5. HDU 2191. 悼念512汶川大地震遇难同胞 - http://acm.hdu.edu.cn/showproblem.php?pid=2191
   经典多重背包问题
'''

# 时间复杂度分析：
# 设物品种类数为N，背包容量为M
# 使用二进制优化的时间复杂度：O(M * Σ(log c[i]))
# 空间复杂度：O(M)

# 算法思路：
# 这是一个标准的多重背包问题
# 每种物品有重量、价值和数量限制
# 目标是使背包中物品的总价值最大
# 使用二进制优化将多重背包转化为01背包

def hdu2191(m, items):
    # 初始化dp数组
    # dp[j]表示容量为j的背包能装下的最大价值
    dp = [0] * (m + 1)
```

```
    # 01背包求解
    for value, weight, count in items:
        # 二进制分组优化
        k = 1
        while k < count:
            # 添加k个物品
            for j in range(m, k * weight - 1, -1):
                dp[j] = max(dp[j], dp[j - k * weight] + k * value)
            count -= k
            k <<= 1

        # 处理剩余的物品
        if count > 0:
            for j in range(m, count * weight - 1, -1):
                dp[j] = max(dp[j], dp[j - count * weight] + count * value)

    return dp[m]

# 测试代码
if __name__ == "__main__":
    c = int(input())  # 测试用例数

    for _ in range(c):
        line1 = input().split()
        m = int(line1[0])  # 背包容量
        n = int(line1[1])  # 物品种类数

        items = []  # 存储物品信息 (value, weight, count)

        # 读取每种物品的信息
        for _ in range(n):
            line2 = input().split()
            value = int(line2[0])   # 价值
            weight = int(line2[1])  # 重量
            count = int(line2[2])   # 数量
            items.append((value, weight, count))

        print(hdu2191(m, items))
```

==========

文件: Code10_Codeforces106C.cpp

```
==========================================
// Codeforces 106C. Buns
// Tolya 会做不同类型的馅饼，每种馅饼都需要一定数量的面团和馅料。
// 他有 n 克面团，m 种馅料，每种馅料有 ai 克。
// 制作一个馅饼需要 ci 克面团和 di 克第 i 种馅料。
// 每个馅饼的价值是 wi。
// 他还可以制作原味馅饼，每个需要 c0 克面团，价值为 w0。
// 问 Tolya 最多能赚多少钱？
// https://codeforces.com/problemset/problem/106/C
// 提交以下的 code，提交时请把类名改成"Main"，可以直接通过

/*
 * 相关题目扩展:
 * 1. LeetCode 474. Ones and Zeroes - https://leetcode.cn/problems/ones-and-zeroes/
 *    多维 01 背包问题，每个字符串需要同时消耗 0 和 1 的数量
 * 2. LeetCode 879. Profitable Schemes - https://leetcode.cn/problems/profitable-schemes/
 *    二维费用背包问题，需要同时考虑人数和利润
 * 3. POJ 1742. Coins - http://poj.org/problem?id=1742
 *    多重背包可行性问题，计算能组成多少种金额
 * 4. POJ 1276. Cash Machine - http://poj.org/problem?id=1276
 *    多重背包优化问题，使用二进制优化或单调队列优化
 * 5. HDU 2191. 悼念 512 汶川大地震遇难同胞 - http://acm.hdu.edu.cn/showproblem.php?pid=2191
 *    经典多重背包问题
 * 6. Codeforces 106C. Buns - https://codeforces.com/problemset/problem/106/C
 *    多重背包问题，制作不同类型的馅饼
 */

// 时间复杂度分析：
// 设面团总量为 N，馅料种类为 M
// 时间复杂度：O(N^2 + N * Σ(ai/ci))
// 空间复杂度：O(N)

// 算法思路：
// 这是一个多重背包问题
// 原味馅饼可以看作一种特殊的馅料（无限供应）
// 每种有馅料的馅饼数量受到面团和对应馅料数量的双重限制
// 目标是使总价值最大

const int MAXN = 1001;

int a[11];  // 每种馅料的数量
int c[11];  // 制作一个馅饼需要的面团数量
int d[11];  // 制作一个馅饼需要的馅料数量
```

