GALWAY-MAYO INSTITUTE OF TECHNOLOGY

## Department of Computer Science & Applied Physics

B.Sc. in Software Development
Object Oriented Programming Main Assignment (50%)

A Multithreaded Language Detector

1. Overview

You are required to develop a Java API that can rapidly compare a query text file against a *n*-gram collection of subject texts and determine the natural language of the query file. The API should uphold the principles of loose-coupling and high cohesion throughout its design by correctly applying abstraction, encapsulation, composition and inheritance. An overview of the system is shown below:
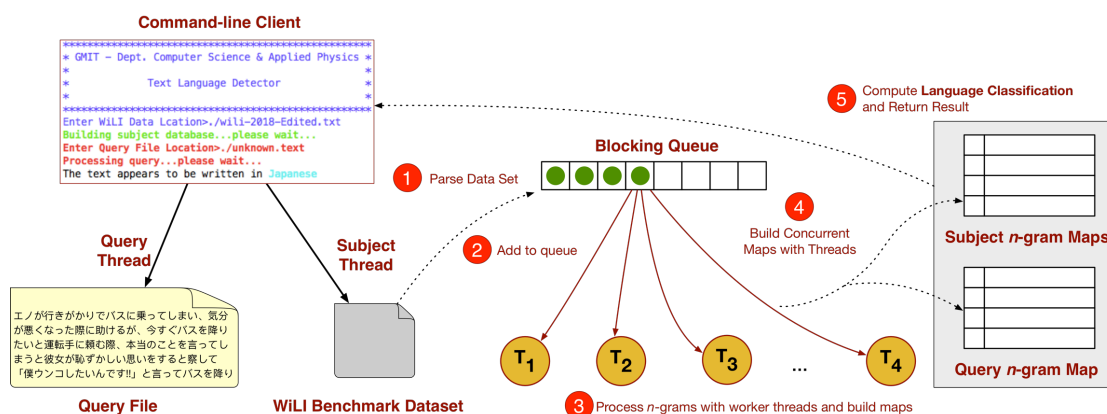
The text file(s) and other resources required for the assignment are available on Moodle.

2. Minimum Requirements

- Use the package name *ie.gmit.sw*. The application must be deployed and runnable using the specification in Section 3.
- Create a console-based *menu-driven UI* to input the name of the language data set file and a query file to analyse. Any other input parameters should be input through the menu-driven UI. Do not use a GUI!
- Each file should be parsed, processed and added to a rapidly searchable data structure using the technique described in Section 5.
- The application should be fully threaded and use *Runnable*, *Callable* and *Future* types where appropriate. You should give careful consideration to using the suite of data structures in the *java.util.concurrent* package.
- The output of the system should be the language name that best matches the text of the input query file.
- Provide a **UML** diagram of your design and **JavaDoc** your code.

3. Deployment and Delivery

The project must be submitted by midnight on Sunday 6th January 2020. Before submitting the assignment, you should review and test it from a command prompt on *__a different computer__* to the one that you used to program the project.

- The project must be submitted as a Zip archive *(not a 7z, rar or WinRar file)* using the Moodle upload utility. You can find the area to upload the project under the "*A Multithreaded Language Detector (50%) Assignment Upload*" heading of Moodle. *Do not add comments to the Moodle assignment upload form.*

- The Zip archive should be named *<id>*.zip where *<id>* is your GMIT student number.

- The Zip archive should have the following structure (do NOT submit the assignment as an Eclipse project):

Marks	Category
oop.jar	A Java **archive** containing your API and runner class with a main() method. You can create the JAR file using Ant or with the following command from inside the "bin" folder of the Eclipse project: **jar –cf oop.jar *** The application should be executable from a command line as follows: **java –cp ./oop.jar ie.gmit.sw.Runner**
src	A directory that contains the packaged **source code** for your application.
README.txt	A text file detailing the main **features** of your application. Marks will only be given for features that are described in the README.
design.png	A UML **class diagram** of your API design. The UML diagram should only show the relationships between the key classes in your design. Do not show private methods or attributes in your class diagram. You can create high quality UML diagrams online at **www.draw.io**.
docs	A directory containing the JavaDocs for your application. You can generate JavaDocs using Ant or with the following command from inside the "src" folder of the Eclipse project: **javadoc -d [*path to javadoc destination directory*] ie.gmit.sw** Make sure that you read the JavaDoc tutorial provided on Moodle and comment your source code correctly using the JavaDoc standard.

