Evil Wordle Game (Scala, 7 Marks)

"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do, it blows your whole leg off."
— Bjarne Stroustrup (creator of the C++ language)

You are asked to implement a Scala program for making the popular Wordle game as difficult as possible.

Important

- Make sure the files you submit can be processed by just calling `scala <<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 Github 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, 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 `M2`. If you want to find out what these functions produce for the list `List("a", "b", "b")`, you would type something like:

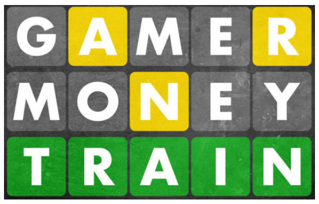
```
$ scala -cp wordle.jar
scala> val secretsURL =
     | """https://nms.kcl.ac.uk/christian.urban/wordle.txt"""

scala> M2.get_wordle_list(secretsURL)
val res0: List[String] = List(aahed, aalii, ...)
```

Hints

Useful data functions: `Source.fromURL`, `Source.fromFile` for obtaining a webpage and reading a file, `.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, `.groupBy` orders lists according to same elements .

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

You probably know the game of Wordle[2] where you are supposed to guess a five-letter word. The feedback for guesses can help with the next guess (green letters are correct, orange letters are present, but in the wrong place). For example:



The idea of the program to be implemented here is to make the Wordle game as evil as possible by finding words that are the most difficult to guess. A word list of five-letter words is available from

`https://nms.kcl.ac.uk/christian.urban/wordle.txt` (78 KByte)

In your program you need to download this list and implement some functions that in the end select the most difficult words (given an input from the user). If bandwidth is an issue for you, download the file locally, but in the submitted version use `Source.fromURL` instead of `Source.fromFile`.

Tasks

(1) Implement the function `get_wordle_list` which takes an URL-string as argument and requests the corresponding file. The function should return the word list appropriately broken up into lines. The result should be a list of strings (the lines in the file). In case the url does not produce a file, return the empty list. [0.5 Marks]

(2) Implement a polymorphic function `removeN`, which removes *n* occurrences of an element from a list (if this element is less than *n* times present, then remove all occurrences). For example

```
removeN(List(1,2,3,2,1), 3, 2)  => List(1, 2, 2, 1)
removeN(List(1,2,3,2,1), 2, 1)  => List(1, 3, 2, 1)
removeN(List(1,2,3,2,1), 2, 2)  => List(1, 3, 1)
removeN(List(1,2,3,2,1), 1, 1)  => List(2, 3, 2, 1)
removeN(List(1,2,3,2,1), 1, 3)  => List(2, 3, 2)
removeN(List(1,2,3,2,1), 0, 2)  => List(1, 2, 3, 2, 1)
```

Make sure you only remove at most *n* occurrences of the element from the list. This function should work for lists of integers but also lists of chars,

[2] https://en.wikipedia.org/wiki/Wordle

strings etc.

[0.5 Marks]

(3) Implement a function `score` that calculates the feedback for a word against a secret word using the rules of the Wordle game. The output of `score` should be a list of 5 elements of type `Tip` representing three outcomes: a letter in the correct position, a letter that is present, but not in the correct position and a letter that is absent. For example given the secret word ″chess″ the score for the word ″caves″ is

```
List(Correct, Absent, Absent, Present, Correct)
```

You have to be careful with multiple occurrences of letters. For example the secret ″chess″ with the guess ″swiss″ should produce

```
List(Absent, Absent, Absent, Correct, Correct)
```

even though the first ′s′ in ″swiss″ is present in the secret word, the ′s′ are already 'used up' by the two letters that are correct. To implement this you need to implement first a function `pool` which calculates all the letters in a secret that are not correct in a word. For example

```
  pool("chess", "caves")  => List(h, e, s)
  pool("chess", "swiss")  => List(c, h, e)
```

Now the helper function `aux` can analyse the arguments secret and word recursively letter-by-letter and decide: if the letters are the same, then return `Correct` for the corresponding position. If they are not the same, but the letter is in the pool, then return `Present` and also remove this letter from the pool in the next recursive call of `aux`. Otherwise return `Absent` for the corresponding position. The function `score` is a wrapper for the function `aux` calling `aux` with the appropriate arguments (recall what is calculated with `pool`). [2 Marks]

