Assignment 3
Sorting: Putting your affairs in order

DESIGN DOCUMENT

Zack Traczyk
CSE13S - Spring 2021

Due: April 25th at 11:59 pm

1 Objective

The objective of this lab is to implement Bubble Sort, Shell Sort, and two Quick-sorts. In addition to these implementations a big O analysis is done.

2 Given

These code snippets are given:

- Stack implementation for Quicksort
- Queue implementation for Quicksort
- Python implementation of the algorithms (for pseudocode)

3 Prelab Questions

3.1 Bubble Sort

a. How many rounds of swapping will need to sort the numbers 8,22,7,9,31,5,13 in ascending order using Bubble Sort?

8, 22, 7, 9, 31, 5, 13 - original
8, 7, 9, 22, 5, 13, 31 - round 1
7, 8, 9, 5, 13, 22, 31 - round 2
7, 8, 5, 9, 13, 22, 31 - round 3
7, 5, 8, 9, 13, 22, 31 - round 4
5, 7, 8, 9, 13, 22, 31 - Sorted

5 Rounds of swapping where required to sort the numbers.

b. **How many comparisons can we expect to see in the worse Case scenario for Bubble Sort? Hint: make a list of numbers and attempt to sort them using Bubble Sort.**

   The worst Case scenario is a list in reverse order. Each round takes n comparisions. Then it takes n iterations to completely sort the list making the worst Case take $O(n^2)$ comparisions.

3.2 Shell Sort

a. **The worst time complexity for Shell Sort depends on the sequence of gaps. Investigate why this is the Case. How can you improve the time complexity of this sort by changing the gap size? Cite any sources you used.**

   After watching sorting visualizations shell sort complexity made more sense. Shell sort's efficiency is dependent on the gaps used to sort. Imagine an array where the 1 is at index 0, 2 is at index n, 3 is at index 1 and four is at index n - 1. If shell sort had a gap size of two then it would be as inefficient as bubble sort. However, if the gap size starts at 1 less than the length of the array and decreases by 1, then the algorithm would be super efficient. Codesdope helped answer this question.

3.3 Quicksort

a. **Quicksort, with a worse Case time complexity of $O(n^2)$, doesn't seem to live up to its name. Investigate and explain why Quicksort isn't doomed by its worst Case scenario. Make sure to cite any sources you use.**

   Quicksort's worst Case happens when a pivot element is picked at either the end or beginning of the array. However, this does not doom Quicksort since the pivot point is decided by the programmer. In other words, $O(n^2)$ only happens because of the programmers own fault rather than intrinsic inefficiency of the algorithm. Baeldung CS helped answer this question.

3.4 General Sorting

a. **Explain how you plan on keeping track of the number of moves and comparisons since each sort will reside within its own file.**

   The number of moves and comparisons will be tracked by using global variables that are defined in a header file and initialized in the test harness.

4 Test Harness

The test harness takes in a couple of arguments specifying which sorts to run and for what size and how many elements. Program arguments are first parsed and stored in a bit field.

1 = True and 0 = False, [Bubble, Shell, Quicksort1, Quicksort2]. This process can be seen in *alg.* (1).

```
if no program arguments then return error;
run_all = False;
execute = 0000;                                              /* bit field */
seed = 13371453;                         /* Initialized with default seed */
size = 100;                              /* Initialized with default size */
elements = 100;            /* Initialized with default number of elements */
while get next argument with do
    switch opt do
        case a do run_all is true;
        case b do execute = execute bitwise or 0001;
        case s do execute = execute bitwise or 0010;
        case q do execute = execute bitwise or 0100;
        case Q do execute = execute bitwise or 1000;
        case r do seed = argument value;
        case n do size = argument value;
        case p do elements = argument value;
        case h do print help message;
    end
end
if execute empty and run_all is not true then return error;
```

Algorithm 1: Parse Program Arguments

After the program arguments are parsed, the program then either runs all or iterates through the execute bitfield produced in the parse. This process can be seen in *alg.* (2). This uses a dynamically sorted array that is defined before execution.

```
functions = {bubble_sort, shell_sort, quicksort_stack, quicksort_heap};
for i from 0 to 4 do
    reset seed;
    populate array with random values until given size;
    if run_all or execute bit i is True then
        functions[i](array, size);
    end
    print analytics and array elements up to elements variable;
end
```

Algorithm 2: Execute Algorithms

5 Algorithm Implementation

Given the python pseudocode in the assignment document, the four sorting algorithms were pretty trivial to implement. The only added component is the metrics for comparisons, swaps, and data structure size if using Quick Sort. A header file, analytics.h, was created to make global variables for these values, then they were called across the various algorithms and referenced in the main sorting.c.

5.1 Stack

The stack for the Quicksort stack implementation was put together using Professor Long's code as well as teaching assistant Eugene Chou's.

5.2 Queue

The queue for the Quicksort queue implementation was adapted from the stack code since the variations are slight. The tail and head are kept track of instead of just top index. Items are appended to the head index and removed from the tail. The items wrap around from n - 1 to 0 to reduce the need for allocating space when empty memory exists from the popped stack elements.