Devan O'Boyle
CSE 13S
Assignment 3

Sorting: Putting Your Affairs in Order

Purpose: In this assignment, a variety of different sorting algorithms will be implemented to order a series of integers from least to greatest. Then each algorithm will be evaluated based on how efficient they were as well as the other strengths and weaknesses that each one brings to the table.

In this assignment, bubble sort, shell sort, and quick sort using a stack and a queue will be implemented.

Values to keep track of throughout the program: moves, compares, size of the random array, max size of the queue, and the max size of the stack

Bubble sort will be done by iterating through pointers to an array using a for loop and comparing two adjacent pointers to one another to see which one is greater. If the pointer to the left is greater than the pointer to the right, then they swap places. This iteration continues to until the program iterates through the entire array and doesn't swap any of the values. This will be done using a while loop which will only break once a boolean variable that keeps track of whether or not a swap has occurred is false. In which case the array has been sorted and the pointer will be returned.

Pseudocode:

- boolean swap - to keep track if the sort needs to keep swapping elements or not
- while swap is true
- set swap to false
- for loop iterating through i for while i is less than or equal to the size of the array
- check if the current element of the array is less than the previous element
- increment the comparisons
- if so swap the current element with the previous element
- this is done by setting a temp variable to the current element then setting the current element to the previous element, after that set the value of the previous element to the value of temp (all of this takes 3 moves, and therefore moves should be incremented 3 times)
- set swap to true to make it so that the array is checked again to see if there are other elements that need to be swapped

Shell sort is done by having a series of gaps, which are essentially numbers that indicate how far apart the numbers are that will swap places with one another if the one to the left is bigger than the one to the right. The different gaps values are given to us, so all we have to worry about is how to iterate through it. First, a for loop will be used to iterate through all of the different gap values. Then a nested for loop will be used to iterate through the pointer of the

array. There should be a check for whether or not the gap value fits into the subset that we are trying to sort. If it is greater than the subset between the iterated value, i, and the length of the array, then that gap value should be skipped. Otherwise, the subset should swap values as long as the iterated value, i, plus the gap value are within the subset.

Pseudocode:

- iterate through all the gaps array with index gap
- iterate through the random array from the current gap element in the gaps array with index i
- set an integer j equal to i
- set a temp integer variable equal to the element of the random array at i
- increment the number of moves by 1 since storing the value into a temp variable counts as one
- while j is greater than or equal to gap element in gaps and temp is less than or equal to j - gap in gaps
- set the random array element j equal to the random array element j - gap element in gaps
- increment the number of moves by 1
- decrement j by gap element in gaps'
- after the while loop, add 1 to comparisons
- set the random array element j equal to temp

Quick sort will be done by using two different methods: one with a stack, and one with a queue.

For the stack, it follows a last in, first out policy (LIFO). First, it starts out with a low index being set to the index of the first element in the array (0), and a high index of the index of the last element in the array. The low index and then the high index will be pushed onto the stack. Then a while loop is used to keep the sort going until the stack is empty meaning that the array has been sorted. Inside the while loop, the high and low indexes will be popped off the top of the stack and set to their own variables. Then the partition function will be called using the high and low indexes. Then a comparison will be made to see if the low index is less than the pivot point (returned by the partition function), if so the low index will be pushed onto the stack followed by the pivot point. Another comparison will be made to check if the high index is greater than the index after the pivot point. If so, then the index after the pivot point as well as the high index are pushed onto the stack. Then in the next iteration of the loop the index after the pivot point will become the new low index and the high point will remain the high index. This will repeat until each index has been sorted.

Pseudocode:

- create integer lo and set it equal to 0 to get the low index
- create integer hi and set it equal to size of the array minus 1 to get the high index
- create the stack
- push lo and hi to the stack

- set the size of the stack to a variable we'll call maxstack
- while the size of the stack isn't zero
- pop hi and lo from the stack
- call the partition with parameters of the random array, lo, and hi to put the elements into groups and set the return value to a variable called pivot
- if lo is less than the pivot, then push the lo and pivot onto the stack
- if hi is greater than the pivot, then add one to the pivot and push it along with hi
- if the size of the stack is greater than the current maxsize, then set maxsize to the current size as the new maximum
- delete the stack to avoid memory leaks

For the queue, it follows a first in first out policy (FIFO). A lot of the logic in this quick sort method is similar to when using the stack, however there are some differences when using the queue. First, the low and high indexes are set to the first and last indexes of the array just as before and are added to the queue. Then the while loop is used to keep the sort going until the queue is empty just like with the stack. However, rather than popping the high index first like we did before with the stack, since we are following first in, first out policy, the low index is popped first and then the high index. Other than that, the rest is the same. The partition will be used to get the pivot point, and then the two comparisons will be made to check if the low index is less than the pivot point and if the high index is greater than the index after the pivot point. If the low index is less than the pivot point, then the low index is added to the queue as is the pivot point. If the high index is greater than the index after the pivot point, then the index after the pivot point is added to the queue as well as the high index. This continues until the queue is empty, in which case the array has been sorted.

