Ryan Hueckel
jrhuecke@ucsc.edu
4/22/2021

CSE 13s Spring 2021
Assignment 3: Putting your affairs in order

Pre-lab Questions:

Part 1:

1. It would take 5 rounds of sorting.
2. Given n numbers, the first pass of sorting there is going to be n-1 pairs of numbers compared. In the next pass, only n-2 pairs need to be compared since we know the largest number will have been put at the end in the previous pass. If the numbers are perfectly out of order, this will continue until it only needs to swap the first two numbers. So the max number of comparisons for a bubble sort will be (n-1) + (n-2) + (n-3)...

Part 2:

1. After messing around with the sorts in python, I found out that the last gap that a shell sort will sort with will be 1, which is essentially just bubble sorting. So, the efficiency of the shell sort is based on how well the previous passes using other size gaps can set up the array so that the last pass does not have to do as much work to move numbers around. With the 14 number long array I messed around with, the best gap sequences I could find were [5, 3, 1] and [3, 2, 1] which both went through the while loop 10 times.

Part 3:

1. The reason quicksort is not doomed by its worst case scenario is due to how unlikely it is to happen. In order for the worst case scenario of quicksort to occur, everytime a pivot is chosen it would have to be either the max or min value of the array. This would result in only 1 number being moved per cycle, making it very inefficient. The likelihood of this occurring is extremely low though. For example, given an array with 20 numbers, the first time it chooses a pivot there is a 1/10 chance of it picking either the min or max value. Each sequential time it chooses a pivot, the program would have to hit that same chance of 2/(num ints left to sort). Rolling that low chance every single time a pivot is chosen is so very unlikely that the worst case scenario for quicksort is not a big deal. (https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort, Khan academy page describing how quicksort works)

Part 4:

1. I will be defining external int variables that will store the moves and comparisons for each type of sort. These external variables will be declared outside of the main function in ‘sorting.c’ and will have to be declared in every function that uses them. (info from Ch 1.10 of textbook) I may also need to utilize pointers.

Program Function:

This program will simulate 3 different types of sorting algorithms: bubble sort, shell stort, and two quicksorts implemented in different ways. The first quicksort will utilize a stack and the second will use a queue. The user uses command options to tell the program what sorting

algorithms to use, how large of an array to sort (default 100), to set a random seed (default 13371453), and how many elements of the array should be printed out (default 100). The program's output will include the number of elements sorted, the number of moves made in each sort, the number of comparisons made with each sort, and then a table containing the specified number of elements from the array.

Program Structure:

Each sorting algorithm will have its own header file that references a file that implements the sort as a function. Shell sort will also have an extra file 'gaps.h' that contains the gap sequence to be used in the shell sort. 'sorting.c' contains the main function which will take input and produce output by calling the sorting algorithms included from the header files.

sorting.c:

This is the file that will contain the main function. By taking inputs through command-options, it sets the random seed, creates the array, and decides what sorting algorithms to push the array through. It then produces the output for each sorting algorithm based on the output specified in the command options. The inputs will be stored using a set.

bubble.c:

This file contains the bubble sort function which takes an array pointer as an input, sorts the array using bubble sort, and does not directly output anything. Statistics on the sorting process will be stored in external variables that are defined at the beginning of the function.

shell.c:

This file contains the shell sort function which takes an array pointer as an input, sorts the array using shell sort, and does not directly output anything. Statistics on the sorting process will be stored in external variables that are defined at the beginning of the function. This file also includes the 'gaps.h' file which it uses to determine the gaps to be used during the shell sorts.

quick.c:

This file contains both quicksort functions and operates the same as the previous files. The first function uses the stack ADT, implemented in 'stack.c', and the second uses the queue ADT, implemented in 'queue.c'.

stack.c:

This file implements the stack ADT that will be used by the first quicksort function.

queue.c:

This file implements the queue ADT that will be used by the second quicksort function.