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 $A = \langle a_1, a_2, \ldots a_n \rangle$

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.

```
for i=1 to  A.length
        if v==A[i]
        return i
```

return NIL

Initialization: before the first iteration there is no index k<i such that A[k]=v

Maintenance: before each loop there is no index k<i such that A[k]=v, and if A[i] is not equal to v, this remains true before the next iteration

Termination: it terminates when it finds the A[i] equal to v and return the index I, or fails to find an item equal to v after trying all items in the array and return 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?

```
for i=1 to A.length
        min=i
        for j=i to A.length
                if A[j]<A[min]
                        min=j
        Swap A[min] and A[i]
```

Invariant: subarray A[1] is sorted

Maintenance: before each loop subarray A[1…i-1] is sorted and after swapping A[min] and A[i] the new subarray A[1…i] is also sorted before the next iteration

Termination: when it terminates the subarray A[1…i-1] consists of all items originally in array A[1…n] and the array is sorted

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 algorithm to have a best case time complexity of O(n) by adding a special case. For example, we can make check if the array is already sorted by iterating through the array once. If it is sorted it will terminate and the complexity is O(n).