Coursework 6 (Scala)

This coursework is about Scala and is worth 10%. The first and second part are due on 16 November at 11pm, and the third part on 21 December at 11pm. You are asked to implement three programs about list processing and recursion. The third part is more advanced and might include material you have not yet seen in the first lecture.

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 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.
- 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 360 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.

Part 1 (3 Marks)

This part is about recursion. You are asked to implement a Scala program that tests examples of the $3n + 1$-*conjecture*, also called *Collatz conjecture*. This conjecture can be described as follows: Start with any positive number n greater than 0:

[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!

- If n is even, divide it by 2 to obtain $n/2$.
- If n is odd, multiply it by 3 and add 1 to obtain $3n + 1$.
- Repeat this process and you will always end up with 1.

For example if you start with 6, respectively 9, you obtain the series

$$6, 3, 10, 5, 16, 8, 4, 2, 1 \quad (= 9 \text{ steps})$$
$$9, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1 \quad (= 20 \text{ steps})$$

As you can see, the numbers go up and down like a roller-coaster, but curiously they seem to always terminate in 1. The conjecture is that this will *always* happen for every number greater than 0.[2]

Tasks (file collatz.scala):

(1) You are asked to implement a recursive function that calculates the number of steps needed until a series ends with 1. In case of starting with 6, it takes 9 steps and in case of starting with 9, it takes 20 (see above). In order to try out this function with large numbers, you should use `Long` as argument type, instead of `Int`. You can assume this function will be called with numbers between 1 and 1 Million. [2 Marks]

(2) Write a second function that takes an upper bound as argument and calculates the steps for all numbers in the range from 1 up to this bound. It returns the maximum number of steps and the corresponding number that needs that many steps. More precisely it returns a pair where the first component is the number of steps and the second is the corresponding number. [1 Mark]

Test Data: Some test ranges are:

- 1 to 10 where 9 takes 20 steps
- 1 to 100 where 97 takes 119 steps,
- 1 to 1,000 where 871 takes 179 steps,
- 1 to 10,000 where 6,171 takes 262 steps,
- 1 to 100,000 where 77,031 takes 351 steps,

[2]While it is relatively easy to test this conjecture with particular numbers, it is an interesting open problem to *prove* that the conjecture is true for *all* numbers (> 0). Paul Erdös, a famous mathematician you might have hard about, said about this conjecture: "Mathematics may not be ready for such problems." and also offered a $500 cash prize for its solution. Jeffrey Lagarias, another mathematician, claimed that based only on known information about this problem, "this is an extraordinarily difficult problem, completely out of reach of present day mathematics." There is also a xkcd cartoon about this conjecture (click here). If you are able to solve this conjecture, you will definitely get famous.

- 1 to 1 Million where 837,799 takes 525 steps

Hints: useful math operators: `%` for modulo; useful functions: `(1 to 10)` for ranges, `.toInt`, `.toList` for conversions, `List(...).max` for the maximum of a list, `List(...).indexOf(...)` for the first index of a value in a list.

Part 2 (3 Marks)

This part is about web-scraping and list-processing in Scala. It uses online data about the per-capita alcohol consumption for each country (per year?), and a file containing the data about the population size of each country. From this data you are supposed to estimate how many litres of pure alcohol are consumed worldwide.

Tasks (file alcohol.scala):

(1) Write a function that given an URL requests a comma-separated value (CSV) list. We are interested in the list from the following URL

```
https://raw.githubusercontent.com/fivethirtyeight/data/
        master/alcohol-consumption/drinks.csv
```

Your function should take a string (the URL) as input, and produce a list of strings as output, where each string is one line in the corresponding CSV-list. This list from the URL above should contain 194 lines.

Write another function that can read the file `population.csv` from disk (the file is distributed with the coursework). This function should take a string as argument, the file name, and again return a list of strings corresponding to each entry in the CSV-list. For `population.csv`, this list should contain 216 lines. [1 Mark]

(2) Unfortunately, the CSV-lists contain a lot of "junk" and we need to extract the data that interests us. From the header of the alcohol list, you can see there are 5 columns

```
country (name),
beer_servings,
spirit_servings,
wine_servings,
total_litres_of_pure_alcohol
```

Write a function that extracts the data from the first column, the country name, and the data from the fifth column (converted into a `Double`). For this go through each line of the CSV-list (except the first line), use the `split(",")` function to divide each line into an array of 5 elements. Keep the data from the first and fifth element in these arrays.

