## 算法设计与分析

**上机报告：最少费用购物问题**

### 问题重述

问题描述：商店中每种商品都有标价。例如，一朵花的价格是 2 元，一个花瓶的价格是5 元。为了吸引顾客，商店提供了一组优惠商品价。优惠商品是把一种或多种商品分成一组，并降价销售。例如， 3 朵花的价格不是 6 元而是 5 元， 2 个花瓶加 1 朵花的优惠价是 10 元。试设计一个算法，计算出某顾客所购商品应付的最少费用。

算法设计：对于给定欲购商品的价格和数量，以及优惠商品价，计算所购商品应付的最少费用。

数据输入：由文件 input.txt 提供欲购商品数据。文件的第 1 行中有 1 个整数 B （ 0 ≦ B ≦ 5 ），表示所购商品种类数。在接下来的 B 行中，每行有 3 个数 C 、 K 和P。 C 表示商品的编码（每种商品有唯一编码），1 ≦ C ≦ 999 ； K 表示购买该种商品总数， 1 ≦ K ≦ 5 ；P是该种商品的正常单价（每件商品的价格 )， 1 ≦ P≦ 999 。注意，一次最多可购买 5 × 5 = 25 件商品。由文件 offer.txt 提供优惠商品价数据。文件的第 1 行中有 1 个整数 S （ 0 ≦ S ≦ 99 ），表示共有 S 种优惠商品组合。接下来的 S 行，每行的第 1 个数描述优惠商品组合中商品的种类数j。接着是j个数字对 (C 、 K), 其中 C 是商品编码，1 ≦ C ≦ 999 ； K 表示该种商品在此组合中的数量， 1 ≦ K ≦ 5 。每行最后一个数字P（1 ≦ P ≦ 9999）表示此商品组合的优惠价。

结果输出：将计算出的所购商品应付的最少费用输出到文件 output.txt 。

### 问题分析

该问题具有最优子结构性质，分析如下：

假若已知前k个所需商品的最少费用Mincost[k]，则前k+1个所需商品的最少费用是以下二者的最小值：

Mincost[k]+Cost[k+1]，即不使用优惠，第k+1件商品单独购买。

使用优惠套餐，检查商品是否满足套餐要求。计算使用套餐的费用。

根据以上思路可以实现代码。

代码中，calculateMinimumCost函数实现数据结构的初始化，cheapestPrice函数实现动态规划，offer函数实现使用套餐的费用的计算。

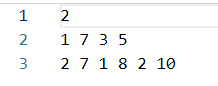
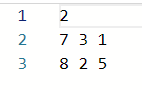
### 代码实现

**import** java.io.File;  
**import** java.io.FileNotFoundException;  
**import** java.io.PrintWriter;  
**import** java.util.\*;  
  
**public class** Main {  
 **public static void** main(String[] args) {  
 **try** {  
 Scanner productScanner = **new** Scanner(**new** File(**"input.txt"**));  
 Scanner offerScanner = **new** Scanner(**new** File(**"offer.txt"**));  
 PrintWriter writer = **new** PrintWriter(**"output.txt"**);  
  
 **int** B = productScanner.nextInt();  
 **int**[][] products = **new int**[B][3];  
  
 **for** (**int** i = 0; i < B; i++) {  
 products[i][0] = productScanner.nextInt();  
 products[i][1] = productScanner.nextInt();  
 products[i][2] = productScanner.nextInt();  
 }  
  
 **int** S = offerScanner.nextInt();  
 **int**[][][] offers = **new int**[S][][];  
 **for** (**int** i = 0; i < S; i++) {  
 **int** j = offerScanner.nextInt();  
 offers[i] = **new int**[j + 1][2];  
 **for** (**int** k = 0; k < j; k++) {  
 offers[i][k][0] = offerScanner.nextInt();  
 offers[i][k][1] = offerScanner.nextInt();  
 }  
 offers[i][j][0] = offerScanner.nextInt();  
 }  
  
