CSE 13S Fall 2021
James Gu
jjgu@ucsc.edu
13 October 2021

Assignment 3: Sorting: Putting your Affairs in Order

Description of Program:
Sorting elements is a very common problem in Computer Science so this program is an implementation of a collection of various sorting algorithms. These include the following: Insertion Sort, Shell Sort, Heap Sort, and Quicksort. Insertion Sort goes through each element one by one and places them in the correct spot. Shell Sort is a variation of Insertion Sort, but first sorts pairs of elements that start with a gap and continues until the gap is one. Heap Sort can be visualized as a tree in which the top most node is the greatest number for max Heap Sort and the least for min Heap. This is called the parent node and its children nodes need to be less than the parent for max heap and greater than for min heap. If the child node doesn't obey the rule of the heap they are switched with the parent and this process is repeated until the entire tree obeys the rules. Each time in this process the top-most node is placed outside the tree since it no longer needs to be switched with anything and is sorted already. Quicksort involves a pivot number which has a partition: the left partition has numbers smaller than it and the right partition bigger than it. Then the sort organizes each partition using the same process as the original pivot.

Files:
insert.c
Implements Insertion Sort

insert.h
Specifies the interface to insert.c

heap.c
Implements Heap Sort

heap.h
Specifies the interface to heap.c

quick.c
Implements Quicksort

quick.h
Specifies the interface to quick.c

set.h
Implements and specifies the interface for the set ADT

stats.c
Implements the statistics module

stats.h
Specifies the interface to the statistics module

shell.c
Implements the Shell sort

shell.h
Specifies the interface to shell.c

sorting.c
Contains main() and may contain any other functions necessary to complete the assignment

Makefile
• CC = clang must be specified.
• CFLAGS = -Wall -Wextra -Werror -Wpedantic must be specified.
• make must build the sorting executable, as should make all and make sorting.
• make clean must remove all files that are compiler generated.
• make format should format all your source code, including the header files.

README.md
This must use proper Markdown syntax. It must describe how to use your program and Makefile. It should also list and explain any command-line options that your program accepts. Any false positives reported by scan-build should be documented and explained here as well. Note down any known bugs or errors in this file as well for the graders.

DESIGN.pdf
This document must be a proper PDF. This design document must describe your design and design process for your program with enough detail such that a sufficiently knowledgeable programmer would be able to replicate your implementation. This does not mean copying your entire program in verbatim. You should instead describe how your program works with supporting pseudocode.

WRITEUP.pdf
This document must be a PDF. The writeup must include the following:
• What you learned from the different sorting algorithms. Under what conditions do sorts perform well? Under what conditions do sorts perform poorly? What conclusions can you make from your findings?
• Graphs explaining the performance of the sorts on a variety of inputs, such as arrays in reverse order, arrays with a small number of elements, and arrays with a large number of elements. Use a UNIX tool — not some website — to produce these graphs. gnuplot is recommended. Attend section for examples of using gnuplot and other UNIX tools

Pseudocode:

insert.c:

```
insertion_sort(A):        //A is an array of random elements.
    Set i to 0.
    For length of A times:
        Set j to i
        Set temp to the i^th element in A.
        While j is greater than 0 and temp is less than the j - 1^th element in A:
            Set the j^th element in A to the j-1^th element in A
            Subtract 1 from j
        Set the j^th element of A to temp.
```

shell.c:

```
gaps(n):        //n is an integer
    Set i to log(3 + 2 * n) / log(3).
    For when i is still greater than 0:
        Yield the current value of (3^i - 1) / 2 to whatever calls gaps()
        Subtract 1 from i.

shell_sort(A):  //A is an array of random elements.
    For the current gap in gaps(length of A):
        Set i = gap.
        For length of A - i times:
            Set j to i.
            Set temp to the i^th element in A.
            While j is greater than or equal to gap and temp is less than the j-gap^th
            element:
                Set the j^th element of A to the j-gap^th element of A
                Subtract j by gap
            Set the j^th element of A to temp
```

