Assignment 5
Sorting

Per-Lab Part 1

1. How many rounds of swapping do you think you will need to sort the numbers 8, 22, 7, 9, 31, 5, 13 in ascending order using Bubble Sort?

   In the first round when 31 is pushed to the end, 4 swaps take place. In the second round when 22 is pushed to second last, 3 swaps take place. Then 1 swap then in two more rounds one swap takes place in each, making it a total of 10 swaps.

2. How many comparisons can we expect to see in the worst-case scenario for Bubble Sort? Hint: make a list of numbers and attempt to sort them using Bubble Sort.

   The time complexity for the worst-case scenario in bubble sort is O(N^2). Let's say you to sort numbers 31,22,13,9,8,7,5 in ascending order. Starting in the loop, the number 31 is compared with every other number in the list, so on the first iteration, 6 comparisons are made. Then on the second iteration, 22 is compared with every number, so there are 5 comparisons. Then on every iteration, the number of comparisons decreases by 1. Because of this pattern, the number of comparisons becomes $n(n-1)/2$, for n being the number of elements in the list, as the comparisons decrease like n-1, n-2, n-3,.... For every iteration.

Pre-Lab Part 2

1. The worst time complexity for Shell sort depends on the size of the gap. Investigate why this is the case. How can you improve the time complexity of this sort by changing the gap size? Cite any sources you used.

   Because the larger the size of gap sequence is, the more you have to loop which causes it to take more time.
2. How would you improve the runtime of this sort without changing the gapp size?

   Refill the gap sequence everytime you loop through the data set to swap.

Pre-Lab Part 3

1. Quicksort, with a worse case time complexity of O(n^2), doesn't seem to live up to its name. Investigate and explain why Quicksort isn't doomed by its worst case scenario. Make sure to cite any sources you use

   The worst case scenario in Quick sort occurs when the smallest or biggest value is chosen as the pivot. Depending on the order we want to sort the numbers, having the smallest or biggest value as pivot doesn't partition the array well in the beginning which

then makes it longer to sort. Although, this worst case time complexity is bad, Quicksort isn't doomed because in most cases, the pivot will not be the smallest or biggest value in the array. The worst case scenario can always be avoided depending on the choice of pivot.

Pre-Lab Part 4

1. Can you figure out what effect the binary search algorithm has on the complexity when it is combined with the insertion sort algorithm?

   Binary search in this sorting algorithm is used to reduce the number of comparisons compared to the comparisons used in insertion sort. Instead of looping backwards again to find where an element should be inserted, binary search is more efficient in picking the location where an element should be inserted. So this brings down the average time complexity as there are less comparisons.

Pre-Lab Part 5

1. Explain how you plan on keeping track of the number of moves and comparisons since each sort will reside within its own file.

   Create a struct for each sort. Every struct for each sort will have variables for moves, comparisons, and any other variables that may be of some help. Pointers will used when creating a variable of these struct so that all the functions makes changes to the space allocated for the sort struct and makes changes to the "static" moves and comparison variables.

Program Pseudocode

Main.c

main();
    Elements_size = 1000, print_count=10, seed = 8222022, #set the variables to default values specified on pdf
    Number = [], messy_num=[] #allocate space in memory for numbers. Numbers will be the ones sorted, but after they are sorted, they returned to their initial messy state which is also stored in messy_num
    Bub,shell, binary, quick = false #booleans to see which sort is called
    while(getopt){
        which ever sort is called, change its bool to true
        'p':
            print_count = given from s argument
        'R':
            Seed = give from r argument
        'N':
            Size = give from n argument

```
        Mask = 1073741823 #This is 0b00111111111111. Used to limit bits down to 30 bits
        srand(s)
        for i in range(size):
                #initializing the numbers
                Number[i] = rand() & mask
                Messy_num[i] = number[i]

If binary called:
        Make binary element and perform binary sort
        print

If quick called:
        Make quicksort element and perform Quicksort
        print
If shell called:
        Make shell element and perform shell sort
        Print
If bubble called:
        Make a bubble sort element and perform bubble sort
        Print

print(number, size, print, messy_num):
        For i till print:
                print number[i]
        For x till size:
                Number[x] = messy_num[x] // bringing number back to messy initial state
```

Bubble.c

```
bubble_create(int size):
        allocate space memory for a pointer variable that can store data type Bubblesort

bubble_sort(numbers, *bs):
        Perform the sorting
```

Shell.c

```
shell_create(size):
        allocate space in memory to store data type of Shellsort

gap_generate(Shellsort *s):
        Allocate space for sequence and follow pseudocode given on pdf
```

shell_sort(Shellsort sh):
    sort and increment the sh.moves and sh.comparisons within the loops where needed

Binary.c

binary_create(size):
    Allocate space in memory to store data type of binarysort
binary_sort(BinaryInsertion bI):
    Perform actual sorting

Quick.c:
qs_create(size):
    Allocate space in memory to store data type of quicksort

Program Details

Main.c: Deals with seeing which sort has been called in the command arguments and calls that sort and prints it. It allocates two sets of space in memory, one in numbers and other in messy_num. Data in both are randomly added and sorted. Once the data in numbers get sorted, it uses messy_num to get back its messy state so it can be sorted again from the next sort.

Note: All the sorting functions have been inspired by the pseudocode given on the assignment pdf.

Bubble.c: Creates a structure called BubbleSort so that moves and comparisons can be accessed from anywhere using pointers. Also has the bubble sort function. Bubble sort works by looping through data and comparing the adjacent values and then swapping them.

Shell.c: Creates a structure called Shellsort so that moves and comparisons and the gap sequence can be accessed from anywhere using pointers. Has the shell sorting function. Shell sort works by comparing and swapping values given at the indexes from the gap sequence.

Binary.c: Creates a structure called Binary_Insertion so that moves and comparisons can be accessed from anywhere using pointers. Also has the binary_insertion sort function. Binary insertion works the exact same way as insertion sort except, it uses binary search to find where an element needs to be inserted.

quick.c : Creates a structure called QuickSort so that moves and comparisons can be accessed from anywhere using pointers. Also has the Quick_sort sort function.