Assignment 2

In this assignment we get a bit more practice breaking down a problem into smaller parts and using the various Haskell facilities to do the appropriate transformations.

The problem we will solve is a voting system called Borda count. The situation is as follows:

- We have a number of candidates.
- Each voter has to order all the candidates in some preference order, with highest preference first.
- We assign decreasing scores to the candidates for that vote, counting down to 1. For example if the voter voted for candidates A, B, C in that order, then A gets score 3, B gets score 2 and A gets score 1.
- We then add up the scores for each candidate, across all the different votes.
- The candidates are then ranked based on those points.

We will write the code that does this. For example our code will be given an input such as:

```
allVotes = [
   ["Peter", "Debra", "Oliver", "John"],
   ["Peter", "Oliver", "John", "Debra"],
   ["Oliver", "Debra", "John", "Peter"],
   ["John", "Oliver", "Debra", "Peter"],
   ["Debra", "John", "Oliver", "Peter"]]

getResults allVotes
-- Will produce the string:
-- Oliver          14
-- Debra           13
-- John            12
-- Peter           11
```

Quick note: To print a string on the interactive console, use putStr or putStrLn, for example:

```
putStrLn "Hello␣there!"
```

Use this only for diagnostic purposes. Your functions should simply be returning strings and not printing directly.

Here is some start code. You may want to add your own tests.

```
module Voting where

import Test.HUnit
import Data.List (sortBy, nub, sort)

type Candidate = String
```

```haskell
type Vote = [Candidate]

allVotes :: [Vote]
allVotes = [
   ["Peter", "Debra", "Oliver", "John"],
   ["Peter", "Oliver", "John", "Debra"],
   ["Oliver", "Debra", "John", "Peter"],
   ["John", "Oliver", "Debra", "Peter"],
   ["Debra", "John", "Oliver", "Peter"]]


-- Function provided for you
getResults :: [Vote] -> IO ()
getResults votes = sequence_ [putStrLn s | s <- formatVotes votes]
   where formatVotes = formatAll . sortedCounts . totalCounts

tests = TestList [
   TestCase $ assertEqual "downToN" [5, 4, 3, 2, 1] (downFromN 5),
   TestCase $ assertEqual "withRanks"
         [("A", 3), ("B", 2), ("C", 1)]
         (withRanks ["A", "B", "C"]),
   TestCase $ assertEqual "allRanks"
         (sort [("A", 3), ("B", 2), ("C", 1), ("B", 3), ("A", 2), ("C", 1)])
         (sort (allRanks [["A", "B", "C"], ["B", "A", "C"]])),
   TestCase $ assertEqual "totalRank"
         5
         (totalRank "B" [("B", 2), ("C", 1), ("B", 3)]),
   TestCase $ assertEqual "allCandidates"
         ["A", "B"]
         (sort (allCandidates [("B", 2), ("B", 3), ("A", 1), ("A", 1)])),
   TestCase $ assertEqual "totalCounts"
         [("A", 5), ("B", 3), ("C", 4)]
         (sort (totalCounts [["A", "B", "C"], ["C", "A", "B"]])),
   TestCase $ assertEqual "cmp1"  EQ (cmp ("A", 2) ("B", 2)),
   TestCase $ assertEqual "cmp2"  LT (cmp ("A", 2) ("B", 1)),
   TestCase $ assertEqual "cmp3"  GT (cmp ("A", 2) ("B", 3)),
   TestCase $ assertEqual "sortedCounts"
         [("A", 5), ("C", 4), ("B", 3)]
         (sortedCounts [("A", 5), ("B", 3), ("C", 4)]),
   TestCase $ assertEqual "neededLength" 5 (neededLength ("Jo", 23)),
   TestCase $ assertEqual "totalNeededLength"
         9
         (totalNeededLength [("Jo", 23), ("Patrick", 4), ("Peter", 123)]),
   TestCase $ assertEqual "formatPair"
         "Jo␣␣␣␣123"
         (formatPair 9 ("Jo", 123)),
   TestCase $ assertEqual "formatAll"
         ["Jo␣␣␣123", "Peter␣23"]
         (formatAll [("Jo", 123), ("Peter", 23)])
   ]
```

Remember to run your tests with:

```
runTestTT tests
```

Note that there are two type aliases, one to represent candidates as strings and another to represent a "Vote" as a list of candidates.

Here are the functions you should implement. You should start by specifying their types and a "stub" implementation that does nothing. That way your tests will be runnable.

- The first is a little helper function called `downFromN`. It takes as input an integer `n` and returns the list of numbers from `n` down to `1`.
- Next is a method `withRanks`. It takes a vote and returns a list of candidate-integer pairs where the ranks have been added. For example the vote `["A", "B","C"]` becomes `[("A", 3), ("B", 2), ("C", 1)]`. You can achieve this easily using `zip`.
- Next is a method `allRanks`. It takes a list of votes and returns a list of all the pairs produced by `withRanks` for each of the votes in the list. So this will merge all the votes together into one big list, which is OK since we know how many points each candidate got from a voter by the rank number. For example the two votes `[["A", "B", "C"], ["B", "A", "C"]]` would produce (in some order) the list `[("A", 3), ("B", 2), ("C", 1), ("B", 3), ("A", 2), ("C", 1)]`.
- Next is a method `totalRank`. This takes in a candidate and a list of candidate-integer pairs, and adds up the integers *only* for those pairs matching the candidate. For example for candidate `"B"` and ranked pairs `[("B", 2), ("C", 1), ("B", 3)]` the answer would be `5`.
- Next is a method `allCandidates` which takes a list of candidate-integers pairs and returns a list of the candidates only, removing duplicates. The function `nub` will help you remove duplicates.
- Next we write a method `totalCounts` which takes a list of votes and returns a list of candidate-integer pairs which lists each candidate once, with corresponding number being the number of points they got. See the example in the test. The order of the candidates does not matter at this point.
- We will next want to sort this list of total counts, to have the winner near the top. In order to do that, we will need to implement a comparison function, so this is next. Write a `cmp` function that takes in two "candidate-integer" pairs and "compares" them by simply comparing the integers using the built-in `compare` function, and in reverse order (so a pair is "smaller" than another pair if its integer is larger). This will list larger integers first. The return type of this function if a bit unusual: it's a type called `Ordering` with the three values `LT` for "less than", `EQ` for "equal" and `GT` for "greater than". You don't need to return these values directly, the call you make to the integer `compare` function does that.
- Next we write a `sortedCounts` function. It takes as input the list of candidate-integer pairs, and returns the "sorted" list using the sort based on the `cmp` function. You can achieve this by calling the `sortBy` function and giving it the `cmp` function as its first parameter (and your list as its second).
- Now we need to work towards producing a printout of the candidate scores. We start with a function `neededLength` that returns, for a candidate-integer pair, the

length needed to properly present it, which should be the length of the candidate followed by one space followed by the length of the integer. You can use the show function to turn the integer into a string. Look at the test for example input.

- Next we define a function totalNeededLength. This takes a list of candidate-integer pairs, and produces the line length needed to be able to show all them (so it should be the largest of their individual lengths). You will need a list comprehension together with the list maximum, which returns the largest number from a list.
- Next we need a formatPair function that takes an integer length and a candidate-integer pair and returns a string of that length that starts with the candidate, ends with the integer, and fills the remaining space with empty spaces.
- Next we have a function formatAll which takes a list of candidate-integer pairs and returns a list of strings, where each pair has been formatted using formatPair, and where the length used is the total needed length produced by totalNeededLength.
- The function getResults puts all these together and is provided for you. You can run it as:

```
getResults allVotes
```