CSIS 10B Assignment 4

Goal

Choose and complete any 10 points from the problems below, which are all included in the download file on the website. Use BlueJ or Eclipse to complete the assignment, then export JAR file and upload to the server using your pass code. You may do more than 10 points of work but the max award will be 11 points.

In this assignment you will practice sorting an array of Student objects, searching and retrieving a student from the array, tackle an algorithm analysis problem to arrive at a Big-O estimate for the billiard algorithm, and finally practice writing a (yet another) recursive method.

Resources

- Chapter 4 and 8: Algorithm Analysis and Basic Sorting
- Chapters 7 and 9: Recursion and Advanced Sorting
- You may also wish to check out the code you wrote for labs 7 and 8 and the lab7 handout.

1. **(4 points) Student Sorter and Searcher** Add code to the Student class and the StudentSorter class. Modify Student so it implements Comparable<Student> using the technique developed in the lab7 BankAccount class. VIDEO HINT

   That is, define a static variable and a method to allow changing the compareType to allow comparing by:
   a. ID
   b. GPA (descending)
   c. lastName (and if two last names are the same, then check by firstNames)

   Then, using the method Arrays.sort (see the javadocs for Arrays) add code to StudentSorter to sort and display the data from the file three ways: 1) sorted by name, 2) sorted by ID and 3) sorted by descending GPA.

   When you are finished verifying your different sorting operations, add code to sort by ID once again. Then ask the user to enter a String for a student ID record. Following the steps described next, use Arrays.binarySearch (see the javadocs for Arrays) to find the index in the array where the desired student is located.

   How to use Binary Search

   To use binarySearch you have to create a special Student object that will be the key or target you are searching for. This student will have blank fields for name and GPA, and only have the ID field set to the ID entered as a string from the user. For example,

```
System.out.println("What student are you searching for?");
int ID = keyboard.next();
Student key = new Student();    // default student with all fields blank.
key.setID( ID);                 // set the ID field for key to the ID entered from the user's search request.
```

   Now you can pass both the database array and the key to the Arrays.binarySearch method, and store the location it returns in an index variable. This will be the location in the database array where the student matching the key student's ID is found. Finally, print the student at the location returned from binarySearch. This will show all the information for the student of interest.

2. **<u>(4 points) Algorithm Analysis</u>** Suppose that you have several numbered billiard balls on a pool table. At each step you remove a billiard ball from the table. If the ball removed is numbered *n*, you replace it with *n* balls whose number is *n* / 2, where the division is truncated to an integer. For example, if you remove the 5 ball, you replace it with five 2 balls. When a ball numbered 0 is removed, nothing more is added back. Here is what an example run of what your program should look like given an initial n of 6 (this assumes the top of the stack is at the right):

```
Enter the value of n, the value of the first billiard ball to put on the table
6
 poolTable contains [6]                        Just add one ball of number 6 into the pool table
 poolTable contains [3 3 3 3 3 3]              Remove the 6 and replace it with six 3's
 poolTable contains [3 3 3 3 3 1 1 1]          Remove the first 3 and replace it with three 1's
 poolTable contains [3 3 3 3 3 1 1 0]          Remove the first 1 and replace it with one 0
 poolTable contains [3 3 3 3 3 1 1]            Remove the zero
 poolTable contains [3 3 3 3 3 3 1 0]          Remove the second 1 and replace with one 0
 poolTable contains [3 3 3 3 3 1]              Remove the zero
   continue in this manner until the stack is empty
```

Write a program that simulates this process using a while loop. Although a bag would be suitable for this problem, just use a **java.util.Stack** of positive Integers to represent the balls on the pool table.

Then, using Big Oh notation, predict the time requirement for this algorithm when the initial stack contains **only the value *n***. In other words, push the value n onto the stack, then start your algorithm. **This means we are looking for O(n) where n is the number on the single ball that is first added to the stack.** Enter your prediction at the top of your main method. Then time the actual execution of the program for various values of *n* and use the spreadsheet **benchTesting.xls** included in the download to record and plot its performance as a function of *n*. (show time on the y-axis, n on the x-axis).

Does the plot match your original prediction? Add a statement reflecting on this after the line showing your original prediction, with any corrections to your estimate brought about by the bench test results.

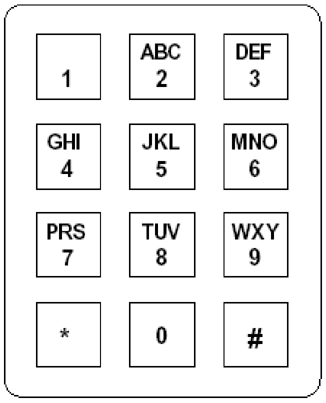
3. **<u>(2 points) Recursion</u>** For additional practice in recursion, add a recursive method **printDashed** to your **BilliardsProblem** program. It has nothing to do with the billiards problem, but this way you don't have to make another class file. Just write the method and test it at the top of main for the billiards solution. This method will accept a string argument and print the characters of the string with dashes (-) between the characters. Make sure your method does not print a leading or trailing dash.

4. **<u>(Optional, 4 points) Advanced Recursion</u>** If you do this problem, also do a Big-O analysis of your solution. It can be fun (but optional) to back up your analysis with data from a bench test of execution speed as the size of n increases. Make sure you don't include the print out in your time measurements.

   Write and test a collection of methods starting with `Vector<String> listMnemonics(String number)`

   (From *Thinking Recursively* by Eric Roberts.) On the standard Touch-Tone™ telephone dial, the digits are mapped onto the alphabet (minus the letters *Q* and *Z*) as shown in the diagram below:

In order to make their phone numbers more memorable, service providers like to find numbers that spell out some word (called a **mnemonic**) appropriate to their business that makes that phone number easier to remember. For example, the phone number for a recorded time-of-day message in some localities is 637-8687 (NERVOUS). Write a method `listMnemonics` that will generate all possible letter combinations that correspond to a given number, represented as a string of digits. For example, if you call



```
Vector<String> result = listMnemonics("723")
```

your method will generate and return a vector containing the following 27 possible letter combinations (strings) that correspond to that prefix:

PAD PBD PCD RAD RBD RCD SAD SBD SCD

PAE PBE PCE RAE RBE RCE SAE SBE SCE

PAF PBF PCF RAF RBF RCF SAF SBF SCF

Note, the order of the strings in the result vector is unimportant.

You will probably want to create the following helper function:

```
   String digitLetters(char ch)
```

This function returns the letters associated with a given digit.

You will find it useful to create a method that does most of the real work:

```
   void recursiveMnemonics(Vector<String> result,       // the vector we are building
                           String  mnemonicSoFar,       // the mnemonic being built up
                           String  digitsLeft);         // the remaining digits
```

This way, the `listMnemonics` method will just kick off the recursive process by invoking `recursiveMnemonics`, for example, here is the definition for `listMnemonics`:

```
Vector<String> listMnemonics(String phoneNumber)
{
   Vector<string> result;
   recursiveMnemonics(result, "", phoneNumber);
                    // when starting the mnemonicSoFar is the empty string
   return result;

}
```

Hint: each invocation of `recursiveMnemonics` will process one digit from the `digitsLeft` parameter, and will generate 3 different recursive calls based on the three letters associated with each digit. The base case will be when the `digitsLeft` string is empty.