CMPUT274—Assignment 3 (Fall 2025)

R. Hackman

Due Date: Friday November 21st, 8:00PM

Per course policy you are allowed to engage in *reasonable* collaboration with your classmates. You must include in the comments of your assignment solutions a list of any students you collaborated with on that particular question.

For each assignment you may not use any Python features or functions not discussed in class! This means, for example, you may not use loops, casting, any built-in functions or methods we have not discussed. If you are in doubt, you may ask on the discussion forum — however remember if you are including your code, or even your plan to solve a question, you must make a private post. Use of disallowed features will result in a grade of zero on the question(s) in which they were used.

All functions you write must have a complete function specification as presented in the course notes. Failure to provide function specifications for functions will lead to the loss of marks.

Unless explicitly forbidden by a question, you are always allowed to write additional helper functions that may help you solve a problem.

1. **Hamming Distance** The *Hamming distance* between two strings is the number of positions in which their characters differ. Typically, Hamming distance is defined between two strings of the same length, however we will define it for two arbitrary strings. In the case where one string is larger than the other, than each character the longer string is longer by is also considered a character in the distance, as it is a position in which the strings differ (one string has a character in that position, while the other does not).

   For example

   ```
   hammingDistance("axyz", "axzz") -> 1
   hammingDistance("abc----", "abc") -> 4
   hammingDistance("grey", "hazy") -> 3
   ```

   You must write the function `hammingDistance` such that takes two strings and produces the natural number that is the Hamming distance between those two strings.

   Restriction: You may not write *any* functions that are explicitly recursive. Instead of using any recursion you must use higher order functions to complete this problem. Solving the problem using recursion would result in no marks awarded.

   You are again provided a skeleton file `q1.py` to fill in. This provides some basic test cases already written for you. While there are three basic test cases provided, they are insufficient to effectively test your function. In addition to filling in the function definition you should write your own test cases to be confident your solution works.

   Once again, you should not edit any files in the repository itself, but instead make your own copy of it to edit. To make a new directory and copy the file for this question into that new directory you can follow the commands below:

   ```
   cd
   mkdir myA3
   cd myA3
   cp ~/F25-CMPUT274/a3/provided/q1.py ./
   ```

   Deliverables: For this question include in your final submission zip the python file `q1.py`

2. **Deduplication** The act of *deduplicating* a sequence is the act of creating a new sequence where only the *first* occurrence of each value appears in the resultant sequence.

   Write the function `dedup` that takes a single `LList` parameter and returns the `LList` that is the result of deduplicating the given input `LList`. For example:

   ```
   dedup(LL(1, 1, 1, 2, 1, 8, 2, 7, 8)) -> (1, 2, 8, 7)
   dedup(LL(2, 3, 5, 7, 9, 9, 7, 9, 5, 2, 3, 2, 7)) -> (2, 3, 5, 7, 9)
   ```

   Restriction: You may not write *any* functions that are explicitly recursive. Instead of using any recursion you must use higher order functions to complete this problem. Solving the problem using recursion would result in no marks awarded.

   Once again, you should not edit any files in the repository itself, but instead make your own copy of it to edit. If you have followed all instructions in the previous questions so far, then you can make a copy of the provided starter file by running the commands below:

   ```
   cd ~/myA3
   cp ~/F25-CMPUT274/a3/provided/q2.py ./
   ```

Deliverables: For this question include in your final submission zip the python file `q2.py`

3. **Word Count** In a textual document the *word count* of a word is the number of times it occurred in the document. For the purpose of this question we will define a "word" as a sequence of non-whitespace characters. We will also define a non-whitespace character as any character that is not one of the characters `" "`, `"\t"`, or `"\n"`.

   You must write the function `wordCounts` that takes a single string parameter and returns a `LList` of `CountPairs`. A `CountPair` is defined as a `LList` of two elements, the first of which is a string and the second of which is a natural number which is the number of times that word appeared in a given text. Your function should return the `LList` in which the pairs appear in the same order their corresponding words showed up in the original string. For example:

   ```
   wordCounts("cook the cook not the cook") -> (("cook", 3), ("the", 2), ("not", 1))
   wordCounts("also don't.\n\nAlso   don't")
       -> (("also", 1), ("don't.", 1), ("Also", 1), ("don't", 1))
   ```

   Note that since our definition of a word is any sequence of non-whitespace characters that means that sequences like "don't." get treated like a word, punctuation and all. Also note that our words are case-sensitive, so you do not need to consider words that are the same but with different case characters to be the same word.

   Once again, you should not edit any files in the repository itself, but instead make your own copy of it to edit. If you have followed all instructions in the previous questions so far, then you can make a copy of the provided starter file by running the commands below:

   ```
   cd ~/myA3
   cp ~/F25-CMPUT274/a3/provided/q3.py ./
   ```

   Deliverables: For this question include in your final submission zip the python file `q3.py`

4. **Optimal Purge Sort** For this question you must understand the concept of a sorting algorithm we will call *purge sort*. The idea behind purge sort is simple: given an arbitrary `LList` L produce a sorted version of that `LList` by producing the result of removing all elements from L that are not already in order. In this way, purge sort does not necessarily produce a sorted version of the original sequence, since some elements may not appear.

   So, for example given the `LList (1, 9, 7, 13, 10, 12)` one implementation of purge sort might produce the result `(1, 9, 13)`, because as you walk the `LList` from left to right you see a 1, then a 9, then a 7 which is not in non-decreasing order so you remove the 7. The next number you see is a 13 which is non-decreasing so it is kept, then 10 and 12 are not in non-decreasing order and as such are removed.

   Another problem with purge sort is that there are multiple possible answers to the resultant list. In the example above it would also be valid to produce the `LList (1, 7, 10, 12)`, and instead delete the 9 and the 13. In fact, one may find this a more desirable result as it removes less information from the original `LList`, deleting only two items instead of three.

   For this question you must write the function `optimalPurgeSort` which takes two parameters.

The first parameter is the `LList` to sort, and the second parameter is the comparison function to use when sorting. Your function should return a sorted `LList` that is the result of applying purge sort to the parameter, but the *longest* possible result.

Note 1: in some cases there can be more than one correct answer, for example sorting the `LList (1, 5, 10, 9)` in non-decreasing order could result in `(1, 5, 10)` or `(1, 5, 9)` both of which have a length of 3. For this reason we will not test your function with an input `LList` where there is more than one correct answer — so you do not need to worry about what to do in this case.

Note 2: this problem can be tricky to do efficiently. Your solution does not need to be particularly efficient, and as such we will not test it with `LLists` with a length larger than 50. Your function may take a few seconds to compute the answer for a `LList` of 50 elements and that is okay. While your solution does not have to be efficient, it is *possible* to solve this problem efficiently. If you want to challenge yourself try to solve the problem in a way that can easily solve for LLists that are thousands of elements long.

Once again, you should not edit any files in the repository itself, but instead make your own copy of it to edit. If you have followed all instructions in the previous questions so far, then you can make a copy of the provided starter file by running the commands below:

```
cd ~/myA3
cp ~/F25-CMPUT274/a3/provided/q4.py ./
```

Deliverables: For this question include in your final submission zip the python file `q4.py`