Scala Part 2

Important (Please Read):

- Make sure the files you submit can be processed by just calling scala <<filename.scala>> on the commandline.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 Arrays or ListBuffers, 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
All major OSes, including Windows, have a commandline. So there is no good reason to not download Scala, install it and run it on your own computer.

Also note that the running time of each part will be restricted to a maximum of 30 seconds on my laptop: If you calculate a result once, try to avoid to calculate the result again. Feel free to copy any code you need from files knight1.scala, knight2.scala and knight3.scala (for Main 4).

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

It seems plagiarism—stealing and submitting someone else's code—is a serious problem at other universities. 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)`. [Mark 1]

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

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

[1 Mark]

(4) Implement first a function that calculates the overlap between two documents, say d1 and d2, 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 (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]

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 on the command line with the option `-cp danube.jar` and then query any function from the template file. Say you want to find out what the function produces: for this you just need to prefix it with the object name `M2`. If you want to find out what these functions produce for the list `List("a", "b", "b")`, you would type something like:

```
$ scala -cp danube.jar
scala> val ratings_url =
     | """https://nms.kcl.ac.uk/christian.urban/ratings.csv"""

scala> M2.get_csv_url(ratings_url)
val res0: List[String] = List(1,1,4 ...)
```

Hints

Use `.split(",").toList` for splitting strings according to commas (similarly for the newline character `\n`), `.getOrElse(..,..)` allows to query a Map, but also gives a default value if the Map is not defined, a Map can be 'updated' by using `+`, `.contains` and `.filter` can test whether an element is included in a list, and respectively filter out elements in a list, `.sortBy(_._2)` sorts a list of pairs according to the second elements in the pairs—the sorting is done from smallest to highest, `.take(n)` for taking some elements in a list (takes fewer if the list contains less than `n` elements).

Main Part 2 (6 Marks, file danube.scala)

You are creating Danube.co.uk which you hope will be the next big thing in online movie renting. You know that you can save money by anticipating what movies people will rent; you will pass these savings on to your users by offering a discount if they rent movies that Danube.co.uk recommends.

Your task is to generate *two* movie recommendations for every movie a user rents. To do this, you calculate what other renters, who also watched this movie, suggest by giving positive ratings. Of course, some suggestions are more popular than others. You need to find the two most-frequently suggested movies. Return fewer recommendations, if there are fewer movies suggested.

The calculations will be based on the small datasets which the research lab GroupLens provides for education and development purposes.

`https://grouplens.org/datasets/movielens/`

The slightly adapted CSV-files should be downloaded in your Scala file from the URLs:

`https://nms.kcl.ac.uk/christian.urban/ratings.csv` (940 KByte)
`https://nms.kcl.ac.uk/christian.urban/movies.csv` (280 KByte)

The ratings.csv file is organised as userID, movieID, and rating (which is between 0 and 5, with *positive* ratings being 4 and 5). The file movie.csv is organised as movieID and full movie name. Both files still contain the usual CSV-file header (first line). In this part you are asked to implement functions that process these files. If bandwidth is an issue for you, download the files locally, but in the submitted version use `Source.fromURL` instead of `Source.fromFile`.

Tasks

(1) Implement the function `get_csv_url` which takes an URL-string as argument and requests the corresponding file. The two URLs of interest are `ratings_url` and `movies_url`, which correspond to CSV-files mentioned above. The function should return the CSV-file appropriately broken up into lines, and the first line should be dropped (that is omit the header of the CSV-file). The result is a list of strings (the lines in the file). In case the url does not produce a file, return the empty list.

[1 Mark]

(2) Implement two functions that process the (broken up) CSV-files from (1). The `process_ratings` function filters out all ratings below 4 and returns a list of (userID, movieID) pairs. The `process_movies` function returns a list of (movieID, title) pairs. Note the input to these functions will be the output of the function `get_csv_url`.

[1 Mark]

(3) Implement a kind of grouping function that calculates a Map containing the userIDs and all the corresponding recommendations for this user (list of movieIDs). This should be implemented in a tail-recursive fashion using a Map as accumulator. This Map is set to `Map()` at the beginning of the calculation. For example

```
val lst = List(("1", "a"), ("1", "b"),
               ("2", "x"), ("3", "a"),
               ("2", "y"), ("3", "c"))
groupById(lst, Map())
```

returns the ratings map

```
Map(1 -> List(b, a), 2 -> List(y, x), 3 -> List(c, a)).
```

In which order the elements of the list are given is unimportant.

[1 Mark]

(4) Implement a function that takes a ratings map and a movieID as arguments. The function calculates all suggestions containing the given movie in its recommendations. It returns a list of all these recommendations (each of them is a list and needs to have the given movie deleted, otherwise it might happen we recommend the same movie "back"). For example for the Map from above and the movie `"y"` we obtain `List(List("x"))`, and for the movie `"a"` we get `List(List("b"), List("c"))`.

[1 Mark]

(5) Implement a suggestions function which takes a ratings map and a movieID as arguments. It calculates all the recommended movies sorted according to the most frequently suggested movie(s) sorted first. This function returns *all* suggested movieIDs as a list of strings.

[1 Mark]

(6) Implement then a recommendation function which generates a maximum of two most-suggested movies (as calculated above). But it returns the actual movie name, not the movieID. If fewer movies are recommended, then return fewer than two movie names.

[1 Mark]