Pre-Lab Part 1

1. It will take 10 rounds of swapping to sort the number using bubble sort
2. We can expect to see n^2 comparisons for bubble sort as the worst case.
3. I would change the algorithm so that the elements would swap if the element before the element being compared was smaller than it.
   For example they would swap if arr[i] > arr[i - 1]

Pre-Lab Part 2

1. The time complexity depends on the sequence of gaps because the gaps allow elements to move farther in the array towards where they are supposed to be. A slower time complexity would be if there was no gap, then it would essentially be a bubble sort. Increasing the gap size would improve time complexity
2. Reduce redundant comparisons within the array.

Pre-Lab Part 3

1. For quicksort to have the worst case complexity of O(n^2) means that the pivot position is the smallest or largest element. This is because all of the elements would still be at the left or right of the element, nothing would move around. Quicksort is not doomed by this scenario because you do not have to pick the pivot position randomly or pick the middle position. One could pick the median element of the array to be the pivot position which would prevent the worst case time complexity.
   https://www.geeksforgeeks.org/can-quicksort-implemented-onlogn-worst-case-time-complexity/#:~:text=The%20worst%20case%20time%20complexity,element%20is%20picked%20as%20pivot.

Pre-Lab Part 4

1. I plan on keeping track of the number of moves and comparisons by creating another file with global variables. I could also use global variables to keep track and reset them to 0 after I print the output.

STACK

Stack *stack_create(void)

allocate memory for the stack using calloc()
Allocate memory for items using calloc()
If either fail then return NULL
Top of stack is initialized at 0
Minimum capacity is 16

void stack_delete(Stack **s)

Frees items
Frees the stack
Sets stack pointer to NULL

bool stack_empty(Stack *s)

Returns a boolean if top is equal to 0 to see if empty

bool stack_push(Stack *s, int64_t x)

If the top == capacity
Then reallocate memory * 2
Put x in the top of the items array
Increment top
Return true

bool stack_pop(Stack *s, int64_t *x)

Check if stack is empty then return false
Decrement the top of the stack
*x = items[top]
Return true

void stack_print(Stack *s)

Loop through stack until top and print each element

SORTING

I am using the pseudocode in python and converting it to C from lab document
For bubble sort, shell sort, quicksort, and heapsort.

TEST HARNESS

Have an array of sorting functions
Loop through the command line arguments
Allocate an array of random elements
Loop through each element and bit-mask it to fit 30 bits
Perform sort if command from getopt()
Print the number of elements, moves, and compares
Reset compare and moves to 0 after function
Free array

SETS

Set set_empty(void)
Returns an empty set (0)

bool set_member(Set s, uint8_t x)
1 is shifted left by x
Return Bitwise AND with set

Set set_insert(Set s, uint8_t x)
1 is shifted left by x
Return Bitwise OR with set

Set set_remove(Set s, uint8_t x)
1 is shifted left by x
Invert the shifter set
Return Bitwise AND with set

Set set_intersect(Set s, Set t)
Returns the set s bitwise AND with set t

Set set_union(Set s, Set t)
Returns the set s bitwise OR with set t

Set set_complement(Set s)
Return the negation of s
~ is bitwise NOT in C

Set set_difference(Set s, Set t)
Returns the set s AND negation of set t

WRITEUP

Get the time complexity for each sorting algorithm
Create graphs based on speed performance

Run different sorting algorithms with reversed ordered arrays and with very small/large arrays and see how they perform

Explain what I learned from the different sorting algorithms