built on 2023/09/11 at 09:04:04 ***due:*** *wed sep 27 @ 11:59pm*

This assignment contains only coding problems. The aim of this assignment is to help familiarize yourself with Scala's programming environment and begin to program in a functional style. We're providing stubs for you; please download the starter code from the course website.

You will zip all your work in one zip file and upload it to Canvas before the due date.

IMPORTANT: Your implementation must strictly follow the functional style. As a rule of thumb, you cannot use features other than what we have done in class so far. In particular, this means, no loops, no mutable variables (cannot use `var`).

Task 1: Aggregate: Min, Mean, Median (4 points)

For this task, save your code in `Aggregate.scala`

Your goal is to define the following *three* functions

```
def myMin(p: Double, q: Double, r: Double): Double
def myMean(p: Double, q: Double, r: Double): Double
def myMed(p: Double, q: Double, r: Double): Double
```

where

- `myMin(p, q, r)` takes 3 numbers and returns the minimum of the three numbers. For example, `myMin(3.0,1.0,9.0)` should return 1.0.
- `myMean(p, q, r)` takes 3 numbers and returns the average of the three numbers. You should recall that the average of p, q, and r is simply $\frac{1}{3}(p+q+r)$. Hence, as an example, `myMean(3, 7, 4)` should return $4.6666\cdots$.
- `myMed(p, q, r)` takes 3 numbers and returns the median of the three numbers. Remember that the median of three numbers is the number where one other number is smaller than it and one other number is larger than it. For instance, `myMed(4,1,5)` should return 4. Also, `myMed(13,5.0,12)` should return 12.

Task 2: Read Aloud (4 points)

For this task, save your code in `Aloud.scala`

When we read aloud the list `[1,1,1,1,4,4,4]`, we most likely say four 1s and three 4s, instead of uttering each number one by one. This simple observation inspires the function you are about to implement. You're to write a function `readAloud(xs: List[Int]): List[Int]` that takes as input a list of integers (positive and negative) and returns a list of integers constructed using the following "read-aloud" method:

> Consider the first number, say m. See how many times this number is repeated consecutively. If it is repeated k times in a row, it gives rise to two entries in the output list: first the number k, then the number m. (This is similar to how we say "four 2s" when we see `[2,2,2,2]`.) Then we move on to the next number after this run of m. Repeat the process until every number in the list is considered.

The process is perhaps best understood by looking at a few examples:

- `readAloud(List())` should return `List()`.
- `readAloud(List(1,1,1))` should return `List(3,1)`.

- `readAloud(List(-1,2,7))` should return `List(1,-1,1,2,1,7)`.
- `readAloud(List(3,3,8,-10,-10,-10))` should return `List(2,3,1,8,3,-10)`.
- `readAloud(List(3,3,1,1,3,1,1))` should return `List(2,3,2,1,1,3,2,1)`.

Task 3: Happy Numbers (Again?) (8 points)

For this task, save your code in `Happy.scala`

Happy numbers are a nice mathematical concept. Just as only certain numbers are prime, only certain numbers are happy—under the mathematical definition of happiness. To test whether a number is happy, we can follow a simple step-by-step procedure:

(1) Write that number down
(2) Stop if that number is either 1 or 4.
(3) Cross out the number you have now. Write down instead the sum of the squares of its digits.
(4) Repeat Step (2)

When you stop, if the number you have is 1, the initial number is *happy*. If the number you have is 4, the initial number is *sad*. There are only two possible outcomes, happy or sad.

By this definition, 19 is a happy number because $1^2 + 9^2 = 82$, then $8^2 + 2^2 = 68$, then $6^2 + 8^2 = 100$, and then $1^2 + 0^2 + 0^2 = 1$. However, 145 is sad because $1^2 + 4^2 + 5^2 = 42$, $4^2 + 2^2 = 20$, and $2^2 + 0^2 = 4$.

