Assignment 3 - Sorting: Putting your Affairs in order Design

Sean Carlyle

October 12 2021

1 Introduction

This is my Assignment 3 - Sorting: Putting your Affairs in order Design document. This program goal is to sort a dynamically allocated array using different sorting functions and print the results. The sorting functions I use in my program are shell sort, insertion sort, heap sort, and quick sort. The user is able to decide whether only one sort is used or multiple sorts sorting the same array. The user can also decide what seed to use (used to randomize elements in the array), the size of the array, and how many elements should be printed. The program also prints the number of moves the sorting algorithm made and the number of comparisons of the array it did. There are command line arguments that can be input into the program. -a employs all sorting functions. -e employs heap sort. -i employs insertion sort. -s employs shell sort. -q employs quick sort. -r seed allows the user to set the size of the seed (default is 13371453). -n size allows the user to set the size of the array (default is 100). -p elements allows the user to set the number of elements to be printed. -h prints out a helper which display program uses.

2 Pseudocode and explanation

2.1 sorting.c

Get the command line arguments from user.

Place the arguments in set.

In a function called make array()

Allocate memory dynamically to the array.

Create an array with random values that are bit masked by 30 bits. The array size will be determined by the user and the seed used to randomize the values.

If no sorting algorithm is called, print help message

If seed, print, or element length is in the set. Change them to user defined values

If heap is in the set.

Call heap sort and print out the sorted array along with statistics, print out as many elements as user tells you to.

If insert is in the set.

Call insertion sort and print out the sorted array along with statistics, print out as many elements as user tells you to.

If shell is in the set.

Call shell sort and print out the sorted array along with statistics, print out as many elements as user tells you to.

If quick is in the set.

Call quick sort and print out the sorted array along with statistics, print out as many elements as user tells you to.

If help is in the set

Print help message

Free memory from the array.

Return a value.

Sorting Explanation

Sorting is the c file that controls my program. Its main function is to create the array that the sorting functions will sort and call them. It also gets user input on which functions should run and how many elements the array should be. It does this through sets, with the wanted commands being inputted in the set and if statements to check if certain commands are members of the set. It also prints out the sorted array along with statistics on how efficiently it sorted the array. A design decision I made was to use enum to help remove magic numbers. Because the sets I use only store integers, I use integers to tell the program what commands to run. This creates magic numbers in my code, so I used enumeration to assign names to these numbers. These names are things like SHELL, INSERTION, etc. for each command.

2.2 insert.c

For each array index starting at 1.

Save the value at the current index (using stats function).

While the saved value at the is less than a certain indexed array value starting at the array indexed behind the current index of the array.

Move the value in the other array into the current array index (using stats function).

Make the certain indexed array go back one index.

Move the original saved array value (using stats function) into the array index once we know it is greater than all previous array values.

Insert Explanation

The function insertion sort() within insert.c accepts a stats structure (see more in stats.c), the array to be sorted, and the length of the array. Insert is quite intuitive. It basically loops through every array value and checks whether any array values before it are greater than the currently looped array value. If they are greater it moves the values up until all values behind the current indexed values are less than it. At this point once it has gone through every array value the array has been sorted. There aren't any difficult design decisions I made since this was a simple sort.

2.3 heap.c

Create a function max child() which finds the child in a heap with higher value

Do this simply by checking if the left or right value of the children of a parent value are greater (using stats function). Return the greater value.

Create a function fix heap() that puts the max heap in the correct order.

Do this by comparing the father heap value with the greatest of the children.

If the child value is greater than the father swap their values and also check the child values of the new child (who was the old father and only if they have any children).

Use stats function when swapping, or comparing

Create a function which builds the heap

Do this by calling fix heap() for the first half of array values.

Create the main function heap sort()

Build a max sorted heap using previous functions

For every value in the array

Swap the first and last values (using stats function)

Fix the heap but this time without the last value in the array

Heap Explanation

The function heap sort() within heap.c accepts a stats structure (see more in stats.c), the array to be sorted, and the length of the array. Heap basically gets the array and sorts it into a max heap. This puts the largest value on top and sorts the array values below it to be in max heap format. Max heap format is when all father heap values are larger than their children. It then makes the first value in the array the last since it should be the largest value. It fixes the heap for every value except the ones already sorted at the top of the array, and keeps doing this for every value until the array is sorted.

2.4 shell.c

Create a gap function which calculates the gap in which the sort will be comparing values by this gap.

For each calculated gap starting at the largest gap.

For the number of indexes between gap and the total number elements in the array.

While indexes are a gap away from each other aren't sorted, sort them by comparing (using stats function).

Move them if they are out of order (using stats function).

Shell Explanation

The function shell sort() within shell.c accepts a stats structure (see more in stats.c), the array to be sorted, and the length of the array. This sort is a little more complex but much like insertion sort. It basically sets a gap which the program sorts values between. For example if the gap was 6, it would sort values 6 away from each other. The gap gets smaller and smaller until it is sorting values one away from each other like insertion. Once this is done the array has been sorted. This sort can be useful because it may get some big values sorted out early before going one by one, since if a big value was at the first index it would have to move it through every single array index in insertion sort.

2.5 quick.c

Partition function()

Places elements less than pivot on left side of array, places elements greater than pivot on right side of array.

Sort each element to each side of the pivot (using stats functions)

Once the two sides are sorted, swap the pivot with the pointer (using stats functions)

Return the index that divides teh partition

Quick sorter()

Keep calling quick sorter() on each side of partition until the partitions are one wide

Call quick sorter() on right side of partition

Call quick sorter() on left side of partition

Quick sort() function

Simply call quick sorter() which takes an array, a low integer, and a high integer

Quick Explanation

The function quick sort() accepts a stats structure (see more in stats.c), the array to be sorted, and the length of the array. Quick sort selects a pivot and sorts the array into sides, one side less than the pivot one side greater than the pivot. Then it breaks those pivots into smaller pivots until the array is sorted.

2.6 stats.c

cmp () function accepts stats struct, and two uint32t values.

Increments compare by 1

Returns -1 if first value is less than second, returns 1 if first value is greater than second, and returns 0 if they are equal.

move () function accepts stats struct, and one uint32t value

Increments moves by 1 and returns the uint32t value

swap () function accepts stats struct and two uint32t values

Increments moves by 3

Stores first value in temp variable

Changes first value to second value

Changes second value to temp value

reset() function accept stats

Changes moves and compares back to 0

Stats Explanation

Stats is the function that tracks the amount of moves and compares the sorts do. A Stats structure has two variables moves and compares. The stats.c increments these variables while also doing practical things such as swapping/comparing values. Swapping is counted as three moves, since a temp variable must be used.

3 User Input Errors

The only design decision I made with user input errors is making the help menu pop up whenever the user inputs a command line argument that won't print anything. If the user doesn't input anything or doesn't input any sorting algorithms, the help menu will show up.