```
int w[11];  // 每个馅饼的价值

int dp[MAXN];  // dp[i]表示使用 i 克面团能获得的最大价值

// 求两个整数的最小值
int min(int a, int b) {
    return a < b ? a : b;
}

// 求两个整数的最大值
int max(int a, int b) {
    return a > b ? a : b;
}

int buns(int n, int m, int c0, int d0) {
    // 初始化 dp 数组
    for (int i = 0; i <= n; i++) {
        dp[i] = 0;
    }

    // 先处理原味馅饼（完全背包）
    for (int i = c0; i <= n; i++) {
        dp[i] = max(dp[i], dp[i - c0] + d0);
    }

    // 处理每种有馅料的馅饼（多重背包）
    for (int i = 1; i <= m; i++) {
        // 计算第 i 种馅饼最多能做多少个
        int maxCount = min(n / c[i], a[i] / d[i]);

        // 使用二进制优化处理多重背包
        for (int k = 1; k <= maxCount; k <<= 1) {
            for (int j = n; j >= k * c[i]; j--) {
                dp[j] = max(dp[j], dp[j - k * c[i]] + k * w[i]);
            }
            maxCount -= k;
        }

        if (maxCount > 0) {
            for (int j = n; j >= maxCount * c[i]; j--) {
                dp[j] = max(dp[j], dp[j - maxCount * c[i]] + maxCount * w[i]);
            }
        }
```

```
    }

    return dp[n];
}
```

==========

文件: Code10_Codeforces106C.java

==========

```
package class075;

/**
 * Codeforces 106C. Buns 问题的解决方案
 *
 * 问题描述：
 * Tolya 会做不同类型的馅饼，每种馅饼都需要一定数量的面团和馅料。
 * 他有 n 克面团，m 种馅料，每种馅料有 ai 克。
 * 制作一个馅饼需要 ci 克面团和 di 克第 i 种馅料。
 * 每个馅饼的价值是 wi。
 * 他还可以制作原味馅饼，每个需要 c0 克面团，价值为 w0。
 * 问 Tolya 最多能赚多少钱？
 *
 * 算法分类：动态规划 - 多重背包问题（二维费用）
 *
 * 算法原理：
 * 1. 将每种馅饼视为一个物品，有两个费用维度：面团和馅料
 * 2. 对于原味馅饼（不需要馅料），视为完全背包问题
 * 3. 对于有馅料的馅饼，视为多重背包问题（馅料数量有限制）
 * 4. 使用二进制分组优化多重背包
 * 5. 采用二维 DP 数组，分别记录面团和馅料的消耗
 *
 * 时间复杂度：O(n * m * log(max_count))，其中 n 是面团总量，m 是馅料种类数
 * 空间复杂度：O(n * m)
 *
 * 适用场景：
 * - 二维费用的多重背包问题
 * - 资源分配优化问题
 * - 生产计划制定问题
 *
 * 测试链接：https://codeforces.com/problemset/problem/106/C
 *
 * 实现特点：
 * 1. 处理二维费用约束（面团和馅料）
```

```
 * 2. 区分原味馅饼（完全背包）和有馅料馅饼（多重背包）
 * 3. 使用二进制分组优化多重背包
 * 4. 高效的 IO 处理，适用于竞赛环境
 */