4. Marking Scheme
Marks for the project will be applied using the following criteria:

Element	Marks	Description
Structure	10	*All or nothing.* The packaging and deployment correct. All JAR, module, package and runner-class names are correct.
README	7	All features and their design rationale are fully documented.
UML	10	Class diagram correctly shows all the important structures and relationships between types.
JavaDocs	8	All classes are fully commented using the **JavaDoc** standard and generated docs available in the *docs* directory.
Robustness	40	The fully threaded application executes perfectly, without any manual intervention, using the specified execution requirements.
Cohesion	10	There is very high cohesion between packages, types and methods.
Coupling	10	The API design promotes loose coupling at every level.
Extras	5	Only relevant extras that have been fully documented in the README.

You should treat this assignment as a project specification. Each of the elements above will be scored using the following criteria:

- 0–30%   **Fail:** Not delivering on basic expectations
- 40-59%   **Mediocre:** Meets basic expectations
- 60–79%   **Good:** Exceeds expectations.
- 80-90%   **Excellent:** Demonstrates independent learning.
- 90-100%   **Exemplary:** Good enough to be used as a teaching aid

5. Automatically Detecting the Language of Text

An *n*-gram, also called a shingle, *l*-mer or *k*-mer is a contiguous substring of text of size *n*. Their use is an example of a ***divide-and-conquer*** technique and *n*-grams have been applied to a wide variety of computing problems including text similarity measurement, text subject determination and language classification.

Consider the text "*object oriented programming is good fun for me*". This string can be decomposed into the following set of 5-grams: {"*objec*", "*t_ori*", "*ented*", "*_prog*", "*rammi*", "*ng_is*", "*_good*", "*_fun_*", "*for_m*", "*e____*"}. A set of *n*-grams can be formed from discrete "chunks" of non-overlapping characters or by tiling across a string using a fixed offset. For example, using an offset of 1, the following set of 5-grams will be formed: {"*objec*", "*bject*", "*ject_*", "*ect_o*", "*ct_or*", "*t_ori*"…}. Note that spaces are included in the *n*-gram set of a text and that the total number of possible *n*-grams in an *n*-spectrum is $\sum^n$, where $\sum$ is the number of symbols in the alphabet for the text, e.g. the full *n*-spectrum for 5-gram lower-case words in English has $26^5 = 11,881,376$ elements. In practical terms, an *n*-spectrum will not be fully realized, as the vast majority of *n*-gram combinations will not be present in text. It should also be clear that the frequency of a 2-gram appearing in a text will be much greater than that of a 5-gram. For example, if we only include lower-case characters in English, each character will appear with a probability of $1/26 = 0.038$. A 2-gram will therefore occur with a probability of $0.038 * 0.038 = 0.001444$ and a 5-gram with a probability of $0.038 * 0.038 * 0.038 * 0.038 * 0.038 = 0.038^5 = 7.92 \times 10^{-8}$.

Empirical evidence shows that the most frequently occurring ~300 *n*-grams are highly correlated to a language, with the next 300-400 *n*-grams more correlated to the subject that a text relates to. Moreover, only ~400 characters of text are required to generated a sufficient set of *n*-grams to allow a language to be identified correctly. Naturally, these observations will vary somewhat from language to language, but they do provide an effective heuristic for language detection.

The following set of steps can be used to build a ***subject database*** to determine the language classification of a text. Although there are 235 languages in the WiLI benchmark dataset, it is probably worth while creating a *Language* enum for this purpose.

- Generate 235 maps capable of relating an *n*-gram to its frequency of occurrence in a specific language.
- For each line of text in the *WiLI benchmark dataset*.
  - Split the line into its text and language tokens using the '@' symbol.
  - Set the current map to point to that denoted by the language token.
  - For each 1-gram, 2-gram, 3-gram and 4-gram in the text.
    - Insert the *n*-gram into the map with a frequency of 1 or increment the frequency if the *n*-gram already exists in the map.
- Sort each of the maps in descending order of occurrence frequency. Only the top ~300 *n*-grams need to be retained.

