1. Correctness of Linear Search

(a) Pseudocode for Linear Search

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) Loop Invariant

If it exists in the array A[1 .. n] then v must exists between the array of A[1 ... n]

- **Initialization**
  Before the loop, i is initialized at i = 1 , therefore, it follows that the key k is not located at A[0] or A[-1] ... A[-i].
- **Maintenance**
  Proof by induction.
  Suppose that the loop is entered in the index i such that the sub-array A[1 ... (i-1)] does not contain key v. In the i^{th} iteration determine whether $A[i] = v$. If it is true then stop at the index i.
  Otherwise, then $A[i] \neq v$ so then from line 1, take an increment of i+1. Then the loop invariant remains true.
- **Termination**
  The code will terminate with two cases, therefore, we will break it up into two cases.
  - When the code finds element v and returns the index in which it found key v at
  - The loop will terminate after looping through all of n elements. Then the index will be i = n+1 and by the loop invariant the subarray A[1 ... n] does not contain key v and therefore returns NIL

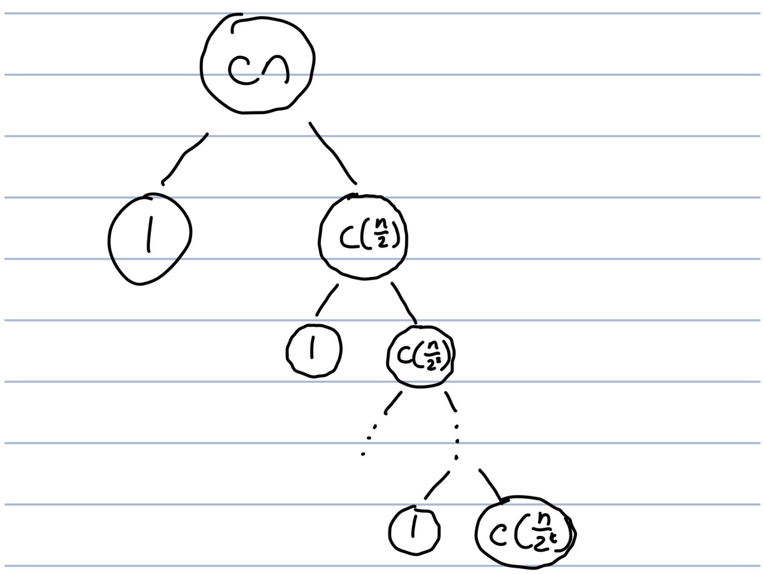
2. Runtime of Binary Search

Algorithm 2 Binary-Search(x, A, low, high)

1: **if** (low > high)
2:     **return** "NOT FOUND"
3: **else**
4:     mid = $\lfloor \frac{(\text{low} + \text{high})}{2} \rfloor$
5: **if** x < A[mid]
6:     **return** Binary-Search(x, A, low, mid-1)
7: **else if** x>A[mid]
8:     **return** Binary-Search(x, A, mid+1, high)
9: **else**
10:     **return** mid

(a) **recurrence relation for Binary Search**
From the direction, it is given that at n = 1 then T(1) = c_1 such that c_1 is a constant. At line 4 we are breaking n into half, therefore, T(n) = $T(\frac{n}{2}) + c_2 + c_3$
Let $c \geq c_1 + c_2 + c_3$ then the recurrence relation of Binary Search is the following T(n)
$= \begin{cases} c & n = 1 \\ T(\frac{n}{2}) + c & n \geq 2 \end{cases}$

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



(c) **Use the Tree to show that Binary Search is $\theta(\mathbf{lg}n)$**
Based on the recursion tree written in part (b) the row total are each c and that the height of the tree is bounded by $\lg n + 1$ since lg(n) is the times one can divide n before reaching 1.

Therefore, we have that $\lg(n) + 1 + c$. Since c and 1 are both constants $\in \mathbb{Z}^+$ then let d $= c + 1$ then it can be further rewritten as $\lg(n) + d$. This follows that the highest power in the equation is $\lg(n)$.
Thus, the running time of a Binary-Search is $\theta(\lg(n))$.

3. Correctness of Bubble Sort

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) Loop Invariant for line 2 - 4 and proof

- **Loop Invariant**
  If the sub-array A[j...n] contains the element originally in A[j...n] with different order, then the in an organized order, A[j] contains the smallest element.
- **Initialization**
  Initially, before entering the loop the element in A[n] is the smallest element.
- **Maintenance**
  In each iteration, the loop compares $A[j]$ and $A[j-1]$ and makes $A[j-1]$ the smallest element. After that the length of the sub-array increments by one and the first element is the smallest element.
- **Termination**
  The loop terminates when hitting j = i. Furthermore, from the loop invariant the sub-array A[j .. n] contains elements from the array A[i..n] such that A[j] in the subarray of A[j ..n] is the smallest element.

(b) Loop Invariant for line 1-4 and its proof

- **Loop Invariant**
  If the sub-array A[i...i-1] contains i-1 smallest elements in the original array then after the start of **line 1-4** the original array of A[1...n] is in sorted order.
- **Initialization**
  Initially the sub-array A[1 .. i - 1] is empty such that by trivial assumption is the smallest element.
- **Maintenance**
  In each iteration for a given value of i. By the loop invariant in part (a) then A[i] is the smallest value of the sorted array in A[1...n].
- **Termination**
  The loop itself terminates when $i =$ A.length.Then the array A[1 ... n] is sorted in order.

(c) – Worst-Case running time for Bubble-Sort
  The worst-case of running time is $O(n^2)$ because at the for loop within line 2 -4 depends on the value i. For instance if i = 1 then the loop will iterate $n-1$ times

of if i = 2 then the loop will iterate $n-2$ times. Therefore, the loop will iterate $n-i$ times. This implies the total iteration will equal

$$\sum_{i=1}^{n-1} n-i = \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i = n(n-1) - \frac{(n-1+1)(n-1)}{2} = n^2 - n - \frac{n^2-n}{2}$$

Notice how the highest power will be a cn^2 where c is a constant. Therefore, the worst case of bubble sort based on the recurrence is O(n^2).

- Best-Case running time for Bubble Sort
  The best-case running time is same as the worst case time for bubble sort and would be O(n^2) because of the algorithm.
- Compare with Insertion-Sort
  Recall that from ICS 211 and Chapter 2.1 of the CRLS textbook that both insertion sort has a best case running time of O(n) and that it has a worst-case running time of $O(n^2)$. Therefore, the running time for both bubble sort and insertion-sort are both equivalent by the worst case run time but insertion has a faster run time in comparison.