

Core 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

- 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 `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.

¹All 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++ 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 `C1`. If you want to find out what these functions produce for the argument `6`, you would type something like:

```
$ scala -cp collatz.jar  
  
scala> C1.collatz(6)  
...  
scala> C1.collatz_max(6)  
...
```

Hints

For the Core Part 1: 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.

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, 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. [1 Mark]

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