Preliminary Part 1 (Scala, 3 Marks)

"The most effective debugging tool is still careful thought, coupled with judiciously placed print statements."
— Brian W. Kernighan, in Unix for Beginners (1979)

Important

- This part is about Scala. It is due on 20 November at 5pm and worth 3%. Any 1% you achieve in the preliminary part counts as your "weekly engagement".
- 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 `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 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++ assignments, the Scala assignments will work like this: you push your files to GitHub and receive (after sometimes a long delay) some automated feedback. In the end we take a snapshot of the submitted files and apply an automated marking script to them.

In addition, the Scala coursework comes with a reference implementation 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 collatz.jar` and then query any function from the template file. Say you want to find out what the functions `collatz` and `collatz_max` produce: for this you just need to prefix them with the object name `CW6a`. If you want to find out what these functions produce for the argument `6`, you would type something like:

```
$ scala -cp collatz.jar

scala> CW6a.collatz(6)
...
scala> CW6a.collatz_max(6)
...
```

Hints

For the Preliminary Part: useful math operators: `%` for modulo, `&` for bit-wise and; useful functions: `(1 to 10)` for ranges, `.toInt`, `.toList` for conversions, you can use `List(...).max` for the maximum of a list, `List(...).indexOf(...)` for the first index of a value in a list.

Preliminary Part (3 Marks, file collatz.scala)

This part is about function definitions and 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:

- 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, say, 6 and 9, you obtain the two *Collatz series*

$$\begin{array}{ll} 6, 3, 10, 5, 16, 8, 4, 2, 1 & (= 8 \text{ steps}) \\ 9, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1 & (= 19 \text{ steps}) \end{array}$$

As you can see, the numbers go up and down like a roller-coaster, but curiously they seem to always terminate in 1. Nobody knows why. The conjecture is that this will *always* happen for every number greater than 0.[2]

Tasks

(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 8 steps and in case of starting with 9, it takes 19 (see above). We assume it takes 0 steps, if we start with 1. 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. [1 Mark]

(2) Write a second function that takes an upper bound as an argument and calculates the steps for all numbers in the range from 1 up to this bound (the bound including). 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]

(3) Write a function that calculates *hard numbers* in the Collatz series—these are the last odd numbers just before a power of two is reached. For this,

[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 heard about, said about this conjecture: "Mathematics may not [yet] 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). If you are able to solve this conjecture, you will definitely get famous.

implement an *is-power-of-two* function which tests whether a number is a power of two. The easiest way to implement this is by using the bit-operator & of Scala. For a power of two, say n with $n > 0$, it holds that n & $(n - 1)$ is equal to zero. I let you think why this is the case.

The function *is-hard* calculates whether $3n + 1$ is a power of two. Finally the *last-odd* function calculates the last odd number before a power of 2 in the Collatz series. This means for example when starting with 9, we receive 5 as the last odd number. Surprisingly a lot of numbers have 5 as last-odd number. But for example for 113 we obtain 85, because of the series

$$113, 340, 170, \boxed{85}, 256, 128, 64, 32, 16, 8, 4, 2, 1$$

The *last-odd* function will only be called with numbers that are not powers of 2 themselves.

Test Data: Some test ranges and cases are:

- 1 to 10 where 9 takes 19 steps
- 1 to 100 where 97 takes 118 steps,
- 1 to 1,000 where 871 takes 178 steps,
- 1 to 10,000 where $6,171$ takes 261 steps,
- 1 to 100,000 where $77,031$ takes 350 steps,
- 1 to 1 Million where $837,799$ takes 524 steps
- 21 is the last odd number for 84
- 341 is the last odd number for 201, 604, 605 and 8600

Preliminary 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

- You are asked to implement a Scala program for measuring similarity in texts. The preliminary part is due on 27 November at 5pm and worth 3%. Any 1% you achieve in the preliminary part counts as your "weekly engagement".
- 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 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 `CW7a`. If you want to find out what these functions produce for the list `List("a", "b", "b")`, you would type something like:

```
$ scala -cp docdiff.jar

scala> CW7a.occurrences(List("a", "b", "b"))
...
```

Hints

For the Preliminary Part: 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.

