Pre-Lab Part 1

1) 11 rounds of swapping is required to sort the numbers using bubble sort.

2) Considering the complexity of the bubble sort, the number of comparisons that would represent the worst case scenario would be n^2.

Pre--Lab Part 2

1) The worst case time complexity of shell short would be n^2 because based on the pseudocode provided, the first for loop only generates the step size, so technically it is not a traditional for loop like in C. The next 2 for loops worst case would be n for both, so 2 for loops stacked is n^2.

2) After doing some research on the run time and complexity of shell sort and how it can be changed, I've learned that the runtime is heavily dependent on the gap sequence it uses, which is briefly explained in the assignment page. The best ratio to be used for gap sequences so far is 2.2, while some experiments observe that the average runtime can be improved when using gap sequences that are proportional to the logarithm of the array size.

Pre-Lab Part 3

1) From an explanation from stack overflow, the only way to obtain a complexity of N^2 is when the pivot for each partition is the greatest or least number in the list. This would result in a partition where one side is one value, and the other is N-1. If you keep choosing the greatest or least number in the list as the partition, you would partition the list again and again until you've gone through all the elements. From an article by geeksforgeeks, finding the median element can be done in linear time, so it would not affect the complexity and would avoid choosing the greatest or least number in the list.

Pre-Lab Part 4

1) Instead of going down the list of elements with insertion sort, using binary search would cut the number of elements to look through in half. Regular insertion sort's time complexity is N^2, however when implementing binary search I would guess the complexity would go down to N log N, since you are halving the search every time.
However, based on answers on Piazza and further research, I've learned that binary insertion sort's time complexity is still N^2, since you still need to swap the elements. The time taken accessing the memory is shortened, but the number of moves, or swaps, is relatively the same.

Pre-Lab Part 5

1) I plan on using extern variables to track the number of moves and comparisons across different files. Putting the extern variables in the header files of each sort, I can access it in the main function if I include the header file in the main .c file.

// Compares values next to each other and swaps if needed

Bubble Sort

```
int bubblesort (int array){
        int arrlength = length(arr);
        int temp;
        for (int i = 0; i <= length(arr) - 1; i++){
                int j = length(arr) - 1;
                while (j > i){
                        if (arr[j] <  arr[j-1]){
                                temp = arr[j-1];
                                arr[j-1] = arr[j];
                                arr[j] = temp;
                        }
                        j -= 1;
                }
        }
        return 0;
}
```

// Sorting is similar to bubble sort, but uses gaps to to sort further values from each other

Shell Sort

```
int shellsort (int arr[]){
        for (int gap = length(arr) * 5 / 11; gap > 1; gap * 5 / 11){
                if (gap <= 2){
                        gap = 1;
                }
                for (int i = gap; gap < length(arr); i++){
                        for (int j = i; i < gap - 1; j -= gap){
                                if (arr[j] < arr[j-gap]){
                                        temp = arr[j];
                                        arr[j] = arr[j-gap];
                                        arr[j-gap] = temp;
                                }
```

```
                        }
                }
        }
        return 0;
}

// Splits the array by placing values less than the pivot on the left, greater on the right
// This is done multiple times to the array, sorting smaller and smaller
```
Quicksort
```
int partition (int arr[], left, right){
        pivot = arr[left];
        lo = left + 1;
        hi = right;

        while (true){
                while (lo <= hi && arr[hi] >= pivot){
                        hi -= 1;
                }
                while (lo <= hi && arr[lo] <= pivot){
                        lo += 1;
                }
                if (lo <= hi){
                        temp = arr[lo];
                        arr[lo] = arr[hi];
                        arr[hi] = temp;
                }else{
                        break;
                }
        }
        temp = arr[left];
        arr[left] = arr[hi];
        arr[hi] = temp;

        return hi;
}

int quicksort (int arr[], left, right){
        if (left < right){
                index = partition(arr, left, right);
```

```
            quicksort(arr, left, index - 1);
            quicksort(arr, index + 1, right);
        }
        return 0;
}
```

// Similar to insertion sort, finds the right place to put the value, but uses binary search instead of
// searching through the whole array

Binary Insertion Sort

```
int binary_insert (int arr[]){
        for (int i = 1; i < length(arr); i++){
            value = arr[i];
            left = 0;
            right = i;
        }

        while (left < right){
            mid = left + ((right - left) / 2);

            if (value >= arr[mid]){
                left = mid + 1;
            }else{
                right = mid;
            }
        }

        for (int j = i; j > left; j -= 1){
            temp = arr[j];
            arr[j] = arr[j-1];
            arr[j-1] = temp;
        }

        return 0;
}
```