Aditya Agrawal
aagraw14@ucsc.edu

CSE13s Spring 2021
Assignment 3: Sorting
Design Document

This program will implement the sorting algorithms of bubble sort, shell sort, and two quicksorts. Additionally a stack and a queue will be implemented in the quicksort algorithms as well as a test harness to test the sorting algorithms. The program will also gather statistics about each sort that reflect its performance.

Pre-lab part 1:

1. It will take 5 rounds of swapping to sort the numbers 8, 22, 7, 9, 31, 5, 13
   Original: 8, 22, 7, 9, 31, 5, 13
   Round 1: 8, 7, 9, 22, 5, 13, 31
   Round 2: 7, 8, 9, 5, 13, 22, 31
   Round 3: 7, 8, 5, 9, 13, 22, 31
   Round 4: 7, 5, 8, 9, 13, 22, 31
   Round 5: 5, 7, 8, 9, 13, 22, 31
2. You can expect to see n-1 comparisons or rounds of sorting in the worst case scenario for bubble sort, with n being the length of the list of numbers. This is because worst case scenario would be if the array was in descending order, for example
   Original: 5, 4, 3, 2, 1
   Round 1: 4, 3, 2, 1, 5
   Round 2: 3, 2, 1, 4, 5
   Round 3: 2, 1, 3, 4, 5
   Round 4: 1, 2, 3, 4, 5
   n=5, n - 1 = 4, this checks out because there were 4 rounds of sorting

Pre-lab part 2:

1. The worst time complexity for Shell Sort depends on the gap because if there are a lot of elements that are out of place and the gap size is low then there will need to be a lot of comparisons/exchanges made to put the elements in the correct positions. That is why it is best to start with a bigger gap size and then progressively make it smaller for each iteration of the sort, this is because when the gap size is larger there are less elements being compared and less exchanges necessary to put elements in the correct positions. So if you start with a large gap size and progressively make it smaller the total number of comparisons/exchanges will be smaller because the larger gap sizes took care of the “heavy lifting” and by the time you reach the small gap sizes there will only need to be a minimal amount of comparisons/exchanges made to place an element in the correct position.
   https://www.youtube.com/watch?v=VxNr9Vudp4Y

Pre-lab part 3:

1. Quicksort is not doomed by its worst case scenario because the worst case scenario only really occurs when the picked pivot is either the smallest or largest element. The

median element of an unsorted array can be found in linear time, so if the median is found first and used as the pivot with the rest of the array partitioned around it the time complexity will be much lower than the worst case scenario. https://www.geeksforgeeks.org/can-quicksort-implemented-onlogn-worst-case-time-complexity/#:~:text=The%20worst%20case%20time%20complexity,(smallest%20or%20largest)%20element.&text=The%20idea%20is%20based%20on,be%20found%20in%20linear%20time.

Pre-lab part 4:

1. At first glance (subject to change) I will keep track of the number of moves and comparisons by assigning the number of moves and comparisons of each sort to static variables that are declared inside a function that returns the value of the variable upon being called. There will be a function of this type in each of the sort files, so when the function is called in the test harness file the function will return the value of the variable and because they are static variables they will retain their values.
   Correction: I declared an int variable in the header file using extern, and kept track of the value in the source file using that variable. Because it;s declared in the header file with extern it can be accessed from other files.

bubble.c

Pseudocode from asgn3.pdf

```
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
```

shell.c

pseudocode from asgn3.pdf

```
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]
                j -= gap
            arr[j] = temp
```

quick.c
pseudocode from asgn3.pdf

Quicksort in Python with a stack

```
def quick_sort_stack(arr):
    lo = 0
    hi = len(arr) - 1
    stack = []
    stack.append(lo) # Pushes to the top.
    stack.append(hi)
    while len(stack) != 0:
        hi = stack.pop() # Pops off the top.
        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)
```

Quicksort in Python with a queue

```
def quick_sort_queue(arr):
    lo = 0
    hi = len(arr) - 1
    queue = []
    queue.append(lo) # Enqueues at the head.
    queue.append(hi)
    while len(queue) != 0:
        lo = queue.pop(0) # Dequeues from the tail.
        hi = queue.pop(0)
        p = partition(arr, lo, hi)
        if lo < p:
            queue.append(lo)
            queue.append(p)
        if hi > p + 1:
            queue.append(p + 1)
            queue.append(hi)
```

```
def partition(arr, lo, hi):
    pivot = arr[lo + ((hi - lo) // 2)]; # Prevent overflow.
    i = lo - 1
    j = hi + 1
    while i < j:
        i += 1 # You may want to use a do-while loop.
        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
```

stack.c
stack_create{
allocate space for stack
set initial values
allocate space for items[]
}
stack_empty{
return (top ==0)
}
stack_full{
return (top == capacity)
}
stack_size{
return top
}
stack_push{
if (stack_full) return false
items[top] = x
top +=1
return true
}
stack_pop{
if (stack_empty) return false
top -= 1
*x = items[top]
return true
}

queue.c
queue_create{
allocate space for queue
set initial values
allocate space for items
}
queue_empty{
return (size ==0)
}
queue_full{
return size == capacity
}
queue_size{
return size
}
enqueue{
if (queue_full) return false
items[tail] = x
tail = next(tail, capacity)
size += 1
return true
}
dequeue{
if(queue_empty) return false
*x = items[head]
head = next(head, capacity)
size -= 1
return true
}

sorting.c

OPTIONS = “absqQrnp”
int main(int argc, char **argv){
if (opt = getopt(argc, argv, OPTIONS))
switch(opt)
case a: set all to true
case b: bool_b = true
case s: bool_s = true
case…

if (bool_b){

```
        enable bubble sort
}
if(bool_s){
        enable shell sort
}
if(bool_…){
        enable/set ….
}
...
...
…
```

correction to sorting.c:
instead of using boolean variables to keep track of if the user entered the flag in the command line I added the flag to a set. After checking all the command line arguments i loop through all the flags and if the flag was in the set I printed out the type of sort specified as well as the array elements.

Other comments:
The comparisons and moves were tracked through extern variables, whenever there was a comparison in a loop condition or if statement i replaced the boolean condition statement with a function which executes the same thing but also increments the variable that stores the number of comparisons. For the moves i increment the variable that stores the number of moves every time a value was ‘moved’ or copied from the array.