ICS 311, Fall 2020, Problem Set 01, Topic 2

#1. Correctness of Linear Search

7 points

(a) Show the pseudocode for Linear Search that you will be analyzing. (It should be code that you understand and believe is correct. You may revise your solution from class, or use the instructor's solution.) Give each line a number for reference in your analysis.

Example Code (yours may differ):

```
linearSearch(A, v)
1  for i = 0 to A.length - 1
2      if A[i] == v
3          return i
4  return NIL
```

(b) Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties (page 19 CLRS).

(Hint: The loop can exit for two reasons. Rather than trying to write and prove an invariant that covers the two cases, use a simpler invariant that deals with only the correctness of the loop completion returned value, as it would be too complex to cover both, and the within-loop returned value is easy to show correct. This is similar to what we did in class for binary search.)

Invariant: The key *k* does not occur in A[0 .. *i*-1]
Properties:

The invariant is true before the first execution of the loop body trivially since *i* is initialized to 0 and the key does not occur in A[0, -1].

The invariant is maintained by the loop body by induction: Suppose we are entering the loop with loop index *i*. By the inductive hypothesis, the key is not in A[0..*i*-1]. Line 1 only increments the loop index to *i+1* if the key is not found at A[*i*] by line 2, as otherwise the loop exits at line 3 before incrementing *i*. Therefore if we reach *i*+1 the invariant is also true for A[0..*i*].

The invariant is true when the loop exits depending on why the loop exited:

- If the exit occurs in line 1, then *i* == A.length: the key is not in the array A[0..A.length-1].
- If the exit occurs in line 3, then line 2 evaluated true, and the key is at A[*i*]: the

loop would have exited earlier if it had been found at *any* position < *i*, so the invariant is true.

The invariant helps to prove the algorithm correct as follows: The loop exits either because the key has been found at position A[*i*] in line 2, in which case the algorithm correctly returns the (first) key in line 3, or because *i* == A.length. In this latter case, we know by the invariant that *k* is not in A[0 .. A.length-1], so the key does not exist in the array, and the algorithm correctly returns NIL in line 4.

#2. Runtime of BinarySearch

8 points

This problem steps you through a recursion tree analysis of BinarySearch to show that it is Θ(lg *n*) in the worst case. Here is a recursive version of BinarySearch:

```
BinarySearch(x, A, low, high)
1    if (low > high)
2         return "NOT FOUND" // or a sentinel such as -1
3    else
4         mid =⌊(low + high) / 2⌋
5         if x < A[mid]
6              return BinarySearch(x, A, low, mid-1)
7         else if x > A[mid]
8              return BinarySearch(x, A, mid+1, high)
9         else
10             return mid
```

(Comments: The problem involves mathematical equations and drawing diagrams. It is your choice whether to figure out how to do this in Google Docs & Drawing, to do it in your favorite program, or to do your work on paper and scan or photograph it. We prefer that you compile the results in one PDF, but if that is difficult compile them into one zip file.)

(a) Write the recurrence relation for BinarySearch, using the formula T(*n*) = *a*T(*n*/*b*) + D(*n*) + C(*n*). (We'll assume T(1) = some constant *c*, and you can use *c* to represent other constants as well, since we can choose *c* to be large enough to work as an upper bound everywhere it is used.)

The base case of lines 1-2 costs some constant c_1.
By inspection of the code, a=1, b=2, D(n) = c_2, C(n) = c_3. So:

$T(1) = c_1$
$T(n) = T(n/2) + c_2 + c_3$

Let $c \geq c_1 + c_2 + c_3$. Then we can write this as:
T(*1*) = *c*
T(n) = T(n/2) + c

(b) Draw the recursion tree for BinarySearch, in the style shown in podcast 2E and in Figure 2.5 of CLRS. Don't just copy the example for MergeSort: it will be incorrect. Make use of the recurrence relation you just wrote!

Obviously we have *no branching*, as half of the data is ignored (rather than processed) on each call. The height of this tree is bounded by lg n + 1, log n being the number of times you can divide n in half before you reach 1, and 1 for the root. (Students may indicate this fact diagramatically.)

$$\begin{tikzcd} c \arrow[d, dash] \\ c \arrow[d, dash] \\ c \arrow[d, dash] \\ \cdots \arrow[d, dash] \\ c \end{tikzcd}$$

(c) Using a format similar to the counting argument in Figure 2.5 of the text or of podcast 2E, **use the tree to show that BinarySearch is Θ(lg *n*) in the worst case**. Specifically,

1. show what the row totals are,
2. write an expression for the tree height (justifying it), and
3. use this information to determine the total computation represented by the tree.

