Homework 5 - 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]
```

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] ≤ A'[2] ≤ … ≤ A'[n] ,

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

   In order to prove that Bubblesort actually sorts, we need to prove that it satisfies the condition of being a permutation of the original array. That is, after sorting, A contains the same elements as before, but in a different order. This can be proved by showing that the elements of A' are a permutation of the elements of 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.

   Loop invariant: At the start of each iteration of the outer loop (line 1), the subarray A[i..n] contains the i smallest elements of A in sorted order.

   Initialization: Before the first iteration of the outer loop, i = 1, so the subarray A[i..n] consists of the entire array, and it contains only one element, which is trivially sorted.

Maintenance: Suppose the loop invariant holds at the start of the i th iteration. Within the i th iteration, the inner for loop checks the elements A[j] and A[j-1], for j from A.length down to i+1, and swaps them if A[j] < A[j-1]. This places the i th smallest element in A[i-1], and so the subarray A[1...i] contains the i smallest elements in sorted order. Thus, the loop invariant holds for the i+1-th iteration.

Termination: At the end of the outer loop, i = n, so the subarray A[i..n] contains the n smallest elements of A in sorted order. Therefore, the entire array A is sorted in non-decreasing order.

c. Using the termination condition of the loop invariant that you proved in part (b), state a loop invariant for the *for* lop 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.

Initialization: Before the first iteration of the outer for loop, i = 1, so the subarray A[1..i-1] is empty and thus trivially sorted.

Maintenance: We assume that at the start of an arbitrary iteration of the outer loop, the subarray A[1..i-1] is sorted. We will show that after the iteration, the subarray A[1..i] is sorted. By using part b, Thus, after the i-th iteration of the outer loop, the subarray A[1..i] consists of i elements in sorted order.

Termination: When the loop terminates, i = A.length, so the subarray A[1..i-1] consists of A.length-1 elements that are in sorted order. By the termination condition of the loop invariant in part (b), we know that the inner loop sorts the last two remaining elements, A[i] and A[i-1]. Therefore, after the outer loop terminates, the entire array A is sorted in non-decreasing order.

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 bubblesort is O(n^2), where n is the length of the input array. In comparison, the worst-case running time of insertion sort is also O(n^2).

However, in practice, insertion sort can be faster than bubblesort for small inputs or for inputs that are already partially sorted. This is because insertion sort has better performance characteristics than bubblesort in these cases, although bubblesort has the advantage of being a stable sort (i.e., it maintains the relative order of equal elements in the input array).

Problem 2: Inversions

Let A[1…n] be an array of *n* distinct numbers (“distinct” means “no duplicates”). If i < j and A[i] > A[j], then the pair (i,j) is called an inversion of A.

a. List the five inversions of the array ⟨ 2, 3, 8, 6, 1 ⟩

   The five inversions of the array are: (2,1), (3,1), (8,6), (8,1), and (6,1).

b. Imagine an array of length *n*, filled only with integers. For an *n* length array, what does the array look like with the most inversions? How many inversions does it have, written in terms of its length *n*?

   If the array is sorted in descending order, then every pair of elements is an inversion. So an n-length array sorted in descending order would have n(n-1)/2 inversions.

c. What is the relationship between the running time of insertion sort and the number of inversions in the input array? Give reasons for your answer.

   The running time of insertion sort is directly proportional to the number of inversions in the input array. This is because insertion sort operates by repeatedly swapping adjacent elements until the current element is in the correct position relative to the previous elements.

   The number of swaps needed for a single element is proportional to the number of elements preceding it that are greater than it, which is the number of inversions involving that element. Therefore, the total number of inversions in the array is a measure of the total number of swaps needed to sort the array using insertion sort.

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.

```
BinarySearch(A, v):
low = 1
high = A.length
while low <= high
   mid = floor((low + high) / 2)
   if A[mid] == v
      return mid
   else if A[mid] < v
      low = mid + 1
   else
      high = mid - 1
return "not found"
```

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

The worst-case runtime of binary search occurs when the target value is not present in the sorted array, and the search needs to continue until the low and high indices meet in the middle.

In this case, the while loop will execute until the range of the search is reduced to a single element. The range of the search is halved at each iteration.

Therefore, the worst-case runtime of binary search is O(log n), where n is the size of the input array.

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

Initialization: At the beginning of the algorithm, the subarray is the entire sorted array. The loop invariant holds because no values have been eliminated yet, and v could still be located in any position of the array.

Maintenance: At the start of each iteration, the algorithm checks the middle value of the subarray against the target value v. If the middle value is greater than v, the right half of the subarray is eliminated. If the middle value is less than v, the left half of the subarray is eliminated. The loop invariant holds because the subarray being searched is reduced by half at each iteration, and v could still be located in the remaining half.

Termination:
The loop terminates when either p > r, which means that v is not in the array, or A[q] == v, which means that we have found the index q where v occurs in the array. In either case, the loop terminates and the first loop invariant is satisfied.

Problem 4: Improve Bubblesort performance

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

```
BubbleSort(A)
  n = A.length
  flag = true
  while flag
    flag = false
    for i = 1 to n - 1
      if A[i] > A[i+1]
        swap A[i] and A[i+1]
        flag = true
```

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

Initialization: Before the first iteration of the outer loop, i is set to 1, so the loop invariant holds vacuously.

Maintenance:
In each iteration, the outer loop compares and swaps adjacent elements if they are out of order. In the modified inner loop, the largest unsorted element is swapped to the right, so that the comparison range is decreased by 1 in each subsequent iteration. The loop invariant is maintained because the sorted elements are in their final sorted positions and the unsorted elements are always to the right of the sorted elements. Thus, after each iteration of the outer loop, the largest unsorted element is moved to the rightmost unsorted position.

Termination: When the outer loop terminates, we have i=n, so the loop invariant implies that the largest n elements of the array are in their correct sorted positions at the end of the array. Therefore, the entire array is sorted.

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

In the worst case, where the input array is in reverse order, the modified algorithm will perform n iterations for the outer loop, as in the original algorithm. It means flag will become "true" every time. However, for the inner loop it will perform n iterations. Therefore, we can conclude that the modified algorithm is still O(n^2), as required.