UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

INFR08014 INFORMATICS 1 - OBJECT-ORIENTED PROGRAMMING

Friday 12 th May 2017

14:30 to 16:30

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 the Kullback Leibler divergence (KLD) between them. You are not expected to have any prior knowledge of KLD. The Kullback Leibler divergence provides an asymmetric measure of the difference between two distributions. 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 **Divergence**

   (a) In the class **Divergence**, 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 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 **Divergence**, 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 input symbol.

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

   [*10 marks*]

   (c) In the class **Divergence**, implement the public static method:

   double kld(double[] p, double[] q)

   The **kld** method should compute the Kullback Leibler Divergence between the double arrays **p** and **q**, that are assumed to represent probability vectors. Specifically, if p_i represents the i-th element of the input array **p** and q_j represents the jth element of input array **q**, then it should compute: $\sum_i p_i \log \frac{p_i}{q_i}$. That is, it should multiply each element in **p** by the (base e)

logarithm of the quotient between the corresponding elements of **p** and **q**, and then sum this quantity over all elements. If the input distributions are the same, a correctly implemented kld function should return zero. You may assume that **p** and **q** have the same length, and that $0 < p_i < 1$ and $0 < q_i < 1$ for all i, so there are no numerical issues.

Expected behaviour: `kld(new double[] {0.75, 0.25}, new double[] {0.6, 0.4})` should compute $0.75 \log \frac{0.75}{0.6} + 0.25 \log \frac{0.25}{0.4}$ and return 0.0499. `kld(new double[] {0.6, 0.4}, new double[] {0.75, 0.25})` should compute $0.6 \log \frac{0.6}{0.75} + 0.4 \log \frac{0.4}{0.25}$ and return 0.0541.

[*10 marks*]

(d) In the class **Divergence**, 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 the frequency count for the corresponding input **String**.

Unlike **charCount**, **charCountArray** should only count chars that exist in *all* input strings. That is each row returned should contain the same number of elements, and the number of columns in the returned matrix should correspond to the number of distinct characters that occur in *every* **String** of the input array. Characters not present in all inputs should be excluded from counting. You may assume there is at least one countable character in all strings.

Expected behaviour: `countFreqArray(new String {"cbbaaa","bbbccd"})` should return {{2,1}, {3,2}} containing the counts corresponding to 'b' and 'c'. 'a' and 'd' are excluded from counting because they do not occur in all inputs.

[*10 marks*]

(e) In the class **Divergence**, 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 counts corresponding to the first input, as computed by **charCount**. Use **Arrays.toString** to convert the result to a suitably formatted string for printing.

ii. The character probabilities corresponding to the first and second input, as computed by **charCount** and **normalise**. Use **Arrays.toString** to convert the **double** array to a suitably formatted **String** for printing.

iii. The shared symbol count, computed by **charCountArray**. Use **Arrays.toString** repeatedly to format the output.

iv. The divergence between shared symbol probabilities obtained from the first and second inputs (use **charCountArray**, **normalise**, and **kld**). 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 Divergence cbbaaa bbbccd` should produce:

Counting cbbaaa : [3, 2, 1]
Symbol probabilities in cbbaaa : [0.5, 0.3333..., 0.1666...]
Symbol probabilities in bbbccd : [0.5, 0.3333..., 0.1666...]
Shared Symbol Count : [[2, 1], [3, 2]]
Divergence PQ : 0.009 [*5 marks*]

- The file you must submit for this question is: **Divergence.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 **Browser.java** and **Plugin.java** containing the classes **Browser** and **Plugin**. Examine these, but do not change them. Your task is to implement the subclass **SnowSquirrelBrowser** extending **Browser**. The cloud-based **SnowSquirrelBrowser** can install plugins. By examining **Plugin**, note that any plugin requires a certain amount of cloud quota, and requires a specific version of **SnowSquirrelBrowser**. 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 **SnowSquirrelBrowser** extending **Browser**. It should have a private instance variable **installedPlugins** of type **HashMap<String,Double>**, storing the names of plugins and the *total* cloud quota used by each plugin. It should also define the private instance variables (fields) **version** of type **String** and **availableQuota** of type **double**, and public getter methods and public getter methods **String getVersion()** and **double getQuota()** [*10 marks*]

   (b) Write a public constructor for **SnowSquirrelBrowser**. The constructor must take a **String** representing the owner's name, and pass this to the **Browser** constructor. It must also take a **double**, representing the cloud quota for this browser, and a second **String** representing the version of the browser, and use these to set the appropriate instance variables. [*5 marks*]

   (c) Write public instance methods **installPlugin** and **uninstallPlugin**.
   installPlugin should take a **Plugin** argument and return a boolean. It should do nothing and return **false** if: the plugin is already installed, or the plugin requires more cloud quota than is available, or the browser version does not match the version required by the plugin exactly (string equality). If the plugin is not already installed, and sufficient quota space is available, and the versions strings match, it should add the Plugin and its cloud quota requirement to the **HashMap installedPlugins**, reduce **availableQuota** appropriately and return **true**.
   uninstallPlugin should take a **Plugin** argument and return a boolean. If the specified plugin is not already installed, it should return **false**. Otherwise it should remove the plugin from the **installedPlugins HashMap** and update **availableQuota** appropriately to reflect the increased available quota after uninstallation. Finally it should return **true** after successful uninstallation. [*15 marks*]

   (d) Write a public instance method **usePlugin**. It accepts a **String** specifying the name of the plugin to be used, and returns boolean. A plugin stores an additional 1 unit of data each time it is used, therefore **installedPlugins** and **availableQuota** should be updated by 1 to reflect the increased memory used by this plugin before returning **true**. If the named plugin is not installed, or insufficient quota is available, it should return **false** and do nothing. [*10 marks*]

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

Expected Behaviour: By way of example, the code:

```
SnowSquirrelBrowser s = new SnowSquirrelBrowser("John Smith",10,"1.0");
s.installPlugin(new Plugin("Shopping",2.0,"1.0"));
s.installPlugin(new Plugin("VPN",3.0,"1.0"));
s.installPlugin(new Plugin("Email",2.0,"2.0"));
s.usePlugin("VPN");
System.out.println(s.getInstalledPlugins());
```

should print out:
[VPN, Shopping]
and result in the field **installedPlugins** containing
{VPN=4.0, Shopping=2.0}

In Summary, **SnowSquirrelBrowser** has the following public interface (omitting what is inherited from **Browser**) with the behavior explained above:

public	class SnowSquirrelBrowser
	SnowSquirrelBrowser(String o, double quota, String version)
double	getQuota()
String	getVersion()
boolean	installPlugin(Plugin p)
boolean	uninstallPlugin(Plugin p)
boolean	usePlugin(String name)
ArrayList<String>	getInstalledPlugins()

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