(4) Implement a function `eval` that gives an integer value to each of the `Tips` such that

$$\begin{aligned} \mathit{eval}\,(\mathit{Correct}) &\overset{\text{def}}{=} 10 \\ \mathit{eval}\,(\mathit{Present}) &\overset{\text{def}}{=} 1 \\ \mathit{eval}\,(\mathit{Absent}) &\overset{\text{def}}{=} 0 \end{aligned}$$

The function `iscore` then takes an output of `score` and sums up all corresponding values. For example for

```
  iscore("chess", "caves")  => 21
  iscore("chess", "swiss")  => 20
```

[0.5 Marks]

(5) The function `evil` takes a list of secrets (the list from Task 1) and a word as arguments, and calculates the list of words with the lowest score (remember we want to make the Wordle game as difficult as possible—therefore when the user gives us a word, we want to find the secrets that produce the lowest score). For this implement a helper function `lowest` that goes through the secrets one-by-one and calculates the score. The argument `current` is the score of the "currently" found secrets. When the function `lowest` is called for the first time then this will be set to the maximum integer value `Int.MaxValue`. The accumulator will be first empty. If a secret is found with the same score as `current` then this word is added to the accumulator. If the secret has a lower score, then the accumulator will be discarded and this secret will be the new accumulator. If the secret has a higher score, then it can be ignored. For example `evil` (the wrapper for `lowest`) generates

```
evil(secrets, "stent").length => 1907
evil(secrets, "hexes").length => 2966
evil(secrets, "horse").length => 1203
evil(secrets, "hoise").length => 971
evil(secrets, "house").length => 1228
```

where `secrets` is the list generated under Task 1. In all cases above the iscore of the resulting secrets is 0, but this does not need to be the case in general.

Note that the template gives as type for `evil`:

```
def evil(secrets: List[String], word: String) = ???
```

where the return type is left unspecified. This return type is not needed when functions are not recursive—`evil` is meant to be just a wrapper that calls `lowest` with appropriate default arguments and returns whatever `lowest` returns. Therefore a return type is not needed. But a slightly more accurate template definition for `evil` is:

```
def evil(secrets: List[String], word: String) : List[String] = ???
```

where also the return type is explicitly given.

[1.5 Marks]

(6) The secrets generated in Task 5 are the ones with the lowest score with respect to the word. You can think of these as the secrets that are furthest "away" from the given word. This is already quite evil for a secret word—remember we can choose a secret *after* a user has given a first word. Now we want to make it even more evil by choosing words that have the most obscure letters. For this we calculate the frequency of how many times certain letters occur in our secrets list (see Task 1). The *frequency* of the letter c, say, is given by the formula

$$freq(c) \stackrel{\text{def}}{=} 1 - \frac{\textit{number of occurrences of c}}{\textit{number of all letters}}$$

That means that letters that occur fewer times in our secrets have a higher frequency. For example the letter 'y' has the frequency 0.9680234350909651 while the much more often occurring letter 'e' has only 0.897286463151403 (all calculations should be done with Doubles).

The function `frequencies` should calculate the frequencies for all lowercase letters by generating a Map from letters (`Char`) to Doubles (frequencies).

[1 Mark]

(7) In this task we want to use the output of `evil`, rank each string in the generated set and then filter out the strings that are ranked highest (the ones with the most obscure letters). This list of strings often contains only a single word, but in general there might be more (see below). First implement a function `rank` that takes a frequency map (from 6) and a string as arguments and generates a rank by summing up all frequencies of the letters in the string. For example

```
rank(frequencies(secrets), "adobe") => 4.673604687018193
rank(frequencies(secrets), "gaffe") => 4.745205057045945
rank(frequencies(secrets), "fuzzy") => 4.898735738513722
```

The return type for `rank` is `Double`:

```
def rank(frqs: Map[Char, Double], s: String) : Double = ???
```

Finally, implement a function `ranked_evil` that selects from the output of `evil` the string(s) which are highest ranked in evilness.

```
ranked_evil(secrets, "abbey") => List(whizz)
ranked_evil(secrets, "afear") => List(buzzy)
ranked_evil(secrets, "zincy") => List(jugum)
ranked_evil(secrets, "zippy") => List(chuff)
```

This means if the user types in "abbey" then the most evil word to choose as secret is "whizz" (according to our calculations). This word has a zero `iscore` and the most obscure letters.

The return type for `ranked_evil` is `List[String]`:

```
def ranked_evil(secrets: List[String], word: String) :
                            List[String] = ???
```

[1 Mark]