Homework 5 – Sorting and Proofs

In this homework, we're going to work through a few CLRS problems, focusing on the basic searches, sorts, and loop invariant problems we've seen so far. Do your best, write answers that are complete sentences, and make sure that you're using the specific notation and pseudocode styles used in CLRS.

Problem 1: Bubblesort

Suppose that Bubblesort on an array A uses the following pseudocode:

```
1 for i = 1 to A.length – 1
2       for j = A.length downto i + 1
3               if A[j] < A[j-1]
4                       exchange A[j] with A[j-1]
```

NOTE: This problem assumes the array starts indexed at 1.

a. Let A' be the output of Bubblesort on A. To prove that Bubblesort is correct, we need to prove that bubblesort terminates, and that

   $A'[1] \leq A'[2] \leq \ldots \leq A'[n]$,

   Where n = A.length. In order to short that Bubblesort actually sorts, what else do we need to prove?

To show that Bubblesort sorts, we also need to show that Bubblesort works towards sorting, so that it terminates and that its output, A' has the same elements and length as A.

b. State precisely a loop invariant for the *for* loop in lines 2-4 in the pseudocode above, and prove that this loop invariant is maintained for the entire algorithm. Use the CLRS loop invariant proof in Chapter 2 as a model for this proof.

The loop invariant is that before each loop, the element A[j] is the smallest element of A[j:n] and that all the elements are in A[j:n] are the same as at the end of the loop.

Initialization: Initially j=n and therefore A[n] is, trivially, the smallest element of A[n:n].

Maintenance: In every loop, we make sure that A[j-1] is the smaller of the elements A[j-1] and A[j], if it was not already. By induction, A[j-1] is the smallest element in A[j-1:n]. As we have only swapped elements A[j-1:n] contains the same elements as at the start of the loop.

Termination: When the loop terminates, j=i+1. Therefore, at the end, A[i] is the smallest element of A[i:n].

c. Using the termination condition of the loop invariant that you proved in part (b), state a loop invariant for the *for* loop in lines 1-4 that will allow you to prove the inequality in (a). Use the CLRS loop invariant proof in Chapter 2 as a model for this proof.

The loop invariant is that before each loop A[1, i-1] is sorted and contains elements that are all smaller than the elements of A[i:n].

Initialization: Initially, A[1:0] is empty so our loop invariant holds trivially.

Maintenance: In every loop, we make sure that A[i] is the smallest element of A[i+1:n]. By induction, A[1:i] is sorted and all the elements in A[0:i] are smaller than the elements of A[i+1:n].

Termination: When the loop terminates, i=n-1. This means that A[1:n-1] is sorted and all these elements are less than A[n]. Therefore, A is sorted.

d. What is the worst-case running time of bubblesort? How does it compare to the worst-case running time of insertion sort?

The worst-case running time of bubble sort is $O(n^2)$. The worst-case running time of insertion sort is also $O(n^2)$. It is important to note though, that bubble sort must always go through comparing n elements then n-1 then n-2… Meanwhile insertions sort does not have to always compare all these elements. Once the correct location for an element is found, the inner loop terminates. This saves running time and makes its best-case scenario running time O(n).

Problem 3: Binary Search

Refer back to the pseudocode you wrote for linear search in the lab. Let's change the problem: let's say that the array A is sorted, and we can check the midpoint of the sequence against *v* and eliminate half of the sequence from our search. The *binary search* algorithm repeats this procedure, halving the size of the remaining sequence every time.

a. Write pseudocode that uses a for or while loop for binary search.

Note: The index starts indexed at 1 like in Problem 1.

```
int binarysearch(array A, int element, int array_size)
        i = 1
        j = array_size
        midpoint = i + (j-i)/2
        while i != j
                if element == A[midpoint]
                        return midpoint
                if element > A[midpoint]
                        i = midpoint
                else
                        j = midpoint
                midpoint = i + (j-i)/2
        return -1 // The element is not in the array
```

b. Make an argument based on this pseudocode for the worst-case runtime.

