a) Discuss what your results mean regarding the theoretical run-time of the different algorithms.

Theoretically based on the COUNT of the first file Num8.txt best sorting algorithms, in order from best to worst

1) Heap Sort O(n log n)
2) Insertion Sort O(n^2)
3) Quick Sort O(n^2)
4) Merge Sort O(n log n)

Theoretically based on the COUNT of the first file Num16384.txt best sorting algorithms, in order from best to worst

1) Heap Sort O(n log n)
2) Merge Sort O(n log n)
3) Quick Sort O(n^2)
4) Insertion Sort O(n^2)

It's hard to definitively say that a specific algorithm is better than another based on looking at the count, because of where the count is placed. If we run Num8.txt but with the values 1,2,3,4…8 in sorted order we can see that the count for insertion sort will be `InsertionSortOutput8.txt count: 0,` the count is measuring the worst case, which for insertion sort will be inside the while loop, so in the instance when the values were in sorted order the body of the while loop is never executed, because the values were already sorted.

Similarly, for merge sort the count is placed inside the for loop

```
for(int k=startPoint; k<=endPoint; k++){
                    count++;
                    if(L[i] <= R[j]){
                           A[k] = L[i];
                           i++;
                    }
                    else{
                           A[k] = R[j];
                           j++;
                    }
             }
```

when the Merging part of the merge sort occurs so no matter what the count for the merge will always be counted. Unlike the heap sort where the count is placed inside the if statement

```
if(LargerValue != parent){
                    swap(A,parent, LargerValue);
                    count++;
                    Max_Heapify(A, LargerValue);
            }
```

which is technically where the swap part of the heap sort occurs, but unlike merge sort where the count part is always executed, the count for the heap is not always executed, this can occur when the heap condition for the specific parent and child we are checking already satisfy the heap condition, in these cases count is not incremented. So the theoretical runtimes for the four sorting algorithms are pretty accurate, n log n algorithms are similar to each other while the n^2 algorithm is the worst

b) Do the sorts really take O(n^2) and O(n log n) steps to run?

Notations like Big-Oh are used to give a worst case scenario of the runtime of an algorithm, however that does not mean that every time insertion sort runs it will be n^2, the worst case scenario for insertion sort occurs when the array is in reversely sorted order, then the insertion sort will take O(n^2), then there can also be a best case scenario, for insertion that scenario will be if the array is sorted already, in that case insertion sorts run time will be O(n). For quick sort picking the right pivot is very important to the outcome of the runtime if a good pivot is picked then the runtime can be O(n log n), however if a bad pivot is picked then the runtime can become O(n^2)

For the O(n log n) algorithms like merge sort and heap sort the worst case and the best case are both the same, because these algorithms do not depend on order of the array. So whether the array is not sorted or already sorted these algorithms take O(n log n) time. We can examine this by changing Num8.txt to a sorted value 1,2, 3,…8. And if we run the code for merge sort and heap sort we can see that the count for merge stays exactly the same, and heap is almost the same(the change is occurred because of where we placed the count, specifically the count is placed where there is a swap occurring) Both merge sort and heap sort use recursion to sort, but merge sort is a divide and conquer algorithm which means it divides the array until it reaches a base case then it can merge the arrays at each level,

heap sort follows the binary tree property so it has to satisfy conditions like not having more than two children etc.

c) Explain how you got your answer to this question.

By understanding how the algorithm works, the way that insertion sort works is that it has two loops one going forward which is the outer for loop and another loop that would move backwards, the outer for loop is always going to execute to be able to check every value in the array, the inner loop checks and makes sure that value to the left of index j is in sorted order. So that's why if an array is in reverse sorted order the runtime can be O(n^2). Similarly, if the array is in sorted order the inner loop is never executed so runtime can be O(n).

For merge sort the runtime whether its best case or worst case is always going to be O(n log n), this is because of the divide and conquer part of the algorithm, even if it is sorted the algorithm has to divide the array and only then can it start conquering. Similarly for heap sort even if an array is already sorted it needs to always build max heap and that will always take the same amount of time based on n. Quick Sort is where it is a little different, quick sort is very dependent on the pivot, picking a good pivot can mean the runtime can be O(n log n), but picking a bad pivot can mean that the runtime will be O(n^2). But however this can be avoided to ensure that O(n^2) can rarely occur.

d) Which of the sorts takes the most steps?

If the array is reverse sorted Insertion sort takes the most amount of time, but at the same time if the array is already sorted then insertion sort is the best for the job, because of its O(n) run time if array is sorted.

e) Which of the two O(n log n) sorts takes the most steps?

Between the two it seems like merge sort takes more steps because of the need to copy value to the helper arrays L and R.

f) Why?

Between the two it seems like merge sort takes more steps because of the two for loops which are used to copy its values to L and R

```
for(int i = 0; i<L_size;i
      L[i] = A[startPoint + i];
}
for(int j = 0; j<R_size;j++){
      R[j] = A[midPoint + j+1];
}
```

This part of the code copies the array into the individual left or right array. If we were to add a count inside these two for loops we can see the count double from what it originally was. So every time we want to split the left and right array we have to copy those values. Not only that it can have effect on the space complexity of the algorithm as the L and R are separate arrays that we need which can end up taking a decent amount of space.

g) Under what circumstances might you prefer to use one of the sorts versus others?

When there is a scenario like if the array is sorted already, the obvious choice would be insertion because the runtime would be O(n). But if the array was in reverse sorted it would be the worst choice as its runtime would be O(n^2). Then other algorithms like mergesort, heapsort, and quicksort might be preferred. QuickSort can be very fast, this algorithm depends heavily on the pivot, picking a good pivot can mean a good runtime and picking a bad pivot can mean a bad runtime.

h) In general, which sort seems preferable?

In general heapsort looks the most preferable.

i) Why?

Heapsort is better than insertion sort because it is more reliable, because heapsorts worst and best case are both O(n log n) compared to insertion sorts best case of O(n) and worst case of O(n^2), so in most scenario it would perform better. Heapsort is better than mergesort, although mergesort and heapsort both have a worst and best case of O(n log n), mergesort used more spaces(as far as the code that I implemented), because helper arrays are needed in order to keep divided parts before merging them, and also because time is taken to store each element into the helper arrays. So in that perspective heapsort performs much better. Heapsort is better than quicksort because it is more reliable(as far as the code that I implemented goes), quicksort is heavily dependent on the picking of a right pivot, so picking a good picot means a good runtime while picking a bad pivot would mean a bad runtime.