Specification

Assignment 3: Sorting

Files needed:

bubble.h: the header file for the bubble sorting algorithm. Where we define the function for the bubble sort so we can end up running it on the sorting.c file.

bubble.c: our c programming implementation of the bubble sorting algorithm. Should be pretty analogous to the python pseudocode that was given to us in the lab document.

gaps.h: the pratt gap sequence is an essential part of the shell sorting algorithm, so we need to include this file in order to make shell sort work.

shell.h: the header file for the shell sorting algorithm. Where we define the function for the shell sort so we can end up running it on the sorting.c file.

shell.c: our c programming implementation of the shell sorting algorithm. Should be pretty analogous to the python pseudocode that was given to us in the lab document.

quick.h: the header file for the 2 quick sorting algorithms. Where we define the function for the shell sort so we can end up running it on the sorting.c file.

quick.c: our c programming implementation of the 2 quick sorting algorithms. Should be pretty analogous to the python pseudocode that was given to us in the lab document.

stack.h: simply the header file so we can incorporate the stack data structure into our sorting algorithms

stack.c: our implementation of the stack.

queue.h: simply the header file so we can incorporate the queue data structure into our sorting algorithms

queue.c: our implementation of the queue.

Sorting.c: very similar to the previous lab in which all the commands are run using getopt(). We will have a list of command line commands to run our sorting algorithms, in which they will be formatted, show how deep it went in the stack or queue, etc.

Makefile: a make file to easily run the program by prompting `make` into the Linux terminal on our Ubuntu virtual machine.

README.md: describes how the program works and runs in addition to how to use the makefile

WRITEUP.pdf - We are asked to analyze the 4 different sorting algorithms with respect to time. Also talk about some observations we made overall about the sorting algorithms, and how we went about experimenting with them. We also need to discuss how long are stack and queue data structures got in order to make sense of the technicality of each algorithm. Provide visuals like graphs where they deem necessary.

Description of the Program: This program involves us incorporating 4 different sorting algorithms. Sorting is one of the fundamental assignments that takes place in computer science and it is imperative that we at least understand what it takes to build these algorithms. These assignments not only involve implementing algorithms, but we also get to use two data structures that we should be familiar with in the stack and the queue to hold things in an array so the sorting can take place. Alot of comparing values will be done in this assignment but we also must understand by the end of it which ones are the fastest and why they are the fastest.

Sorting.c main()

```
main(int argc, char **argv)
While loop(opt=getopt(argc,argv,OPTIONS)) != -1
bubble flag = 0
shell flag = 0
quick flag = 0
QUICK flag = 0
Seed flag = 0
Size flag = 0
Elements flag = 0
Switch statement(opt)
    Case -a:
        bubble flag = 1
        shell flag = 1
        quick flag = 1
        QUICK flag = 1
    Case -b:
        bubble flag = 1
    Case -s:
        shell flag = 1
    Case -q:
        quick flag = 1
    Case -Q:
        QUICK flag = 1
    Case -r:
        Seed flag = 1
    Case -n:
        Size flag = 1
    Case -p:
        Element flag = 1

If bubble flag = 1
    Print bubble sort algorithm
If shell flag = 1
    Print shell sort algorithm
If quick flag = 1
    Print quick sort algorithm (stack)
If QUICK flag = 1
    Print quick sort algorithm (queue)
If seed flag = 1
    Randomize seed
If size flag = 1
    Change size of array
If element flag = 1
    Print elements
```

Very similar to assignment 2, we will be utilizing the getopt command to not only link all the files but to be able to use it as the main interface for the program. When compiling and running the test the makefile will be technically compiling this file just like we had the previous assignment make the mathlib-test file. Overall there doesn't seem to be much that needs to be said other than the fact that the all command line arguments will be run through this file.

bubble.c
//source lab document: Darrell Long Python Pseudocode

```
Def bubble_sort(arr):
  N = len(arr)
  Swapped = True
  While swapped:
    Swapped = False
    For i in range(1,n)
      If arr[i] < arr[i - 1]:
        Arr[i], arr [i - 1] = arr[i - 1], arr[i]
        Swapped = true
    N -=1
```