    1. The row totals are each c.
    2. The tree height is bounded by lg n + 1 (see above).
    3. Then total computation is c lg n + c = Θ(lg *n*)

#3. Correctness of BubbleSort

15 points

BubbleSort is a well known but inefficient sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order, and doing so enough times that there can be no more elements out of order. From CLRS:

```
BUBBLESORT(A)
for i = 1 to A.length - 1
  for j = A.length downto i + 1
      if A[j] < A[j - 1]
          exchange A[j] with A[j - 1]
```

Let A' (an array) denote the output of BubbleSort. In order to prove that BubbleSort is correct, we need to show that

1. A' is a permutation of A; that is, A' has the same elements as A.
2. A'[1] ≤ A'[2] ≤ A'[3] ... ≤ A'[n-1] ≤ A'[n]; that is, A' is sorted

The first part can be proven by showing that BubbleSort never removes or adds items to the array: it merely swaps them in line 4. In this homework, you will use loop invariants to prove the second part, that the resulting array is in order. Since there are nested loops this requires two loop invariants, handled in (a) and (b).

(a) State precisely a loop invariant for the **for** loop in lines 2-4, and prove that this loop invariant holds. Your proof should use the structure of the loop invariant proof presented in Chapter 2 of CLRS.

> **Loop invariant:** At the start of each iteration of the **for** loop of lines 2–4, A[j] = min { A[k] : j ≤ k ≤ n}
>
> **Initialization:** Initially, j = n, and the subarray A[j . . n] consists of single element A[n], which trivially is the minimum.
>
> **Maintenance:** By the loop invariant at entry, A[j] is the smallest of A[j … n]. In lines 3-4, A[j-1] and A[j] swap values if A[j] is smaller than A[j-1]. Thus, after line 4, A[j-1] is the smallest element among A[j-1 … n]. Decrementing j keeps the invariant true for the next iteration.
>
> **Termination:** Loop terminates when j reaches i. By the loop invariant A[i] contains the smallest element among A[i … n]

(b) Using the termination condition of the loop invariant proved in part (a), state a loop invariant for the **for** loop in lines 1-4 that will allow you to prove the inequality A'[1] ≤ A'[2] ≤ A'[3] ... ≤ A'[n-1] ≤ A'[n]. Your proof should use the structure of the loop invariant proof presented in Chapter 2 of CLRS.

> **Loop invariant:** At the start of each iteration of the **for** loop of lines 1–4, the subarray A[1 . . i − 1] consists of the i − 1 smallest values originally in A[1 . . n], in sorted order.
>
> **Initialization:** Before the first iteration of the loop, i = 1. The subarray A[1 . . i − 1] is empty, so the invariant is vacuously true.
>
> **Maintenance:** Consider an iteration for a given value of i. By the loop invariant, A[1 . . i − 1] consists of the i smallest values in A[1 . . n], in sorted order. Part (a) showed that after executing the **for** loop of lines 2–4, A[i] is the smallest value in A[i . . n] (that is, it is the next smallest item after those in A[1 .. i−1]), and so A[1 . . i] is now the i smallest values originally in A[1 . . n], in sorted order.
>
> **Termination:** The for loop of lines 1–4 terminates when i = n + 1, so that i − 1 = n. By the statement of the loop invariant, A[1 . . i − 1] is the entire array A[1 . . n], and it consists of the original array A[1 . . n], in sorted order.

(c) Three parts:

- Give the <u>worst-case</u> running time of BubbleSort, in big-O notation, with justification.
- Give the <u>best-case</u> running time of BubbleSort, in big-O notation, with justification.
- How do these compare to the running time of InsertionSort?

The only conditional portion that depends on the data is lines 3-4. The worst case will be when the line 3 test always evaluates true, so the line 4 swap always has to be done. However, comparison and swap both require constant time, so overall lines 3-4 require constant time regardless of the data (i.e., whether a swap is done). So, worst-case and best-case will be asymptotically the same.

For data of size *n*, the outer loop runs *n*-1 = O(*n*) times. The number of times the inner loop runs depends on *i*, running once when *i* = *n*-1, ranging up to *n*-1 = O(*n*) times when *i* = 1. Since we are doing worst case analysis, we can use O(*n*) as the upper bound for the inner loop. Since the inner loop is nested in the outer, we multiply to get O(n^2).

A more precise analysis follows, accounting for the changing iterations of the inner loop.

$$\begin{aligned}
\sum_{i=1}^{n-1}\sum_{j=n}^{i+1} c &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} c \\
&= \sum_{i=1}^{n-1} c(n-(i+1)-1) \\
&= \sum_{i=1}^{n-1} c(n-i) \\
&= c\left(\sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i\right) \\
&= c\left(n(n-1) - \tfrac{n(n-1)}{2}\right) \\
&= c \cdot \tfrac{n(n-1)}{2} \\
&= O(n^2)
\end{aligned}$$

The worst-case runtime of BubbleSort is the same as the worst-case runtime of insertion sort. However, the best case of BubbleSort is also O(n^2), which is worse than the O(*n*) best case of insertion sort.