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

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

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

   ANS

   In addition to proving the inequality above, we also need to prove that the elements of *A' (output)* are also the elements in *A(input).*

   But since Bubblesort is an algorithm that sorts in-place, I think that proving the inequality only is *enough.*

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.

Invariant: *A*[*j*] is the smallest value in range of *A*[*j ...A.length*]

Initialization:Before the first iteration of the loop, *j =A.length and t*he subarray *A* [*j*] only has one elements. *A*[*j*] is the smallest.

Maintenance: For each iteration of the loop, we compare *A*[*j*] with *A*[*j-1*] and put the smaller element at the position *A*[*j-1*]. At the beginning of next iteration, new *j = j -1* of the last iteration with the smaller element, and all these show that each iteration maintains the loop invariant.

Termination: While *j < i + 1* which means *j = i,* we terminate the loop. In the last iteration while *j= i + 1,* we've already compared *A*[*j*] and *A*[*j-1*]*(which equals to A[i] and A[i+1])* and put the smaller element at the position *j (which is also i).*All value in *A [1...A.length],* the algorithm is correct.

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.

Invariant: The subarray *A* [*1...i-1*] *is sorted. A*[*i-1*] is the smallest value in range of

A [*i...A.length*]

Initialization: Before the first iteration of the loop*, i =1 and t*he subarray *A* [*1...i -1*] is empty.

Maintenance: Before the i times iteration of the loop, the subarray *A* [*1...i-1*] *is sorted and A*[*i-1*] is the smallest value in range of *A* [*i...A.length*]. In this iteration, according to the answer of b, while we terminate the inner loop*(which means j = i)*, *A*[*i*] is the smallest value in range of *A* [*i+1...A.length*].

Thus, the subarray *A* [*1...i*] *is sorted and A*[*i*] is the smallest value in range of *A* [*i+1...A.length*].

Termination: While *i > A.length - 1* which means *i = A.length,* we terminate the loop. All value in *A* [*1...A.length -1*] are sorted and less than the value of *A.length.*By definition, the values in *A[1...A.length]* are sorted.Hence, the algorithm is correct.

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 is that we have a reverse array need to sort and we need to run the fourth statement for every time.

Statement	Cost	Times
for i = 1 to A.length – 1	c_1	n
for j = A.length downto i + 1	c_2	$\sum_{j=2}^{n} j$
if A[j] < A[j-1]	c_3	$\sum_{j=2}^{n} (j-1)$
exchange A[j] with A[j-1]	c_4	$\sum_{j=2}^{n} (j-1)$

$T(n) = c_1\, n + c_2 \sum_{j=2}^{n} j + c_3 \sum_{j=2}^{n} (j-1) + c_4 \sum_{j=2}^{n} (j-1)$
The big-Oh complexity is $O(n^2)$.

Comparing to the worst-case running time of Insertion sort.

$T(n) = c_1 n + c_2(n-1) + c_4\,(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8\,(\mathrm{n}-1)$

In the worst-case, they have the same big-Oh complexity $O(n^2)$.

Name: Cuiting Huang Homework 05 - Proofs

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 ⟩

$(1,5),(2,5)(3,5)(4,5)$
$(3,4)$

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*?

The array is a reverse array. *⟨ n, n-1, … 1 ⟩.*
The number of inversions of the array is

$$(n-1)+(n-2)+\ldots+2+1=\frac{n(n-1)}{2}=c_n^2$$

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.

When there is an inversion, the body of ***the while loop*** of insertion sort will operate once.
The running time of the while loop in insertion sort is $O(n+m)$, m is the number of inversions in the input array.

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.
ANS

```
BINARY-SEARCH(A)
1       int min = 1
2       int max = A.length
3       int value v
4       while(min<=max)
5               mid=(min+max)/2
6               if A[mid]=value v then
7                       return mid
8               else if A[mid]>value v then
9                       max=mid-1
10              else if A[mid] <value v then
11                      min=mid+1
12      return -1 // means we find nothing
```

b. Make an argument based on this pseudocode for the worst-case runtime.
ANS
The worst case:
There is a sequence array *A*[*1....n*]
max = n ,min =1, value v, compare A[*mid*] with value, the rest numbers of array is *n/2,*
max = n/2, min = 1 compare again, the rest numbers of array is *n/4,*
max = n/4, min = 1 compare again, the rest numbers of array is *n/16,*
.....
every time, the rest numbers of array is n/(2^m). When n/(2^m) = 1, we could get the *value v* or nothing.
Thus, *n = (2^m),*the worst case runtime of binary search is *m =* $log_2(n)$. The big-Oh is *O(logn).*

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

Invariant: There is a sequence array *A*[*i....j*], in the middle of array, *mid =(i+j)/2*, A[mid] is not equals to the target value *v.*

Initialization: Before the first iteration of the loop,we try to search the target value from *i* to *j* of the array. *i = 1* , *j =* the number of array elements.

