Analysis of Algorithms
CS 375, Fall 2022
Problem Set 3
Due **AT THE BEGINNING OF CLASS** Wednesday, November 9

- For this assignment, standard file naming conventions apply: Please submit your typewritten answers in a PDF file named `CS375_PS3_<userid>.pdf` where `<userid>` is replaced by your full Colby userid, and submit it to your `SubmittedWork` folder. Please reach out to me right away with any questions or concerns about this!

- **IMPORTANT:** Some of these exercises may build upon topics covered in our Oct. 17 class meeting; they are included here "early" so you can see all of the exercises on this assignment.

- Please recall the essential, general style guidelines for writing algorithms in CS375—including restrictions on `break` and other flow-of-control statements—as presented on PS1. Unless otherwise specified, they apply for all work in CS375.

- Please recall the guidelines for algorithms given on previous HW assignment sheets. They continue to apply to all HWs in CS375.

- In general, there may be multiple correct ways of presenting an algorithm, although excessively inefficient or inelegant solutions may not receive full credit. If you have questions about whether your proposed solution is excessively inefficient or inelegant, please ask your Prof.!

- Exercises on this problem set use the *LList* data structure as defined in our lecture notes. As with all LList exercises in CS375, the only functions you can use as primitives are the three given in the definition (*first*, *rest*, *cons*), and the check if a list is empty. You must write any others yourself for these exercises, along with correctness arguments. This is not intended to suggest that any new or additional LList functions are necessary to solve any particular exercise; if you find them helpful in some cases, however, this is intended to clarify that you are welcome to create helper functions or other new functions for use on LLists to use as part of your solutions.

- Exercises on this problem set require *inductive arguments* of correctness. Please see the Lecture Notes of Oct. 26 and Oct. 31 for reminders, and please feel free to see me with any questions!

- *A general note for CS375:* When writing up your homework, please present your answers neatly and **explain your answers clearly**, giving all details needed to make your answers easy to understand. Graders may not award full credit to incomplete or hard to understand solutions. Clear communication *is* the point, on every assignment.

  In general in CS375, unless explicitly specified otherwise, answers should be accompanied by explanations. Answers without explanations may not receive full credit. Please feel free to ask me any questions about explanations that might come up!

Exercises

1. Using our *IBT* data structure, come up with a recursive algorithm that returns the sum of the elements in a tree.

   ```
   # Input: IntBinTree T
   # Output: The sum of all of the integers in tree T
   ```

   What did you decide that the algorithm should return on an empty tree as input? Explain your reasoning for that decision (a sentence or so could be sufficient), and give an inductive explanation of the algorithm's correctness. You do not need to give a complexity argument for this algorithm.

2. **(Common elements with Problem Set 0!)** In contrast to the iterative algorithm you created for HW1, here, design a *recursive* algorithm to find all the common elements in two sorted LLists of numbers. (Please be sure to use the LList data structure from class!) For example, for input lists $[2, 5, 5, 5]$ and $[2, 2, 3, 5, 5, 7]$, the output should be the list $[2, 5, 5]$. What is the maximum number of comparisons between list elements your algorithm makes if the lengths of the two input lists are m and n, respectively?

   Please give both a pseudocode description and an English description, to make it as easy as possible to understand the algorithm, and give an inductive explanation of your algorithm's correctness. (As usual for these LList exercises, the only functions you can use as primitives are the three given in the definition, and the check if a list is empty. You must write any others yourself for this exercise, along with correctness arguments. This does not, however, suggest that any are necessary to solve this problem!) You do not need to give a complexity argument for this algorithm.

NOTE: Exercises below refer to nested LLists—an LList can, in the usual way, have an LList as an element. For example, $[55, [77, 42], [11, 42], 88]$ is a valid LList.

For terminology, we say that an element of an LList is *top-level* to distinguish it from elements of an LList nested inside another LList. For example, in LList $[55, [77, 42], [11, 42], 88]$, 55 is top-level, while 77, 11, and both instances of 42 are not top-level. (Please talk with your Prof. if there are any questions about this definition!).

3. Using the LList data structure, write a recursive algorithm for the *LLRemoveAll* problem on lists:

   ```
   # Input: Item x and LList L.  Assume x is not a list.
   # Output: List L' containing exactly the elements of L
   #         not equal to x, in the order in which they occur in L
   #         (see examples below)
   ```

   This removes all—and only!—top-level occurrences of x in L. For examples,

   - `LLRemoveAll(42,` $[55, 77, 42, 11, 42, 88]$`)` returns $[55, 77, 11, 88]$
   - `LLRemoveAll(42,` $[55, [77, 42], [11, 42], 88]$`)` returns $[55, [77, 42], [11, 42], 88]$. (Note that 42 is not top-level!)

- `LLRemoveAll(`$[77, 42]$`,` $[55, [77, 42], [11, 42], 88]$`)` returns $[55, [11, 42], 88]$. (Note that $[77, 42]$ *is* top-level!)
- `LLRemoveAll(42,` $[42, 67, 42, [42, 42, 43], 47]$`)` returns $[67, [42, 42, 43], 47]$. (Only the top-level 42's are removed!)

As usual, give a short English explanation of correctness; because the algorithm is recursive, make sure it's an inductive explanation. You do not need to give a complexity argument for this algorithm.

4. Unlike the *LLRemoveAll* function, which only removes an item that is top-level in an LList, an *LLDeepRemoveAll* function removes an item at any level of nesting. Using the LList data structure, write a recursive algorithm for the *LLDeepRemoveAll* problem on lists:

```
# Input: Item x and LList L.  Assume x is not a list.
# Output: List L' with the same structure and elements as L
#         (in that order) except all occurrences of x are removed
#         at any level of nesting in L (see examples below)
```

For examples,

- `LLDeepRemoveAll(42,` $[42, 67, 42, [41, 42, 43], 47]$`)` returns $[67, [41, 43], 47]$.
- `LLDeepRemoveAll(42,` $[55, [77, 42], [11, 42], 88]$`)` returns $[55, [77], [11], 88]$.
- `LLDeepRemoveAll(47,` $[42, 47, [1, 2, [47, 48, 49], 50, 47, 51], 52]$`)` returns $[42, [1, 2, [48, 49], 50, 51], 52]$.
- `LLDeepRemoveAll(3,`$[3, [[[3]]], [3, 4]]$`)` returns [[[[ ]]], [4]].

For this exercise, you will also need to test whether or not an element in a list is a list itself. For this, you can use the following (or something very similar) in your pseudocode to test whether or not an item x is a list:

```
if type(x) == list
  # if x is a list...
else
  # if x is not a list...
```

As usual, give a short English explanation of correctness; because the algorithm is recursive, make sure it's an inductive explanation. You do not need to give a complexity argument for this algorithm.