

Analysis of Algorithms
CS 375, Spring 2020

Homework 4

Due **AT THE BEGINNING OF CLASS** Monday, February 24

- Unless otherwise specified, exercises will be from the CLRS textbook and will be named on HW assignments by exercise number used in the book.
- 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

Some of the 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!)

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

```
# Input: Item  $x$  (assume  $x$  is not a list) and LList  $L$ 
# 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(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.

2. 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 (assume x is not a list) and LList L
# 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 a list element *e* is itself a list:

```
if type(e) == list
    # if e is a list...
else
    # if e 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.

3. Here are a few exercises on properties of logarithms: You'll be asked to show that a few properties of logarithms are true. The objective is to help you become more familiar with some properties of logarithms that are especially important for analysis of recursive algorithms.

As an example, here's how we might show that $\log_b xy = \log_b x + \log_b y$. In the below, recall from the definition of logarithm that a logarithm really stands for an exponent— $x = \log_b n$ means that $b^x = n$, i.e., x is the exponent we raise b to, to get n .

To make it easier to talk about raising some relevant expressions to a power, we'll introduce a few variables: let $k = \log_b xy$, $\ell = \log_b x$, and $m = \log_b y$. Then, by the definition of logarithm (above), $b^k = xy$, $b^\ell = x$, and $b^m = y$.

(Note: Do you see why?) Therefore, $b^k = b^\ell b^m = b^{\ell+m}$, and thus, by standard properties of exponents, $k = \ell + m$, which is what we had set out to show.

Now, pick 2 of the following 3 properties of logarithms (your choice!) and show that they are true. (If you turn in all 3 of them, only your best 2 will be counted for your grade, so I hope you'll give all 3 of them a try!)

Please give detailed explanations, as shown in the example above! Assume that a, b, c, n are positive real numbers (but note—they are not necessarily integers).

- (a) $\log_b a^n = n \log_b a$.
- (b) $\log_b n = \frac{\log_c n}{\log_c b}$. (This shows that changing the base of a logarithm—say, from $\log_b n$ to $\log_c n$ —really means multiplying by a constant factor, because this could be rewritten as $\log_b n = \frac{1}{\log_c b} \cdot \log_c n$.)
- (c) $a^{\log_b n} = n^{\log_b a}$. (This result is sometimes called the “log-switching theorem” since it says that we can “switch” a and n .)