Write another function that processes the population size list. This is already of the form country name and population size.[3] Again, split the strings according to the commas. However, this time generate a `Map` from country names to population sizes. [1 Mark]

(3) In (2) you generated the data about the alcohol consumption per capita for each country, and also the population size for each country. From this generate next a sorted(!) list of the overall alcohol consumption for each country. The list should be sorted from highest alcohol consumption to lowest. The difficulty is that the data is scraped off from "random" sources on the Internet and annoyingly the spelling of some country names does not always agree in both lists. For example the alcohol list contains `Bosnia-Herzegovina`, while the population writes this country as `Bosnia and Herzegovina`. In your sorted overall list include only countries from the alcohol list, whose exact country name is also in the population size list. This means you can ignore countries like Bosnia-Herzegovina from the overall alcohol consumption. There are 177 countries where the names agree. The UK is ranked 10th on this list by consuming 671,976,864 Litres of pure alcohol each year.

Finally, write another function that takes an integer, say `n`, as argument. You can assume this integer is between 0 and 177 (the number of countries in the sorted list above). The function should return a triple, where the first component is the sum of the alcohol consumption in all countries (on the list); the second component is the sum of the `n`-highest alcohol consumers on the list; and the third component is the percentage the `n`-highest alcohol consumers drink with respect to the the world consumption. You will see that according to our data, 164 countries (out of 177) gobble up 100% of the World alcohol consumption. [1 Mark]

Hints: useful list functions: `.drop(n)`, `.take(n)` for dropping or taking some elements in a list, `.getLines` for separating lines in a string; `.sortBy(_._2)` sorts a list of pairs according to the second elements in the pairs—the sorting is done from smallest to highest; useful `Map` functions: `.toMap` converts a list of pairs into a `Map`, `.isDefinedAt(k)` tests whether the map is defined at that key, that is would produce a result when called with this key; useful data functions: `Source.fromURL`, `Source.fromFile` for obtaining a webpage and reading a file.

[3] Your friendly lecturer already did the messy processing for you from the Worldbank database, see https://github.com/datasets/population/tree/master/data for the original.

Advanced Part 3 (4 Marks)

A purely fictional character named Mr T. Drumb inherited in 1978 approximately 200 Million Dollar from his father. Mr Drumb prides himself to be a brilliant business man because nowadays it is estimated he is 3 Billion Dollar worth (one is not sure, of course, because Mr Drumb refuses to make his tax records public).

Since the question about Mr Drumb's business acumen remains open, let's do a quick back-of-the-envelope calculation in Scala whether his claim has any merit. Let's suppose we are given \$100 in 1978 and we follow a really dumb investment strategy, namely:

- We blindly choose a portfolio of stocks, say some Blue-Chip stocks or some Real Estate stocks.
- If some of the stocks in our portfolio are traded in January of a year, we invest our money in equal amounts in each of these stocks. For example if we have \$100 and there are four stocks that are traded in our portfolio, we buy \$25 worth of stocks from each.
- Next year in January, we look how our stocks did, liquidate everything, and re-invest our (hopefully) increased money in again the stocks from our portfolio (there might be more stocks available, if companies from our portfolio got listed in that year, or less if some companies went bust or were de-listed).
- We do this for 39 years until January 2017 and check what would have become out of our \$100.

Until Yahoo was bought by Altaba this summer, historical stock market data for such back-of-the-envelope calculations was freely available online. Unfortuantely nowadays this kind of data is difficult to obtain, unless you are prepared to pay extortionate prices or be severely rate-limited. Therefore this coursework comes with a number of files containing CSV-lists with the historical stock prices for the companies in our portfolios. Use these files for the following tasks.

Tasks (file drumb.scala):

(1.a) Write a function `get_january_data` that takes a stock symbol and a year as arguments. The function reads the corresponding CSV-file and returns the list of strings that start with the given year (each line in the CSV-list is of the form `year-01-someday,someprice`).

