COMP125: Assignment 3

Shopping dilemma

Initial due date: Sunday 20th October 2013, 11pm.
Final due date: Sunday 3rd November 2013, 11pm.

Updates

8th October: Assignment released.

9th October: Some typos fixed.

11th October: Mistake fixed in the template `comp125-assignment3-start.zip`. Namely in the file `Bag.java`, line 191, i.e.
`selection = bag.getMostExpensiveFirstSelection();`
has been replaced by
`selection = bag.getOptimalSelection();`.

Directions

Assignment 3 is independent of assignments 1 and 2.

It has two deliverables and two corresponding deadlines:

Deliverable 1: Provide a "reasonable" implementation of the method `pickMostExpensiveFirst`. This will be assessed automatically by a few tests. If you do not submit a "reasonable" attempt, your final mark will be reduced by 20%, i.e. effectively multiplied by a 0.8 coefficient.

First deadline: Sunday 20th October 2013, 11pm.

Deliverable 2: Provide your final and complete solution.

Final deadline: **Sunday 3rd November 2013, 11pm**.

Note that there will be a testrun based on automated feedback for submissions made before Sunday 27th October 2013, 11pm.

No late submission is allowed.

The maximum mark is 100, if you submit deliverables 1 and 2. This mark will be scaled to 10% for the overall unit assessment.

Contact: `christophe.doche@mq.edu.au`

Assignment Outcomes

In this assignment we will test your ability to:

1. Solve an optimisation problem;
2. Maintain information in ArrayList structures; and
3. Write Java code featuring good programming and in-code documenting style.

Background

Assume that you have a list of $n > 0$ items, each having a value v_i and a weight w_i, as well as a bag which can carry items whose combined weight is $\leqslant W$. The problem is to select items from the list such that their total value is maximised under the constraint that the sum of their weights must be $\leqslant W$. Any item can be picked at most once.

To model this problem, you are given two classes:

- The class `Item` is used to represent generic items having a certain value and weight;
- The class `Bag` is used to determine which items should be picked among a given list of objects of the class `Item`, called `totalListofItems`. You can assume that the list `totalListofItems` always contains at least one item.

We are going to investigate two strategies to solve the problem:

1. Strategy A: pick at any time the most expensive item that is available and that can fit in the bag. The corresponding method, determining which items are picked, is called `pickMostExpensiveFirst`. Unfortunately, this strategy does not always return an optimal selection;
2. Strategy B: efficiently go through all the possible selections of items, in order to determine an optimal one, i.e. a selection of items having maximal value under the constraint that they all fit in the bag. The corresponding method is called `findOptimalSelection`.

See below additional rules to deal with ties.

Your Task

You are provided with a template, namely `comp125-assignment3-start.zip` located in an ilearn folder. This is an Eclipse project containing the outline of the classes definition and a set of JUnit tests.

The project contains four classes:

- The class `Item`. You need to become familiar with it, but you must not modify it;
- The class `Bag`. It contains some incomplete methods;
- The classes `ItemTest` and `BagTest` implementing some tests.

To decide which items should be picked, we use a list of `Boolean`. For instance, if we have four items in total, a list containing `false true true false` means that the items at index 0 and 3 are not selected, whereas the items at index 1 and 2 are picked.

In this spirit, the class `Bag` contains two private members `ArrayList` of type `Boolean`: `mostExpensiveFirstSelection` and `optimalSelection`.

After the method `pickMostExpensiveFirst` executes, the list `mostExpensiveFirstSelection` should be updated with the selection of items according to strategy A.

Similarly, after the method `findOptimalSelection` executes, the list `optimalSelection` should contains a selection of items, i.e. the selection according to strategy B.

Your solution should implement the following rules to deal with ties:

- Regarding `pickMostExpensiveFirst`
  - In case there are two or more candidate items with the same value in the list, then the item with minimal weight should be picked first;
  - In case there are two or more candidate items with the same maximal value and the same minimal weight in the list, then the item with minimal index in the list should be picked first.
- Regarding `optimalSelection`
  - In case there are two or more selections with the same maximal total value, the one with the lowest combined weight should be chosen;

  - In case there are two or more selections with the same maximal total value and the same lowest combined weight, then any selection satisfying these two conditions is fine. A test will check that your method `findOptimalSelection` creates a list `optimalSelection` which gives rise to an optimal value and weight.

The efficiency of `findOptimalSelection` is paramount. Some hard tests will involve large lists and you will have a fixed amount of time (around 1 second) to pass each test. So try to come up with a correct implementation of `findOptimalSelection` first, then you can address its efficiency, and see how you can improve its running time.

You must not modify the type and parameters of `pickMostExpensiveFirst` and `findOptimalSelection`. You may add new methods to the class `Bag` but you must not modify the existing ones.

To summarise, you need to:

1. Insert your name and student ID in the file `Bag.java`.
2. Implement the methods `pickMostExpensiveFirst` and `findOptimalSelection` in the class `Bag` according to the specifications outined in this document. You may add other methods to the class, but do not modify existing ones;
3. Insert meaningful comments, detailing your general approach for each method. You should also include comments where appropriate, for example to explain the prurpose of a complex set of statements. Remember that there is little value in comments like
   `x++; // add one to x`;
4. Analyse the time and space complexity of each method. Provide your analysis in the comments just before the implementation of each method.

Testing Your Solution

We provide a few tests so that you can get started. The automarking of the methods `pickMostExpensiveFirst` and `findOptimalSelection` will be based on a set of `JUnit` tests and you are strongly encouraged to design your own tests.

Submitting Your Work and Marking

You must submit the file `Bag.java` only.

You need to submit a "reasonable" implementation of `pickMostExpensiveFirst` by Sunday 20th October 2013, 11pm. Failure to do so will attract a penalty of 20% on your final mark.

Then, submit your final version of `Bag.java` before Sunday 3rd November 2013, 11pm. Only this last submission will be marked. It will be assessed based on the correctness of your approach, on the quality of your code, documentation and comments that you provide, and on the complexity analysis of your implementations.

The entire assignment will be marked out of 100. These are allocated as follows:

- 25 marks: Number of tests passed by the method `pickMostExpensiveFirst`;
- 25 marks: Number of regular tests passed by the method `findOptimalSelection`;
- 10 marks: Number of hard tests passed by the method `findOptimalSelection`;
- 20 marks: The readability and quality of your Java code;
- 10 marks: Completeness, quality, and readability of the documentation you provide in your code;
- 10 marks: Complexity analysis (time and space) of `pickMostExpensiveFirst` and `findOptimalSelection` in the worst case.