Question 1:

To merge lists [1, 25, 31, 16] and [-3, 0, 16, 27] we will need to use mergesort.

I will do this by comparing the elements from both lists one by one and insert them into a new list in sorted order.

Here is the pseudocode for the prompt:

```
merge(A, B):
   Result = [] //Create an empty list "result"
   Let i = 0  // pointer for list A
   Let j = 0  // pointer for list B

   // compare elements from both lists and insert the smaller one
   while i < length(A) AND j < length(B):
     /if A[i] ≤ B[j]:
         result.append(A[i]) //add A[i] to result
         i = i + 1
      else:
         result.append(B[j]) //add B[j] to result
         j = j + 1

   //If there are remaining elements in A, add them
   while i < length(A):
     result.append(A[i]) //add remaining elements from A
     i = i + 1

   // If there are remaining elements in B, add them
   while j < length(B):
      result.append(B[j]) //add remaining elements from B
      j = j + 1

   Return result;
```

//Final Output:
[-3, 0, 1, 16, 16, 25, 27, 31]

Question 2:

I will show the step-by-step process of sorting a list using the insertion sort algorithm.
The unsorted list is: [-1, -5, 67, -10, 21, 8, 4, 1]

First, I will start with the second element, assuming the first element is already sorted.
For the first iteration, the key is -5, and the index is 1.
Compare -5 with -1, and we will find that -5 is smaller.
Because it is smaller, shift -1 to the right.
Insert -5 at position 0.
The list now looks like this: [-5, -1, 67, -10, 21, 8, 4, 1]

Second, we will insert 67 at index 2.
Compare 67 with -1, and we will find that 67 is larger.
Because it is larger no movement is needed.

Third, we will insert -10 at index 3.
Compare -10 with 67.
Because it is smaller, shift 67 to the right.
Compare -10 with -1.
Because it is smaller, shift -5 to the right.
Insert -10 at position 0.
The list now looks like this: [-10, -5, -1, 67, 21, 8, 4, 1]

Fourth, we will insert 21 at index 4.
Compare 21 with 67.
Because it is smaller, shift 67 to the right.
Compare 21 with -1.
Because it is larger, no movement is needed.
Insert 21 at position 3.
The list now looks like this: [-10, -5, -1, 21, 67, 8, 4, 1]

Fifth, we will insert 8 at index 5.
Compare 8 with 67.
Because it is smaller, shift 67 to the right.
Compare 8 with 21.
Because it is smaller, shift 21 to the right.
Compare 8 with -1.
Because it is larger, no movement is needed.
Insert 8 at position 3.
The list now looks like this: [-10, -5, -1, 8, 21, 67, 4, 1]

Sixth, we will insert 4 at index 6.
Compare 4 with 67.
Because it is smaller, shift 67 to the right.

Compare 4 with 21.
Because it is smaller, shift 21 to the right.
Compare 4 with 8.
Because it is smaller, shift 8 to the right.
Compare 4 with -1.
Because it is smaller, no movement is needed.
Insert 4 at position 3.
The list now looks like this: [-10, -5, -1, 4, 8, 21, 67, 1]

Seventh, we will insert 1 at index 7.
Compare 1 with 67.
Because it is smaller, shift 67 to the right.
Compare 1 with 21.
Because it is smaller, shift 21 to the right.
Compare 1 with 8.
Because it is smaller, shift 8 to the right.
Compare 1 with 4.
Because it is smaller, shift 4 to the right.
Compare 1 with -1.
Because it is larger, no movement is needed.
Insert 1 at position 3.
The list now looks like this: [-10, -5, -1, 1, 4, 8, 21, 67]

Since there are no more values to sort through, the list has fully been sorted with insertion sort.

Question 3:

I will show the step-by-step process of sorting a list using the quicksort algorithm.
The unsorted list is: [-5, 42, 6, 19, 11, 25, 26, -3]

First, we will choose a pivot. In this case, we will pick the last element, -3.
Partition the list so that elements smaller than -3 go to the left, and elements larger go to the right.
Compare -5 with -3. Since -5 is smaller, leave it.
Compare 42 with -3. Since 42 is larger, move it to the right.
Compare 6 with -3. Since 6 is larger, move it to the right.
Compare 19 with -3. Since 19 is larger, move it to the right.
Compare 11 with -3. Since 11 is larger, move it to the right.
Compare 25 with -3. Since 25 is larger, move it to the right.
Compare 26 with -3. Since 26 is larger, move it to the right.
Swap -3 with the first larger element (42).
The list now looks like this: [-5, -3, 6, 19, 11, 25, 26, 42]

Next, we recursively sort the left and right partitions.
The left side only contains [-5], so it is already sorted.
Now, we sort the right side: [6, 19, 11, 25, 26, 42]

We pick the last element, 42, as the pivot.
Compare 6 with 42. Since 6 is smaller, leave it.
Compare 19 with 42. Since 19 is smaller, leave it.
Compare 11 with 42. Since 11 is smaller, leave it.
Compare 25 with 42. Since 25 is smaller, leave it.
Compare 26 with 42. Since 26 is smaller, leave it.
Since all elements are smaller, the pivot remains in place.
The list remains: [-5, -3, 6, 19, 11, 25, 26, 42]

Next, we sort the sublist [6, 19, 11, 25, 26]
Pick the last element, 26, as the pivot.
Compare 6 with 26. Since 6 is smaller, leave it.
Compare 19 with 26. Since 19 is smaller, leave it.
Compare 11 with 26. Since 11 is smaller, leave it.
Compare 25 with 26. Since 25 is smaller, leave it.
Since all elements are smaller, the pivot remains in place.
The list remains: [-5, -3, 6, 19, 11, 25, 26, 42]

Now, we sort the sublist [6, 19, 11, 25]
Pick 25 as the pivot.
Compare 6 with 25. Since 6 is smaller, leave it.
Compare 19 with 25. Since 19 is smaller, leave it.
Compare 11 with 25. Since 11 is smaller, leave it.

