Diwa Ashwini Vittala
CSE 13S - Fall 2021

⁰❀ Assignment 3: Design ❀。

SORTING: PUTTING YOUR AFFAIRS IN ORDER

About the program:

The program uses various sorting methods to sort integers in a random array. The user has the option to ask the program to work with a certain number of elements, set the seed, and size of the array. The sorting algorithm options include heap sort, quick sort, shell sort, and insertion sort.

Heap Sort:

> This sorting method restructures the array into a "tree" or "heap" style where the largest number is at the top of the heap. Then, it pulls that large number out of the heap and adds it to the end of the array and fixes the heap again with the remaining elements. This process keeps repeating until the array is sorted from smallest to largest element order.

Shell Sort:

> This sorting method begins by creating a gap value, and comparing the two elements at the end points of that gap. If the value on the left is greater than the right, it swaps them and then moves the gap to compare the left + 1 and right + 1 elements and repeats the same process until it reaches the end of the array. The gap then decreases, and the values at the two ends are compared then moved just as before. This process repeats until the gap is only 1 element wide (so we are comparing elements next to each other), and the end result is an array sorted from smallest to largest element order.

Insertion Sort:

> This sorting method is the classic sorting algorithm where each element is compared to the element next to it until its rightful place is found. The result is an array sorted from smallest to largest element order.

Quick Sort:

> This sorting method parts the function at one of the numbers in the array, places all values smaller than the partition to the left and all values larger than the partition to the right. The algorithm then uses recursion to sort through the left and right of that partition using new partitions, until the entire array is sorted from smallest to largest element order.

About the file sorting.c:

Input(s): in line command of a, e, i, s, q, r (with a number), n (with a number), p (with a number), and h are given during execution (for example, ./sorting -e). Multiple commands may be given.

-a all sorting algorithms
-e Heap Sort
-i perform Insert Sort
-s perform Shell Sort
-q perform Quicksort
-r set a random seed, with default as 13371453

-n set array size, with default as 100
-p set number of elements to print out from the array, with default as 100
-h help
Output: The program outputs the specified number of elements from the sorted array using the given seed and array size information. It also prints out information about the number of elements in the array, as well as the number of moves and comparisons each sorting algorithm uses.

Other files involved:
Makefile, README.md, WRITEUP.pdf, heap.c, set.h, stats.c, heap.h, quick.c, shell.c, stats.h, insert.c, quick.h, shell.h, insert.h, sorting.c

Pseudocode/solution breakdown for:
WRITEUP.pdf:
Includes analysis about the functions and includes graphs that compares the functions in the math library with the functions in our library.

sorting.c:
Include header files for the sorts, stats, and set.
Include standard headers and getopt headers
A function to print out the error and proper usage message:
    Print statement that shows how to properly use the sorting.c file
Enumerate the sort options
Define all the possible options for the function
Start the main function and allow for a string of character inputs
    Create an empty set
    Declare variables for the seed (13371453 by default), size (100 by default),
        elements (100 by default) and the masking value of 30 bits.
    While the input options are valid:
        Start a switch statements to evaluate the options
        Possibility that help was selected:
            Call the message function to print the proper usage
            Return 0 so we stop running the program
        Possibility that a was selected:
            Insert all the sorts into our empty set, then break
        Possibility that e was selected:
            Insert heap sort into our empty set, then break
        Possibility that s was selected:
            Insert shell sort into our empty set, then break
        Possibility that i was selected:

Insert insertion sort into our empty set, then break
Possibility that q was selected:
Insert quick sort into our empty set, then break
Possibility that r was selected:
Set the seed value to the given input, then break
Possibility that n was selected:
Set the size value to the given input, then break
Possibility that p was selected:
Set the elements to the given input, then break
Default:
Set the help option to true, then break
If the user entered an empty set:
Encourage the to select at least one sort in a print statement,
Call the proper usage function and return 0 to end the program
If help was selected, print proper usage and synopsis
Set stats and make sure they are reset
Set random seed using the user input
Set up array using the size of the array and allocate memory to the array
For index 0 to the size of the array:
Set the index of the array to a masked random number
If heap is a member of the set,
Perform the heap sort function
Print out the moves and comparisons for the array given some size
If the size of the array is less than specified amount to print:
Set the printing size to the size of the array
Print out the elements, moving to the next line after 5 elements or at the end
If shell is a member of the set,
Reset the random array
Perform the shell sort function
Print out the moves and comparisons for the array given some size
If the size of the array is less than specified amount to print:
Set the printing size to the size of the array
Print out the elements, moving to the next line after 5 elements or at the end
If insert is a member of the set,
Reset the random array
Perform the insertion sort function
Print out the moves and comparisons for the array given some size
If the size of the array is less than specified amount to print:
Set the printing size to the size of the array
Print out the elements, moving to the next line after 5 elements or at the end