Pseudocode:

- create integer lo and set it equal to 0 to get the low index
- create integer hi and set it equal to size of the array minus 1 to get the high index
- create the queue
- enqueue lo and hi to the queue
- set the size of the queue to a variable we'll call maxqueue
- while the size of the queue isn't zero
- dequeue lo and then hi from the queue
- call the partition with parameters of the random array, lo, and hi, to put the elements into groups and set the return value to a variable called pivot
- if lo is less than the pivot, then enqueue the lo and pivot onto the queue
- if hi is greater than the pivot, then add one to the pivot and enqueue it along with hi
- if the size of the queue is greater than the current maxsize, then set maxsize to the current size as the new maximum
- delete the queue to avoid memory leaks

As for the partition, it takes in the array, low, and high indexes as parameters. It sets the pivot point to value at the midpoint between the low and high indexes. Then it creates two variables, one to store the index before the low index, so low index - 1, and the other stores the index after

the high index, so high index + 1. Then a while loop is used iterating through the indexs of the low index - 1, until it is no longer less than the high index + 1. Each iteration increments the low index - 1 by 1. A while loop is then used to continue incrementing the low index so long as the value at that index is less than the value at the pivot point. Then the high index is decremented by 1, and another while loop is used to continue decrementing the high index so long as the value at that index is greater than the value of the pivot point. Then if the low index is less than the high index, then the values at the low and high indexes are swapped. Once overarching while loop breaks, then the pivot point is returned.

Pseudocode:

- create a value of pivot and set it equal to the midpoint between hi and lo, which is (hi + lo)/2 + lo
- then create an integer i and set it equal to lo minus 1
- create an integer j and set it equal to hi plus 1
- while i is less than j
- increment i by 1
- while the element i in the array is less than pivot, increment i by one, and also increment the comparisons by 1
- after that while loop, increment j by one
- while the element j in the array is greater than pivot, decrement j by one, and also increment the comparisons by 1
- after that while loop, check if i is less than j
- if so swap the element i in the array with the element j in the array by setting the element i to a temp variable, then set the element i equal to the element j and set the element j equal to temp (this counts as 3 moves so the moves variable should be incremented by 3)
- after the if statement and the overarching while loop, return j a.k.a. the value of the pivot point

Each sort will be called on using cases in the command line. Boolean variables will be used for each sort to tell which case is being called on. Each case will set the corresponding boolean to true. The all case sets all of the sorts to true. There are also three other options that allow the user to modify the size of the array, the random seed, and the number of elements printed.

If a sort's corresponding boolean is true, it's function is called. Then the sorted array will be printed in columns of 5 along with the corresponding statistics on how many moves and comparisons the sort made along with the max size of the stack or queue if quick sort is called.

Pre-lab Questions:
Part 1:

1. I'm going to assume that the rounds of swapping is the amount of times that the array pointer is iterated through until it is fully sorted. In which case, I went by hand and drew out each iteration of the loop:

8 22 7 9 31 5 13
8 7 9 22 5 13 31
8 7 9 5 13 22 31
8 7 5 9 13 22 31
8 5 7 9 13 22 31
5 7 8 9 13 22 31

   As we can see, it goes through six rounds of swapping before we reach the sorted array.

2. n^2, because there is a nested for loop inside a while loop, this results in the maximum amount of iterations to be the length squared.

Part 2:

1. Shell sort sets itself apart, because it allows the smaller values that are at the end of the array to be moved to the front of the array much quicker. The same goes for the larger values at the front of the array, which can be moved to the end of the array far earlier than with bubble sort. Say if there is an array of 100 and the gap sequence was 4, 2, 1, then there would probably take a considerable amount of time longer than if the gap sequence was expanded to say 25, 15, 10, 6, 4, 2, 1. That way, those smaller values that are caught in the last 10 spots of the array have a chance of moving back to the front much quicker than if we were to just swap them all in gaps of 4, 2, and 1. This will likely result in fewer iterations and swaps having to be performed.
   Information source: https://en.wikipedia.org/wiki/Shellsort

Part 3:

1. Although quicksort does have a worse case run time of $O(n^2)$, it's expected runtime is O(nlog(n)) which is far more efficient. Not to mention quicksort also performs in place sorting which means that extra space doesn't have to be allocated for it to work.

Part 4:

1. In order to keep track of the number of moves and comparisons, external variables will be used which will be incremented in each of the sorting functions and then printed along with the rest of the output in the main function.