Main Part 2 (Scala, 7 Marks)

You are asked to implement a Scala program for recommending movies according to a ratings list. This part is due on 15 January at 5pm.

Important

- 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

It should be understood that the work you submit represents your **own** effort! You have not copied from anyone else. 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 commandline. So there is no good reason to not download Scala, 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 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 `CW7b`. 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> CW7b.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 (7 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]

(7) Calculate the recommendations for all movies according to what the recommendations function in (6) produces (this can take a few seconds). Put all recommendations into a list (of strings) and count how often the strings occur in this list. This produces a list of string-int pairs, where the first component is the movie name and the second is the number of how many times the movie was recommended. Sort all the pairs according to the number of times they were recommended (most recommended movie name first).

[1 Mark]