Each loop, we eliminate half of the elements left in the list. That means if N is of the form 0.5^G, the number of loops we need to eliminate all the elements but 1, which is the worst-case, is on the order of G. As G=log(N), this means the worst-case runtime is on the order of O(log(N)).

c. Prove that binary search solves the search problem on a sorted array by using a loop invariant proof.

The loop in variant is that before each loop is that if the element is in the array, then it in in A[i:j].

Initialization: Initially, A is sorted, i is the index of the first element in the list, and j is the index of the last element in the list.

Maintenance: Each loop, if the element is compared to the element at index of the middle of the array A[i:j]. If the element is the element, we are looking for we return it. If we are looking for a larger integer, as the array is sorted, we change A[i:j] so that i is now the index of the middle of the array, where the element must be located if it in the array. If we are looking for a smaller integer, as the array is sorted, we change A[i:j] so that j is now the index of the middle of the array. Each iteration j-i is halved, and the two indices get closer to each other.

Termination: The loop terminates when the element is found in the middle of A[i:j] or when i=j. If i=j, i=j=midpoint and we have also previously checked if either i or j is the index where the element is found. In this case, then the element is not in A[i:j] and by induction it is not in A.

Problem 4: Improve Bubblesort performance

a. Modify the pseudocode of Bubblesort to increase its efficiency marginally.

Suppose that Bubblesort on an array A uses the following pseudocode:

```
1  for i = 1 to A.length – 1
2          already_sorted = True // a place holder: if we make any swaps this is not true
3          for j = A.length downto i + 1
4                  if A[j] < A[j-1]
5                          already_sorted = False
6                          exchange A[j] with A[j-1]
7          if already_sorted = True
8                  Break
```

b. Write a proof showing that this modification still solves the sorting problem, using loop invariants.

The inner loop invariant is that before each loop, the element A[j] is the smallest element of A[j:n] and that all the elements are in A[j:n] are the same as at the end of the loop.

Initialization: Initially j=n and therefore A[n] is, trivially, the smallest element of A[n:n].

Maintenance: In every loop, we make sure that A[j-1] is the smaller of the elements A[j-1] and A[j], if it was not already. By induction, A[j-1] is the smallest element in A[j-1:n]. As we have only swapped elements A[j-1:n] contains the same elements as at the start of the loop.

Termination: When the loop terminates, j=i+1. Therefore, at the end, A[i] is the smallest element of A[i:n].

The outer loop invariant is that before each loop A[1: i-1] is sorted and contains elements that are all smaller than the elements of A[i:n].

Initialization: Initially, A[1:0] is empty so our loop invariant holds trivially.

Maintenance: In every loop, we make sure that A[i] is the smallest element of A[i+1:n]. By induction, A[1:i] is sorted and all the elements in A[0:i] are smaller than the elements of A[i+1:n]. We also check every loop to see if we've swapped any integers. If we have not, then we terminate the loop.

Termination: In one case, the loop terminates when i=n-1. This means that A[1:n-1] is sorted and all these elements are less than A[n]. Therefore, A[1:n] is sorted. In the second case, the loop terminates when there were no swaps necessary in the inner loop. In this case A[1:i-1] is sorted and all its elements are less than those in A[i:n]. Also, every element from A[i:n] is less than the one to their right and therefore A[i:n] is also sorted. This means that A is sorted.

c. Prove that this algorithm is still $O(n^2)$

In the worst-case, the array is in reverse order: such as [5, 4, 3, 2, 1]. In this case, we must make n-1 swaps, then n-2 swaps, …, all the way down 1 swap in the last loop. This arithmetic series is on the order of $O(n^2)$. So, while this modification makes the best case O(n), when the list is already sorted, the worst-case scenario for this algorithm is still $O(n^2)$.