Homework 5 - Proofs

These problems focus 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: Linear Search pseudocode

The searching problem can be defined in the following way, using CLRS notation:

Input: A sequence of n numbers $A = \langle a_1, a_2, \ldots a_n \rangle$

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

a. Write pseudocode in CLRS style for linear search, which scans through the sequence, looking for a target number t.

```
Linear_Search(A, T):

For i = 1 to A.length

        If A[i] == t

        return i

Return NIL
```

b. Using a loop invariant, prove that your algorithm is correct and solves the searching problem. Make sure that your loop invariant fulfills the three necessary properties.

Loop Invariant: At the start of each iteration of for loop, sub-array A[1..i-1] does not contains t

1.Initialization: Before first iteration, i=1, sub-array is empty, the loop invariant is true

2.Maintainance: Suppose loop invariant is true before ith iteration

1) if A[i] is equal to t, algorithm ends and return index i, ends correctly

2) if A[i] is not equal to t, we go to i+1 th iteration, since A[1..i] does not contains t, we can infer that A[1..i+1] also does not contains t.

3. Termination: loop ends when i is greater than A.length, in this case, sub-array A[1..n] has been searched and does not contains t, so loop invariant holds because A[1..n] does not contains t, and returns NIL.

Problem 2: Binary Search

a. Write pseudocode for binary search in CLRS notation that uses a while loop.

```
Binary_Search(A, t)
1  Low = 1
2  high = A.length
3  while low <= high
4          mid = floor((low + high) / 2)
5          if A[mid] == t
6                  return mid
7          else if A[mid] > t
8                  high = mid - 1
9          else
10                 low = mid + 1
11 return NIL
```

b. Identify the run time of this algorithm by counting operations (show your work), and prove the algorithm's big O.

line 1, 2: 1 operation

line 3, 4: log2(n) operations

line 5, 6: 1 operation

line 7-10: 2log2(n) operations

In line 11: 1 operation

$T(n) = 5 + 4 \log(n)$

By definition f(n) is O(g(n)) if there exists constants `c` and `k` such that $|f(n)| <= c*|g(n)|$ whenever $n > k$

Proof:

$5 + 4\log(n) <= 5\log(n) + 4\log(n)$ $(n > 1) = 9 \log(n)$

By definition, there exist $c = 9$ and $n = 1$ so that $|f(n)| <= c*|g(n)|$

So we proved that f(n) is O(log(n))

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

Loop invariant: At the start of each iteration of the while loop, the subarray A[low..high] only contains the target value t when t is originally in A[1..A.length]

Initialization: Before the first iteration, low equals 1 and high equals A.length. So the subarray is A[1..A.length] which is the original array, so we know A[low..high] can either contain t or not depending on the original sorted array, its true in first iteration.

Maintenance: If loop invariant is true before i-th iteration, we compute mid = int((low+high)/2), we compare A[mid] to x, if A[mid] is greater than t, then we know x is in left half, and set high = mid-1. If A[mid] is less than t, we know x is in right half of subarray, we set low = mid+1. Then we reduce the size of subarray by half, and still ensure that A[low..high] contains target value t when t is originally in A[1..length] before i+1 th iteration. So loop invariant is true.

Termination: we stop the loop when low > high, so that the array is empty, and target is not in subarray[low..high], and returns NIL. So if t is not in original array, then we have t is not in A[low..high] and algorithm returns NIL. So loop invariant is true, and we proved the correctness of this algorithm.

Problem 3: Insertion sort using binary search

a. Look at the pseudocode on page 19 of CLRS for Insertion-Sort. Line 5 includes a while loop that performs a backward linear search to identify the correct position for the key. Rewrite the pseudocode to perform a backwards binary search at this point in the algorithm.

```
INSERTION-SORT(A)
1 for j = 2 to A.length
2       key = A[j]
3       //insert A[j] into sorted sequence A[1..j-1]
4       i = j - 1
5       low = 1
6        high = i
7       while low <= high
8               mid = floor((low+high) / 2)
9               if A[mid] > key
10                      high = mid - 1
11              else
12                      low = mid + 1
13      for j = i - 1 downto high + 1
14              A[j+1] = A[j]
15      A[high+1] = key
```

b. Count the operations in this new version of insertion sort, and prove its big O.

line 1 : n operations

line 2-6: n-1 operations

Line 7-12: $\sum_{i=2}^{n} \log(i-1)$ operations

Line 13-14: n-1 operations in worst case

Line 15: n-1 times in worst case

T(n) = n + (n-1) * 4 + log((n-1)!) + (n-1)*3 = 8n - 7 + log((n-1)!) <= 8n - 7 + (n-1)log(n-1), since we have (n-1)! < (n-1)^(n-1) when n > 1

By definition f(n) is O(g(n)) if there exists constants `c` and `k` such that |f(n)| <= c*|g(n)| whenever n > k

Proof:

(n-1)log(n-1) + 8n - 7 <= nlog(n) + 8nlog(n) + 7nlog(n) = 16 * nlog(n) when n is greater than 1

By definition, there exist c = 16 and n = 1 so that |f(n)| <= c*|g(n)|

So we proved that f(n) is O(n*logn)

c. Write a proof of correctness showing that this modification still solves the sorting problem, using loop invariants. Each loop will need its own loop invariant, as well as discussion of that loop invariant's state for initialization, maintenance, and termination.

for loop(1-15):

Loop invariant: at the start of each iteration of for loop, subarray A[1..j-1] is sorted

Initialization: before the first iteration, j = 2 and A[1..1] is sorted.

