算法

Pratice 3

14130130231

吴天成

1. ***0-1 knapsack problem.***

***Instance : weight capacity is 100***

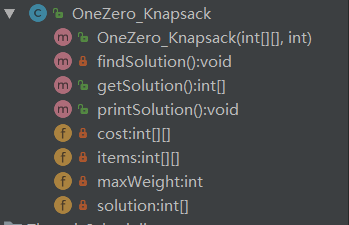
|  |  |  |
| --- | --- | --- |
| ***item*** | ***weights*** | ***values*** |
| ***A*** | ***50*** | ***200*** |
| ***B*** | ***30*** | ***180*** |
| ***C*** | ***45*** | ***225*** |
| ***D*** | ***25*** | ***200*** |
| ***E*** | ***5*** | ***50*** |

01背包问题

即用能装100的背包最多装多少价值的货物，货物必须全部拿走或者全部留下了

此问题不适用与贪心算法，因此我们使用动态规划

OneZero\_Knapsack中方法和变量如下



其中private int maxWeight为背包的最大值

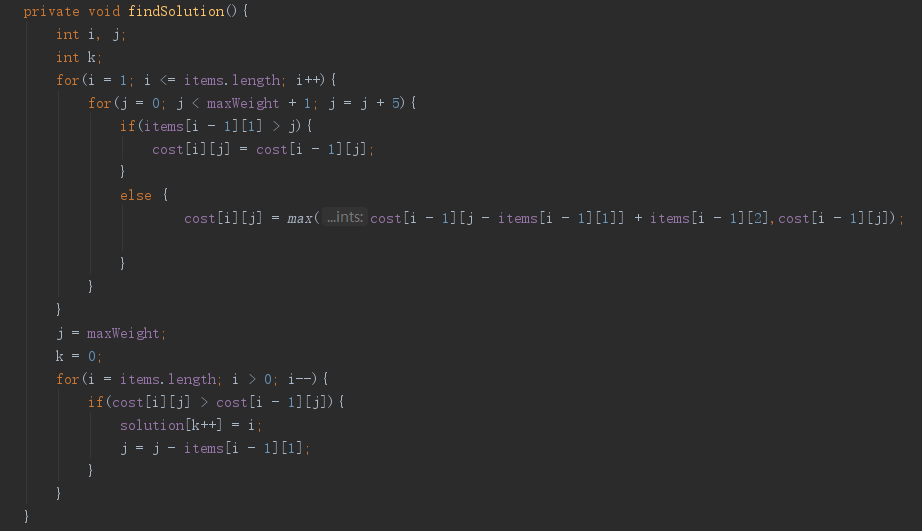
int[ ] [ ]Items为存储物品的名称重量和价值的数组，item[][0]为名称，item[][1]为重量,item[][2]为价值

printSolution用于格式化输出

cost[i][j]代表前i个物品放入容量为j 的背包所能存储物品的最大价值，cost数组的行数和列数均比item的行数和背包最大值大一，因为包含了有0个物品和容量为0的情况。

solution[]数组代表放入最大价值时背包里的物品

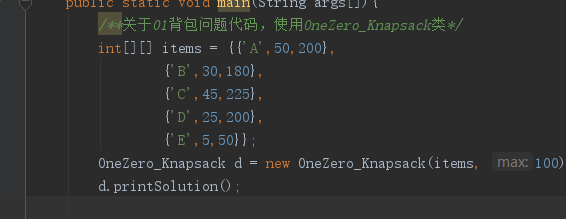
其中核心方法findSolution()代码如下



其中有两重循环，第一重循环i从1到物品数量最大值，第二重循环j从0到背包最大值。在最内层循环比较第i个物品能否装进背包，不能则cost[i][j] = cost[i -1][j]，若可以的话则比较前i-1个物品在j – 第i个物品的重量时能装的最大价值加上第i个物品的价值与前i – 1个在背包容量为j时能装的最大值，将这两个值的最大值赋给cost[i][j]。

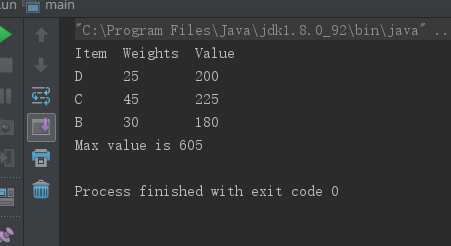
方法中还有一个一重循环，该循环通过比较cost[i][j] 和cost[i -1][j]的值来确定solution数组。

Main函数代码如下



实例化类时要传入一个有关物体名称质量和价值的数组以及背包的最大值

输出结果如下



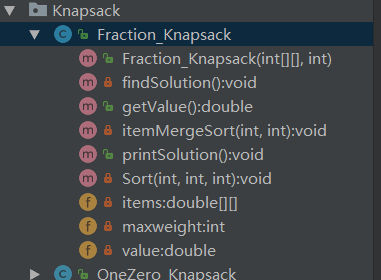
***2． Fractional knapsack problem***

***Instance: same as 1***

分数背包问题，即可以拿走部分的物品

此问题可以采用贪心算法的思路，首先对物品的平均质量价格进行又大到小的排序，接着从大到小将物品放入背包直至背包被装满。

Fraction类结构如下所示



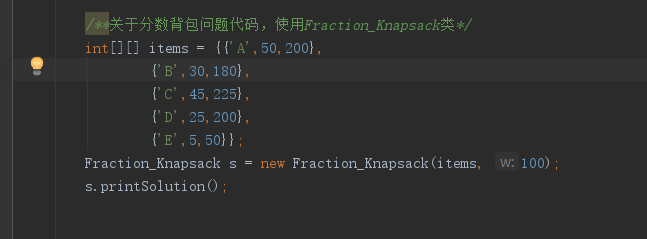
因为需要对物品的平均质量价格进行排序，所以引入了Sort和MergeSort方法来进行归并排序，printSolution用于格式化输出，getValue方法用来得到最大值，其中item类增加了一列来记录物品的平均质量价格以便排序，并且因为可能存在小数，item的数组类型变为double。相比于上一题，取消了cost[i][j]二维数组来记录前i个物品放入容量为j的背包的最大价值。Maxweight代表背包的最大值

核心方法findSolution代码如下

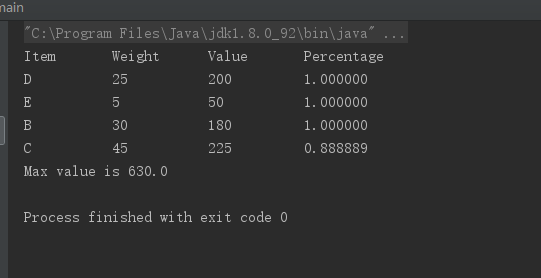


方法中第二个循环表示若第i个物品还能装入背包，则将整个物品装入背包，价值的值加入物品的价值，若装不下去，则装入背包能装入的量，将所装入的比例记在item[i][3]中，价值的值加入物品所加比例的价值。最后可以得到最优价值。

Main函数代码如下



运行结果如下

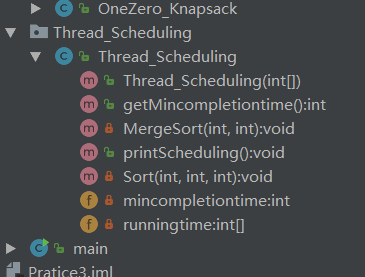


***3． A simple scheduling problem. We are given jobs j1, j2… jn, all with known running time t1, t2… tn, respectively. We have a single processor. What is the best way to schedule these jobs in order to minimize the average completion time. Assume that it is a non-preemptive scheduling: once a job is started, it must run to completion. The following are some instances:***

***(j1, j2, j3, j4) : (15，8，3，10)***

最短平均轮转时间

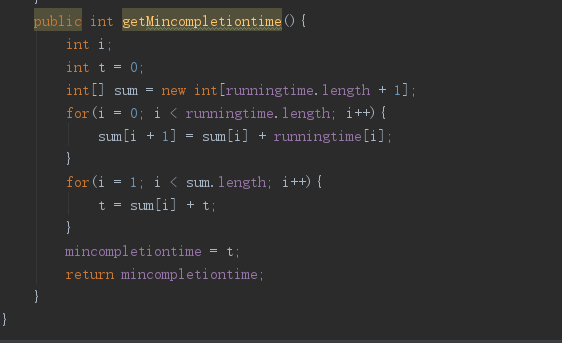
可以得知最短作业时间可得最短平均轮转时间



MergeSort和Sort为归并排序，runningtime为传入的运行时间数组，printScheduling为格式化输出

该类初始化时就调用MergeSort对runningtime进行排序

核心方法getMincompletiontime代码如下



数组Sum[i]为运行到第i个任务时的时间

因为runningtime数组已经有序，该方法从runningtime数值首元素开始计算该元素的completiontime。此时得到的轮转时间最小。

***4． Bin Packing: We have a number of bins each with a capacity of 1, and we have a set of objects all with different seizes, s1,s2,…,sn between 0 and 1. What is the fewest number of bins that would be needed to store all of the objects?***

