1. *Describe the time complexity of the linear search algorithm. Choose the tightest asymptotic representation, from Θ, O, or Ω, and argue why that is the tightest bound.*

The time complexity of the linear search algorithm is O(n), where n is the number of elements in the array.

This is the tightest asymptotic upper bound because, in the worst case, the algorithm must search through all n elements of the array to find the target item x. The number of iterations in the while loop is directly proportional to the size of the array, so the growth of the time complexity is linear with respect to the size of the input.

In the best case, when the target item x is found in the first iteration, the time complexity is Ω(1), but this is not a tight bound because the algorithm could potentially take O(n) time in the worst case. Thus, O(n) is the tightest asymptotic upper bound for the linear search algorithm.

2. *Analyze the following binary search algorithm. Assume the input A is a sorted list of elements; x may or may not be in A.*

(a) The time complexity of the binary search algorithm is O(log n), where n is the number of elements in the array.

This is the tightest asymptotic upper bound because, in the worst case, the algorithm will make log base 2 (n) comparisons before finding the target item x or determining that x is not in the array. This is because with each iteration, the size of the search interval is halved. As a result, the running time of the algorithm grows logarithmically with respect to the size of the input.

In the best case, when the target item x is found in the first iteration, the time complexity is Ω(1), but this is not a tight bound because the algorithm could potentially take O(log n) time in the worst case. Thus, O(log n) is the tightest asymptotic upper bound for the binary search algorithm.

(b) To improve the runtime of the binary search algorithm, we can make one small change by adding a check for the target item x at the end of the while loop before checking if x = A[i]. The revised algorithm is as follows:

Binary Search (Improved):

Input: sorted array A (indexed from 1), search item x
Output: index into A of item x if found, zero otherwise

```
function BINARY-SEARCH(x, A)
    i = 1
    j = len(A)
    while i < j do
        m = (i + j)/2
        if x = A[m] then
            return m
        else if x > A[m] then
            i = m + 1
        else
    if x = A[i] then
        return i
    else
        return 0
```

With this change, the time complexity remains the same: O(log n), where n is the number of elements in the array. This is because the running time of the algorithm is still logarithmic with respect to the size of the input, but the average number of comparisons made is reduced. In the best case, when the target item x is found in the first iteration, the time complexity is O(1).

3. *Use the Master Theorem to find the asymptotic bounds*

For the recurrence relation $T(n) = 4T(\frac{n}{4}) + n^2$, we have:

$a = 4,\ b = 4,\ f(n) = n^2$

We apply Case 3 of the Master Theorem, which states that if $f(n) = \Theta(n^{log_b a+\varepsilon})$ for some $\varepsilon > 0$, then $T(n) = \Theta(f(n^2))$.

In this case, $log_4 4 = 1$, and $n^2 = \Theta(n^{1+\varepsilon})) \ where\ \varepsilon = 1$, so the conditions for Case 3 are met.

For the $af(\frac{n}{b}) \leq cf(n)$ regularity condition, we can have $4\frac{n^2}{4^2} = \frac{n^2}{4} \leq cn^2$ then have $\frac{1}{4} \leq c$.

Therefore, by Case 3 of the Master Theorem, $T(n) = \Theta(f(n)) = \Theta(n^2)$.

So the asymptotic upper bound for $T(n)$ is $O(n^2)$, and the asymptotic lower bound is $\Omega(n^2)$.

4. *Use the Master Theorem to find the asymptotic bounds*

For the recurrence relation $T(n) = 3T(\frac{n}{3} + 1) + n$, let $u = n + 3$ for substitution, we have $T(u - 3) = 3T(\frac{u}{3}) + u - 3$ and then $a = 3,\ b = 3,\ f(n) = u - 3$.

We apply Case 2 of the Master Theorem, which states that if $f(n) = \Theta(n^{log_b a})$ for, then $T(n) = \Theta(n^{log_b a} lg\ n)$.

In this case, $log_4 4 = 1$, and $u - 3 = \Theta(u)$, so the conditions for Case 2 are met.

Therefore, by Case 2 of the Master Theorem,
$T(n) = T(u - 3) = \Theta(u\ lg\ u) = \Theta((n - 3)\ lg\ (n - 3))$ =
$\Theta(n\ lg\ (n - 3) - 3lg\ (n - 3)) = \Theta(n\ lg\ (n - 3)) = \Theta(n\ lg\ n)$.

So the asymptotic upper bound for $T(n)$ is $O(n\ lg\ n)$, and the asymptotic lower bound is $\Omega(n\ lg\ n)$.

5. *Collaborative Problem*

(a) Insertion sort takes $\Theta(k^2)$ time to sort a sublist of length k, and there are $\frac{n}{k}$ sublists, so the total time to sort all sublists is $\Theta(k^2 * \frac{n}{k}) = \Theta(nk)$.

(b) Merging two sublists of length k takes $\Theta(k)$ time, and there are $\frac{n}{k}$ sublists, so the total time to merge all sublists in the same level is $\Theta(k * \frac{n}{k}) = \Theta(n)$ (total n-1 compares). Since there are $lg(\frac{n}{k})$ levels of merging, the total time to merge all sublists is $\Theta(n * lg(\frac{n}{k}))$.

(c) To have less or equivalent running time as standard merge sort $\Theta(nlgn)$, we need $\Theta(nk + n * lg(\frac{n}{k})) \leq \Theta(nlgn)$.

Solving for k, we get $k = lgn$, where we have $\Theta(nk + n * lg(\frac{n}{k}))$ = $\Theta(nlgn + n * lg(\frac{n}{lgn})) = \Theta(nlgn + n(lgn - lg(lgn)) = \Theta(nlgn)$ for $n \geq 4$.

Hence, for all k between $1 \leq k \leq \lfloor lgn \rfloor$, the inequality $\Theta(nk + n * lg(\frac{n}{k})) \leq \Theta(nlgn)$ holds and when $k = lgn$, the modified algorithm has the same running time as standard merge sort.

(d)

Pros:

- Less runtime memory consumption
  Since merge sort has to use an extra space whenever it merges the current level, we can save the runtime memory space with fewer merging times. Hence, even though the modified merge sort runs similarly fast as the original one, it is still worth using it when the input size is reasonable.
- less runtime
  As we prove in the last question, when $1 \leq k \leq \lfloor lgn \rfloor$, the modified merge sort can have a faster computation time than the original one (at least in the asymptotic sense).

Cons

- higher operation overhead
  If the input size is relatively slow, we should still use insertion sort instead of using the hybrid method since the modified merge sort still has the sins(merging overhead) from the merge sort.