Project 1: Scala Warm-up

Notes

The problems in this assignment are based on those available at Ninety-Nine Scala Problems.

Objectives

- Practice writing recursive functions.
- Practice what you have learned so far about programming in Scala.
- Practice working with lists and trees in addition to applying pattern matching.
- Learn about breaking down a big problem into smaller parts.

Code

You will need to add your code to the provided project template available at: `https://github.itu.dk/omsh/BDM2019/tree/master/Projects/BIDMT_F19_P1`
Note: Make sure that you do not change the signature of any of the functions given to you. You can certainly add your helper functions, but we will test your code by calling the ones in the given code template.

Part 1: Flatten a Nested List Structure

Input: A nested list structure is a list that contain integers or another nested list structure. Example: List(List(1, 1), 2, List(3, List(5, 8))).
Requirement: You are required to write the code for the function $flatten$ that takes a nested list structure and returns all the elements it. When applying flatten for the example above, the output should be: List(1, 1, 2, 3, 5, 8). Therefore, you need to write the code for the following function:

```
def flatten (nestedList : List[Any]) : List[Any] = ???
```

Note: You should not use list flat or flatmap provided with Scala.

Part 2: Sorting Lists

Input: A list of lists, where each sub list is composed of characters. Example: val listOfLists = `List(List('a', 'b', 'c'), List('d', 'e'), List('f', 'g', 'h'), List('d', 'e'), List('i', 'j', 'k', 'l'), List('m', 'n'), List('o'))`
Requirement: You are required to sort the input list in the following two approaches:

- **Sort based on the length of the sublist:** implement the function *sortListLength* that sorts the sublists of the input list in an ascending order based on the lengths of these sublists. For example the sorted order of $listOfLists$ is: `List(List(o), List(d, e), List(d, e), List(m, n), List(a, b, c), List(f, g, h), List(i, j, k, l))`

- **Sort based on the frequency of the lengths of the sublists:** implement the function *sortListFreq* that sorts the sublists of the input list in an ascending order based on the frequency of the lengths of these sublists. If sublists of length x rarely appear in the list and sublists of length y are very common, then the former sublists should appear before the latter sublists in the sorted version. For example, the sorted list of $listOfLists$ will be `List(List(i, j, k, l), List(o), List(a, b, c), List(f, g, h), List(d, e), List(d, e), List(m, n))`. Lists of length 4 and 1 appeared in the first and second places because they appear only once in $listOfLists$. Lists of length 3 appeared in the third and fourth places because they appear twice in $listOfLists$. Lists of length 2 appeared in the fifth, sixth, and seventh places because they appear three times in $listOfLists$.

Therefore, you need to complete the following two functions:

```
def sortListLength(listOfLists: List[List[Char]]): List[List[Char]] = ???

def sortListFreq(listOfLists: List[List[Char]]): List[List[Char]] = ???
```

Part III: Huffman Coding

Overview

Huffman coding is an approach for lossless data compression of strings. Standard coding approaches such as ASCII assign a code of a fixed length to all characters. However, Huffman coding assigns codes with variable lengths to different characters according to the frequency of their occurrences. Assigning shorter codes to the most frequent characters in a message leads to shorter (compressed) messages.
A Huffman code can be represented as a binary tree, which has the following properties:

- Leaves of the tree are the symbols that are encoded. Each leaf node has a weight that represents its relative frequency.
- Each no-leaf node is a set of all the symbols in the subtree rooted by this node and the weight of all the frequencies of the leaves in that sub-tree.

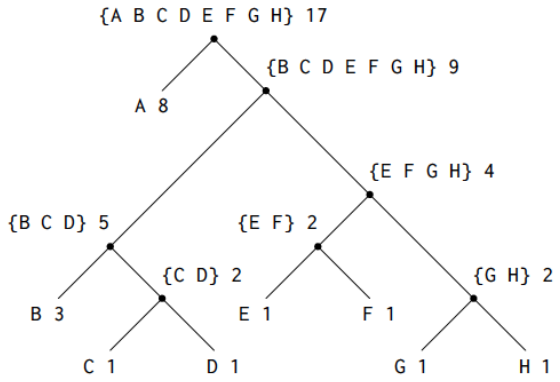

Figure 1: Huffman tree example.

Figure 1 shows an example Huffman coding tree for letters A trough H. The tree shows that A has a relative frequency 8, B has a relative frequency 3, and all the remaining letters have relative frequencies of 1. For a given tree, we can do the following:

- **Encode a text:** For each letter in that text, we start at the root of the tree and move down until the lead representing this letter is reached. Each time, we move left, a 0 is added to the code, and each time, we move right, a 1 is added to the code.
- **Decode a text:** We start at the root of the tree, consume a code entry (0 or 1) to decide to move left or right in the tree. When a leaf is reached, a letter is generated in the output decoded message and we start over from the root of the tree.