Subtask I: First, you'll implement a function `def sumOfDigitsSquared(n: Int): Int` that takes a positive number `n` and sum the squares of its digits. For example,

- `sumOfDigitsSquared(7)` should return `49`
- `sumOfDigitsSquared(145)` should return `42` (i.e., `1**2 + 4**2 + 5**2 = 1 + 16 + 25`)
- `sumOfDigitsSquared(199)` should return `163` (i.e., `1**2 + 9**2 + 9**2 = 1 + 81 + 81`)

Subtask II: Then, you'll write a function `def isHappy(n: Int): Boolean` that takes as input a positive number `n` and tests if `n` is happy. You may wish to use what you wrote in the previous subtask.

- `isHappy(100)` should return `true`
- `isHappy(111)` should return `false`
- `isHappy(1234)` should return `false`
- `isHappy(989)` should return `true`

Subtask III: The k-th happy number is the k-th smallest happy number. This means, the 1st happy number is the smallest happy number, which is 1. The 2nd happy number is the second smallest happy number, which is 7. Below is a list of the first few happy numbers:

> 1, 7, 10, 13, 19, 23, 28, 31, 32, 44, 49, 68, 70, 79, 82, 86, 91, 94, 97, 100, 103, 109, 129, 130, 133, 139, 167, 176, 188, 190, ...

Implement a function `def kThHappy(k: Int): Int` that returns the k-th happy number. For example:

- `kThHappy(1)` returns 1
- `kThHappy(3)` returns 10
- `kThHappy(11)` returns 49
- `kThHappy(19)` returns 97

Pro Tips: Use what you have already written. Don't repeat yourself.

Task 4: Roman Numerals (4 points)

For this task, save your code in `Roman.scala`

You surely have encountered Roman numerals: I, II, III, XXII, MCMXLVI, etc. They are everywhere. Roman numerals are represented by repeating and combing the following seven characters: `I` = 1, `V` = 5, `X` = 10, `L` = 50, `C` = 100, `D` = 500, `M` = 1000. To understand how they must be composed, we borrow the following excerpt from *Dive Into Python*:

- Characters are additive. `I` is 1, `II` is 2, and `III` is 3. `VI` is 6 ("5 and 1"), `VII` is 7, and `VIII` is 8.
- The tens characters (`I`, `X`, `C`, and `M`) can be repeated up to three times. At 4, you need to subtract from the next highest fives character. You can't represent 4 as `IIII`; instead, it is represented as `IV` ("1 less than 5"). The number 40 is written as `XL` (10 less than 50), 41 as `XLI`, 42 as `XLII`, 43 as `XLIII`, and then 44 as `XLIV` (10 less than 50, then 1 less than 5).
- Similarly, at 9, you need to subtract from the next highest tens character: 8 is `VIII`, but 9 is `IX` (1 less than 10), not `VIIII` (since the `I` character cannot be repeated four times). The number 90 is `XC`, 900 is `CM`.
- The fives characters cannot be repeated. The number 10 is always represented as X, never as `VV`. The number 100 is always `C`, never `LL`.
- Roman numerals are always written highest to lowest, and read left to right, so the order the of characters matters very much. `DC` is 600; `CD` is a completely different number (400, 100 less than 500). `CI` is 101; `IC` is *not* a valid Roman numeral (because you can't subtract 1 directly from 100; you would need to write it as `XCIX`, for 10 less than 100, then 1 less than 10).

YOUR TASK: Implement a function `def toRoman(n: Int): String` that takes an integer n ($1 \leq n < 4,000$) and returns a string that represents the number n in Roman numerals.

Despite the complexity this problem may seem at first, there are nice (and not tedious) ways to implement the logic to convert an integer into a Roman numeral using only the features we have learned so far. You should not write more than 20 lines of code. The model solution contains about 15 lines of code.

Task 5: Turn It Around (4 points)

For this task, save your code in `TurnIt.scala`

Using the only collection datatype we know so far (the `List`), we can represent a 2-dimensional array as a list of lists, i.e. `List[List[T]]`. In this problem, you will compute what is known as the "transpose" of a given array. If `A[][]` is an m-by-n array, the transpose of `A` is the an array `B[][]`, which has dimension n-by-m, where `B[i][j] = A[j][i]`.