***Instance: 0.5,0.7,0.3,0.9,0.6,0.8,0.1,0.4,0.2,0.5***

装箱问题，此问题为NP问题，只有approximation（近似）算法。

一个2-approximation的多项式算法。

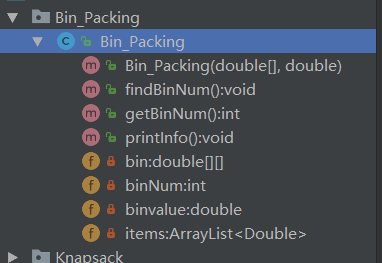
算法：

先打开一个箱子，然后逐一放入物品，如果当前的物品没有一个已打开的箱子可以放入，那么就新打开一个箱子来装它。

算法是2-approximation的，证明：

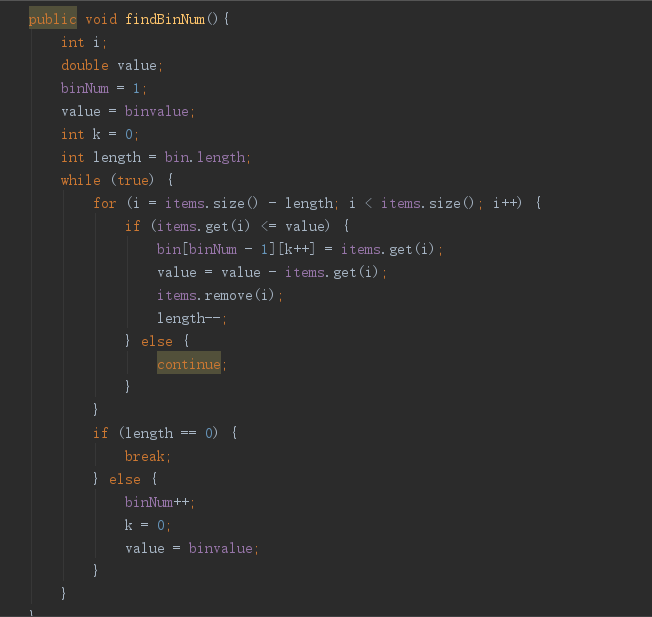
假设最优解需要B\*个箱子，上述算法需要B个。那么，考虑使用上述算法的完成情况，至少会有B-1个箱子是半满的（箱子使用了多余一半的容量），这是因为，如果有两个箱子使用量不足一半，那么根据算法，后一个箱子的物品要被放入第一个箱子，因此上述结论正确。那么全部物品的总量S>(B-1)\*0.5，即，B-1<2\*S，又考虑到B是整数，则B-1<=B，因此有B<=2\*S，而S<=B\*，所以B<=2\*B，因此是2-approximation算法。

Bin\_Packing类如下



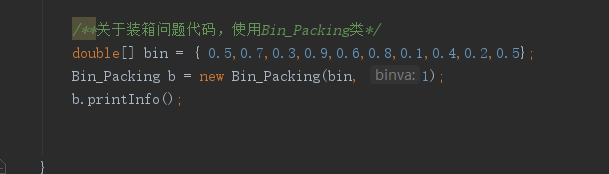
类中有一个double类型的ArrayList，用来存储物品的值，double类型的二维数组bin用来存储装在同一个bin中的物品序号，binvalue为背包的大小，binNum为背包数量。printInfo用于格式化输出，getBinNum用于得到私有变量binNum。

核心方法findBinNum代码如下



Length为剩余物品的数量0，while循环中包括一个for循环，for循环的i从0开始，对第i个物品，若可以仿佛则将此物品从item中去除，箱子的值减去物品的值，并继续寻找剩余物品能否装入。若不能装入，则寻找其他物品能否装入箱子，若不能则打开一个新的箱子。继续开始for循环。直至物品全部装入箱子。

Main函数代码如下



结果如下

