Programming Assignment #2

Due Apr 20 by 7am **Points** 20 **Available** Apr 10 at 7am - Jun 7 at 8am about 2 months

Objectives:

- Solve the **subset-sum** problem.
- Define the new class(es): **GroceriesFileReader**, **SubsetSum** class(es)
- Practice using the **java.util.ArrayList** class.
- Test the class **ShoppingBag**
- Get in the habit of dividing larger projects into smaller parts.
- Use EGit features such as commit and clone to write and read from your repository.
- For extra credit test the class **FoothillTunesStore** using your **SubsetSum** class.

Material from:

- Syllabus and Program Guidelines.
- Material from Week 1 modules.
- Class **ShoppingBag**, **FoothillTunesStore** under the **subsets** package, **cs1c** package, and example data files (included under the **resource** folder) in your assigned GitHub repository.

First a note...

Parts I is **required**.

Part II is **optional** and is worth four points extra credit.

Part I.

class ShoppingBag

Write the classes **GroceriesFileReader** and **SubsetSum** such that class **ShoppingBag** and **main()** method will successfully:

- Read the input file that contains the prices of the different items.

- Then given the condition of how much the user's budget is, determines a list of items we can buy.

  We're at a cash only store. So, no purchases via checks or credit! We're working with a specific target in the class **SubsetSum** and **findSubset()** method.

  NOTE: Your static **findSubset()** method is static should be *self-contained*. This means that it should *not* depend on a **static** or class field. If your implementation relies on these, then these class fields should be *initialized* every single time the user call the **static** methods.

- Finally, captures and outputs the estimated run time of your implementation and reports it back.

The format of the input file is:

```
bananas,2.50
beans,4
beef,11.50
chicken,7
fish,15
juice,4
milk,6
ramen,8
```

Output of example (with input from standard in shown in **bold**):

```
The list of groceries has 8 items.
Groceries wanted:
[2.5, 4.0, 11.5, 7.0, 15.0, 4.0, 6.0, 8.0]

Enter your budget:
20

Algorithm Elapsed Time: 0 hrs : 0 mins : 0 sec : 1 ms : 944393 ns
Purchased grocery prices are:
[2.5, 11.5, 6.0]
Done with ShoppingBag.
```

NOTE: Regarding result...
The items in the sub-list L' appear in the order that it appears in the set S (i.e. groceries wanted):

```
[2.5,11.5,6.0]
```

Also, note there maybe more than one correct answer. However the sub-list(s) which have the closest sum to the target are

correct.

Your output should cover these three **test cases**:

- When **budget** is **>** sum of all elements.
- When **budget** is **<** sum of all elements and a subset of elements is found with a total that has an exact match our budget amount.
- When **budget** is **<** sum of all elements and there is no exact match. So, we return a subset of elements with closest match.

Clearly distinguish the different test cases.

Suggestion: Use the following template for your RUN.txt file:

```
--------------------------------------------------
--------------------------------------------------
Test file: resources/inputFile01.txt
budget: 2000

NOTES: Testing target budget > sum of all elements.
--------------------------------------------------

[Paste your output here.]


--------------------------------------------------
--------------------------------------------------
Test file: resources/inputFile02.txt
budget: 17.5

NOTES: Testing set of elements found with sums
       to exactly to target budget.
--------------------------------------------------
```

Part II.

class FoothillTunesStore

Use your implementation of class **SubsetSum** such that class **FoothillTunesStore** and **main()** method will successfully:

- Creates an object of type **MillionSongSubset** which parses the JSON file provided.
  NOTE: This has already been done for you.

- Then given the condition of how much user's play list target duration is, determines a list of songs to add to the play list.

- Finally, captures and outputs the estimated run time of your implementation and reports it back.

The format of the input file is:

```
{ "songs" : [
    {
        "genre": "classic pop and rock",
        "artist_name": "Blue Oyster Cult",
        "title": "Mes Dames Sarat",
        "duration": "246.33424",
        "song_id": "000001"
    },
    {
        "genre": "classic pop and rock",
        "artist_name": "Blue Oyster Cult",
        "title": "Screams",
        "duration": "189.80526",
        "song_id": "000002"
    },
    ...
]}
```

