

Analysis of Algorithms
CS 375, Spring 2020

Homework 6

Due **AT THE BEGINNING OF CLASS** Monday, March 2

- From your textbook (CLRS), please read Chapters 2.1 and 2.2.
- 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.!
- *A general note for CS375:* 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 full credit to incomplete or illegible 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. 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. Write a recursive algorithm that takes a natural number n as input, returns nothing, and prints number n one line at a time on the screen. For example, on input 375, the output on the screen would be

3
7
5

In your algorithm, **for full credit, you may not use any string functions (other than print)**—the input is a number and should be treated that way. In addition, **for full credit, your algorithm should use only the basic arithmetic functions $+$, $-$, $*$, $/$, and the modulo operator $\%$** when performing number-valued arithmetic operations. (Recall from an exercise on the previous HW that the modulo operator $\%$ can be thought of as returning the *remainder* of a division—for examples, $5 \% 2 = 1$ and $7 \% 4 = 3$, because the remainder when dividing 5 by 2 is 1, and the remainder when dividing 7 by 4 is 3.)

Note: This function is different from others so far in CS375, in that it does not have a return value—instead, it simply generates output for the screen. The specification for it therefore includes *side effects*:

```
# Input: natural number n
# Output: none (no return value)
# Side Effect: prints number n to screen, one digit per line
```

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

3. Unlike the *LLReverse* function from HW3, 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. (See HW4 for a reminder of what “top-level” means in context, if you’d like!)

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 (which means the only ways to access and manipulate a list are those given in class), 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.