Pre Lab Questions:

1. 22<->7, 22<->9, 31<->5, 31<-> 13, 8<->7, 22<-> 5, 22<->13,9<->5,8<->5,7<->5
   10 rounds of swapping
2. The worst case for the bubble sort would be if the array was sorted in the descending order. If we had an array with 5 values in descending order, it would be a total of 10 swaps. The more values in the array the more tedious it would be if the values were in the worst case. Therefore the worst case time complexity would be n^2. source:https://en.wikipedia.org/wiki/Bubble_sort.

Bubble sort is what first comes to mind when I think of sorting. It requires us to do multiple passes in which we know at least the last element is in the right place after a pass. And after every pass you just subtract an extra 1 from the elements you need to check in that pass. The way this sorting works is that it starts at the beginning of the array and does a comparison with the n+1 element. If n > n+1 they swap, if n < n+1 they stay in place. Therefore it is the most tedious of the bunch, because it checks every element one by one until it reaches the end of the array before it starts the next pass.

Shell.c
//source: lab document, Darrell Long's python pseudocode

```
Def shell_sort(arr):
  For gap in gaps:
    For i in range(gap, len(arr):
      j = i
      Temp arr[i]
      While j >= gap and temp < arr[j - gap]:
        Arr[j], arr[j - gap] = arr[j - gap], arr[j]
        J -= gap
      Arr[j] = temp
```

Pre lab Questions 2:

1. The time complexity relies heavily on the gap because the more elements you have in an array the less efficient the algorithm actually is. For example if you have 100 elements you will be checking every n and n + 50th element before you check every n + 50/2th element and so on. So the more elements you have, the less efficient this algorithm will be.

Shell Sort from what I see is very inefficient. As explained in the lab question, the more elements that are in the array the less efficient it will tend to be. The reason being is because the gap is calculated by all the elements in an array divided by 2. So it will be checking the nth element while checking the nth element it will also check the nth element * (all elements/2). But it's not like we split the array in half and are conducting two swaps at once, the shell sort is literally just doing the bubble sort with a big gap in between which I think is very inconvenient.

Quick.c

Partition in Python: SOURCE - Lab Document, Darrell Long's Python Pseduocode

```
Def partition(arr, lo ,hi):
  Pivot = arr[lo + ((hi - lo) // 2)]
  i = lo - 1
  J = hi + 1
  While i < j:
    I += 1
    While arr[i] < pivot:
      I += 1
    j-=1
    While arr[j] > pivot:
       j -= 1
    If i < j:
      Arr[i], arr[j] = arr[j], arr[i]
  Return j
```

quick.c (stack)

SOURCE: lab document Darrell Long's Python Pseudocode

```
Def quick_sore_stack(arr):
  lo = 0
  hi = len(arr) - 1
  Stack = []
  stack.append(lo)
  stack.append(hi)
  While len(stack) != 0:
    Hi = stack.pop()
    Lo = stack.pop()
    P = partition(arr, lo, hi)
    If lo < p
      stack.append(lo)
      stack.append(p)
    If hi > p + 1:
      stack.append(p+1)
      stack.append(hi)
```

Quick.c (queue)
SOURCE: lab document Darrell Long's Python Pseudocode

```
Def quick_sort_queue(arr):
  Lo = 0
  Hi = len(arr) - 1
  Queue = []
  queue.append(lo)
  queue.append(hi)
  While len(queue) != 0
    Lo = queue.pop(0)
    Hi = queue.pop(0)
    P = partition(asrr, lo, hi)
    If lo < p:
      queue.append(lo)
      queue.append(p)
    If hi > p + 1:
      queue.append(p + 1)
      queue.append(hi)
```

Pre lab Question 3

1. First off it is highly unlikely that the time complexity for the quick sort occurs. However it still can occur if the leftmost or rightmost element is chosen as the pivot but it is like I said highly unlikely. Even then we can simply avoid this by choosing a pivot from the middle of the array. If we avoid the worst case then it is one of the fastest sorting algorithms. If it is the worst case however it is pretty slow.
   Source: https://www.baeldung.com/cs/quicksort-time-complexity-worst-case

