1.

Algorithm	Complexity (i.e., growth rate)			Justification
	Sorted arrays	**Reverse-order arrays**	**Arrays of identical items**	
Insertion sort	O(n)	O(n^2)	O(n)	Insertion sort traverses through each element in the list and places the element in the correct order as it goes. If the list is already in order or the items are identical, then it has nothing else to do, so it traverses through just once. In the worst case, it must traverse through the list and for each element, relocate it to the beginning and then traverse through again, which is quadratic behavior.
Selection sort	O(n^2)	O(n^2)	O(n^2)	Selection sort always has the same time complexity, as it traverses through the list once for each element in the list in order to correctly order the elements. Performing a O(n) operation n times is quadratic.
Mergesort	O(n log n)	O(n log n)	O(n log n)	Since mergesort works by repeatedly halving the array, we can picture it like a perfect binary tree, where each split is another layer in the tree. When we merge the layers together, it is like traversing through the binary tree where the height is log n. Since we have a linear operation at each level with merge, then that means we have n log n in all cases.
Quicksort	O(n^2)	O(n^2)	O(n^2)	Although quicksort has an expected time complexity of O(n log n), its worst case for any order is quadratic. This happens when the pivot happens to be the first or last

				element in the array. Here we are assuming that it is always the first element in the array. Because of this, we must actually traverse through the entire list each time instead of partitioning and traversing through n elements n times is quadratic.
Heapsort	O(n log n)	O(n log n)	O(n log n)	Just like with mergesort, heapsort is also n log n in all cases. It is even based off of a binary heap. Constructing the heap is a linear time operation and each operation on the heap is a logarithmic time operation by definition. Therefore it is n log n.

2. Sort the array [3, 1, 4, 1, 5, 9, 2, 6, 5, 3] by quicksort, use median of 3 pivot selection and the partitioning strategy covered in class, show step by step

1st level
[3, 1, 4, 1, 5, 9, 2, 6, 5, 3]
1st pivot: median(3, 5, 3)= 3
The pivot is already the last element, so we don't need to swap it
[3, 1, 4, 1, 5, 9, 2, 6, 5, 3]
 i j
i moves to the next element larger than (or equal to) the pivot and j moves to the next element smaller than the pivot
[3, 1, 4, 1, 5, 9, 2, 6, 5, 3]
 i j
i < j, so we swap the elements
[2, 1, 4, 1, 5, 9, 3, 6, 5, 3]
 i j
Now we move i and j again
[2, 1, 4, 1, 5, 9, 3, 6, 5, 3]
  i j
i < j, so we swap the elements
[2, 1, 3, 1, 5, 9, 4, 6, 5, 3]
  i j
move i and j again
[2, 1, 3, 1, 5, 9, 4, 6, 5, 3]
  i j
i < j, so we swap the elements

[2, 1, 1, 3, 5, 9, 4, 6, 5, 3]
i j
move i and j again
[2, 1, 1, 3, 5, 9, 4, 6, 5, 3]
j i
since i > j, we no longer swap the elements and instead, swap i with the pivot
[2, 1, 1, 3, 3, 9, 4, 6, 5, 5]
j i
2nd level
Now we are left with [2, 1, 1, 3] and [9, 4, 6, 5, 5]
2nd pivots: median(2, 1, 3)=2, median(9, 6, 5)=6
Left first: [2, 1, 1, 3]
Swap the pivot with the last element
[3, 1, 1, 2]
i j
i < j, so swap the elements
[1, 1, 3, 2]
i j
move i and j again
[1, 1, 3, 2]
j i
since i > j, we no longer swap the elements and instead, swap i with the pivot
[1, 1, 2, 3]
j i
right: [9, 4, 6, 5, 5]
swap the pivot with the last element
[9, 4, 5, 5, 6]
i j
i < j, so swap the elements
[5, 4, 5, 9, 6]
i j
move i and j again
[5, 4, 5, 9, 6]
j i
since i > j, we swap i with the pivot
[5, 4, 5, 6, 9]
j i
3rd level
Now we are left with [1,1] from the left and [5,4,5] from the right
Pivots: 1 and 5
[1, 1]
i,j
we are done since there is nothing to do (technically we swap the 1s when we are done as 1 is the pivot)

[5, 4, 5]
We do not need to swap, since the pivot is already the last element
[5, 4, 5]
 i j
i < j, so swap
[4, 5, 5]
 i j
move i and j again
[4, 5, 5]
 j i
we are done (like before, we swap the 5s)
2nd level merging:
[1, 1, 2, 3] and [4, 5, 5, 6, 9]
1st level merging:
[1, 1, 2, 3, 3, 4, 5, 5, 6, 9]
Sorted array: [1, 1, 2, 3, 3, 4, 5, 5, 6, 9]

3. Shellsort on the input [19, 18, 17, 16, 15, 14, 13, 12, 10] using the increment sequence {1, 3, 7}, show array after each execution of insertion sort, i.e. for each increment/gap