If quick is a member of the array,
    Reset the random array
    Perform the quick sort function
    Print out the moves and comparisons for the array given some size
    If the size of the array is less than specified amount to print:
        Set the printing size to the size of the array
    Print out the elements, moving to the next line after 5 elements or at the end
Free the space taken by the array in memory

set.h:
The Professor's code - provided for us
Include all the necessary header files
The file contains a function that makes an empty set, a function that returns true/false based on if a given value is part of a set, a function that inserts elements into an array, a function that deletes elements, a function that returns the union between two sets, a function that returns the intersection of two elements, a function that returns elements in one set but not the other, and a function that finds the complement of a given set.

stats.h:
The Professor's code - provided for us
Include necessary files
Form a structure with moves and compares as uint64_t
The file contains declarations for functions to count comparisons, moves, and functions that swap and reset elements.
Moves are the number of times that elements are moved.
Compares are the number of times elements of the array are compared directly.

stats.c:
The Professor's code - provided for us
Includes stats.h
This file contains functions to count comparisons, moves, and functions that swap and reset elements.

heap.h:
The Professor's code - provided for us
Includes necessary files
Declares the heap sort function

heap.c:
Include all necessary files

The first function finds the maximum child of the heap (or the branches if looking
    at the heap visually as a tree):
    The left child can be found using 2 times the index number of the parent
    The right child can be found by adding 1 to the left child
    If the right child is smaller than the last digit, and the value is
        greater than left in comparison (use compares to change statistics):
        The right child is the max
    Otherwise, the left child is the max
The second function fixes the heap:
    Set a variable to keep track of found (the proper place of an element) value as false
    Declare the mother and max child of a branch
    While the mother is on the left half (it is less than the last/2) and the value is not
        found (use compares to change statistics):
        If the value of the mother is smaller than the max child
            Swap the mother and max child values (use swap)
            Let the mother be the new max child value
            Find the max child of mother and last in the array to
                find the new max child
        Otherwise, it is true that we have found the value's spot
The third function will be building the heap:
    For a value father, which is last/2, as long as father is greater than
        the first value and decrementing by 1:
        Fix the heap
The last function is main heap sort function:
    We begin by resetting the statistics so we start without moves or comparisons
    Set the first value to 1 and the last value to the size of the array
    Build the heap
    For every leaf (which is the last value), as long as the left is greater than
        the first value and decrementing by 1:
        Swap the value before first with the value before
            leaf (use swap)
        Fix the heap

insert.h:
The Professor's code - provided for us
Includes necessary files
Declares the insertion sort function

insert.c:
Includes the necessary files

Start the insertion sort function:
    Reset the statistics so we start without moves or comparisons
    For i = 1, as long as i is less than n and we increment by 1:
        Declare a temporary variable j to the current iteration count
        Declare another variable to the "i"th term of the array (use moves
            to change statistics)
        While the iteration count is greater than 0 and the "i"th term is less than
            the term in the array in the index j -1 (use compares to change statistics):
            Move the value of the "j-1"th array element into "j"th array
                position (use moves to change statistics)
            Decrement j by 1
        Move the value of the temp element in the array to the "j"th array
            position (use moves to change statistics)

quick.h:
The Professor's code - provided for us
Include necessary files
Declares the quick sort function

quick.c:
Include all necessary files
Begin the partition function:
    Let a value to be one below the low value, referenced as "i" in this function
    For a value j = low, as long as it is less than high and incrementing by 1:
        If the value - 1 is less than than the value below high (use compares
            to change statistics):
            Increment i up by 1
            Swap the value below i with the value below the low (use swap)
    Swap the "i"th value of the array with the value of the array high - 1 (use swap)
    Return i + 1
Begin a sorter function:
    If the low value is smaller than the high value:
        Set the partition
        Sort the left side of the partition (from low to the value below partition)
        Sort the other side of the partition (from the value above partition to high)
Begin the main quicksort function:
    Reset the statistics so we start without moves or comparisons
    Send the array to the sorter function

shell.h:

The Professor's code - provided for us
Include all the necessary files
Declares the shell sort function

shell.c:

Include the necessary files
Add the professor's power function from discord, which returns the power of a number.
Begin the main shell sort function:
    Reset the statistics so we begin with no moves or comparisons
    Declare variables to keep track of the gap, max gap, and temporary variables
    For the max gap, as long as the gap is greater than or equal to 1 and
        decrementing the gap by 1 every round:
        Set the gap
        For the gap as long as it is less than the array size, and incrementing by 1:
            Set a temporary variable to the gap (j) and another to the gap's
                value in the array (temp) (use moves to change statistics)
            While the value of j is greater than or equal to the gap and the
                temp is smaller than j-gap in comparison:
                Move the value in the "j-gap"th index of the array
                    to the "jth" position in the array (use moves to
                    change statistics)
                Decrement j by the gap
            Move temp into the "j"th value of the array (use moves to
                change statistics)