Lab Assignment 5

You are to write a program to use an `ArrayList` with an insertionsort and then to do breakpoint processing for totalling.

Background on the data

First, you have two raw data files labelled

- `mushroom.dat`
- `mushroomsubset.dat`

This is "itemset" data from the web. Itemset processing is the same thing as what is often called "the Walmart problem". Retail stores that sell a large number of items would like to know what sets of items are sold together. The classic statement (which may or may not be true) is that of beer and diapers. Allegedly, if beer is placed in the store near the diapers, then when men are sent to buy diapers, they will think "am I out of beer?", and then make an impulse purchase of beer just to make sure.

The input data is simply a list of numbers, one per line. You might consider these to be the bar code SKU numbers of items being sold, with each line being one retail purchase.

For example, the first line of data in the main `mushroom.dat` file reads

```
1 3 9 13 23 25 34 36 38 40 52 54 59 63 67 76 85 86 90 93 98 107 113
```

and means that items 1, 3, 9, 13, etc., were all bought in one purchase.

Note that the item numbers are always in increasing order in each line. This will be important.

Your lab assignment is

1. to create an insertionsort that will read in the unsorted list (currently named `mushroom.dat`) and output a list sorted on *the first four* item numbers in each line.

2. and to follow that sorted list with the counts of the number of times each of those first-four sequences appear in the file, by doing breakpoint processing on the sorted list.

The current `zout` file should be the correct output for the larger input file. I have given you the shorter version of 20 lines so you can test your code on a small file whose answers you can compute without a computer.

Note that the first four items in any row are all numbers, so your `Scanner` can use `nextInt` to read them, and then you can just use `nextLine` to read all the way to the rest of the line.

Insertionsort

You will need an `ArrayList<OneLine>` for each line, and each `OneLine` entry in that will have five instance variables–four integers and a string. You will need to write your `OneLine` class so as to implement the interface `IOneLine`. The data should be read in the `OneLine` class by a method that returns an instance of the class with the data loaded into the instance.

We have gone over the insertionsort, but it's worth doing again.

1. Read one lines of data into an instance of `OneLine` and add them to the `ArrayList`.
2. Now, until you run out of data, read in an instance of `OneLine` and add it to the (end of the) `ArrayList`. This is the outer loop. Then run an inner loop to pull the `OneLine` instance forward through the `ArrayList` until you find the appropriate position for it.

You might want to create a private method internal to the `Itemset` class that pulled forward one entry in the list until it was in proper place.

The inner loop is thus exactly the inner loop in Chapter 4 and in the powerpoint. The outer loop is a read of data and an add of the data to the end of the `ArrayList`.

DO NOT WORRY ABOUT MAINTAINING THE REST OF THE LIST IN SUBSORTED ORDER. You only need make sure that all the lines are gathered together in blocks by the first four item numbers.

Feel free to steal and reuse driver programs and file utilities programs from previous exercises.

The program you turn in for the lab will use only the `mushroom.dat` data.

Breakpoint Processing

Now that you have the list, sorted by candidate/contest, you can compute totals by candidate/contest by doing breakpoint processing.

In essence, just run down the list, keeping a “last four items read” variable. As long as the current data matches the “last” data, total up counts. When the next entry doesn’t match the “last” data, you have walked into the next block of data. At this point you output the last data line and the total for that data line, reset the “last” variable, and continue.

COMMON GLITCH ONE: You will have to initialize your “last” variable with a dummy value before starting the loop. This means that when you read the first line, there’s a change, and you could—but shouldn’t—write a total for the first dummy value.

COMMON GLITCH TWO: When you run out of data, your loop processing stops. This means that it’s easy to make the mistake of forgetting to output the total you have been accumulating for the very last block of data. This might then require a second output block outside the loop.