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) = 14 comps
pass 2:	3 2 1 4 5	(3 comps) 5 items
pass 3:	2 1 3 4 5	(2 comps)
pass 4:	1 2 3 4 5	(1 comps)

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.

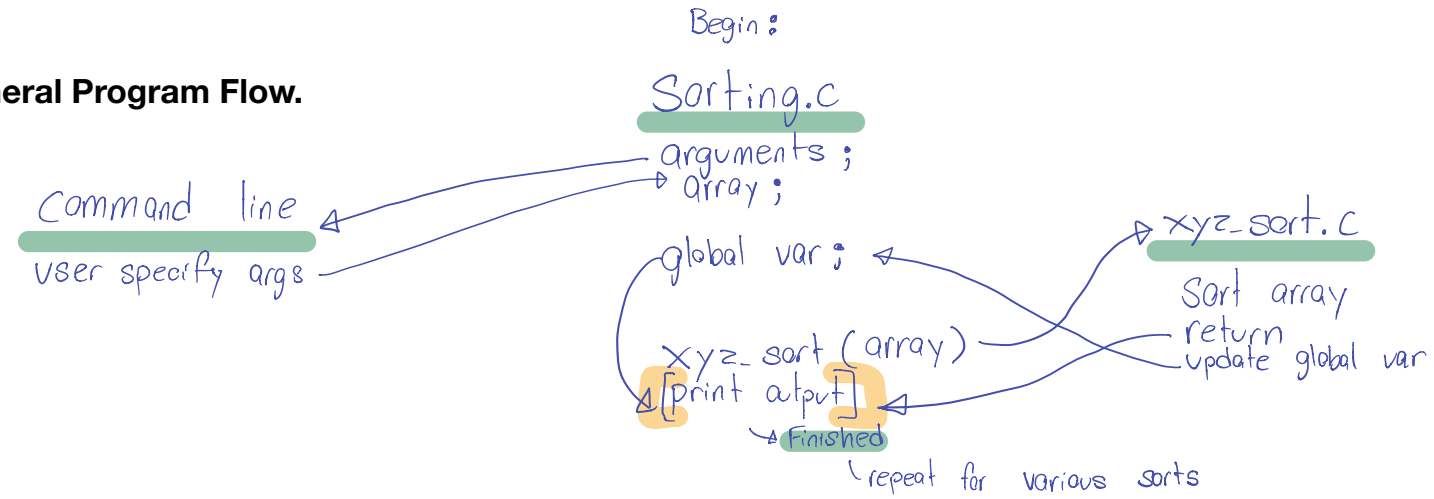
For this reason, not all functions with growth function O(n log n) are equally fast. Indeed, here quicksort has proven, through nothing but modeling and testing, to be considerably more efficient than it's competitors.

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.

General Program Flow.



As illustrated in this flow chart, the program essentially needs to generate 2 pieces of information for it to finish printing output for a particular sort:

The global variables holding sort effiency information

The sorted array, which is passed via pointer to and from the sorting algorithm.

Global variables: These to be declared in the sorting.c file. They are modified by the sort function currently running before being printed out by the sorting.c file's code after every sort function is called individually.

Note: Italicized code pertains to global tracking variables

Comparisons

*Moves (*Note that swapping elements involves **3** moves)

Max_stack_size

Max_queue_size

Array: The array is to be declared and initialized to the user's length and seed specification in sorting.c. It is then passed by reference (pointer) to the sorting algorithms which modify it (sort it) again through

pointers. These sort functions do not return anything, upon their completion, sorting.c will simply have it's array back in sorted form.

Note: In order for all sort types to be run, copies of the original array will have to be made for each algorithm to manipulate.

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

```
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 **
                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**
    return xhi
```

**Swapping requires the use of a temporary storage variable, each swap is therefore counted as three moves.

```
Swap(Array[1], Array[2]):
    temp = Array[1]
    Array[1] = Array[2]
    Array[2] = temp
    moves += 3
```

Implementation of tester file:

Use set data structure. The idea is to add all command line arguments to the set in the form of integers (for this I use ASCII string to integer representation). Then the set will, as sets do, assure that replicas are not repeated.
Finally, for every item in the set, at this point there are no repetitions, the corresponding output function will be run.

Pseudocode for input management

```
Argset = Set()
Elements = 100, seed = xxxxx, print = 100
Switch:
    Arg = a, b, s, q, Q
        set.add arg
    Arg = r, n, p
        elements = n arg, seed = r arg, print = p arg
```

** For the above switch statement, I rely on the getopt class's built in functions to track command line arguments, I found my information in this topic at: azrael.digipen.edu.

Pseudocode for output generation functions (mirrored by all four output options).

Printoutput_bubble(elements, seed, print)
    Dynamically allocate array(elements)
    For i in array:
        array i = random(seed)

    bubblesort(array, elements)

    Print(elements, numComparisons, numMoves)
    for int i in print:
        print array(i)
            Make sure to print the above in a 5 column tabular format.

