1. Correctness of Linear Search

(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.

Algorithm 1 Linear-Search(A,v)

```
1: for i = 1 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 the three necessary properties (page 19 CLRS).

- **Initialization**
  Before the for loop is executed, the variable i is initiated, therefore, this implies that the key, otherwise A[i], is not located at A[0], A[-1], to A[-i].
- **Maintenance**
  As the loop continues to iterate, it moves from left to right of the array, with the value increasing from A[i-1], A[i-2]...,A[i-n]. As the loop continues, it must be verified that A[i] == key. If these values are not equal to the key, then the algorithm returns to the index of the loop, and moves to the next.
- **Termination**
  Since the code initially has two cases, it will break up into two cases.
  - The code will initially find the element denoted as v and will return the index in the array in which it found the key at.
  - The for loop will completely terminate once it cycles through the n number of elements. Since the index is initialized as i = n+1, and because the loop invariant at A[1...n] doesn't contain the key, denoted as v, it will return NIL.

2. Runtime of Binary Search

This problem steps you through a recursion tree analysis of BinarySearch to show that it is $\theta(lgn)$ in the worst case. We can use the same code as we used in class:

Algorithm 2 Binary-Search(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
```

(a) **Write the recurrence relation for BinarySearch**, using the formula $T(n) = aT(\frac{n}{b} + D(n) + C(n))$. (We'll assume $T(1)$ = some constant 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.)

It is noted that the base condition of BinarySearch is given as $n = 1$. Since the logic statement of whether or not the low index > high index, the loop will not enter and will return no index found; else, it will cycle through the loop. This returns $T(n) = O(1)$.

Lines 3-10 will then generate a recursive call, where the average of the variables of low and high return the midpoint of the array. The recurrence relation can then be written as follows:

$$\text{T(1)} = \text{c such that c=1}$$

$$\text{T(n)} = \tfrac{n}{2} + c \text{ such that n} \geqslant 2$$

(b) **Draw the Recursion Tree for Binary Search**, 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!

Recursion Tree For Binary Search

c
|
c
|
...
|
c

There is no branching because the data that is broken in half is ignored on each call. Since the data is broken and half, and because there is no manipulation in the data, there will be no branching in this Recursion Tree. Therefore each call of $T(n/2)...$ will be equal to $O(1)$ which is equivalent to the constant c. In addition, since this is a binary search, there is no branching involved.

(c) Using a format similar to the counting argument in Figure 2.5 of the text or of podcase 2E, **use the tree to show that BinarySearch is $\theta(lgn)$ 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.

On the basis that this information is determined by the recursion tree drawn in part (b), it is noted that each level of this tree is denoted by $lg(n)+1$. 1 is a representation of the constant values of this code, while $lg(n)$ is a representation of the height of the given tree, since it is true that $lg(n)$ can be the number of times the given n can be divided before reaching the bottom of the tree, denoted as 1. Therefore, it is given that the running time of a Binary-Search is $\theta(\lg(n))$.

3. Correctness of Bubble Sort

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:

Algorithm 3 Bubble-Sort(A)

```
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) 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 pre-

sented in Chapter 2 of CLRS.

- **Loop Invariant**
  At the beginning of the iteration on line 2, it is given that the array of A[j...n] is to be sorted, starting from the index, and traversing through until the end of the array is reached, given the end of the array is the nth iteration. Once A[j...n] is sorted, the loop invariant terminates, given that the array is sorted and the smallest position of the array is to be A[j].
- **Initialization**
  The loop invariant follows that A[j] is indeed, the smallest element in the given array. If this is true, then the array is sorted, and it contains one element that is the smallest.
- **Maintenance**
  To maintain this loop invariant, the loop compares the array, A[j] to A[j-1], and determines which of these two values are smaller. If A[j] > A[j-1], then the indices are swapped. This continues for all R n, until the array is completely sorted and until all elements within the array have been traversed.
- **Termination**
  If the array of A[j...n] contains the elements of A[i...n], and once j=i, the loop terminates. The format concludes that A[j] is indeed the smallest element of the array A[j...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 $A'[1]'[2]'[3]...'[n-1]'[n]$. Your proof should use the structure of the loop invariant proof presented in Chapter 2 of CLRS.

- **Loop Invariant**
  At the start of the iteration on line 1 of this algorithm, the array A[i...i+1] is sorted, starting from the index of the array to the end of the iteration, given the iteration reaches its last nth item. Once all of the elements within A[1...n] are sorted, the loop invariant terminates.
- **Initialization**
  The first element is started at the subarray of A[i...i-1]. It is thus assumed that the subarray is sorted, and therefore the first element is to be the smallest within the array.
- **Maintenance**
  Within the loop, A[i] is compared to the next given iteration, A[i-1]. If the value of A[i-1] is smaller than the given value of A[i], then these adjacent elements are swapped. Given n is an integer within this same array, the pattern follows a sequential order, in which A[n] continues until the array has been fully sorted.

- **Termination**
  The loop is terminated when the variable i = A.length - 1, which is similar to when the array has been fully traversed. Once the loop has reached the end of the array, it is given through the algorithm that A[1...n] is completely sorted.

(c) Three Parts:

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

  - **Worst-Case running time for Bubble-Sort**

    The worst-case running time for Bubble-Sort is $O(n^2)$. This case follows that none of the elements within the array are sorted, and that every given value within the array with all elements denoted as n need to be rearranged and swapped. If the array is arranged within a reverse order, the loop must go through n-i iterations, meaning that it does two times the amount of work as oppose to one.

  - **Best-Case running time for Bubble Sort**

    Bubble Sort has a best case running time of $O(n)$. In this best case, it is assumed that all the elements within the array have already been sorted, or some elements in the array are sorted. It is then determined that the loop will only iterate once, which gives the best case running time of $O(n)$.

  - **Compare with Insertion-Sort**

    Both insertion sort has a best case running time of $O(n)$ and that it has a worst-case running time of $O(n^2)$. Due to this truth, given in CLRS and Data Structures, the given running time and performance for bubble sort and insertion-sort are of equivalence through comparison.