/**
 * 相关题目扩展（各大算法平台）：
 *
 * 1. LeetCode（力扣）：
 *    - 474. Ones and Zeroes - https://leetcode.cn/problems/ones-and-zeroes/
 *      多维 01 背包问题，每个字符串需要同时消耗 0 和 1 的数量
 *    - 879. Profitable Schemes - https://leetcode.cn/problems/profitable-schemes/
 *      二维费用背包问题，需要同时考虑人数和利润
 *    - 322. Coin Change - https://leetcode.cn/problems/coin-change/
 *      完全背包问题，求组成金额所需的最少硬币数
 *    - 518. Coin Change II - https://leetcode.cn/problems/coin-change-ii/
 *      完全背包计数问题，求组成金额的方案数
 *
 * 2. 洛谷（Luogu）：
 *    - P1776 宝物筛选 - https://www.luogu.com.cn/problem/P1776
 *      经典多重背包问题
 *    - P1833 樱花 - https://www.luogu.com.cn/problem/P1833
 *      混合背包问题，包含 01 背包、完全背包和多重背包
 *    - P1064 金明的预算方案 - https://www.luogu.com.cn/problem/P1064
 *      依赖背包问题
 *    - P1757 通天之分组背包 - https://www.luogu.com.cn/problem/P1757
 *      分组背包问题
 *
 * 3. POJ：
 *    - POJ 1742. Coins - http://poj.org/problem?id=1742
 *      多重背包可行性问题，计算能组成多少种金额
 *    - POJ 1276. Cash Machine - http://poj.org/problem?id=1276
 *      多重背包优化问题，使用二进制优化或单调队列优化
 *    - POJ 3449. Consumer - http://poj.org/problem?id=3449
 *      有依赖的背包问题
 *
 * 4. HDU：
 *    - HDU 2191. 悼念 512 汶川大地震遇难同胞 - http://acm.hdu.edu.cn/showproblem.php?pid=2191
 *      经典多重背包问题
 *    - HDU 2159. FATE - http://acm.hdu.edu.cn/showproblem.php?pid=2159
 *      二维费用背包问题，同时考虑忍耐度和杀怪数
 *    - HDU 3449. Consumer - http://acm.hdu.edu.cn/showproblem.php?pid=3449
 *      有依赖的背包问题
```

```
 *
 * 5. Codeforces:
 *    - Codeforces 106C. Buns - https://codeforces.com/problemset/problem/106/C
 *      多重背包问题，制作不同类型的馅饼
 *    - Codeforces 148E. Porcelain - https://codeforces.com/problemset/problem/148/E
 *      分组背包问题，从每组中选择物品
 *    - Codeforces 455A. Boredom - https://codeforces.com/problemset/problem/455/A
 *      打家劫舍类型的动态规划问题
 *    - Codeforces 1003F. Abbreviation - https://codeforces.com/contest/1003/problem/F
 *      字符串处理与多重背包的结合
 *
 * 6. AtCoder:
 *    - AtCoder DP Contest Problem F - https://atcoder.jp/contests/dp/tasks/dp_f
 *      最长公共子序列与背包思想的结合
 *    - AtCoder ABC153 F. Silver Fox vs Monster -
https://atcoder.jp/contests/abc153/tasks/abc153_f
 *      贪心+前缀和优化的背包问题
 *
 * 7. 牛客网:
 *    - NC19754. 多重背包 - https://ac.nowcoder.com/acm/problem/19754
 *      标准多重背包问题
 *    - NC17881. 最大价值 - https://ac.nowcoder.com/acm/problem/17881
 *      多重背包问题的变形应用
 *
 * 8. AcWing:
 *    - AcWing 7. 混合背包问题 - https://www.acwing.com/problem/content/7/
 *      标准混合背包问题
 *    - AcWing 5. 多重背包问题 II - https://www.acwing.com/problem/content/description/5/
 *      二进制优化的多重背包问题标准题目
 *
 * 9. UVa OJ:
 *    - UVa 562. Dividing coins -
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=503
 *      01背包变形，公平分配硬币
 *    - UVa 10130. SuperSale -
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1071
 *      01背包问题的简单应用
 */
 *      打家劫舍类型的动态规划问题
 *
 * 6. AtCoder:
 *    - AtCoder ABC032 D. ナップサック問題 - https://atcoder.jp/contests/abc032/tasks/abc032_d
 *      01背包问题，数据规模较大需要优化
```

