Fabrice Kurmann
fkurmann@ucsc.edu
19 April, 2021

CSE 13S Spring 2021
Assignment 3: Sorting: Putting your Affairs in Order
Design Document

Prelab Questions:

Part 1, Bubble Sort:

1) How many rounds to swap 8, 22, 7, 9, 31, 5, 13?

begin:	8	22	7	9	31	5	13
	8< 22	22 >7	22>9	31>22	31>5	31>13	31>13
pass 1:	8	7	9	22	5	13	31
	8>7			22>5	22>13		
pass 2:	7	8	9	5	13	22	31
			9>5				
pass 3:	7	8	5	9	13	22	31
		8>5					
pass 4:	7	5	8	9	13	22	31
	7>5						
pass 5:	5	7	8	9	13	22	31 - End

This problem is sorted after 5 passes.

2) Number of comparisons for worst case of bubble sort.

Assuming perfectly backwards = worst case: 5 items = 14 comparisons

begin:	5	4	3	2	1	(4 comps)
pass 1:	4	3	2	1	5	(4 comps)
pass 2:	3	2	1	4	5	(3 comps)
pass 3:	2	1	3	4	5	(2 comps)
pass 4:	1	2	3	4	5	(1 comps)

= 14 comps
5 items

Part 2: Shell Sort

1) General worst case = $O(n^2)$ using Shell's gap, $O(n \log^2 n)$ using $2^p 3^q$ gap

Choosing gaps is a question of optimization, for certain inputs, different sequences will be faster than others. Passes run more slowly with fewer gaps, meanwhile an excessive amount of gaps is often unnessasary and thus increases the number of comparisons for no benefit.

Information Sources: Priyansh Mangal atCodesdope.com (gap sequence comparison), Codinggeek.com (gap sequence trade offs).

Part 3: Quicksort

1) Despite it's worst case time complexity being slower than competitors quicksort is often faster since it's average case is faster and the worst case, by nature happens rarely. The average case for quicksort actually results in a time complexity of O(n log n) which is the same as many competitors including merge sort. At this point, it is important to consider the defenition of time complexity, which is not an exact measure of how long or how many comparisons must be made, but rather a measure of how a program's complexity grows in proportion the size of it's input, in this case the unsorted array.

For this reason, not all functions with growth function O(n log n) are equally fast. Indeed, here quicksort has proven, through nothing but modeling and testing, to be considerably more efficient than it's competitors.

Information sources: Paul Hseih at Azillionmonkeys.com (guide to sorts), Rob Bell at rob-bell.net (guide to big o notation).

Part 4: Returning infomation

1) Since the functions for the various types of sort do not return anything (void) information regarding their comparison and movement counts as well as, for quick.c, maximum stack/queue size should be stored in global variables that can then be accessed outside of the sort functions by the main function in sorting.c. The functions will be allowed to modify and access this value directly, before it is read. Static variables are not an option since they exist only in the file they are declared and this program covers multiple source code files.

Implementation of Sorts:

Citation: Sort functions are based on pseudocode provided by Professor Long in the assignment handout.

Pseudocode:

Italicized code pertains to global tracking variables

Global variables:

Comparisons

Moves

Max_stack_size

Max_queue_size

bubblesort(array, length):

```
        Reset global variables
        bool swapped = true
        while swapped == true
                swapped = false
                for (int i = 1, i < length, I++)
                        comparisons ++
                        if array(I) < array(I-1)
                                swap
                                swapped = true
                                moves ++
                length —



Gaps = array() // Provided in the gaps.h header file
Shellsort(array, length):
        Reset global variables
        for item in gaps:
                for (int i = item, i<length, i++):
                        index = i, holder = array[i]

                        while (index >= gap and array[index-gap] > holder):
                                swap (array[index], array[index-gap])
                                index -= gap
                        array[index] = holder



Quicksort(array, length): (Quicksort for stacks and queue differ only in stack versus queue calling)
        Reset global variables
        lo = 0, hi = length - 1;
        push to stack: lo and hi
        Max_stack_size +=2

        while (stack != empty):
                partition = partition(array, hi, lo)

                comparisons ++
                if (partition > lo):
                        push to stack: lo and partition
                        Max_stack_size += 2
                comparisons ++
                if (partittion + 1 < hi)
                        push to stack: partition + 1 and hi
                        Max_stack_size += 2



Partition(array, lo, hi):
        pivot = 1/2*(hi-lo)
        xlo = lo - 1, xhi = hi - 1
        while (xlo < xhi):
                comparisons++
                x_lo ++
                while array[xlo] < pivot:
```

```
            comparisons++
            xlo++
        xhi—
        while array[xhi] > pivot:
            comparisons++
            x_hi—

        if xlo < xhi:
            comparisons++
            swap(array[xlo], array[xhi]
    return xhi
```

Implementation of tester file:

Use set data structure.

Implementation of stack and queue:

Citation: Stack and Queue constructors and functions are based on code provided by Professor Long in the assignment handout and Stack/Queue lecture.

Required Stack Methods:
- constructor/destructor
- empty check/full check/size check
- print (for debugging)
- stack push - add a new item to top
- stack pop - pop from top

Required Queue Methods:
- constructor/destructor
- empty check/full check/size check

print (for debugging)
dequeue - remove one item from the head of the queue
enqueue - add one item to the tail of the queue