heap.c:

```
max_child(A, first, last):        //A is an array of random elements, first and last are integers
    Set left to 2 * first
    Set right to left + 1
    If right is less than or equal to last and the right-1^th element of A is greater than the left-1^th
    element of A:
        Return right
    Return left

fix_heap(A, first, last):         //A is an array of random elements, first and last are integers
    Set found to false
    Set mother to first
    Set great to max_child(A, mother, last)
    While mother is less than or equal to last/2 and found is false:
        If the mother-1th element of A is less than the great-1th element of A:
            Set temp to the mother-1th element of A
            Set the mother-1th element of A to the great-1th element of A
            Set the great-1th element of A to temp
            Set mother to great
            Set great to max_child(A, mother, last)
        Else:
            Set found to true

build_heap(A, first, last):     //A is an array of random elements, first and last are integers
    Set father to last/2
    For when father is still greater than first - 1:
        fix_heap(A, father, last)
        Subtract 1 from father

heap_sort(A):       //A is an array of random elements
    Set first to 1
    Set last to the length of A
    build_heap(A, first, last)
    Set leaf to last
    For when leaf is still greater than first:
        Set temp to the first-1th element of A
        Set the first-1th element of A to the leaf-1th element of A
        Set the leaf-1th element of A to temp
        fix_heap(A, first, leaf-1)
        Subtract 1 from leaf

quick.c:
partition(A, lo, hi):   //A is an array of random elements, lo and hi are integers
    Set i to lo - 1
    Set j to lo
    For hi - lo times:
        If the j-1th element of A is less than hi-1th element of A:
            Add 1 to i
            Set temp to the i-1th element of A
            Set the i-1th element of A to the j-1th element of A
            Set the j-1th element of A to temp
    Set temp to the i-1th element of A
    Set the i-1th element of A to the hi-1th element of A
    Set the hi-1th element of A to temp
    Return i + 1

quick_sorter(A, lo, hi):        //A is an array of random elements, lo and hi are integers
```

```
    If lo is less than hi:
        Set p to partition(a, lo, hi)
        quick_sorter(A, lo, p-i)
        quick_sorter(A, p+1, hi)

quick_sort(A):        //A is an array of random elements
    quick_sorter(A, 1, length of A)

sorting.c:
    Set a_flag to false
    Set e_flag to false
    Set i_flag to false
    Set s_flag to false
    Set q_flag to false
    Set r_flag to false
    Set n_flag to false
    Set p_flag to false
    Set h_flag to false
    While the user wants to test functions using the command line:
        If case is ‘a’:
            Set e_flag, i_flag, s_flag, q_flag to true
            break
        If case is ‘e’:
            Set e_flag to true
            Break
        If case is ‘i’:
            Set i_flag to true
            Break
        If case is ‘s’:
            Set s_flag to true
            Break
        If case is ‘q’:
            Set q_flag to true
            Break
        If case is ‘r seed’:
            Set r_flag to true
            If no seed was given:
                Set random_Seed to 13371453
            Else:
                Set random_Seed to seed
            break
        If case is ‘n size’:
            Set n_flag to true
            If no size was given:
                Set array_Size to 100
            Else:
                Set array_size to size
            break
        If case is ‘p elements’:
            Set p_flag to true
            If elements is less than array_size:
                Print the entire array
            Else if elements wasn’t given:
                Print 100 elements from array
            Else:
                Print out elements number of elements from array
            break
        If case is ‘h’:
            Set h_flag to true
            break
    Create a random array using random_Seed and size of array_Size
    If e_flag is true:
        Run heap.c
    If i_flag is true:
        Run insertion.c
    If s_flag is true:
        Run shell.c
    If q_flag is true:
        Run quick.c
    If h_flag is true:
        Print out program usage
```