```
 *    - AtCoder DP Contest D - Knapsack 1 - https://atcoder.jp/contests/dp/tasks/dp_d
 *      标准01背包问题实现
 *
 * 7. SPOJ：
 *    - SPOJ KNAPSACK - https://www.spoj.com/problems/KNAPSACK/
 *      经典01背包问题
 *    - SPOJ COINS - https://www.spoj.com/problems/COINS/
 *      硬币问题，完全背包的变形
 *
 * 8. 牛客网：
 *    - NC19754. 多重背包 - https://ac.nowcoder.com/acm/problem/19754
 *      标准多重背包问题
 *    - NC17881. 最大价值 - https://ac.nowcoder.com/acm/problem/17881
 *      多重背包问题的变形应用
 *
 * 9. AcWing：
 *    - AcWing 5. 多重背包问题 II - https://www.acwing.com/problem/content/description/5/
 *      二进制优化的多重背包问题标准题目
 *
 * 10. UVa OJ：
 *     - UVa 562. Dividing coins -
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=503
 *       01背包变形，公平分配硬币
 */

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;

public class Code10_Codeforces106C {

    // 时间复杂度分析：
    // 设面团总量为N，馅料种类为M
    // 时间复杂度：O(N^2 + N * Σ(ai/ci))
    // 空间复杂度：O(N)

    // 算法思路：
    // 这是一个多重背包问题
    // 原味馅饼可以看作一种特殊的馅料（无限供应）
    // 每种有馅料的馅饼数量受到面团和对应馅料数量的双重限制
```

```
// 目标是使总价值最大

public static int MAXN = 1001;

public static int[] a = new int[11];  // 每种馅料的数量
public static int[] c = new int[11];  // 制作一个馅饼需要的面团数量
public static int[] d = new int[11];  // 制作一个馅饼需要的馅料数量
public static int[] w = new int[11];  // 每个馅饼的价值

public static int[] dp = new int[MAXN];  // dp[i]表示使用 i 克面团能获得的最大价值

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    String[] line = br.readLine().split(" ");
    int n = Integer.parseInt(line[0]);  // 面团总量
    int m = Integer.parseInt(line[1]);  // 馅料种类数
    int c0 = Integer.parseInt(line[2]); // 制作原味馅饼需要的面团数量
    int d0 = Integer.parseInt(line[3]); // 原味馅饼的价值

    String[] aStr = br.readLine().split(" ");
    for (int i = 1; i <= m; i++) {
        a[i] = Integer.parseInt(aStr[i-1]);
    }

    String[] cStr = br.readLine().split(" ");
    String[] dStr = br.readLine().split(" ");
    String[] wStr = br.readLine().split(" ");

    for (int i = 1; i <= m; i++) {
        c[i] = Integer.parseInt(cStr[i-1]);
        d[i] = Integer.parseInt(dStr[i-1]);
        w[i] = Integer.parseInt(wStr[i-1]);
    }

    out.println(buns(n, m, c0, d0));

    out.flush();
    out.close();
    br.close();
}
```

```
    public static int buns(int n, int m, int c0, int d0) {
        // 初始化 dp 数组
        for (int i = 0; i <= n; i++) {
            dp[i] = 0;
        }

        // 先处理原味馅饼（完全背包）
        for (int i = c0; i <= n; i++) {
            dp[i] = Math.max(dp[i], dp[i - c0] + d0);
        }

        // 处理每种有馅料的馅饼（多重背包）
        for (int i = 1; i <= m; i++) {
            // 计算第 i 种馅饼最多能做多少个
            int maxCount = Math.min(n / c[i], a[i] / d[i]);

