UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

INFR08014 INFORMATICS 1 - OBJECT-ORIENTED PROGRAMMING

Friday 12th May 2017

09:30 to 11:30

INSTRUCTIONS TO CANDIDATES

1. **Note that all questions are compulsory**
2. **Remember that a file that does not compile, or does not pass the simple JUnit tests provided, will get no marks.**
3. **This is an open book exam. You may bring in your own material on paper. No electronic devices are permitted.**
4. **CALCULATORS MAY NOT BE USED**

Convener: I. Simpson
External Examiner: I. Gent

THIS EXAMINATION WILL BE MARKED ANONYMOUSLY

1. In this question you will explore modeling discrete probability distributions and their entropy. Entropy provides a measure of the uncertainty in a probability distribution. You are not expected to have any prior knowledge of entropy. In this question you will use the java utility function **Arrays.toString**, and may wish to also use a sorting function such as **Arrays.sort**. You may need to look at the JDK documentation for these. You are advised to read all question parts before beginning. Note that this question assumes that input strings contain only lowercase letters a-z throughout. Create a public class **Entropy**

   (a) In the class **Entropy**, implement the public static method:
   int[] charCount(String s)
   that returns an **int** array containing the frequencies of each character occurring in the **String s**. The length of the returned array should correspond to the number of distinct characters in **s**. It should return null if **s** is empty or null. Assume the string contains only lowercase letters. Return the counts, sorted in alphabetical order of the corresponding characters. (The correct counts in incorrect order will receive partial credit).

   Expected behaviour: `countFreq("abbc")` should return {1,2,1}.
   `countFreq("xxxa")` should return {1, 3}.

   [*15 marks*]

   (b) In the class **Entropy**, implement the public static method:
   double[] normalise(int[] c)
   that returns a normalised array of **double** probabilities corresponding to the specified **int** array of counts **c**. If the ith element of the input array is denoted c_i, then return the array **p** where $p_i = c_i / \sum_i c_i$. That is, divide each element in the input array by the sum of the input elements, and return the result. It should return null if the input is null or empty.

   Expected behaviour: `normalise(new int {2, 1, 1})` should return {0.5, 0.25, 0.25}.

   You can verify that with these routines, applying normalise to the output of charCount will estimate the probability of each symbol in the input string.

   Expected behaviour: `normalise(countFreq("abbc"))` should return {0.25, 0.5, 0.25}.

   [*10 marks*]

   (c) In the class **Entropy**, implement the public static method:
   double entropyOf(double[] p)
   The **entropyOf** routine should compute the entropy of the **double** array **p**, that is assumed to represent a probability vector. If p_i represents the i-th element of the input array, then it should compute: $-\sum_i p_i \log p_i$. That is, it should return the negative sum of each input element multiplied by its (base e) logarithm. Assume that $0 < p_i < 1$ for all i, so there are no numerical issues.

Expected behaviour: `entropyOf(new double [] {0.5, 0.25, 0.25})` should compute $-0.5 \log 0.5 - 0.25 \log 0.25 - 0.25 \log 0.25$, and return 1.0397. [*10 marks*]

(d) In the class **Entropy**, implement the public static method:
int[][] charCountArray(String[] a)
this method should count the frequencies of characters in each **String** in array **a** and return an **int** array of arrays containing the counts in each input string. The returned array of arrays should have as many rows as there are **String**s in the input array. Each row will contain a character frequency count for the corresponding input **String**. Unlike **charCount**, **charCountArray** should only count those characters that exist in exactly one input string. That is, each row returned should contain the counts of characters that are *unique* to the corresponding input string. Any distinct character present in more than one input **String** should be excluded from the counts of all input strings. The input **String**s should again be assumed to contain lowercase letters, and the resulting counts should again be sorted in alphabetical order of corresponding characters. Assume there is at least one unique character in each input.

Expected behaviour: `charCountArray(new String {"abbcccxx","bbccyzdd"))` should return {{1,2},{2,1,1}} because: 'b' and 'c' occur in multiple inputs so are excluded from counting, 'a' and 'x' uniquely occur in the first input with frequency 1 and 2 and 'd', 'y' and 'z' uniquely occur in the second input with frequency 2, 1 and 1. [*10 marks*]

(e) In the class **Entropy**, write a main method. The main method should assume that two strings are presented as command line arguments, and print on five lines:

i. The character probabilities corresponding to the first argument. Use **charCount** and **normalise**, and use **Arrays.toString** to convert the result to a formatted **String** for printing.
ii. The entropy of character probabilities in the first and second argument.
iii. The entropy of *unique* characters in the first and second argument (use **charCountArray**, **normalise** and **entropyOf**). Use at least 3 decimal places.

If you did not complete one of the required methods in (a)-(d), leave the corresponding line blank after the colon.

Expected behaviour: The output should use formatting shown in the example below. Invoking `java Entropy hello world` should produce:

Character Probabilities in hello : [0.2, 0.2, 0.4, 0.2]
Entropy of hello : 1.332
Entropy of world : 1.609

Entropy of unique chars in hello : 0.693
Entropy of unique chars in world : 1.099

[*5 marks*]

- The file you must submit for this question is: **Entropy.java**. Before you submit, check that it compiles, passes the basic JUnit tests provided, and does not use packages, otherwise it will get 0.

2. You are given files **Phone.java** and **App.java** containing the classes **Phone** and **App**. Examine these, but do not change them. Your task is to implement the subclass **FruitySmartPhone** extending **Phone**. A FruitySmartPhone can install Apps. By examining **App**, note that any App has a name, a memory requirement, and a CPU speed requirement. In this question you will use the java classes **HashMap** and **ArrayList**. You may need to look at the JDK documentation for these. You are advised to read all question parts before beginning.

   (a) Define the class **FruitySmartPhone** extending **Phone**. It should have a private instance variable **installedApps** of type **HashMap<String,Integer>**, storing the names of Apps and the *total* memory used by each App. It should also define private instance variables (fields) **cpuSpeed** and **freeMemory** both of type **int**, and public getter methods **int getCpuSpeed()** and **int getFreeMemory()**. [*10 marks*]

   (b) Write a public constuctor for **FruitySmartPhone**. The constructor must take a **String** representing the owner's name, and pass this to the **Phone** constructor. It must also take an **int**, representing the available memory of the phone, and a second **int** representing the CPU speed of the phone, and use these to set the appropriate instance variables. [*5 marks*]

   (c) Write public instance methods **installApp** and **uninstallApp**.
   installApp should take an **App** argument. If the app is already installed, insufficient memory is available, or the app's required CPU speed is greater than the phone's CPU speed, then it should do nothing and return **false**. If the app is not already installed, and space is available, and the cpu speed is sufficient then installation should proceed. It should add the app and its space used to the **HashMap installedApps**, reduce **freeMemory** appropriately and return **true**.
   uninstallApp should take an **App** argument. If the app is not already installed, it should return **false**. Otherwise it should remove the app from the installedApps **HashMap**, and update the instance variable **freeMemory** to reflect the increased available memory after uninstalling the app. Finally it should return **true** after successful uninstallation. [*15 marks*]

   (d) Write a public instance method **useApp**. It accepts a **String** specifying the name of the app to be used and returns boolean. An app takes up 1 additional unit of data each time it is used, therefore **installedApps** and **freeMemory** should be updated by 1 to reflect the increased memory used by this app before returning **true**. If the named app is not installed, or insufficient memory is available, it should return **false** and do nothing. [*10 marks*]

   (e) Write a public instance method **getInstalledApps**. This should return an **ArrayList<String>** listing the names of all the installed apps, *sorted in descending order of memory usage*. If no apps are installed, it returns an empty **ArrayList<String>**. [*10 marks*]

Expected Behaviour: By way of example, the code:

```
FruitySmartPhone p = new FruitySmartPhone("John Smith",100,10);
p.installApp(new App("Camera",1,2));
p.installApp(new App("Music",2,3));
p.installApp(new App("Podcast",2,11));
p.useApp("Music");
System.out.println(p.getInstalledApps());
```

would print out:
[Music, Camera]
and result in the field **installedApps** containing {Music=3, Camera=1}.

In Summary, **FruitySmartPhone** has the following public interface (omitting what is inherited from **Phone**) with the behviour explained above:

public	class FruitySmartPhone
	FruitySmartPhone(String o, int m, int s)
int	getMemory()
int	getCpuSpeed()
boolean	installApp(App a)
boolean	uninstallApp(App a)
boolean	useApp(String name)
ArrayList<String>	getInstalledApps()

- The file you must submit for this question is **FruitySmartPhone.java**. Before you submit, check that it compiles, passes the basic JUnit tests provided, and does not use packages, otherwise it will get 0.