As described in the pre lab question, the quick sort algorithm is one of the best sorting algorithms as long as the worst case is avoided. We will be needing to implement this algorithm with both the stack and the queue which will be described relatively soon. This algorithm is what we call a divide and conquer where we have a pivot, and the rest of the values will be partitioned into two sub arrays and be compared to the pivot to see which sub-array they go into. Thereby making it far less work than both the shell short and bubble sort. Then they will

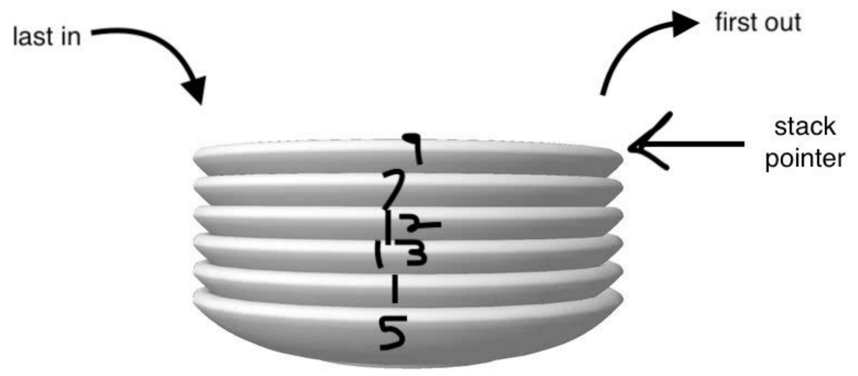
Stack.c

Struct definition: SOURCE DARRELL LONG

```
Struct stack {
  Uint32_t top;
  Uint32_t capacity;
  Int64_t *items;
};
Stack create{
*create stack
}
Stack delete{
Deletes stack
}
Stack full{
*stack full = top of stack = capacity
}
Stack empty{
*Stack empty = top is at 0
}
Stack size{
Returns stack size
}
Stack push{
*push element to stack. LIFO
}
Stack pop{
*take last element in first out
}
Stack print{
Print elements in a stack
}
```

Stack is a data structure that we will need to use in the quicksort. Because we aren't doing a recursive calling quicksort we are utilizing two data structures. The first being a stack. The stack is a last in first out meaning the last element that was put into the stack is the first one coming out. A common analogy is that of a stack of dishes. Refer to the visual below:

Front



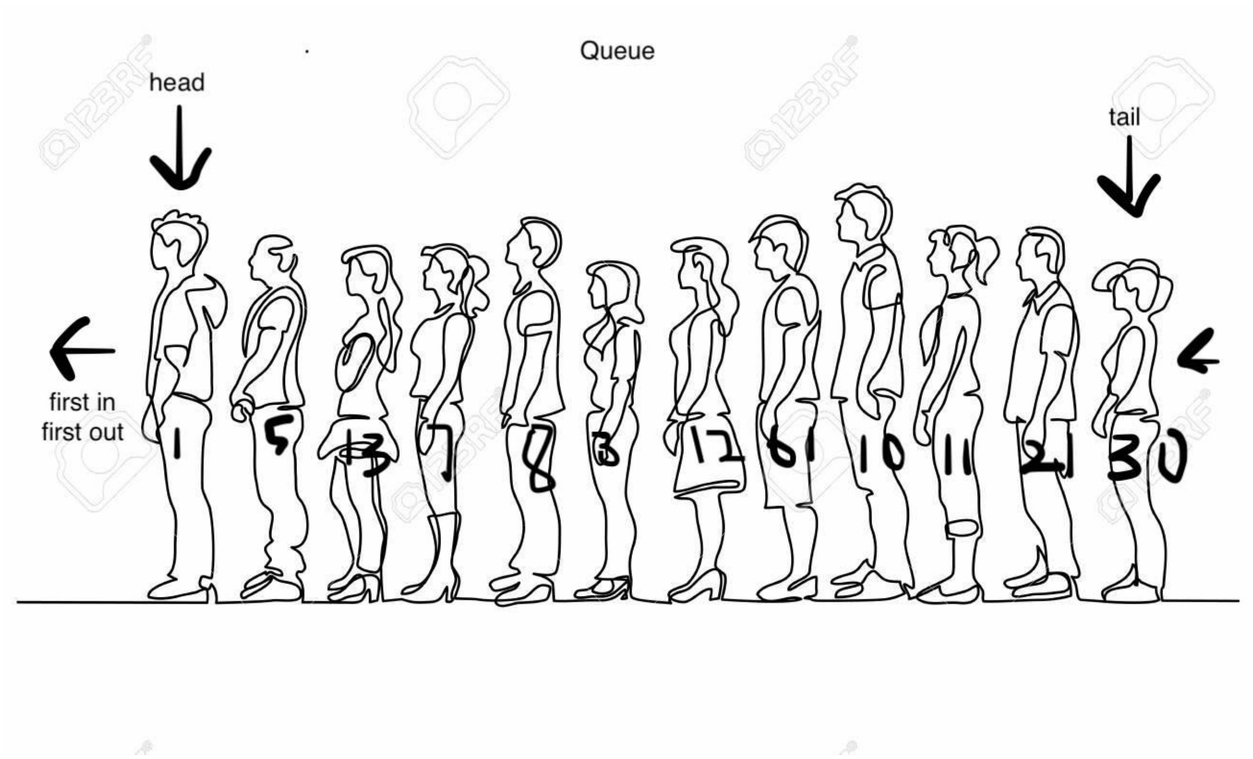
The first value stored in the stack was 5, but the first value to leave the stack is 9. The stack pointer keeps track of what was the last value. We are going to be using pointers, malloc, calloc to dynamically allocate memory for the stack and use the addresses in the quick sort. Once the memory is allocated we will perform the quicksort in which we will push and pop values, while also being able to check if the stack is full or empty, etc. We will also be calculating the amount of moves that goes down as well.

Queue.c

Struct definition: Source Darrell Long

```
Struct queue{
  Uint32_t head;
  Uint32_t tail;
  Uint32_t size;
  Uint32_t capacity;
  Uint32_t *items
Queue constructor{
*make queue
}
Queue destructor{
*destroy queue
}
Enqueue{
Add element to the queue first in first out
}
Dequeue{
First element in comes out
}
Queue size{
Returns size of queue
}
Queue empty{
If queue size = 0
}
Queue full
If queue size = capacity
}
```

Queue is a data structure needed to build the other implementation of quicksort. Like the stack it is a data structure, but unlike the stack it is a first in first out. If you think about it, it is almost as if it is the opposite of the stack. Here is an visual example of the queue:



As you can see the queue is exactly what you visualized. It is a line for memory addresses to be stored in which the first values stored in memory are the first values being pulled out of memory. So in the quicksort, we will be storing values in the queue and it will be distributed based on whether the value is less than or greater than the pivot.

<u>Prelab Question 4:</u>

1. I made some extra pass by reference variables which will count and return the moves of every swap, will count all the compares and return the compares, and for the quick sort will count all the queue and stack maxsize and return it. I realized that I could print strictly from the sorting functions but thought that using pass by references would be more efficient in tracking and overall more practical.

Sources for images:
Stack Image:
https://3dwarehouse.sketchup.com/model/60ae0c35-b5c4-4ac9-9a42-9893f7f7d709/Stack-of-White-Plates
Queue Image:
https://www.123rf.com/photo_114461545_stock-vector-continuous-one-line-drawing-group-of-people-waiting-in-line-silhouette-isolated-on-white-background-.html

Updates:
-Based all sorting algorithms on Darrell Long's Python Pseudocode.
-They look identical to his pseudocode.
-Counting for compares and moves are very off for quicksort.
- count for max stack size and max queue size is very off because I used a pass by reference.
- the -a command line argument prints out 3 extra times and I don't know why.