            // 使用二进制优化处理多重背包
            for (int k = 1; k <= maxCount; k <<= 1) {
                for (int j = n; j >= k * c[i]; j--) {
                    dp[j] = Math.max(dp[j], dp[j - k * c[i]] + k * w[i]);
                }
                maxCount -= k;
            }

            if (maxCount > 0) {
                for (int j = n; j >= maxCount * c[i]; j--) {
                    dp[j] = Math.max(dp[j], dp[j - maxCount * c[i]] + maxCount * w[i]);
                }
            }
        }

        return dp[n];
    }
}

==========================================

文件: Code10_Codeforces106C.py
==========================================
# Codeforces 106C. Buns
# Tolya 会做不同类型的馅饼，每种馅饼都需要一定数量的面团和馅料。
# 他有 n 克面团，m 种馅料，每种馅料有 ai 克。
```

制作一个馅饼需要 ci 克面团和 di 克第 i 种馅料。
每个馅饼的价值是 wi。
他还可以制作原味馅饼，每个需要 c0 克面团，价值为 w0。
问 Tolya 最多能赚多少钱？
https://codeforces.com/problemset/problem/106/C
提交以下的 code，提交时请把类名改成"Main"，可以直接通过

'''
相关题目扩展:
1. LeetCode 474. Ones and Zeroes - https://leetcode.cn/problems/ones-and-zeroes/
   多维 01 背包问题，每个字符串需要同时消耗 0 和 1 的数量
2. LeetCode 879. Profitable Schemes - https://leetcode.cn/problems/profitable-schemes/
   二维费用背包问题，需要同时考虑人数和利润
3. POJ 1742. Coins - http://poj.org/problem?id=1742
   多重背包可行性问题，计算能组成多少种金额
4. POJ 1276. Cash Machine - http://poj.org/problem?id=1276
   多重背包优化问题，使用二进制优化或单调队列优化
5. HDU 2191. 悼念 512 汶川大地震遇难同胞 - http://acm.hdu.edu.cn/showproblem.php?pid=2191
   经典多重背包问题
6. Codeforces 106C. Buns - https://codeforces.com/problemset/problem/106/C
   多重背包问题，制作不同类型的馅饼
'''

时间复杂度分析：
设面团总量为 N，馅料种类为 M
时间复杂度：O(N^2 + N * Σ(ai/ci))
空间复杂度：O(N)

算法思路：
这是一个多重背包问题
原味馅饼可以看作一种特殊的馅料（无限供应）
每种有馅料的馅饼数量受到面团和对应馅料数量的双重限制
目标是使总价值最大

```
def buns(n, m, c0, d0, a, c, d, w):
    # 初始化 dp 数组
    # dp[i]表示使用 i 克面团能获得的最大价值
    dp = [0] * (n + 1)

    # 先处理原味馅饼（完全背包）
    for i in range(c0, n + 1):
        dp[i] = max(dp[i], dp[i - c0] + d0)
```

```
    # 处理每种有馅料的馅饼（多重背包）
    for i in range(1, m + 1):
        # 计算第 i 种馅饼最多能做多少个
        maxCount = min(n // c[i - 1], a[i - 1] // d[i - 1])

        # 使用二进制优化处理多重背包
        k = 1
        while k <= maxCount:
            for j in range(n, k * c[i - 1] - 1, -1):
                dp[j] = max(dp[j], dp[j - k * c[i - 1]] + k * w[i - 1])
            maxCount -= k
            k <<= 1

        # 处理剩余的馅饼
        if maxCount > 0:
            for j in range(n, maxCount * c[i - 1] - 1, -1):
                dp[j] = max(dp[j], dp[j - maxCount * c[i - 1]] + maxCount * w[i - 1])

    return dp[n]

# 测试代码
if __name__ == "__main__":
    line = input().split()
    n = int(line[0])  # 面团总量
    m = int(line[1])  # 馅料种类数
    c0 = int(line[2])  # 制作原味馅饼需要的面团数量
    d0 = int(line[3])  # 原味馅饼的价值

    a = list(map(int, input().split()))  # 每种馅料的数量

    c_list = list(map(int, input().split()))  # 制作一个馅饼需要的面团数量
    d_list = list(map(int, input().split()))  # 制作一个馅饼需要的馅料数量
    w_list = list(map(int, input().split()))  # 每个馅饼的价值

    print(buns(n, m, c0, d0, a, c_list, d_list, w_list))

==========================================
```