(1.b) Write a function `get_first_price` that takes again a stock symbol and a year as arguments. It should return the first January price for the stock symbol in the given year. For this it uses the list of strings generated by `get_january_data`. A problem is that normally a stock exchange is not open on 1st of January, but depending on the day of the week on a later

day (maybe 3rd or 4th). The easiest way to solve this problem is to obtain the whole January data for a stock symbol and then select the earliest, or first, entry in this list. The stock price of this entry should be converted into a double. Such a price might not exist, in case the company does not exist in the given year. For example, if you query for Google in January of 1980, then clearly Google did not exist yet. Therefore you are asked to return a trade price as `Option[Double]`...`None` will be the value for when no price exists.

(1.c) Write a function `get_prices` that takes a portfolio (a list of stock symbols), a years range and gets all the first trading prices for each year in the range. You should organise this as a list of lists of `Option[Double]`'s. The inner lists are for all stock symbols from the portfolio and the outer list for the years. For example for Google and Apple in years 2010 (first line), 2011 (second line) and 2012 (third line) you obtain:

```
List(List(Some(311.349976), Some(27.505054)),
     List(Some(300.222351), Some(42.357094)),
     List(Some(330.555054), Some(52.852215)))
```

[2 Marks]

(2.a) Write a function that calculates the *change factor* (delta) for how a stock price has changed from one year to the next. This is only well-defined, if the corresponding company has been traded in both years. In this case you can calculate

$$\frac{price_{new} - price_{old}}{price_{old}}$$

If the change factor is defined, you should return it as `Some(change factor);` if not, you should return `None`.

(2.b) Write a function that calculates all change factors (deltas) for the prices we obtained under Task 1. For the running example of Google and Apple for the years 2010 to 2012 you should obtain 4 change factors:

```
List(List(Some(-0.03573992567129673), Some(0.5399749442411563))
      List(Some(0.10103412653643493), Some(0.2477771728154912)))
```

That means Google did a bit badly in 2010, while Apple did very well. Both did OK in 2011. Make sure you handle the cases where a company is not listed in a year. In such cases the change factor should be `None` (see 2.a).

[1 Mark]

(3.a) Write a function that calculates the "yield", or balance, for one year for our portfolio. This function takes the change factors, the starting balance and the year as arguments. If no company from our portfolio existed in that year, the balance is unchanged. Otherwise we invest in each existing company an equal amount of our balance. Using the change factors computed under Task 2, calculate the new balance. Say we had \$100 in 2010, we would have received in our running example involving Google and Apple:

```
$50 * -0.03573992567129673 + $50 * 0.5399749442411563
                                        = $25.21175092849298
```

as profit for that year, and our new balance for 2011 is \$125 when converted to a `Long`.

(3.b) Write a function that calculates the overall balance for a range of years where each year the yearly profit is compounded to the new balances and then re-invested into our portfolio.

[1 Mark]

Test Data: File `drumb.scala` contains two portfolios collected from the S&P 500, one for blue-chip companies, including Facebook, Amazon and Baidu; and another for listed real-estate companies, whose names I have never heard of. Following the dumb investment strategy from 1978 until 2017 would have turned a starting balance of \$100 into roughly \$30,895 for real estate and a whopping \$349,597 for blue chips. Note when comparing these results with your own calculations: there might be some small rounding errors, which when compounded lead to moderately different values.

Hints: useful string functions: `.startsWith(...)` for checking whether a string has a given prefix, `_ ++ _` for concatenating two strings; useful option functions: `.flatten` flattens a list of options such that it filters way all `None`'s, `Try(...) getOrElse ...` runs some code that might raise an exception—if yes, then a default value can be given; useful list functions: `.head` for obtaining the first element in a non-empty list, `.length` for the length of a list.

Moral: Reflecting on our assumptions, we are over-estimating our yield in many ways: first, who can know in 1978 about what will turn out to be a blue chip company. Also, since the portfolios are chosen from the current S&P 500, they do not include the myriad of companies that went bust or were de-listed over the years. So where does this leave our fictional character Mr T. Drumb? Well, given his inheritance, a really dumb investment strategy would have done equally well, if not much better.