Maintenance: If before j-th iteration, subarray[1..j-1] is sorted, in j-th iteration we insert the key into its position in sorted sub-array A[1..j-1], and we maintain A[low] <= key <= A[high], in line 13-15, we shift elements right and insert the element its into correct position, so that before j+1-th iteration subarray A[1..j] is sorted.

Termination: when outer loop ends, j > A.length, we have subarray a[1..j-1] is sorted, and whole array is sorted, so the algorithm is correct.

Inner while loop(7-12)

Loop invariant: at the start of each iteration of while loop, we have A[low] <= key <= A[high]

Initialization: before first iteration, low is 1 and high is i, the subarray A[1..i] is sorted because of for loop, so we have A[low] <= key <= A[high]

Maintenance: Suppose before k-th iteration, we have A[low] <= key <= A[high]. During k-th iteration and before k+1th iteration, we compare A[mid] with key, if its greater than key we update high, and if is lower we update lower bound, if A[mid]=key loop terminates and invariant holds, since array is still sorted, we have A[low] <= key <= A[high].

Termination: When loop ends, we have low is greater than high or A[mid] = key, in both situation the whole array is sorted, and A[low] <= key <= A[high] holds.

Inner for loop(13-15)

Loop invariant: at the start of each iteration of for loop, subarray A[high+1..j-1] is shift one position right

Initialization: before first iteration, j=i-1, subarray A[high+1..j-1] is empty, so loop invariant is true

Maintenance: Suppose before k-th iteration, we have subarray A[high+1..j-1]is shift one position right, before k+1 th iteration, we shift A[j] = A[j+1], as a result subarray A[high+1..j] shift one position right, so loop invariant holds.

Termination: when loop ends, j = high, and A[high+1.. high-1] is empty, since empty array is trivial for shifting so loop invariant holds.

Problem 4: Big and Small sort

a) Write CLRS pseudocode for an algorithm that solves the sorting problem by iterating through the unsorted array and finding both the smallest and the largest item in the array in each pass through the array, and placing them both in the correct position in the array. Make this algorithm as efficient as possible.

```
BIG-SMALL-SORT(A)
1 for i = 1 to floor(A.length / 2)
2       minIdx = i
3       maxIdx = i
4       for j = i+1 to n-i+1
5               if A[j] < A[minIdx]
6                       minIdx = j
7               else if A[j] > A[maxIdx]
8                       maxIdx = j
9       if minIdx != i
10              swap A[i] with A[minIdx]
11      if maxIdx == i
12              maxIdx = minIdx
13      if maxIdx != n-i+1
14              swap A[n-i+1] with A[maxIdx]
```

b) Determine the big O runtime of this algorithm by counting operations (show your work). Prove its big O using the definition of big O.

line1-3: floor(A.length/2) operations

line 4-8: floor(A.length/2) * A.length/2 times

line 9-14: floor(A.length/2) operations

T(n) = 3*n/2 + 5*n/2 * (n/2) + 6*(n/2) = 5/4 * n^2 + 9/2 * n

By definition f(n) is O(g(n)) if there exists constants `c` and `k` such that |f(n)| <= c*|g(n)| whenever n > k

Proof:

$$\frac{1}{4}\left(5n^2\right)+\frac{9n}{2} \le \frac{1}{4}\left(5n^2\right)+\frac{1}{2}\left(9n^2\right)=\frac{23}{4}n^2$$, when n is greater than 1

By definition, there exist c = 23/4 and n = 2 so that |f(n)| <= c*|g(n)|

So we proved that f(n) is O(n^2)

c) Prove that this algorithm solves the sorting problem using a loop invariant proof. Each loop will need its own loop invariant, as well as discussion of that loop invariant's state for initialization, maintenance, and termination.

Outer for loop:

Loop invariant: At the start of each iteration of loop, A[1..i-1] is sorted

Initialization: Before the start of 1st iteration, i = 1, A[1..0] ie empty and loop invariant is true

Maintenance: Suppose loop invariant is true in i-th iteration, in i+1-th iteration, we find smallest and largest element from A[i+1 : n-i+1], and we swap the smallest with A[i+1], and largest element with A[n-i+1], so that smallest element and largest element are in their correct position, so A[1..i] is sorted and loop invariant holds true.

Termination: for loop ends when i > floor(A.length/2), since when i = floor(A.length/2) first half and second half is sorted, we know that A[1..A.length] is sorted. Loop invariant is true and we prove the correctness.

Inner for loop:

Loop invariant: At the start of each iteration of inner loop, min and max element of A[i..n-i+1] is not in their correct sorted position.

Initialization: before first iteration, i = 1 so A[1..n] is unsorted, and min and max element are not in correct sorted position.

Maintenance: Suppose loop invariant is true before ith iteration. In i+1-th iteration, we find min and max element of A[i+1..n-i+1] and compare them with min and max of subarray A[i..n-i]. If we update minIdx and maxIdx. So before i+1-th iteration, min and max element of A[i..n-i+1] is not in their correct sorted position is true.

Termination: In termination, we have j > n-i+1, we find min and max element of whole array and swap them into their correct position, the whole array is sorted.