Core Part 2 (Scala, 3 Marks)

"What one programmer can do in one month,
two programmers can do in two months."
— Frederick P. Brooks (author of The Mythical Man-Month)

Important

- Make sure the files you submit can be processed by just calling `scala-cli compile <<filename.scala>>` on the command line.[1] Use the template files provided and do not make any changes to arguments of functions or to any types. You are free to implement any auxiliary function you might need.
- **Do not leave any test cases running in your code because this might slow down your program!** Comment out test cases before submission, otherwise you might hit a time-out.
- Do not use any mutable data structures in your submissions! They are not needed. This means you cannot create new `Array`s or `ListBuffer`s, for example.
- Do not use `return` in your code! It has a different meaning in Scala than in Java. It changes the meaning of your program, and you should never use it.
- Do not use `var`! This declares a mutable variable. Only use `val`!
- Do not use any parallel collections! No `.par` therefore! Our testing and marking infrastructure is not set up for it.

Also note that the running time of each part will be restricted to a maximum of 30 seconds on my laptop.

Disclaimer

It should be understood that the work you submit represents your **own** effort! You have implemented the code entirely on your own. You have not copied from anyone else. Do not be tempted to ask Copilot for help or do any other shenanigans like this! An exception is the Scala code I showed during the lectures or uploaded to KEATS, which you can freely use.

[1] All major OSes, including Windows, have a command line. So there is no good reason to not download scala-cli, install it and run it on your own computer. Just do it!

Reference Implementation

Like the C++ part, the Scala part works like this: you push your files to GitHub and receive (after sometimes a long delay) some automated feedback. In the end we will take a snapshot of the submitted files and apply an automated marking script to them.

In addition, the Scala part comes with reference implementations in form of `jar`-files. This allows you to run any test cases on your own computer. For example you can call `scala-cli` on the command line with the option `--extra-jars docdiff.jar` and then query any function from the template file. Say you want to find out what the function `occurrences` produces: for this you just need to prefix it with the object name `C2`. If you want to find out what these functions produce for the list `List("a", "b", "b")`, you would type something like:

```
$ scala-cli --extra-jars docdiff.jar

scala> C2.occurrences(List("a", "b", "b"))
...
```

Hints

For the Core Part 2: useful operations involving regular expressions:

```
reg.findAllIn(s).toList
```

finds all substrings in `s` according to a regular regular expression `reg`; useful list operations: `.distinct` removing duplicates from a list, `.count` counts the number of elements in a list that satisfy some condition, `.toMap` transfers a list of pairs into a Map, `.sum` adds up a list of integers, `.max` calculates the maximum of a list.

Core Part 2 (3 Marks, file docdiff.scala)

It seems plagiarism—stealing and submitting someone else's code—is a serious problem at other universities.[2] Detecting such plagiarism is time-consuming and disheartening for lecturers at those universities. To aid these poor souls, let's implement in this part a program that determines the similarity between two documents (be they source code or texts in English). A document will be represented as a list of strings.

Tasks

(1) Implement a function that 'cleans' a string by finding all (proper) words in the string. For this use the regular expression `\w+` for recognising words and the library function `findAllIn`. The function should return a document (a list of strings). [0.5 Marks]

(2) In order to compute the overlap between two documents, we associate each document with a `Map`. This Map represents the strings in a document and how many times these strings occur in the document. A simple (though slightly inefficient) method for counting the number of string-occurrences in a document is as follows: remove all duplicates from the document; for each of these (unique) strings, count how many times they occur in the original document. Return a Map associating strings with occurrences. For example

```
occurrences(List("a", "b", "b", "c", "d"))
```

produces `Map(a -> 1, b -> 2, c -> 1, d -> 1)` and

```
occurrences(List("d", "b", "d", "b", "d"))
```

produces `Map(d -> 3, b -> 2)`. [1 Mark]

(3) You can think of the Maps calculated under (2) as memory-efficient representations of sparse "vectors". In this subtask you need to implement the *product* of two such vectors, sometimes also called *dot product* of two vectors.[3]

For this dot product, implement a function that takes two documents (`List[String]`) as arguments. The function first calculates the (unique) strings in both. For each string, it multiplies the corresponding occurrences in each document. If a string does not occur in one of the documents, then the product for this string is zero. At the end you need to

[2] Surely, King's students, after all their instructions and warnings, would never commit such an offence. Yes?

[3] https://en.wikipedia.org/wiki/Dot_product

add up all products. For the two documents in (2) the dot product is 7, because

$$\underbrace{1 * 0}_{''a''} + \underbrace{2 * 2}_{''b''} + \underbrace{1 * 0}_{''c''} + \underbrace{1 * 3}_{''d''} \qquad = 7$$

[1 Mark]

(4) Implement first a function that calculates the overlap between two documents, say d_1 and d_2, according to the formula

$$\texttt{overlap}(d_1, d_2) = \frac{d_1 \cdot d_2}{max(d_1^2, d_2^2)}$$

where d_1^2 means $d_1 \cdot d_1$ and so on. You can expect this function to return a `Double` between 0 and 1. The overlap between the lists in Task (2) is 0.5384615384615384.

Second, implement a function that calculates the similarity of two strings, by first extracting the substrings using the clean function from (1) and then calculating the overlap of the resulting documents.

[0.5 Marks]