 **int** result = *calculateMinimumCost*(B, products, S, offers);  
 writer.println(result);  
  
 productScanner.close();  
 offerScanner.close();  
 writer.close();  
 } **catch** (FileNotFoundException e) {  
 e.printStackTrace();  
 }  
 }  
 **private static int** calculateMinimumCost(**int** B, **int**[][] products, **int** S, **int**[][][] offers) {  
 *// Create data structures to store product and offer information* Map<Integer, Integer> buyList = **new** HashMap<>();  
 Map<Integer, Integer> priceList = **new** HashMap<>();  
 Map<**int**[][], Integer> discount = **new** HashMap<>();  
 List<**int**[][]> offerDetail = **new** LinkedList<>();  
  
 *// Populate buyList and priceList from the input products* **for** (**int** i = 0; i < B; i++) {  
 **int** productId = products[i][0];  
 **int** quantity = products[i][1];  
 **int** price = products[i][2];  
  
 buyList.put(productId, quantity);  
 priceList.put(productId, price);  
 }  
  
 *// Populate offerDetail and discount from the input offers* **for** (**int** i = 0; i < S; i++) {  
 **int** numProductsInOffer = offers[i].**length** - 1;  
 **int**[][] offer = **new int**[numProductsInOffer][2];  
  
 **for** (**int** j = 0; j < numProductsInOffer; j++) {  
 offer[j][0] = offers[i][j][0];  
 offer[j][1] = offers[i][j][1];  
 }  
  
 **int** offerPrice = offers[i][numProductsInOffer][0];  
 offerDetail.add(offer);  
 discount.put(offer, offerPrice);  
 }  
  
 *// Calculate the minimum cost using the provided function* **return** *cheapestPrice*(buyList, priceList, discount, offerDetail);  
 }  
  
 **private static int** cheapestPrice(Map<Integer, Integer> buyList, Map<Integer, Integer> priceList, Map<**int**[][], Integer> discount, List<**int**[][]> offerDetail) {  
 **int** minFee = 0;  
 **int**[] needBuy = buyList.keySet().stream().mapToInt(Integer::intValue).toArray();  
  
 **if** (*isEmpty*(needBuy, buyList)) {  
 **return** minFee;  
 } **else** {  
 minFee += *offer* (buyList, discount, offerDetail);  
 **for** (**int** productId : needBuy) {  
 **if** (buyList.get(productId) > 0) {  
 minFee += buyList.get(productId) \* priceList.get(productId);  
 }  
 }  
 }  
  
 **return** minFee;  
 }  
  
 **private static int** offer (Map<Integer, Integer> buyList, Map<**int**[][], Integer> discount,  
 List<**int**[][]> offerDetail) {  
 **int** val = 0;  
 **for** (**int** i = offerDetail.size() - 1; i >= 0; i--) {  
 **int**[][] tmp = offerDetail.get(i);  
 **int** cnt = 0;  
 **for** (**int** j = 0; j < tmp.**length**; j++) {  
 **if** (buyList.get(tmp[j][0]) >= tmp[j][1]) {  
 cnt++;  
 }  
 }  
 **if** (cnt == tmp.**length**) {  
 **for** (**int** j = 0; j < tmp.**length**; j++) {  
 buyList.put(tmp[j][0], buyList.get(tmp[j][0]) - tmp[j][1]);  
 }  
 val += discount.get(tmp);  
 }  
 }  
 **return** val;  
 }  
  
 **private static boolean** isEmpty(**int**[] arr, Map<Integer, Integer> buyList) {  
 **for** (**int** productId : arr) {  
 **if** (buyList.get(productId) != 0) {  
 **return false**;  
 }  
 }  
 **return true**;  
 }  
}

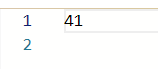
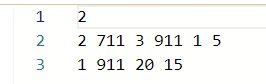
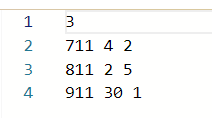
### 运行结果展示

代码运行结果如下：

第一组验证数据：



第二组验证数据：



### 总结

动态规划可以解决最少费用购物问题。但是，本次实现的代码有些混乱，动态规划体现的不清晰，虽然运行结果无误，但还有改进的空间。