

Analysis of Algorithms
CS 375, Spring 2021

Problem Set: Intro to Asymptotic Complexity (and Sorting Algorithms)

Due **BY THE BEGINNING OF CLASS** Wednesday, March 24

- From your textbook (CLRS), please read all of Chapters 2 and 3. Although not all of it will show up on CS375 exercises, it is all good material to know.
- To submit this Problem Set, the standard file naming and HW submission conventions for CS375 apply: Please put your answers in a PDF file named

`CS375_PS_AC_<userid>.pdf`

where `<userid>` is replaced by your full Colby userid (including your class year)—for example, my file would be called `CS375_PS_AC_eaaron.pdf`—and submit it to your `SubmittedWork` folder.

- Unless otherwise specified, exercises will be from the CLRS textbook and will be named on HW assignments by exercise number used in the book.
- **IMPORTANT:** Some of these exercises may build upon topics covered in class meetings after the day this is assigned; they are included here “early” so you can see all of the exercises on this assignment.
- 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.
- 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:* As always, please present answers cleanly and **explain them clearly and thoroughly**, giving all details needed to make your answers easy to understand; typed-up (rather than handwritten) answers are especially appreciated. (Feel free to talk with your Prof. or TA’s about using L^AT_EX to typeset your answers!) 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. List the following functions of n according to their order of growth—that is, how fast each function grows as n gets big—from lowest to highest:

$$(n-2)!, \quad 5\lg(n+100)^{10}, \quad 2^{2n}, \quad 0.001n^4 + 3n^3 + 1, \quad \ln^2 n, \quad \sqrt[3]{n}, \quad 3^n.$$

(As is conventional, the $\lg$ function is logarithm base 2; the $\ln$ function is the *natural logarithm*, logarithm base e ; and $\ln^2 n$ is common notation for $(\ln n)^2$.) Although you don't need to explain every part of the ordering for this exercise, please give short explanations (1–2 sentences) for the following:

- (a) how you know the second-smallest comes before the third-smallest; and
- (b) how you know the second-largest comes after the third-largest.

NOTE: This exercise is not about the definition of big-O notation, and it does not require that definition! It's an exercise on mathematical background, as previous CS375 exercises were (e.g., those about logarithms), in preparation for the complexity analysis work we'll be doing as the semester goes along.

2. **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 of numbers, 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.

SLIGHT DIGRESSION: Do you wonder which is *asymptotically* more efficient, the iterative Insertion Sort, or the recursive Insertion Sort? Or are they in the same asymptotic complexity class? Please **do not submit an answer to this question**, but it's an interesting and important question, and we'll learn how to answer it when we cover methods for analyzing the complexity of recursive algorithms—coming soon to CS375!

3. Prof. E. Nigma of the Portland Institute of Technology hired you to analyze the algorithm given here in pseudocode, but as usual, Prof. Nigma neglected to explain what the algorithm does.

```
// Input: A matrix A[0..n-1, 0..n-1] of integers
for i = 0 to n-2 do
    for j = i+1 to n-1 do
        if A[i,j] != A[j,i]
            return False
return True
```

In the above, recall that a matrix is essentially just a two-dimensional array, so $A[i, j]$ might in some languages be written as $A[i][j]$.

- (a) What does this algorithm do? Give an English description of what inputs lead to it returning True and what inputs lead to it returning False. (You do not need to give examples as part of your answer, but you are welcome to include example 2D arrays along with the English description, if it would make your answer clearer.)
- (b) Give an *exact* count of the number of array accesses (count $A[x, y]$ as a single array access) done by this algorithm in the worst case on input of size n , and based on that, give the Θ complexity class for this algorithm. Be sure to explain the details behind your answer: show all work that you did to count those operations and arrive at a summation that expresses the running time; and explain how you got from that summation to the Θ complexity bound, giving the relevant threshold n_0 and leading constants c_1, c_2 used to establish that Θ bound.