Maintenance: For each iteration of the loop, we check *A*[*mid*] is not the target number, and get the new value of *i* and *j.* if *A*[*mid*] > value *k, i = i, j = mid - 1;* if *A*[*mid*] < value *k, i = mid + 1, j = j, which is* showing that each iteration maintains the loop invariant.

Termination: If we terminate the loop with returning index *mid*, it is when we already find an element *A*[*mid*] equals to the value *v.* In this case, our algorithm is correct.

If we terminate the loop without returning index *mid*, which means there is not an element of *A*[*i...j*] equals to the value *k.* And we return -1. Hence, the algorithm is correct.

Name: Cuiting Huang Homework 05 - Proofs

Problem 4: Improve Bubblesort performance

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

ANS

```
1 for i = 1 to A.length – 1
2       int flag = 1 //set a flag
3       for j = A.length downto i + 1
4               if A[j] < A[j-1]
5                       exchange A[j] with A[j-1]
6                       flag = 0 // if we make an exchange, give a new value to flag
7       if flag == 1
8               break   // if the array is sorted  break the whole loop.
```

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

Invariant: The subarray *A* [*1...i-1*] *is sorted. A*[*i-1*] is the smallest value in range of

A [*i...A.length*]

Initialization: Before the first iteration of the loop*, i =1 and t*he subarray *A* [*1...i -1*] is empty.

Maintenance: Before the i times iteration of the loop, the subarray *A* [*1...i-1*] *is sorted and A*[*i-1*] is the smallest value in range of *A* [*i...A.length*].

In this iteration, in the inner for loop, we compare *A*[*j*] with *A*[*j-1*] ,if *A*[*j*] is the smaller one we swap the two numbers and give the flag new value, for each iteration of inner loop, we traverse all element in the array.

While we terminate the inner loop*(which means j = i)*, *A*[*i*] is the smallest value in range of *A* [*i+1...A.length*].

Thus, the subarray *A* [*1...i*] *is sorted and A*[*i*] is the smallest value in range of *A* [*i+1...A.length*].

Termination: if *i <A.length,* we terminate the loop with flag = 1, which means for all loop, All value in *A*[*1...A.length -1*] are sorted, *A*[*i*] is always bigger than *A*[*i-1*] and less than the value of *A.length.*By definition, the values in *A[1...A.length]* are sorted. Hence, the algorithm is correct.

While *i > A.length - 1* which means *i = A.length,* we also terminate the loop. For all loop, *j=i.* All value in *A* [*1...A.length -1*] are sorted and less than the value of *A.length.*By definition, the values in *A* [*1...A.length*] are sorted.Hence, the algorithm is correct.

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

Statement	Cost	Times
for i = 1 to A.length – 1	c_1	n
int flag = 1	c_2	n
for j = A.length downto i + 1	c_3	$\sum_{j=2}^{n} j$
if A[j] < A[j-1]	c_4	$\sum_{j=2}^{n} (j-1)$
exchange A[j] with A[j-1]	c_5	$\sum_{j=2}^{n} (j-1)$
flag = 0	c_6	$\sum_{j=2}^{n} (j-1)$
if flag = 1	c_7	n

$$T(n) = c_{1n} + c_2 n + c_3 \sum_{j=2}^{n} j + c_4 \sum_{j=2}^{n} (j-1) + c_5 \sum_{j=2}^{n} (j-1) + c_6 \sum_{j=2}^{n} (j-1) + c_7 n$$

$$T(n) = c_1 n + c_2 n + n * c_3 (n-1) + n * (c_4 + c_5 + c_6)(n-3) + c_7 n$$

In the best case, we have an sequence array, c3,c4 will be operated for one time and c5.c6 will never be operated. We operate c_7 for once and break the loop.
$T(n) = c_1 n + c_2 n + c_3 (n-1) + c_4 (n-3) + c_7 n$
$T(n) = (c1 + c_2 + c_3 + c_4 + c_7) n - c_3 - 3c_4$
The big-Oh complexity is $O(n)$.

In the other cases, every statement will be operate at least twice
$T(n) = (c_3 + c_4 + c_5 + c_6) n^2 + (c1 + c_2 - c_3 - 3c_4 - 3c_5 - 3c_6 + c_7) n$
The big-Oh complexity is still $O(n^2)$.