Evan Edelstein
EN.605.621.84.FA25
HW 1
The work in this exercise is mine alone without un-cited help. No AI was used to answer these questions.

1) The best-case scenario of linear search on a sorted list is Ω (1) since the first element could be the value searched for. The worst-case runtime is O(n), when the element is the last item of the list or not in the list at all. Since the best- and worst-case runtimes are not equivalent a tight Θ() bound cannot be evaluated.

   (also do arrays typically start from index 1?)

2)
   a. The algorithm sorts the list by finding pairs of neighboring unsorted elements. The smaller second element is then swapped with previous items in the element until the list is sorted. In the best-case, the elements are already sorted, and the algorithm is Ω (n) as each element, except the last, in the list will still be iterated over. For example [1,2,3,4,5] would look like this

      Brackets denote comparison on line 4:
      0: {1,2} ,3,4,5
      1: 1,{2 ,3},4,5
      2: 1,2 ,{3, 4},5
      3: 1,2 ,3 {4,5}

   b. In the worst case each element is compared with each other element behind it. This would lead to a O(n^2) runtime. For example, the array [5,4,3,2,1] would show this:
      Brackets denote positions to swap on line 5:

      0: {5, 4}, 3, 2, 1
      1: 4, {5, 3 }, 2, 1
      0: {4, 3}, 5, 2, 1
      2: 3, 4, {5, 2}, 1
      1: 3, {4, 2}, 5, 1
      0: {3, 2}, 4, 5, 1
      3: 2, 3, 4, {5, 1}
      2: 2, 3, {4, 1}, 5
      1: 2, {3, 1}, 4, 5
      0: {2, 1 }, 3, 4, 5

   c. Since the Since the best- and worst-case runtimes are not equivalent a tight Θ() bound cannot be evaluated. The general runtime of this algorithm would be bound by O(n^2) and Ω (n)

3)

a. Since k is constant it contributes a Θ (1) factor to the runtime resulting in an Θ ((n) * Θ (1) which simplifies to Θ (n). A tight bound (Θ) can be used since the upper and lower bound are both linear.
b. One optimization that can be made is to move the statement on line 4, which relies only on Ai outside the loop since it does not rely on j. This would remove redundant work. Also, if the expected size of k and n are known we could make the outer loop reliant on the larger of the two values.

4) $T(n) = 3T\left(\frac{n}{3} + 1\right) + f(n)$

Simplify $\frac{n}{3} + 1 = \frac{n+3}{3}$
Let: $n = m - 3$
Substitute m: $T(m-3) = 3T\left(\frac{m-3+3}{3}\right) + f(m-3) \rightarrow 3T\left(\frac{m}{3}\right) + f(m-3)$
a=3, b=3
Question states: $f(n) = n$
By masters theorem: $f(n) = n^{log3(3)} = n^1 = n$
This is case 2 of the masters theorem and thus $T(n) = \theta\left(n^{logb(a)} * \lg(n)\right) = \theta(nlg(n))$

5) The problem states the fifth number in a Fibonacci sequencing beginning with f1 and f2 is 20. From the recurrence relation we know the fifth Fibonacci number can be expressed as $F_x = F_{x-1} + F_{x-2}$:

$$F_5 = F_4 + F_3$$
$$F_4 = F_3 + F_2$$
$$F_3 = F_2 + F_1$$

$$F_5 = (F_3 + F_{2)} + (F_2 + F_1) = ((F_2 + F_1) + F_2) + (f_2 + F_1) = 20$$
$$3F_2 + 2F_1 = 20$$
$$Let: F_2 = F_1$$
$$5F_1 = 20 \rightarrow F_1 = 4$$

Thus 4 is a valid value for F1 and F2. The next number in the series would be:

$$F_6 = F_5 + F_4 = 20 + (((F_2 + F_1) + F_2) = 20 + 4 + 4 + 4 = 20 + 12 = 32$$

6)

a. Insertion sort worst case runtime is $\theta(x^2)$ where x is the length of the input. In the merge-insertion sort we have (n/k) number of sublists to sort, each with k elements therefore the worst-case runtime would be the product $\theta(k^2) * \theta\left(\frac{n}{k}\right) = \theta(nk)$

b. Regular Mergesort has a runtime of $\theta(n \lg n)$ which means that performing merge sort on a list of size k would have a runtime of $\theta(k\ lg\ k)$. Since there are $\theta\left(\frac{n}{k}\right)$ subarrays the overall time spent sorting the subarrays with regular merge sort is $\theta(k\ lg\ k) * \theta\left(\frac{n}{k}\right) = \theta(n \lg k)$. By subtracting the length k subarray time from the total runtime we can compute the amount of time dedicated to merging the subarrays. $\theta(n\ lg\ n) - \theta(n\ lg\ k) \rightarrow$ $\theta(n\ (\lg n - \lg k)) \rightarrow \theta(n \lg \frac{n}{k})$

   (the visualization from the office hours really helped with this)

c. The goal is to find the largest k that still satisfies the equation Θ(nk + n lg (n/k)) = Θ(n lg (n)). Looking at the first part of the left hand side we see that O(nk) cannot be asymptoticly bigger than O(nlogn) canceling the leading n term shows that k must be less than O(lg n). Setting k to lg n: $\theta\left(nk + n \lg \frac{n}{k}\right) \rightarrow \theta(nk + n \lg n - n \lg k) \rightarrow \theta(n \lg n + n \lg n - n \lg \lg n) \rightarrow \theta(2nlg\ n) \rightarrow \theta(n \lg n)$

d. The main reason to use this method is to take advantage of insertions sort "local" improvement for small arrays where the runtime is O(n) as opposed to merge sort's O(nlgn). One downside of this approach is the necessity to choose a value for k that minimizes the probability of insertion sort reaching its O(n^2) runtime which would be worse than just using merge sort.