Preliminary Part (3 Marks, file docdiff.scala)

It seems source code 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 this 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 (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]

Preliminary Part 3 (Scala, 3 Marks)

"[Google's MapReduce] abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages."
— Dean and Ghemawat, who designed this concept at Google

Important

- This part is about the shunting yard algorithm by Dijkstra. The preliminary part is due on ~~4 December~~ 11 December at 5pm and worth 3%. Any 1% you achieve in the preliminary part counts as your "weekly engagement".
- 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

This Scala assignment comes with two 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 postfix.jar` and then query any function from the `postfix.scala` file (similarly for file `postfix2.scala`). As usual you have to prefix the calls with `CW8a` and `CW8b`, respectively.

```
$ scala -cp postfix.jar

scala> CW8a.syard(CW8a.split("( 5 + 7 ) * 2"))
val res0: CW8a.Toks = List(5, 7, +, 2, *)
```

Hints

For the Preliminary Part: useful operations for determining whether a string is a number are `.forall` and `.isDigit`. One way to calculate the the power operation is to use `.pow` on `BigInts`, like `BigInt(n).pow(m).toInt`.

Preliminary Part (3 Marks, files postfix.scala, postfix2.scala)

The Shunting Yard Algorithm has been developed by Edsger Dijkstra, an influential computer scientist who developed many well-known algorithms. This algorithm transforms the usual infix notation of arithmetic expressions into the postfix notation, sometimes also called reverse Polish notation.

Why on Earth do people use the postfix notation? It is much more convenient to work with the usual infix notation for arithmetic expressions. Most modern pocket calculators (as opposed to the ones used 20 years ago) understand infix notation. So why on Earth? ...Well, many computers under the hood, even nowadays, use postfix notation extensively. For example if you give to the Java compiler the expression $1 + ((2 * 3) + (4 - 3))$, it will generate the Java Byte code

```
ldc 1
ldc 2
ldc 3
imul
ldc 4
ldc 3
isub
iadd
iadd
```

where the command `ldc` loads a constant onto the stack, and `imul`, `isub` and `iadd` are commands acting on the stack. Clearly this is the arithmetic expression in postfix notation.

The shunting yard algorithm processes an input token list using an operator stack and an output list. The input consists of numbers, operators ($+$, $-$, $*$, $/$) and parentheses, and for the purpose of the assignment we assume the input is always a well-formed expression in infix notation. The calculation in the shunting yard algorithm uses information about the precedences of the operators (given in the template file). The algorithm processes the input token list as follows:

- If there is a number as input token, then this token is transferred directly to the output list. Then the rest of the input is processed.
- If there is an operator as input token, then you need to check what is on top of the operator stack. If there are operators with a higher or equal precedence, these operators are first popped off from the stack and moved to the output list. Then the operator from the input is pushed onto the stack and the rest of the input is processed.
- If the input is a left-parenthesis, you push it on to the stack and continue processing the input.
- If the input is a right-parenthesis, then you pop off all operators from the stack to the output list until you reach the left-parenthesis. Then you discharge the (and) from the input and stack, and continue processing the input list.
- If the input is empty, then you move all remaining operators from the stack to the output list.

Tasks (file postfix.scala)

(1) Implement the shunting yard algorithm described above. The function, called `syard`, takes a list of tokens as first argument. The second and third arguments are the stack and output list represented as Scala lists. The most convenient way to implement this algorithm is to analyse what the input list, stack and output list look like in each step using pattern-matching. The algorithm transforms for example the input

```
List(3, +, 4, *, (, 2, -, 1, ))
```

into the postfix output

```
List(3, 4, 2, 1, -, *, +)
```

You can assume the input list is always a list representing a well-formed infix arithmetic expression. [1 Mark]

(2) Implement a compute function that takes a postfix expression as argument and evaluates it generating an integer as result. It uses a stack to evaluate the postfix expression. The operators $+$, $-$, $*$ are as usual; / is division on integers, for example $7/3 = 2$. [1 Mark]

Task (file postfix2.scala)

(3/4) Extend the code in (1) and (2) to include the power operator. This requires proper account of associativity of the operators. The power operator is right-associative, whereas the other operators are left-associative. Left-associative operators are popped off if the precedence is bigger or equal, while right-associative operators are only popped off if the precedence is bigger. [1 Marks]