Since all elements are smaller, the pivot remains in place.
The list remains: [-5, -3, 6, 19, 11, 25, 26, 42]

Now, we sort the sublist [6, 19, 11]
Pick 11 as the pivot.
Compare 6 with 11. Since 6 is smaller, leave it.
Compare 19 with 11. Since 19 is larger, move it to the right.
Swap 11 with the first larger element (19).
The list now looks like this: [-5, -3, 6, 11, 19, 25, 26, 42]

Now, we sort the sublist [6], which is already sorted.
Since there are no more values to sort, the final sorted list is:
[-5, -3, 6, 11, 19, 25, 26, 42]

Since there are no more values to process, the list has fully been sorted with quicksort.

Question 4:

I will show the step-by-step process of sorting a list using the shell sort algorithm.
The unsorted list is: [15, 14, -6, 10, 1, 15, -6, 0]

First, we choose a gap value. We typically start with half the list size.
The list has 8 elements, so we start with a gap of 4.

We will compare and sort elements that are 4 positions apart.
Compare 15 with 1. Since 15 is larger, swap them.
Compare 14 with 15. No swap is needed.
Compare -6 with -6. No swap is needed.
Compare 10 with 0. Since 10 is larger, swap them.
The list now looks like this: [1, 14, -6, 0, 15, 15, -6, 10]

Next, we reduce the gap to 2 and sort elements that are 2 positions apart.
Compare 1 with -6. Since 1 is larger, swap them.
Compare 14 with 0. Since 14 is larger, swap them.
Compare -6 with 15. No swap is needed.
Compare 0 with 10. No swap is needed.
The list now looks like this: [-6, 0, 1, 10, 14, 15, -6, 15]

Next, we reduce the gap to 1, which is just insertion sort.
Compare 0 with -6. No swap is needed.
Compare 1 with 0. No swap is needed.
Compare 10 with 1. No swap is needed.
Compare 14 with 10. No swap is needed.
Compare 15 with 14. No swap is needed.
Compare -6 with 15. Since -6 is smaller, shift 15 to the right.
Shift 14 to the right.
Shift 10 to the right.
Shift 1 to the right.
Shift 0 to the right.
Insert -6 at position 1.
The list now looks like this: [-6, -6, 0, 1, 10, 14, 15, 15]

Since there are no more values to process, the final sorted list is:
[-6, -6, 0, 1, 10, 14, 15, 15]

Since there are no more values to process, the list has fully been sorted with shell sort.

Question 5:

In this document, I will rank six sorting algorithms in order of their expected runtimes, from fastest to slowest, based on asymptotic analysis.

1. **Merge Sort (O(n log n))**
    - Merge Sort is an efficient algorithm with a guaranteed runtime of O(n log n).
    - It performs well on large datasets and is stable, but requires additional space for merging.
2. **Quick Sort (O(n log n) on average, O(n^2) worst case)**
    - Quick Sort is often faster in practice due to efficient partitioning.
    - However, its worst-case runtime of O(n^2) can occur if poor pivot choices are made.
3. **Heap Sort (O(n log n))**
    - Heap Sort has a guaranteed O(n log n) runtime.
    - It does not require extra space but is often slower in practice due to heap operations.
4. **Shell Sort (O(n log n) to O(n^2), depending on gap sequence)**
    - Shell Sort is better than Insertion Sort because it uses larger initial gaps, reducing swaps.
    - With an optimal gap sequence, it can approach O(n log n), but usually performs worse.
5. **Insertion Sort (O(n^2) worst case, O(n) best case for almost completely sorted data)**
    - Performs well on small or almost completely sorted lists but is inefficient for large, random lists.
6. **Bubble Sort (O(n^2))**
    - Bubble Sort is the slowest because it repeatedly swaps adjacent elements.
    - Even with slight improvements, its O(n^2) complexity makes it inefficient for large lists.

Ties:

- Merge Sort, Quick Sort (on average), and Heap Sort all achieve O(n log n) efficiency.
- Insertion Sort and Shell Sort can sometimes perform better than their worst cases, but in general, their performance is slower than O(n log n) algorithms.

In conclusion:

- **Fastest algorithms (O(n log n))**: Merge Sort, Quick Sort (avg.), Heap Sort.
- **Slower algorithms (O(n^2))**: Insertion Sort, Shell Sort, Bubble Sort.
- **Bubble Sort is the slowest overall.**

Question 9:

Question 10:

From my experiments, I saw that the faster sorting algorithms handled large lists much better than the slower ones. Here's what I noticed:

- Merge Sort, Quick Sort, and Heap Sort were the best for big lists. Quick Sort was usually the fastest, but it slowed down sometimes depending on the pivot choice.
- Shell Sort was faster than Insertion and Bubble Sort but not as fast as the top three.
- Insertion Sort worked okay for small lists but got much slower as the list got bigger.
- Bubble Sort was by far the slowest and struggled with large lists.

My results mostly matched my predictions, but not always.

- Quick Sort was often faster than Merge Sort because it used memory more efficiently.
- Heap Sort was slower than expected because it did a lot of extra swaps.
- Shell Sort's speed changed a lo**t** depending on how I picked the gaps between the numbers being compared.

When sorting really big lists, some algorithms, especially Bubble and Insertion Sort, took way too long. At a certain point, the computer just couldn't handle it, or the time measurements became unreliable. To fix this, I had to limit how big the lists were for slow algorithms.

In conclusion, the faster sorting methods worked as expected, but small details, like how they use memory, affected actual speed in ways that the theory didn't predict.

Question 12: