1.

a)

[10 points] Describe the complexity of the linear search algorithm.

```
LINEAR-SEARCH(x, A)
1  i = 0
2  while i < n and x ≠ a_i
3        i = i + 1
4  if i < n
5        location = i
6  else location = nil
7  return location
```

Complexity O(n) time, O(1) space

b)

[10 points] Describe the time complexity of the binary search algorithm in terms of number of comparisons used (ignore the time required to compute m = b(i + j)=2c in each iteration of the loop in the algorithm) as well as showing the worst case complexity. Note: Do not use the algorithm on page 799 of the book, use the following algorithm.

```
BINARY-SEARCH(x, A)
1  i = 0
2  j = n-1
3  while i < j
4        m = ⌊(i + j)/2⌋
5        if x > a_m
6              i = m + 1
7        else j = m
8  if x = a_i
9        location = i
10 else location = −1
11 return location
```

Every comparison cuts the list of size n in half, thus

$$2^{\#\ comparisons} = n$$

Reorganizing:

$$comparisons = \lg(n)$$

And the complexity is O(lg(n)) time and O(1) space

2. [10 points] Give asymptotic upper and lower bounds for T(n) = 3T(n/2) + n lg n, assuming T(n) is constant for sufficiently small n. Make your bounds as tight as possible. Prove that your bounds are correct.

By master we have a = 3, b = 2, and $n^{log_2 3} \sim n^{1.585} > n \lg n$ (for sufficiently large n)
Case 1 gives us $T(n) = O(n^{log_2 3})$

3. [15 points] Give asymptotic upper and lower bounds for T(n) = T(n^1/2) + 1, assuming T(n) is constant for sufficiently small n. Make your bounds as tight as possible. Prove that your bounds are correct.

let $m = lgn$

$$S(m) = S(m/2) + 1$$

By master we have a = 1, b = 2, and $n^{log_2 1} = n^0 = O(1)$
Case 2 applies, thus the solution is O(lgn * m) = O(lg^2(n))

4. [20 points] Give asymptotic upper and lower bounds for T(n) = 2T (n/3 + 1) + n, assuming T(n) is constant for sufficiently small n. Make your bounds as tight as possible. Prove that your bounds are correct.

For sufficiently large n, n/3 + 1 asymptotically approaches n/3, thus simplifies the problem to T(n) = 2T (n/3) + n
By master we have a = 2, b = 3, $n^{log_3 2} \sim n^{.9542} < n$
Case 3 applies, thus the solution is O(n)

5. [35 points] **Collaborative Problem** –CLRS 2-1: Although merge sort runs in _(n lg n) worst-case time and insertion sort runs in _(n2) worst-case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. Thus, it makes sense to coarsen the leaves of the recursion by using insertion sort within merge sort when subproblems become sufficiently small. Consider a modification to merge sort in which n/k sublists of length k are sorted using insertion sort and then merged using the standard merging mechanism, where k is a value to be determined.
(a) [10 points] Prove that insertion sort can sort the n=k sublists, each of length k, in O(nk) worst-case time.

Insertion sort can be done in O(n^2) time
length of list to sort is k
number of lists to sort is n/k
O(k^2) * O(n/k) = O(nk)

(b) [10 points] Prove how to merge the sublists in O(n lg (n/k)) worst-case time.

Sorting c sublists of length k takes takes T(c) = 2T(c/2) + ck, as merging sublists requires dividing them into groups of c/2 lists, merging groups, and combining the results in c * k steps.

By master, a = 2, b = 2, $c^{log_2 2} = c = O(ck)$
Case 2 applies, thus the solution is O(n lg(c) = O(n lg(n/k))

(c) [10 points] Given that the modified algorithm runs in O(nk + n lg (n/k)) worst-case time, what is the largest value of k as a function of n for which the modified algorithm has the same running time as standard merge sort, in terms of O-notation?

Merge sort has time O(nlgn)
O(nk + nlgn – nlgk)
With k = O(lgn)
O(nlgn + nlgn – nlglgk)
= O(nlgn)
Any value larger than k = O(lgn) will produce algorithm with worse runtime than O(lgn)

(d) [5 points] How should we choose k in practice?

k should be the largest length of sublist with insertion sort runtime faster than the merge sort