

Analysis of Algorithms
CS 375, Fall 2019

Homework 6

Due **AT THE BEGINNING OF CLASS** Wednesday, October 2

- From your textbook (CLRS), please read Chapters 2.1 and 2.2.
- 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.!
- *A general note:* When writing up your homework, please write neatly and **explain your answers clearly**, giving all details needed to make your answers easy to understand. Graders may not award credit to incomplete or illegible solutions. Clear communication *is* the point, on every assignment.

Exercises

1. Imagine we had a function $minus(x, y)$ that took two numbers x, y as input and returned $x - y$. We'll use it in this exercise.

In class, we went over two implementations of the *reduce* function on lists, one on LLists/LISP-style lists, and one on Java/Python-style lists. Each implementation depended on its foundations, the underlying definition of the list on which it operates.

For which of the implementations—or both, or neither—would $reduce(minus, L, 0)$ be iterated subtraction on a list of numbers, as we conventionally use it? As a concrete example, for which implementation (or both, or neither) does $reduce(minus, [a, b, c, d, e], 0)$ return $a - b - c - d - e$?

As always, explain your answer in sufficient detail to show command of the relevant concepts—here, that will include sufficient detail to show understanding of how both implementations of *reduce* work.

2. (As discussed in class, this exercise is repeated from HW3, but with additional instructions. You are welcome to resubmit your paper from HW3 as all or part of this exercise.) Using the *LList* data structure from class, write a recursive algorithm for the *reverse* problem on lists:

```
# Input: List  $L = [a_0, a_1, \dots, a_n]$ 
# Output: List  $L' = [a_n, \dots, a_1, a_0]$  with the same elements as in  $L$ 
#           but in reverse order
```

As usual, give a short English explanation of correctness; because the algorithm is recursive, make sure it's an inductive explanation.

Note: You are welcome to create other functions for use on LLists to use as part of your solution! For each such function, however, you must explicitly write the algorithm as part of this exercise.

As usual, give a short English explanation of correctness for every algorithm given for this exercise—at least *LLReverse*, but also any other algorithms you create for use here. For each recursive algorithm on LLists (and I anticipate any algorithm you write for this exercise would be recursive!), make sure the accompanying explanation is an inductive explanation.

3. Unlike the *LLReverse* function, which reverses the top-level elements of an LList, an *LLDeepReverse* takes as input an LList L and returns as output an LList L' that results from reversing all nested lists in L as well as L itself.

For examples,

- *LLDeepReverse*([1, 2, 3]) returns [3, 2, 1]. (This is the same as what *LLReverse* would return, because there are no nested lists—all elements are top-level.)
- *LLDeepReverse*([1, [2, 3], 4]) returns [4, [3, 2], 1]. Note that this is different from what *LLReverse* would return: *LLReverse*([1, [2, 3], 4]) returns [4, [2, 3], 1].
- *LLDeepReverse*(([1, [2, [3, 4], [5, [6, 7], 8]]])) returns [[[8, [7, 6], 5], [4, 3], 2], 1]. Note that *LLReverse* would return [[2, [3, 4], [5, [6, 7], 8]], 1].
- *LLDeepReverse*([2, [], [[[3]]], [3, 4]]) returns [[4, 3], [[[3]]], [], 2].

Using the LList data structure, write a recursive algorithm for the *LLDeepReverse* problem on LLists. For this exercise, you may assume the following two things: you have a correctly functioning *LLReverse* function to use in your algorithm without arguing its correctness; and 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...
```

Note: You are welcome to create other functions for use on LLists to use as part of your solution! For each such function, however, you must explicitly write the algorithm as part of this exercise. (The exception, as noted above, is *LLReverse*.)

As usual, give a short English explanation of correctness for every algorithm given for this exercise—at least *LLDeepReverse*, but also any other algorithms you create for use here. For each recursive algorithm on LLists (and I anticipate any algorithm you write for this exercise would be recursive!), make sure the accompanying explanation is an inductive explanation.

4. **Recursive Insertion Sort!** In this exercise, you'll write a pseudocode algorithm for a recursive version of Insertion sort, a different way of expressing the same underlying algorithmic idea as the iterative version from class.

We'll do this in two parts, ending up with an algorithm to sort LLists of numbers (i.e., using the LList data structure from class for the sequence to be sorted). For this exercise, sorting is taken to mean in non-decreasing order.

- (a) Write a recursive *LLInsert* algorithm that inserts a number x in the proper location in a sorted LList L .

```
# Input: Number  $x$  and sorted LList  $L = [a_0, a_1, \dots, a_n]$ ,
#         where  $a_0 \leq a_1 \leq \dots \leq a_n$ 
# Output: List  $L' = [b_0, b_1, \dots, b_{n+1}]$  containing input  $x$  and the
#          $n + 1$  elements of  $L$ , in sorted order
#          $b_0 \leq b_1 \leq \dots \leq b_{n+1}$ 
```

- (b) Using the *LLInsert* function, write a recursive *LLInsertionSort* algorithm that takes an LList L , possibly unsorted, and returns a sorted LList L' with the same elements as L in sorted order, consistent with the specification of the sorting problem.

```
# Input: LList  $L = [a_0, a_1, \dots, a_n]$ 
# Output: List  $L' = [b_0, b_1, \dots, b_n]$  containing exactly the
#         elements of  $L$ , in sorted order  $b_0 \leq b_1 \leq \dots \leq b_n$ 
```

As usual, explain your algorithms and give correctness arguments.