    free(array)
    return

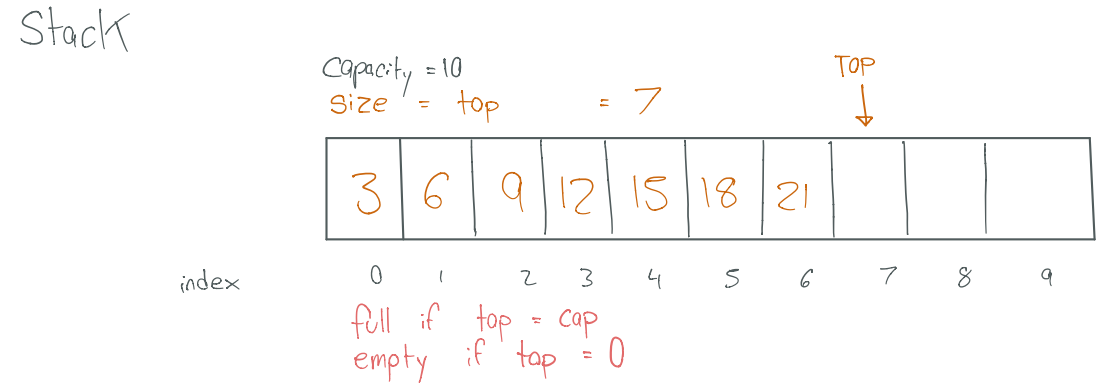
Implementation of stack and queue:

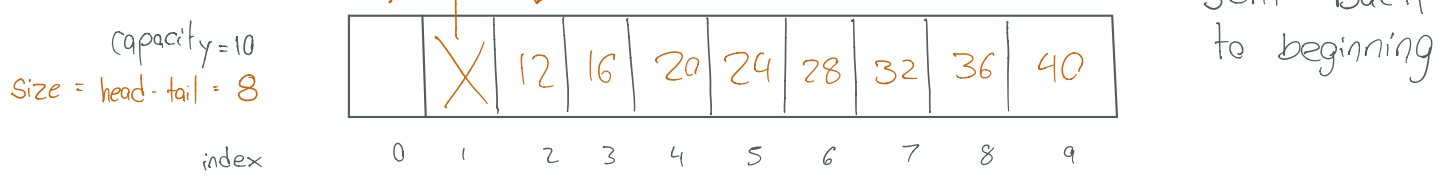
Citation: Stack and Queue constructors and functions are based on code provided by Professor Long in the assignment handout and Stack/Queue lecture.

Information to store for stack:
    Top
    Bottom
    Items

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



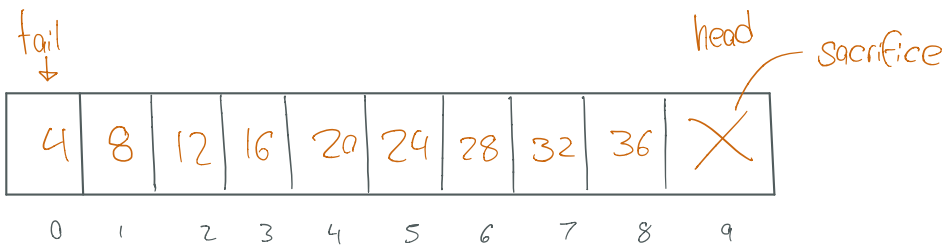


to beginning

empty if top = tail

full if head + 1 = tail (mod) cap

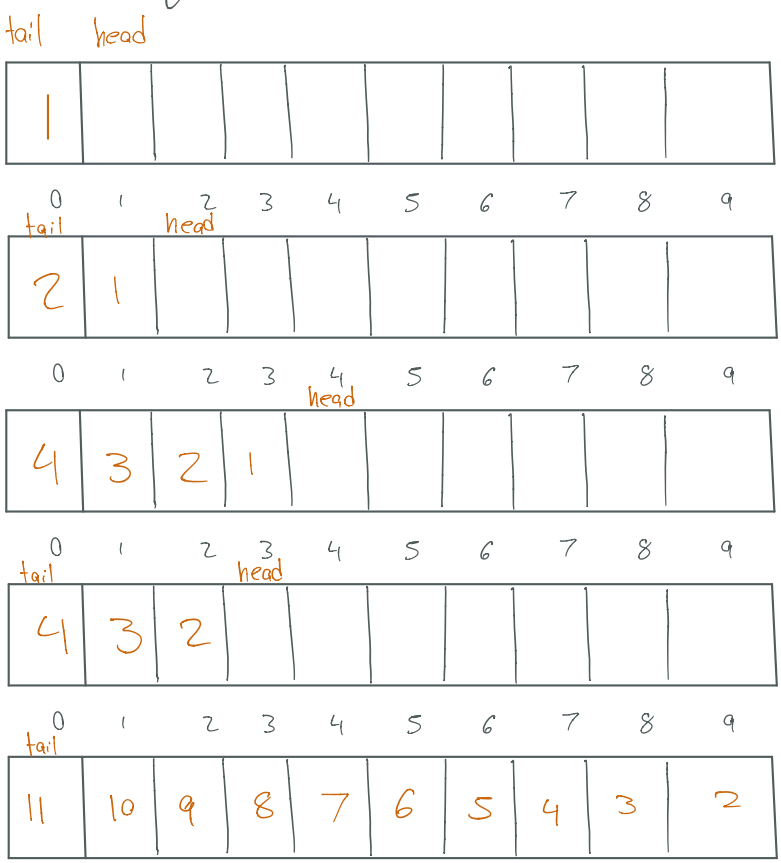
$1 \neq 2 \% 10 = 2$



$0 + 1 = 0 \bmod 10 \neq \text{full}$

$9 + 1 = 0 \bmod 10 = \text{full}$

enqueue at tail, dequeue at head



Above are different drawings to understand different cases for stacks and queues. Stacks are far simpler since the bottom value doesn't move. For this reason, a stack in the form of an array is able to

be relocated.

Meanwhile, as illustrated above, the queue, assuming it is a circular queue, which is the only type that is efficient to push and pop, may run out of space when the tail is in the middle. If this happens, reallocating the array is not possible since the items ahead of the tail cannot be pushed foreword. Based on this illustration, I decided to go with a fixed length array for my queue and a dynamic length for my stack.