Pre-Lab Part 1

1. It would take 10 swaps to sort the numbers using Bubble Sort. The algorithm would need to iterate completely through the numbers five times.
2. N_2 comparisons can be expected in the worst-case scenario for Bubble Sort.

Pre-Lab Part 2

1. The worst-case time complexity for this shell sort is $N_{5/3}$. However, the worst case time complexity for shell sort overall depends on the gap sequence.
2. The runtime can be improved by finding a more efficient gap sequence, which would decrease the constant.

Pre-Lab Part 3

1. Quicksort isn't doomed by its worst-case time complexity because its average case time complexity is NlogN, which is the best complexity for sorting. Additionally, the worst case occurs when it tries to sort an already sorted list, and this case can be avoided if a pivot is picked from the middle rather than from either end.

Pre-Lab Part 4

1. While normal insertion sort has a time complexity of N_2, introducing binary search reduces this to NlogN because it can much more efficiently search for where to insert the element in question so it is in order.

Pre-Lab Part 5

1. I plan on using extern variables to keep track of the moves and compares so that the values can be accessed across files.

Assignment Pseudocode:

bubble.c:

```
extern moves = 0
extern compares = 0

// Derived from DDEL pseudocode
// Input – array: array to sort
// No output
def bubble_sort(array):
        swapped = false
        for (i = 0; i < length(array) - 1; i++):
                j = length(array) – 1
                while (j > i):
                        compares++
                        if (array[j] < array[j-1]):
                                swap array[j], array[j-1]
                                moves += 3
                                swapped = true
                        j--
                // Check if any swaps occurred – no swaps means it’s already sorted
                if (!swapped):
                        return
        return
```

shell.c:

```
extern moves = 0
extern compares = 0

gaps[]
// Derived from DDEL pseudocode
// Input – length: length of array to make gaps for
```

```
// No output
def create_gap(length):
        i = 0
        while (length > 1):
                if length <= 2:
                        gaps[i] = 1
                else:
                        gaps[i] = 5 * (n / 11)
                i++
        return

// Derived from DDEL pseudocode
// Input – array: array to sort
// No output
def shell_sort(array):
        for (step = 0; step < length(gaps); step++):
                for (i = step; i < length(array); i++):
                        for (j = i; j >= step; j -= step):
                                compares++
                                if (array[j] < array[j – step]):
                                        swap array[j], array[j – step]
                                        moves += 3
        return
```

quick.c:

```
extern moves = 0
extern compares = 0

// Derived from DDEL pseudocode
// Input – array: array to sort, left: index of leftmost element to sort, right: index of rightmost
element to sort
```

```
// Output: hi – index of middle of sort
def partition(array, left, right):
        pivot = array[left]
        lo = left + 1
        hi = right

        while (true):
                compares++
                while ((lo <= hi) and (arr[hi] >= pivot)):
                        hi--
                compares++
                while ((lo <= hi) and (arr[lo] <= pivot)):
                        lo++
                if (lo <= hi):
                        swap array[lo], array[hi]
                        moves += 3
                else:
                        break
        swap array[left], array[hi]
        return hi

// Derived from DDEL pseudocode
// Input – array: array to sort, left: index of leftmost element to sort, right: index of rightmost
element to sort
// No output
def quick_sort(array, left, right):
        if (left < right):
                index = partition(array, left, right)
                quick_sort(array, left, index – 1)
                quick_sort(array, index + 1, right)
        return
```

binary.c:

```
extern moves = 0
extern compares = 0

// Derived from DDEL pseudocode
// Input – array: array to sort
// No output
def binary_insertion_sort(array):
        for (i = 1; i < length(array); i++):
                value = array[i]
                left = 0
                right = i

                while (left < right):
                        mid = left + ((right - left) / 2)
                        compares++
                        if (value >= array[mid]):
                                left = mid + 1
                        else:
                                right = mid

                for (j = i; j > left; j--):
                        swap array[j – 1], array[j]
                        moves += 3
        return
```

sorting.c:

```
extern moves
extern compares
```

```
// Function to print array
// Input – array: array to print, number: number of elements to print
// No output
print_array(array, number):
        for (i = 0; i < number; i++):
                print array[i]
        return

// Function to print the results
// Input – sort_name: string with name of sort, array: sorted array, number: number of elements
to print
// No output
def print_results(sort_name, array, number):
        print sort_name
        print moves
        print compares
        print array(array, number)
        return

int main(argc, **argv):
        // Order: bubble, shell, quick, binary insertion
        sorts = [false, false, false, false]
        number_printed = 100
        seed = 8222022
        array_size = 100

        command = command line arguments
        while (not at end of arguments):
                if (command == 'A'):
                        sorts[0] = true
                        sorts[1] = true
```

```
                sorts[2] = true
                sorts[3] = true
        if (command == ‘b’):
                sorts[0] = true
        if (command = ‘s’):
                sorts[1] = true
        if (command == ‘q’):
                sorts[2] = true
        if (command == ‘i’):
                sorts[3] = true
        if (command == ‘p’):
                number_printed = optarg
        if (command == ‘r’):
                seed = optarg
        if (command == ‘n’)
                array_size = optarg
        if (number_printed > array_size):
                print error
                exit

srand(seed)
array[array_size] = malloc(int * array_size)
for (i = 0; i < array_size; i++):
        element = rand()
        bit_mask = 0x3FFFFFFF
        // Makes number a 30 bit number by erasing 2 bits
        element AND bit_mask
        array[i] = element

if (sorts[0]):
        bubble_array = copy of array
```

```
        bubble_sort(bubble_array)
        print_results(“Bubble sort”, bubble_array, number_printed)
if (sorts[1]):
        shell_array = copy of array
        shell_sort(shell_array)
        print_results(“Shell sort”, shell_array, number_printed)
if (sorts[2]):
        quick_array = copy of array
        quick_sort(quick_array, 0, array_size)
        print_results(“Quick sort”, quick_array, number_printed)
if (sorts[3]):
        binary_insertion_array = copy of array
        binary_insertion_sort(binary_insertion_array)
        print_results(“Binary insertion sort”, binary_insertion_array, number_printed)

return 0
```