# Lab 5 - Proofs

In this lab, we are going to work to become comfortable with the CLRS notation and answering basic problems about algorithmic performance and correctness

## Problem 1: Insertion Sort, Descending order

Using CLRS’s notation, rewrite pseudocode for Insertion Sort so that it sorts in descending order:

For j = 2 to A.length

Key = A[j]

// Insert A[j] into the sorted sequence A[1…j-1]

i = j-1

while i > 0 and A[i] < key:

A[i +1] = A[i]

i = i -1

A[i +1] = key

## Problem 2: Linear Search pseudocode

We’re going to define the searching problem in CLRS terms.

Input: A sequence of *n* numbers A = 〈 *a1, a2, … an* 〉

Output: An index *i* such that *v = A[i]* or the special value NIL if *v* does not appear in *A*.

Write pseudocode in CLRS style for linear search, which scans through the sequence, looking for *v*. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

Pseudocode:

for i = 1 to A.length

if A[i] == *v*

return i

return NIL

Loop invariant:

The invariant: for every index *k* that we’ve seen, there’s no index *k* < *i* such that A[k] == v.

At each turn of the loop, we check to determine that A[i] is not equal to v, to maintain that loop invariant.

If we exit the loop by returning *i*, we have found a value equal to *v* and return its index.

If the loop terminates without returning a value *i*, it has not found a match and we have tried every possible value in A, and so it returns the special value NIL.

## Problem 3: Selection Sort

In CLRS notation, write pseudo for selection sort. Assume that selection sort works this way: It sorts *n* numbers stored in array *A* by first finding the smallest element of *A* and exchanging it with the first element in *A*. Then it finds the second smallest item in *A* and exchanges it with the second item in *A*. It continues for the first *n-1* elements in *A*.

Write pseudocode for selection sort.

What loop invariant does this algorithm maintain?

Why does it need to run for only the first *n-1* elements, instead of for all *n* elements?

What are the best- and worst-case runtimes of this algorithm in big O terms, and why?

Selection Sort Pseudocode:

for i = 1 to A.length-1:

min = i

for j = i + 1 to A.length-1

if A[j] < A[min] then

min = j

Swap A[min] and A[i]

The loop invariant is that, at each iteration of the for loop, the subarray A[1…i-1] contains the i-1 smallest elements of A in increasing order. We know this because in the first iteration, we find the smallest item by taking the first item i and comparing it to A[j] where j is every value greater than i in the array. After n-1 iterations of the loop, the n-1 smallest elements of A are in the subarray, which has length n-1. At that point, the nth element is the largest element left, so we do not need to run a full loop.

Best and worst case are both O(n2), because this algorithm will iterate through the list n-1 times no matter what.

## Bonus: Problem 4: Why we never consider best case

How can we modify any sorting algorithm to have a good best case running time, returning a definitely sorted array in O(n)? Remember that, in the best case, you can assume the data is in the most beneficial format that you need for your solution to work.

Answer: We can modify the algorithm to always check if the initial array is in sorted order by iterating through the array once. If it is, it can return the array as-is, returning a sorted output in O(n) time.