Julian Garcia
CSE 344: HW3

-I certify that every answer in this assignment is the result of my own work; thatI have neither copied off the Internet nor from any one else's work; and I have not shared my answers or attempts at answers with anyone else

1. TRUE - Insertion sort is a stable sorting algorithm because it is a comparison-based algorithm in that it checks the size of the current element in the list and compares it to the next element in the list, only changing the order if the next element in the list is less than the current, so the order is preserved when both elements are of equal value.
2. Based on this algorithm:

```
HEAPIFY(A, i) ▷ MAX-HEAPIFY
1   ▷ Assumption: A: array, i: index; almost complete binary tree rooted at
2   ▷ node i (i.e., A[i]) may not be a heap, but those rooted at
3   ▷ nodes left(i) and right(i) are heaps
4   l ← left(i)
5   r ← right(i)    ▷ Note: l, r are node numbers, not values!
6   if l ≤ A.heapsize and A[l] > A[i]
7       then largest ← l
8       else largest ← i
9   if r ≤ A.heapsize and A[r] > A[largest]
10      then largest ← r
11  ▷ largest contains the index of the node containing
12  ▷ the maximum of the values in nodes i, left(i), right(i)
13  if largest <> i then
14      exchange A[i] with A[largest]
15      HEAPIFY(A, largest)
```

The only recursive call here is on line 15: Heapify(A, Largest), and this is the only statement that will have an effect on stack space.
So using recurrence relations we have:
C(1) = Θ(1) = c'
C(h) = c' + C(n - 1)

C(1) = c'
C(2) = 2c'
C(3) = 3c'
C(4) = 4c'
C(n) = nc'
C(n) = Θ(n)

3. HEAPINCREASEKEY(A, I, ∆)

```
A[i] <- A[i] + ∆  \\ Increase/decrease value at i by ∆
l <- left(i)              \\ Gives node of left child
r <- right(i)             \\ Gives node of right child
p <- parent(i)            \\ Gives parent of node at i

\\ check to see if a parent or child is out of order, or if we're at top of heap
While i > 1 and (A[l] > A[i] or A[r] > A[i] or A[p] < A[i])
    if A[l] > A[i]
        exchange A[i] and A[l]
        i <- l
    if A[r] > A[i]
        exchange A[i] and A[r]
        i <- r
    if A[p] < A[i]
        exchange A[i] and A[p]
        i <- p
```

Why it's O(logn):

In the worst case this while loop goes from the top of the heap to the bottom, but in that case it only traverses one path in the whole heap, so it only traverses the height of the heap, thinking of the entire heap as our input n, the height of the heap would be lg(n), since this algorithm only iterates through the height of the heap, it is therefore bound by O(lg(n)).