[19, 18, 17, 16, 15, 14, 13, 12, 10]

gap = 7, p = 7	**12**	18	17	16	15	14	13	**19**	10
p = 8	12	**10**	17	16	15	14	13	19	**18**
gap = 3, p = 3	**12**	10	17	**16**	15	14	13	19	18
p = 4	12	**10**	17	16	**15**	14	13	19	18
p = 5	12	10	**14**	16	15	**17**	13	19	18
p = 6	12	10	14	**13**	15	17	**16**	19	18
p = 7	12	10	14	13	**15**	17	16	**19**	18
p = 8	12	10	14	13	15	**17**	16	19	**18**
gap = 1, p = 1	**10**	**12**	14	13	15	17	16	19	18
p = 2	10	**12**	**14**	13	15	17	16	19	18
p = 3	10	12	**13**	**14**	15	17	16	19	18
p = 4	10	12	13	**14**	**15**	17	16	19	18
p = 5	10	12	13	14	**15**	**17**	16	19	18
p = 6	10	12	13	14	15	**16**	**17**	19	18
p = 7	10	12	13	14	15	16	**17**	**19**	18
p = 8	10	12	13	14	15	16	17	**18**	**19**

sorted array: [10, 12, 13, 14, 15, 16, 17, 18, 19]

4. If you are given an array whose first n-f(n) elements are sorted, but whose last f(n) are not, how would you sort the entire array in O(n) time if f(n) is as follows? Be specific
    a. F(n) = 10

In this case, since the last 10 elements are unsorted regardless of the size of the array, we can accomplish O(n) if we use insertion sort. This is because for a sorted array, insertion sort has linear time complexity. For an unsorted array, this changes to $O(n^2)$, but since 10 is a constant, that means that the time complexity will always be O(n + 100). We only care about the largest power when dealing with time complexity, so we ignore the 100, simplifying to just O(n).

b. F(n) = log n

Since the last log(n) elements are unsorted, we can use the 1-log(n)th element as the pivot and use quickselect to find each element in the unsorted area. Since quickselect is O(n), all we must do find the ith element in the unsorted section and then assign the element to the correct index by swapping, which can be done in constant time since we know the index. Since we have a linear time operation and a constant time operation, this leaves us with linear time.

c. F(n) = sqrt(n)

Just like before, we can use quickselect to find each element in the unsorted sqrt(n) section at the end of the array. Quickselect can be done in O(n) since we are partitioning logically and once again, the swaps can be done in constant time. Therefore, just like with log n, we have accomplished linear time.

Assignment 5

5. union(8,9)

9 ← 8

8	9
9	-2

union(6,11)

9 ← 8 11 ← 6

6	8	9	11
11	9	-2	-2

union(4,6)

9 ← 8 11 ← 6 ← 4

4	6	8	9	11
6	11	9	-2	-3

union(1,9)

1 → 9 ← 8 11 ← 6 ← 4

1	4	6	8	9	11
9	6	11	9	-2	-3

union(1,12)

12 ← 9 ← 8 11 ← 6 ← 4

1

1	4	6	8	9	11	12
9	6	11	9	12	-3	-3

union(1,2)

2 ← 12 ← 9 ← 8 11 ← 6 ← 4

1

1	2	4	6	8	9	11	12
9	-4	6	11	9	12	-3	2

union(11,0)

2 ← 12 ← 9 ← 8 0 ← 11 ← 6 ← 4

1

0	1	2	4	6	8	9	11	12
-4	9	-4	6	11	9	12	0	2

union(7,0)

2 ← 12 ← 9 ← 8 0 ← 11 ← 6 ← 4

1 7

0	1	2	4	6	7	8	9	11	12
-4	9	-4	6	11	0	9	12	0	2

union(8,12)

2 ← 12 ← 9 ← 8 0 ← 11 ← 6 ← 4

1 7

0	1	2	4	6	7	8	9	11	12
-4	9	-4	6	11	0	9	12	0	2

union(3,5)

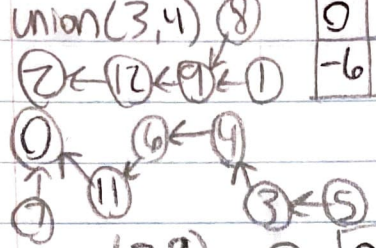
2 ← 12 ← 9 ← 8 3 ← 5 11 ← 6 0 ← 7

1 9 11

0	1	2	3	4	5	6	7	8	9	11	12
-4	9	-4	-2	6	3	11	0	9	12	0	2

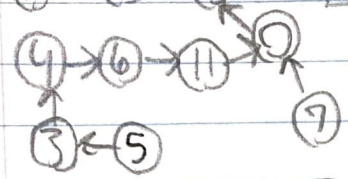
union(3,4)

0	1	2	3	4	5	6	7	8	9	11	12
-6	9	-4	4	6	3	11	0	9	12	0	2



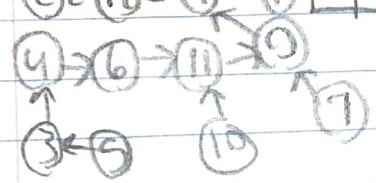
union(7,9)

0	1	2	3	4	5	6	7	8	9	11	12
9	9	-9	4	6	3	11	0	9	12	0	2



union(10,11)

0	1	2	3	4	5	6	7	8	9	10	11	12
9	9	-9	4	6	3	11	9	9	12	11	0	2



find(5) = 2

0	1	2	3	4	5	6	7	8	9	10	11	12
2	9	-3	2	2	2	2	0	9	2	11	2	2