Assignment objectives

1. Design and implement a binary-search-like Decrease and Conquer algorithm.
2. Compare the running times of O(n) and O(logn) algorithms.

Introduction (back-story)

Student pranksters from the School of Business are invading our computer labs! They have been sneaking into our labs on the weekends, throwing accounting-and-pizza bacchanals, burning out our printers with reams of TPS reports, and filling our whiteboards with endless financial scribbles. And they've been leaving behind their messes for us to clean up on Monday mornings!

In response, Stephanie is going to generate new numeric passcodes for our labs every week. She needs to inform everyone whenever there is a new passcode, but if she simply emails it out, it will be intercepted by the interlopers, who will continue their meddlesome invasions.

Stephanie decides to obscure each new passcode using a clever method that we programmers can crack, while making it harder for the troublemakers. The message will consist of a very long, unsorted list of *incorrect passcodes* (all the same length). The correct new passcode is the smallest number that *does not appear* anywhere on the list.

Overview

You are going to write *two functions* that will find the passcode for a given list of data. Both functions do the same thing, but differently. In addition, your program will (in a rudimentary way) estimate the running time of each function by counting the number of times a particular operation is performed.

Stephanie's data file contains numbers that are all within the range (inclusive) of 0 to "all nines" (99 or 999 or 9999 or ... different for each data file). Important: The data file does not contain any duplicated numbers. Your task is to find *the smallest number that is missing.*

To keep the business students very confused, the data file is *scrambled*, so you will need to sort the list after you read the data file. But we are not implementing a sort algorithm for this lab; you may just use one of Java's built-in sort() methods—e.g. Arrays.sort() or ArrayList.sort().

These are the two search functions you will write:

Brute Force passcode-finding algorithm: Starting at the beginning of the (sorted) list, examine each number until you discover that one is missing.

Binary Search passcode-finding algorithm: Perform a binary-search-like algorithm to find the smallest number that is missing from the list.

Passcode class details

You must provide your search functions as public methods in a class called PasscodeFinder. Below is a description of the class name and public methods that you MUST provide. You are free to declare any other private members and helper functions that you wish to use.

Class name: **PasscodeFinder**

Integer getPasscodeBruteForce(Integer[]) – The argument is a sorted array of non-negative integers with no duplicates. Return value is the smallest number *not present* in the array. The value could be zero. You may not use any Java collection types or methods inside this function—your algorithm must work only with the given plain array and other simple data types.

This function *should not sort the data.* It assumes that the given array is already sorted.

This function also counts the number of comparisons performed by the algorithm, and records this result in a (private) class variable. Count *all* the comparisons and *only* the comparisons that involve array elements in the list of fake passcodes. For example, if A[] is your list of passcodes, then a statement such as "if (A[i] <= 37)" counts as one comparison (regardless of the true/false outcome). But a statement such as "if (x > y)" for some other variables x and y does not count.

Note also that "conditional tests" includes not only if statements, but also the terminating conditions of all loops (if they involve tests against the values in the passcode list).

int bruteForceComparisons() – No arguments; returns the number of comparisons that were performed by the most recent invocation of getPasscodeBruteForce().

Integer getPasscodeBinarySearch(Integer[]) – Same argument as the Brute Force algorithm. Same return value as the Brute Force algorithm. Same rules for counting comparisons as the Brute Force algorithm.

int binarySearchComparisons() – No arguments; returns the number of comparisons that were performed by the most recent invocation of getPasscodeBinarySearch().

Hint about the Binary Search algorithm

This is an example of how you can use the decrease-and-conquer "pattern" of binary search to do something different than simply finding a key in a list. In this case you are looking for something that *is NOT there*!

For example, suppose this is your input array after sorting:

00 01 02 03 04 05 06 07 08 09 10 11 12 13 16 17 18 19 20 21 23

Then the correct new passcode should be 14. *How do you know when you've found it?* Draw this array with the indices showing and see if you can spot any clues.

Also note: You may need to examine not only the "middle" element of an array but also one or both of its neighbours at a given moment. Watch out for array-out-of-bounds errors!

Other details

For purposes of testing your Passcode, you will need to write code to read a data file, build and sort the list, and call your functions. This can be part of the Passcode class itself, in a main() function or otherwise, or in a separate driver class. This part of your code is not graded, and submitting it is optional.

It is OK to use ArrayList while you are reading the data, but remember that the public methods for finding the passcode are *required* to take a plain array of Integers as the only argument. It is

also OK to use ArrayList.sort() or Arrays.sort() for sorting your data. But you cannot use any Java collection types or methods within your search functions.

Results from several sample data files are shown below. You do not need to produce a table like this. This information is just for you to reference while you are testing your algorithms.

File information		Brute Force		Binary Search	
Name	**Length**	**Passcode**	**Number of comparisons**	**Passcode**	**Number of comparisons****
pc44.txt	68	44	45	44	8
pc99.txt	99	99	99	99	14
pc429.txt	626	429	430	429	18
pc0930.txt	4261	0930	931	0930	23
pc8589.txt	9104	8589	8590	8589	20
pc00000.txt	50092	00000	1	00000	15
pc00037.txt	50216	00037	38	00037	18
pc38424.txt	83756	38424	38425	38424	28
pc037373.txt	108656	037373	37374	037373	24
pc00000259.txt	364	00000259	260	00000259	14

****Note**: Your number of comparisons *may not be exactly the same* as these. It can depend on the logical structure of your binary search algorithm. But they should be generally consistent with the expected O(logN) performance of binary search. For example, if your data file has about 10000 data items, and your algorithm is doing about 2 comparisons per iteration/level of recursion, then your result should be about $2 \times \log_2(10000)$ or about 26. (Or fewer ... because binary search can "get lucky" and finish early, e.g. if the desired value is exactly in the middle of the array at any step along the way.)

Technically a similar "Note" might be true about Brute Force if you come up with an unusual algorithm, but I'd be surprised. There's pretty much one "obvious" way to do this via Brute Force, and that is going to take "passcode+1" number of comparisons. (Then again, I've been surprised before.)

Submission information

Due date: As shown on Learning Hub.

Submit the following to the drop box on Learning Hub:

- Source code (*.java file) for Passcode class.
- (Optional) If you have an additional "Main" driver class, you may submit it, but this is not necessary.
- File names of the above are not important.
- Please *do not zip* or otherwise archive your code. Plain Java files only.
- Please *do not zip* or include your entire project directory.

Marking information

This lab is worth 20 points.

Virtual donut

This part is not required; it is just for fun.

Create a *data file* that satisfies the following two conditions:

1. It is a valid data file for this program.
2. It causes your Binary Search algorithm to perform *exactly* 37 comparisons.

If you attempt this portion, be sure to submit the data file you have created, and also let me know what the expected passcode value is.