The language of the **query text** can be determined using the following steps:
- Generate a single map that relates an *n*-gram to its frequency of occurrence.
- Parse the first ~400 characters from the text into a query sequence.
- For each 1-gram, 2-gram, 3-gram and 4-gram in the text.
  - Insert the *n*-gram into the map with a frequency of 1 or increment the frequency if the *n*-gram already exists in the map.
- Sort the map in descending order of occurrence frequency.
- Compute a **distance measure** for the map compared to all 235 maps in the subject database and output the score for each map.
- Output the language of the map with the highest score as the language classification.

Computing a Distance Metric

The **distance measure** can be computed using a wide variety of different approaches. In this assignment, we will use the **Out-of-Place Measure** as a distance metric. For each *n*-gram, the **Out-of-Place** distance measure if computed as $d(n\text{-}gram) = R_s(n\text{-}gram) – R_q(n\text{-}gram)$, where $R_s(n\text{-}gram)$ is the rank of an n-gram in a subject database and $R_q(n\text{-}gram)$ is the rank of the same n-gram in the query database. The sum of all the **Out-of-Place** values for all *n*-grams is the distance measure of the document for that language. Consider the following example:

The **Out-of-Place** distance between just the first 9 elements in the query map and each of the subject maps is **d**("ACH") + **d**("GUS") + **d**("AGU") + **d**("ILE") + **d**("INE") + **d**("ART") + **d**("CHU") + **d**("EAR") + **d**("CHT"). As none of the first 9 query elements match any of the first 9 English *n*-grams, the distance value will be relatively large. However, for Gaeilge, the distance is (1 - 1) + (3 - 2) + (2 - 3) + (8 - 4) + (5 - 5) + (6 - 6) + (7 - 7) + (4 - 8) + (9 - 9) = 0, indicating that the query text is written in the Irish language.

Computational Overhead of n-gram Mapping

We can compute the set of *n*-grams and their frequency in a text by relating each *n*-gram to an integer value in a map. A naïve implementation of this will declare the map as *Map<String, Integer>*. The problem with this approach is that each String key in the map has a minimum memory usage (Java 7) of $8 * (((length * 2) + 45) / 8)$. The String "JOHN" will therefore consume 53 bytes of memory! A better approach is to declare the map as *Map<Integer, Integer>*, using the *hashCode()* method of each *n*-gram String as a key, allowing the key to be represented more efficiently as a fixed-length 32-bit integer that can be manipulated in a number of different ways. This approach does have the disadvantage of permitting two different strings to have the same hash code and, although hash-collisions will be rare, they will undoubtedly occur when using large data sets.

If we limit the maximum length of an *n*-gram to 4, we can compress a String into a 64-bit long

and declare the map as *Map<Long, Integer>*. This is shown below where each 16-bit Unicode value can be "packed" into a bit vector for a 64-bit long.

char[]	'J'	'O'	'H'	'N'
Decimal	74	79	72	78
Binary	0000000001001010	0000000001001111	0000000001001000	0000000001001110
	16 bits	16 bits	16 bits	16 bits

= 20829487583723598L = 64 bit long

The following method illustrates how the process is implemented in Java. The *<<* operator shifts the bits in the long 16 places to the left to accommodate a UTF-16 Unicode character. The pipe, "|", symbol is bitwise OR. You should alter the *if* statement to allow a 1-gram to be used.

```java
public long encode(String s) throws Exception{ //
    if (s.length() < 2 || s.length() > 4) throw new Exception("Can only encode n-grams with 2, 3, or 4 characters");

    long sequence = 0x0000000000000000L; //Set all 64 bits to zero
    for (int i = 0; i < s.length(); i++){ //Loop over the String
        sequence <<= 16; //Shift bits left by 16 bits
        sequence |= s.charAt(i); //Bitwise OR. Sets the first (right-most) bits to the Unicode binary value

    }
    return sequence; //return the chars encoded as a long!
}
```