More details about Huffman coding are presented in Section 2.3.4 in [1]. Next, the structure of the tree and the three main functions that you are required to build are described.

Huffman Coding Representation Tree

In the code provided to you, the Huffman tree is represented as follows:

```
abstract class HuffmanCodingTree
class HuffmanCodingTreeLeaf(symbol: Char, weight: Int)
     extends HuffmanCodingTree
class HuffmanCodingTreeNonLeaf(symbols: List[Char], weight: Int,
     left: HuffmanCodingTree, right: HuffmanCodingTree)
     extends HuffmanCodingTree
```

Building the Huffman Coding Tree

The next step is to construct a Huffman tree given a text represented as a list of characters. The signature of the function is as follows:

```
def createHuffmanTree(text: List[Char]):HuffmanCodingTree = ???
```

You will need to follow these steps to construct the tree:

- Generate a list of all the characters appearing in the $text$ and pair each character with its frequency based on how many times it appears in the $text$. Therefore, you need to write code for this function:

  ```
  def extractCharFrequencies(text: List[Char]):  List[(Char,Int)]= ???
  ```

- From the above created list, generate a list of $HuffmanCodingTree$ nodes. Initially, this list contains a leaf tree node for each character in the $text$. Therefore, you need to write code for this function:

  ```
  def makeTreeLeaves(charFreqs : List[(Char,Int)])
                : List[HuffmanCodingTree] = ???
  ```

- Repeat the following **until there is only one tree node in the list**: (1) find two nodes with the list that has the lowest frequencies; (2) merge these two nodes into one tree by creating a new tree node with these nodes are its children; (3) remove the two nodes from the list and add the newly generated tree node to it. We suggest that you complete and use the following functions:

```
    def makeNonLeaf(a : HuffmanCodingTree, b : HuffmanCodingTree)
                  : HuffmanCodingTree = ???

    def insertAsc(treeNode : HuffmanCodingTree,
                      listTreeNodes : List[HuffmanCodingTree])
                  : List[HuffmanCodingTree] = ???

    def generateTree(treeLeaves: List[HuffmanCodingTree])
                  : HuffmanCodingTree = ???
```

An example of the construction of the tree in Figure 1 is as follows:

```
Initial leaves {(A 8) (B 3) (C 1) (D 1) (E 1) (F 1) (G 1) (H 1)}
       Merge {(A 8) (B 3) ({C D} 2) (E 1) (F 1) (G 1) (H 1)}
       Merge {(A 8) (B 3) ({C D} 2) ({E F} 2) (G 1) (H 1)}
       Merge {(A 8) (B 3) ({C D} 2) ({E F} 2) ({G H} 2)}
       Merge {(A 8) (B 3) ({C D} 2) ({E F G H} 4)}
       Merge {(A 8) ({B C D} 5) ({E F G H} 4)}
       Merge {(A 8) ({B C D E F G H} 9)}
     Final merge {({A B C D E F G H} 17)}
```

Encoding

Given a Huffman tree and a message, you are required to write a function that encodes this message into a code of zeros and ones. The signature of the function is as follows:

```
def encode(tree: HuffmanCodingTree, code: List[Char]): List[Int]
```

You will probably need to write the following function and call for each character in the input message:

```
def encodeChar(tree: HuffmanCodingTree, c:Char): List[Int] = ???
```

Decoding

Given a Huffman tree and a code of zeros and ones, you are required to write a function that decodes this code into a message represented as list of characters. The signature of the function is as follows:

```
def decode(tree: HuffmanCodingTree, code: List[Int]): List[Char] = ???
```

You will probably need to write the following function and call for each code of a character in the input:

```
def getCharCode(tree:HuffmanCodingTree , code: List[Int]): (Char, List[Int]) = ???
```

Deliverable

You need to deliver the following:

- Your code. **Only the scala files.**
- A report (or a link to a short video that you have prepared) that includes
  - a brief description of your solutions for the three parts.
  - the test cases (test units or calls from the main function) that you ran to test each problem. Please show the inputs and the outputs that you tested your code with.
  - any choices that you made to improve the performance of your applications.
  - the status of the submitted code. What is running, what went wrong, and if there are any errors or problems that you have experienced.
  - If you are submitting a video, include in that video a demo of how you are running your code.

References

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and interpretation of computer programs*. Justin Kelly, 1996.