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:

```
INSERTION-SORT(A)
  for j = 2 to length[A]
    key = A[j]
    /* Insert A[j] into the sorted sequence A[1..j-1] in descending order */
    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 = $\langle a_1, a_2, \ldots a_n \rangle$
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.

```
Linear-Search(A, v)
  for i = 1 to A.length
    if A[i] == v
      return i
  return NIL
```

The loop invariant is that at the start of each iteration of the loop, the subarray A[1...i-1] does not contain the value v. The three necessary properties of a loop invariant are initialization, maintenance, and termination. Initialization is that before the loop begins, the subarray A[1...i-1] is empty and does not contain the value v. Maintenance is that at each iteration, if the current element A[i] is not equal to v, then the subarray A[1...i] still does not contain the value v. Termination is that when the loop terminates, either i = n+1 and the value v was not found, or

A[i] = v and the index i is returned as the solution. This proves the correctness of the linear search algorithm.

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.

```
SELECTION-SORT(A)
  for i = 1 to n - 1
    smallest = i
    for j = i + 1 to n
      if A[j] < A[smallest]
        smallest = j
    exchange A[i] with A[smallest]
```

What loop invariant does this algorithm maintain?
The loop invariant for selection sort is that at the start of each iteration of the outer loop, the subarray A[1...i-1] consists of the i-1 smallest elements of A in sorted order.

Why does it need to run for only the first n-1 elements, instead of for all n elements?
The algorithm only needs to run for the first n-1 elements because after the n-1 smallest elements have been placed in the correct order, the nth element must be the largest remaining element and will automatically be in the correct place when the other elements are sorted. Therefore, it is not necessary to include it in the sorting process.

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.

We can modify any sorting algorithm to have a good best case running time by checking if the input array is already sorted, and if so, simply returning the sorted array without performing any additional sorting operations. This check can be done in O(n) time. In the best case, where the input array is already sorted, this modified algorithm would have a running time of O(n).