Output of example:

```
Welcome! We have over 59600 in FoothillTunes store!

Enter the duration of your play list (in minutes):
81.5

Songs in play list:
[Mes Dames Sarat, 0:4:7, Blue Oyster Cult, classic pop and rock
, Screams, 0:3:10, Blue Oyster Cult, classic pop and rock
, Dance The Night Away, 0:2:39, Blue Oyster Cult, classic pop and rock
, Debbie Denise, 0:4:11, Blue Oyster Cult, classic pop and rock
, (Don't Fear) The Reaper, 0:5:8, Blue Oyster Cult, classic pop and rock
, Morning Final, 0:4:15, Blue Oyster Cult, classic pop and rock
, The Revenge Of Vera Gemini, 0:3:51, Blue Oyster Cult, classic pop and rock
, True Confessions, 0:2:58, Blue Oyster Cult, classic pop and rock
, Out Of Stadion - Original, 0:1:22, Blue Oyster Cult, classic pop and rock
, Ginger Snaps - Original, 0:1:35, Blue Oyster Cult, classic pop and rock
, Demon's Kiss - Original, 0:3:53, Blue Oyster Cult, classic pop and rock
, Remodeling - Original, 0:2:14, Blue Oyster Cult, classic pop and rock
```

Additional Requirements for Part I and Part II:

- Your program must read input from files located in a directory called **resources**. Make sure to use relative path instead of specifying the entire path (such as /Users/alicew/myworkspace/so_on_and_so_forth).

- If the first exact match is found, stop class **SubsetSum findSubset()** method and print it out. If there is no exact match found, go through all combination of sublists and print out the first *closest* match.

- Show enough runs to prove your algorithm works, and show at least one run that does not meet target, exactly.
  Also, provide a special test to quickly dispose of the case in which the target is larger than the sum of all elements in the master set. This way, any target that is too large will not have to sit through a long and painful algorithm. Demonstrate that this works by providing at least one run that seeks a target larger than the sum of all master set elements.
  Modify the test file to include additional test cases.

Answer the Related Discussion Question

When checking an application our goal is to verify our implementation with various test cases. Share your test cases in this

week's discssusion forum titled:

Answer the Instructor's Weekly Questions Here -> Week 2: What test cases did you use?

FAQ: How do I read user input?

Read user input from standard in (i.e. the keyboard).

Read the input file by using a parser. For Part I you have to write your own parser.

FAQ: Should I include additional test runs?

Yes. Paste all the test cases in the same RUN.txt file.

FAQ: Does my output have to match the example output?

Your output style can be different from mine, as long as it is contains equivalent information and is easy to understand. However, your input files must match. Otherwise, I will not be able parse and therefore grade your submission.

FAQ: Can I submit more than once?

Yes, make sure to create and test one feature at a time, however small...rather than implementing a lot of code that you never test.
If your implementation so far compiles and runs successfully, commit to your repo. Then, continue working on the next feature.

FAQ: Can I include additional attributes, methods and or classes?

Yes, add as many as you see fit outside of the test classes **ShoppingBag** and **FoothillTunesStore**.

Keep in mind that I run **two test files** when checking your work:

1) I first run your test file (i.e. the one you submit and perhaps modify).

2) Then, I run the instructor test file, which will be testing for different input.

If your test class creates and depends on **classX** or **methodY()**, but the original test class (i.e. one or more test classes that you received as the starting code in your assigned repository) had never created an object of type **classX** or called **methodY()**, then the result will be compilation errors in the instructor test class. This is because the interfaces the instructor test class *relies* on is the same interfaces (i.e. class types and method calls) as the original test class (es).

In general, do *not* change method signatures. This includes:

- the package name,
- list of file imports,
- visibility of methods,
- capitalization of method names,
- order of arguments,
- handling of exceptions, etc.

I will *not* be grade your project if it results in compilation or runtime errors.

FAQ: For Part II may I optimize the algorithm to avoid running out of memory?

Yes, you may modify the algorithm to optimize it.

A potential optimization is to throwing away unnecessary subsets. If we are able to construct the bigger subset L + x (which is based on the smaller one), then we can throw away the smaller subset L.
For example, if you have 1,2,3,4, and I already have {1,2,3} and {1,2,4} less than target, do you still want {1,2} in your list?

FAQ: For Part I and Part II should I sort the given set first?

That depends on your implementation.

However, consider how expensive it would be to first sort a vary large library of songs. For example, I highly recommend agains using BubbleSort.java class.

Your implementation should not take more than a few seconds to run. If it does, then you need to optimize your implementation.

FAQ: Do I need to submit Javadocs for Part I and Part II?

Submitting documentation in the form of Javadocs for **Part I** is required. You must submit documentation for all .java files you are using (this includes **ShoppingBag.java** class).

If you would like to receive extra credit for **Part II** then submit Javadocs here as well.

REMINDER: Check that the **index.html** file of your **docs** folder to make sure the Javadocs are generated correctly.

As a reminder follow the Syllabus and **Program Guideline** for all required files, etc.

Into Student View

view the course as a brand new student.

Leave Student View