You will implement a function `def transpose(A: List[List[Int]]): List[List[Int]]` that takes in a 2-dimensional array `A` and returns the transpose of `A`.

Your function must not be excessively slow. That is, transposing a 1000-by-1000 array should take less than a few seconds. **Do not use the built-in transpose.**

Task 6: Zombies, Revisited (4 points)

For this task, save your code in `Zombies.scala`

We want you to implement the `merge` routine for merge sort. But to spice things up a little, we'll bring back your fond memories from Data Structures.

In a remote village known as Salaya, zombies and humans have lived happily together for many decades. In fact, no one can quite tell zombies and humans apart. However, when these "people" line

up in a single row, all sorts of trouble ensue, including this weird phenomenon: human beings will line themselves up from tall to short, but zombines act erratically.

In particular, if `line` is an array of heights of the population of this village, we would expect that `line`$[i] \geq$ `line`$[j]$ for $i \leq j$. But this simply isn't true in many cases especially with zombies around. Hence, one nobleman—or is he a zombie?—came to you for help: he wants to know how many pairs of his people violate this social norm.

Your Task: Write a function `def countBad(hs: List[Int]): Int` that takes a list of n numbers and returns the number of pairs $0 \leq i < j < n$ such that `hs`$[i] >$ `hs`$[j]$ (i.e., the number of pairs that violate the social norm).

For example:

- `countBad(List(35, 22, 10)) == 0`
- `countBad(List(3,1,4,2)) == 3`
- `countBad(List(5,4,11,7)) == 4`
- `countBad(List(1, 7, 22, 13, 25, 4, 10, 34, 16, 28, 19, 31)) == 49`

Performance Expectations: We expect your code to run in at most $O(n \log n)$ time, where n is the length of the input array. Here are some tips to get started:

- Eventually you'll want to implement a merge-sort-like algorithm. To begin, ask yourself how can you split a front-access list in half (by size).
- How do you merge? Remember in a cons-list (a front-access list), the only thing you can access cheaply is the head.

Task 7: All Permutations (4 points)

For this task, save your code in `AllPerm.scala`

You will implement a *recursive* function `allPerm(n: Int): List[List[Int]]` that takes an integer n > 0 and returns a list containing all the permutations of $1, 2, 3, \ldots, n$. Each permutation is represented as a list as well. For example:

```
allPerm(1) == List(List(1))
allPerm(2) == List(List1,2), List(2,1))
allPerm(3) == List(List(1,2,3), List(1,3,2), List(2,1,3),
                   List(2,3,1), List(3,1,2), List(3,2,1))
```

Notice that the sample output list is "sorted" lexicographically, but your code's output can be arbitrarily ordered. The running time should be $O(n \cdot n!)$.

Hints: Consider how we can use `allPerm(2)` to create the answer for `allPerm(3)`. Perhaps the following diagram will help (pay close attention to the color coding and numbers in boldface fonts):

`allPerm(2)` == {(1,2), (2,1)}

`allPerm(3)` == {(**3**,1,2), (1,**3**,2), (1,2,**3**), (**3**,2,1), (2,**3**,1), (2,1,**3**)}

`allPerm(4)` == {(**4**,3,1,2), (3,**4**,1,2), (3,1,**4**,2), (3,1,2,**4**),
(**4**,1,3,2), (1,**4**,3,2), (1,3,**4**,2), (1,3,2,**4**),
(**4**,1,2,3), (1,**4**,2,3), (1,2,**4**,3), (1,2,3,**4**),
(**4**,3,2,1), (3,**4**,2,1), (3,2,**4**,1), (3,2,1,**4**),
(4,2,3,1), (2,4,3,1), (2,3,4,1), (2,3,1,4),
(**4**,2,1,3), (2,**4**,1,3), (2,1,**4**,3), (2,1,3,**4**)}