Assignment 5- Sorting: Putting Your Affairs in Order

Pre-Lab:
Part 1:

1. In order to sort the numbers 8, 22, 7, 9, 31, 5, 13 in ascending order using Bubble Sort, 10 rounds of swapping would need to occur.
   Original: 8 22 7 9 31 5 13
   Swap 1: 8 7 22 9 31 5 13
   Swap 2: 8 7 9 22 31 5 13
   Swap 3: 8 7 9 22 5 31 13
   Swap 4: 8 7 9 22 5 13 31
   Swap 5: 7 8 9 22 5 13 31
   Swap 6: 7 8 9 5 22 13 31
   Swap 7: 7 8 9 5 13 22 31
   Swap 8: 7 8 5 9 13 22 31
   Swap 9: 7 5 8 9 13 22 31
   Swap 10: 5 7 8 9 13 22 31 (sorted in ascending order)

2. In the worst case scenario for Bubble Sort, we can expect to see the same number of comparisons happening as the best case scenario in Bubble Sort, which would be: (n-1) + (n-2) + (n-3) +... + 1, where n is the number of elements in a given array. The only difference between the worst case and best case is that the number of swaps being made in the worst case is higher than that in best case— however, the number of comparisons remains the same for arrays of the same size. The number of comparisons made in Bubble Sort is only affected by the number of elements in an array.

3. In order for the smallest element to float to the top instead, I would revise the algorithm so that when comparing each pair of elements, instead of the smaller element of the two being swapped to the bottom, the larger number is swapped to the bottom. In this way, when done for each pair of elements, the larger numbers will congregate to the bottom of the array and the smaller numbers will float to the top.

Part 2:

1. The worst case time complexity for Shell Sort, depends on the sequence of gaps because if the gap size is small in relation to the size of the array, then it takes each element a longer amount of time— somewhat comparable to the amount of time it takes in a Bubble Sort— to get to their final destination in the sorted array. The time complexity of this sort can be improved by changing the gap size so that it is appropriately large relative to the size of the array. This would allow the elements, even if they are far from where they

need to end up, to get to the approximate region of the final position a lot quicker than in a less efficient, though stable, sort like Bubble Sort where each element is only compared to the adjacent element, which means that only small changes in position can be made for every pass.

Site used:
https://www.codingeek.com/algorithms/shell-sort-algorithm-explanation-implementation-and-complexity/#:~:text=Time%20complexity%20of%20Shell%20Sort,is%20still%20an%20open%20problem

Part 3:

Quicksort, even with a worst time complexity of $O(n^2)$, is not necessarily doomed because of the fact that the worst case almost always is avoided when Quick Sort is randomized, which essentially allows random pivot elements to be selected— which in turn ensures, to a certain extent, that the input array is partitioned relatively evenly.

Site(s) used:
https://www.youtube.com/watch?v=COk73cpQbFQ
https://www.baeldung.com/cs/quicksort-time-complexity-worst-case

Part 4:

I plan on keeping track of the number of swaps and comparisons made for each sort algorithm within another separate file(with its own header file), which increments the number of comparisons made by 1 for every pass through the array. Similarly, the number of swaps will be accounted for by incrementing the swap counter by 3— because every swap involves three moves— every time a swap is made.

Assignment 4 Description: When this program runs, it will implement one/many sorting algorithms to sort a random array of user-specified length and will output a sorted array(if the user chooses) along with the statistics of how many elements were sorted and the number of moves and comparisons each sorting algorithm made in the process of sorting the array.

Pseudocode:

**For an array: [39, 24, 37, 51, 47, 24, 31]

//followed provided Python pseudocode

1) bubble.c: parameters(int *arr, int arraySize)

compare pairs of adjacent elements while iterating through entire array

```
for (i = 0, i< n-1, i++)
    for (j = i+1, j < n, j++)
        call EVAL from counter.c to compare adjacent elements
```

```
                if EVAL == true
                        call SWAP

2) shell.c(int *arr, arraySize)
        for(i = 0, i < length of gaps, i++)
                while gaps[i] < n & > 0
                        int j = i
                        temp = arr[i]
                        while j >= gaps[i] & EVAL(j, j-gaps[i])
                                SWAP
                                j = j-gaps[i]
                                arr[j] = temp

3) heap.c(int *arr, int arraySize)
        int max_child
                gets the element with the least val
        make heap
                If current parent node < child node
                        SWAPS (to build max heap)
        build heap
                for a parent node within "heap" of array
                sends parent to make heap to compare nodes
        sort heap
                for created max heap
                        COMPARES parent and child nodes
                        SWAPS if child < parent

4) quick.c(int *arr, int arraySize)
        partition
                partitions given array into relatively equal halves
                For an element in array
                        If greater than partition/pivot
                                Places to right of pivot
                        Else
                                Places to left of pivot
        sort quick
                creates stack
                pushes left and right elements of pivot to stack
                placeholder to pop later for use in partition
```

```
5) set.c (for use with getopt options) //referred to Sahiti's code
        Set set_empty(void)
                empty set —> so return 0
        bool set_member(Set s, uint8_t x) [same as get_bit]
                mask = 1 << x%32
                return (s & mask) >> (x%32)

        bool set_insert(Set s, uint8_t x) [same as set_bit]
                mask = 1 << x%32
                return (s|mask)
        Set set_remove [same as clr_bit]
                mask = ~(1 << x%32)
                return s & mask
        Set set_intersect
                return s & t
        Set set_union
                return s|t
        Set set_complement
                return ~s
        Set set_difference
                return s & ~t
6) stack.c //referred to Sahiti's code
        Struct Stack stack
                u32 top, capacity
                int 64 *items

        stack_create
                calloc memory to stack(*s) first—> (*Stack)calloc(1, stack)
                top = 0
                capacity = min capacity(macro)
                *items mem allocation(calloc to rows only(not 2-D)) —> int64
                return s

        bool stack_empty
                return top == 0

        bool stack_push(*s, int64 val_to_push)
                if top == capacity
                        capacity = 2*capacity
                        items = mem reallocation—> int64(items)(capacity*int64)
```

```
                if items == NULL
                        false
        items[top] = val_to_push
        increment top
        return true

    bool stack_pop(*s, int64 *val_to_pop)
        if top == 0
                false
        else
                top = top -1
                *val_to_pop = items[top]
                true
```

7) counter.c (counts swaps and comparisons)

```
    bool EVAL(int *i, int *j, int *count)
        (...compare array[i] to array[j]...)
        evalCount++
        *count = evalCount

    int SWAP(int *i, int *j)
        (...swaps array elements at i and j using temp int var...)
        swapCount += 3
        *count = swapCount
```

8) sorting.c (test harness) //referred to Eugene and Sahiti's code

```
    getopt (options)-- same as asgn3 & 4
        -a: all sorting algorithms specified
        -b: enables Bubble Sort
        -s: enables Shell Sort
        -q: enables Quicksort
        -h: enables Heapsort
        -r(seed): set the random seed to seed— the default seed should be 7092016
        -n(size): set the array size to size— the default size should be 100
        -p(elements):  print out number of elements from the array.
    implements sets for whichever case/combination of cases are specified
        using enum
                bubble = 0
                shell = 1 ….
        for ex) if bubble sort corresponds to bit 1
                        set bit 1-> 0000 0001
```

```
iterate through valid positions for bit setting (in this case 0-3)
    if specific member is set
        call respective sorting alg
```