Programming Languages and Paradigms
COMP 302, Winter 2018
Assignment 1

Due date: Monday, February 5, 2018
6pm

This assignment focuses on basic use of Scala and functional programming. Your code must run without error or modification using Scala 2.12.

Note that you must follow the given naming, input, and output requirements precisely. All code should be well-commented, in a professional style, with appropriate variables names, indenting (uses spaces and avoid tabs), etc. **The onus is on you to ensure your code is clear and readable. Marks will be very generously deducted for bad style, lack of clarity, or failing to follow the required instructions.**

Your code should endeavour to follow a pure functional programming style. In particular, and unless specifically stated otherwise, all data types must be *immutable*, and *data may not be modified once assigned or bound*. Note that this means you may not use `var` declarations, `while` or `do`-loops, `ArrayBuffers` or other mutable structures, nor may you reassign `Array` element values after creation.

1. Euler showed that the infinite series $\sum_{n=1}^{\infty} \frac{1}{n^2}$ converges to $\frac{\pi^2}{6}$.   **4**

   Use this property to approximate the value of π. Define a function `euler` which receives an `Int` argument and returns an approximation of π using the first k terms of the series, as a `Double` result.

   Note that this function does not converge very fast, and you may need a fairly large value of k to get many digits of π correct. If you are using a command-line scala invocation you can increase the stack size by passing a stack size option to the underlying JVM, like "`scala -J-Xss4M yourprogram.scala`".

2. Lists can be understood as recursive structures, as constructed by Scala's right-associative "`::`" operator.   **4** For example, the list "`1::2::3::4::Nil`" is a simplified view of how it is actually constructed, as "`1::(2::(3::(4::Nil)))`".

   Define a function `show` which accepts a `List` of `Any` type, and returns a `String` representation that reflects its actual construction. For example,

   ```
   show(1::2::3::4::Nil) // returns:  1::(2::(3::(4::Nil)))
   show(1::Nil) // returns:  1::Nil
   show(Nil) // returns:  Nil
   ```

3. A Fibonacci function is easy to define recursively, but done naively it is inefficient, re-computing the   **5** same Fibonacci value many times as it recursively descends down both recursive calls. For example, given a definition,

   ```
   def fib(n:Int):Int = { if (n<2) 1 else fib(n-1)+fib(n-2) }
   ```

   calling `fib(5)` ends up computing `fib(3)` twice, `fib(2)` thrice, and the base case of `fib(1)` or `fib(0)` 8 times during the course of its recursive unfolding.

   Come up with a more efficient version, `fib2` (with the same type signature) that only ever computes a given Fibonacci number once.

   Note that you can time your code to observe the difference. For this you can use,
   ```
   import System.nanoTime
   def profile[R](code:  => R, t:  Long = nanoTime) = (code, nanoTime - t)
   ```

which executes the given code, returning a pair of values consisting of the result, and the time taken (in nanoseconds). For example, on my systems, executing "`profile { fib(33) }`" results in "(5702887,32452005)" (with significant variance in the timing)

Your `fib2` version should be at least an order-of-magnitude faster (for non-trivial inputs).

nb: Do not actually include any timing code or executions in your solution, just provide the improved function definition.

4. Lex Luther stole too many cakes. He actually only wants to have stolen exactly 40, but ended up with **5** more, and so has come up with a complicated scheme for returning some of them. Having stolen n cakes, he decided he can return cakes subject to the following, recursively applied conditions:

If n is even, he may return $n/2$ cakes.
If n is divisible by 3 or 4, multiple together the last 2 digits of n, and he may return that many cakes.
If n is divisible by 5, he can give back exactly 40 cakes

This approach may or may not allow him to reduce the number of cakes he has to 40. Come up with a Scala function, `cakes` which accepts an `Int` argument n and returns a `Boolean` value, true if n cakes can be reduced to 40, and false otherwise.

Your function should give a correct answer given any non-negative integer. To help test it, note that:

```
for ( i <- 38 to 100 if (cakes(i)) ) yield i
```

should return a collection consisting of: `40, 64, 75, 80`.

Of course other values will be tested too.

5. Recall (or discover) that a simple polygon is a closed sequence of $n > 2$ non-intersecting, connected **6** straight-line segments. We can represent it as a series of ordered pairs of numbers (coordinates), moving either clockwise or counter-clockwise around the boundary. For example the polygon,
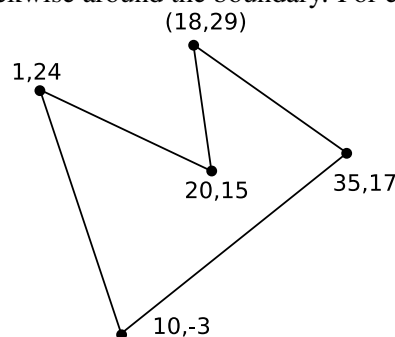
can be represented in a text file as n pairs of numbers,

1 24
20 15
18 29
35 17
10 -3

The "Shoelace Formula" is a well known technique for computing the area of any simple polygon. It computes the area from the formula,

$$\text{Area} = \frac{1}{2} \left| \left(\sum_{i=1}^{n-1} x_i y_{i+1} \right) + x_n y_1 - \left(\sum_{i=1}^{n-1} x_{i+1} y_i \right) - x_1 y_n \right|$$

Give a scala function that computes the area of a polygon. Your `shoelace` function should receive the name of file containing coordinates as a `String`, and return the area as a `Double` value.

You can assume the coordinate file contains nothing other than the coordinates, although the amount of "whitespace" (spaces, tabs, newlines, carriage-returns) between numbers may vary (but they will be separated by some).

6. Scala has many ways of doing simple processing from one String into another; making a String upper-case or lower-case is trivial, you can reverse, sort, filter characters, and many other operations. We can chain these operations together fairly easily, but it can turn into a long sequence of syntax. **8**

    The goal here is to construct a function that can assemble multiple operations together, returning a new function that internally applies several String operations in sequence to its input.

    Define a function `stringPipeline`. It should accept a single argument, a `String` consisting only of characters in the set {'U','T','l','r','s','*'}. It returns a function that accepts a `String` and returns a `String`.

    The input to your `stringPipeline` function encodes the operations you want to perform.

    'U' convert the string to upper-case
    'l' convert the string to lower-case
    'T' convert the string to title-case (capitalize the first non-space character, and the first character of each word, where words are sequences of characters separated by one or more spaces)
    'r' reverse the string
    's' sort the characters
    '*' delete all space-characters

    For example, an input of "Ts*" means to convert it to title case, sort it, and then delete all spaces.

    The returned function encapsulates the multiple operations given by the input string. It can then be applied to a string, performing all the operations requested, in order. For example,

    ```
    val p = stringPipeline("Ts*")
    ```

    gives a function that will do the 3 operations mentioned above. Thus,

    ```
    p(" abc def ghi x !  1") // result:  !1ADGXbcefhi
    ```

What to hand in

Submit your assignment to *MyCourses*. Note that clock accuracy varies, and late assignments will not be accepted without a medical note: **do not wait until the last minute**. Assignments must be submitted on the due date **before 6pm**.

For each question n, include a file `qn.scala` with the source code of your answer. Do not define a main.

This assignment is worth 10% of your final grade. **32**