Sorting Algorithms

Mignot Mesele

January 2, 2024

Purpose

The purpose of this program is to sort an array of pseudo randomly generated numbers using 4 sorting algorithms and giving us the results of the algorithms based on what the user inputs in the command argument.The command options are controlled by through the use of sets to organize and return the correct output. The results consists of the number of elements, moves, and comparisons that the sorting algorithm go through to have a sorted array.

How to Use the Program

To use the program you'd simply type in the command argument, ./sorting along with a command option. This would specify what sorting algorithms to be implemented. Options:

- -a: runs all sorting algorithms, resulting in printing all the
- -i: run Insertion Sort
- -s: run Shell sort
- -b: run Batcher Sort
- -q: run Quick Sort
- -r seed: Sets the random seed to what you put for the seed. The default is 13371453.
- -n size:Sets the array size to what is put for size in argument. The default is 100.
- -p elements: Prints out "element" number of elements from the array. The default is 100. -p 0 disables printing elements.
- -h: displays help message

You need to use the Makefile, which enables you to run the commands, make filename and make clean. The make command converts the c files into executable files to be able to use sorting.c. The make clean command removes all files created by make, clearing any issues that the object and executable files could cause.

Program Design

Data Structures

Data Structures:

- arrays-This is used to hold randomly generated numbers to be sorted using the algorithms.
- sets-This is being used to track what command options are being inputted.

Algorithms

This section should include an explanation of your series, and how it approximates to the value you are trying to find.

```
Insertion sort algorithm
    loop from k = 1 to element size of array
        assign k to another variable like x
        temp varable for the value of the element at k
        loop as long as x > 0 and temp variable < value of the element at x-1
            index x of array is assigned the index x-1 of array
            x decremented by 1
        index x of array is assigned the temp variable
```

```
Shell sort algorithm
    loop through the elements in gaps
        loop from element in gaps to size of given array and assign a counter
            assign counter to variable x
            assign the value of the index[counter] of the array to temp variable
            loop as long as x >= element of gap and temp variable < value of index[x-element of gap]
                index x of array is assigned value of index[x-element of gap]
                x decremented by element of gap
            index x of array is assigned value of temp variable
```

```
Batcher's Odd-Even Merge sort algorithm:
    if array size is zero
        return nothing
    assign the array size to a variable x
    assign the bit size of x to a variable y
    assign the result of 1 logical shifted left by y-1 to z
    loop as long as z > 0
        assign the result of 1 logical shifted left by y-1 to a
        assign 0 to a variable b
        assign z to a variable c
        loop as long as c>0
            loop from 0 to x-c tracking a counter
                if counter & z is equal to b
                    call function comparing array[counter]>array[counter+c] and swaping
            c assigned a-z
            a logical shifted right by 1
            b assigned z
```

```
Quick sort algorithm:
partition algorithm
    assign low-1 to a variable x
    loop in range from low to high assigning a counter
        if array[counter-1] < array[high-1]
            x incremented by 1
            array[x-1] = array[counter-1]
            array[counter-1]= array[x-1]
    array[x]=array[high-1]
    array[high-1]=array[x]
    return x+1
quick sort algorithm
```

```
if low < high
    assign the function call partition to a variable like x
    recurse quick sorter(array,low, and x-1)
    recurse quick sorter(array, x+1, high)
function call quick sorter(array, 1, size of array)
```

Function Descriptions

For each function in your program, you will need to explain your thought process. This means doing the following

- Set set[underscore]empty(void)-This function's purpose is to give an empty set, in which every bit is zero. This function has no inputs.
- Set set[underscore]universal(void)-This function's purpose is to give the universal set, which consists of all the bits being 1. This function has not inputs.
- Set set[underscore]insert(Set s, int x)-This function's inputs are a set and an integer, with the purpose of inserting the integer x into the set s, using bit wise OR and logical shifting, to return a set with x in it.
- Set set[underscore]remove(Set s, int x)-This function's inputs are a set and an integer, with the purpose of removing the integer x into the set s, using bit wise AND, negation, and logical shifting, to return a set without x.
- bool set[underscore]member(set s, int x)-This function's inputs are a set and an integer, with the purpose of indicating if the integer x is in the set s, using bit wise AND and logical shifting, to return true or false.
- Set set[underscore]union(Set s, Set t)-This function's inputs are two sets, with the purpose of creating a union between the Set s and t, using bit wise OR, to return a union of the two sets.
- Set set[underscore]intersect(Set s, Set t)-This function's inputs are two sets, with the purpose of creating an intersection between the Set s and t, using bit wise AND, to return an intersection of the two sets.
- Set set[underscore]difference(Set s, Set t)-This function's inputs are two sets, with the purpose of creating a set of the difference between the Set s and t, using bit wise AND and the negation of t, to return the set of elements in s that aren't in t.
- Set set[underscore]complement(Set s)-This function's input is a set, with the purpose of getting the complement of Set s, using bit wise NOT, to return the complement of Set s.
- insertion[underscore]sort(array a)-This function's input is an array, with the purpose of sorting it by checking every element with every other element and placing them in the correct position, returning a sorted array.
- shell[underscore]sort(array a)-This function's input is an array, with the purpose of sorting by ordering pairs of elements far from each other and reducing the gap between them, returning a sorted array.
- partition(array a, int low, int high)-This function's inputs are an array and two integers being a high and low number of a pair. Its purpose is to arrange elements on the left and right side of a value and returns an index.
- quick[underscore]sort(array a)-The input is an array and it uses the array to call on the quick sorter function with the array as its input, returning the function's output.
- quick[underscore]sorter(array a, int low, int high)-This function's inputs are an array and two integers. This function checks if low is less than high to assign the output of a function call to partition to a variable, like x, and calls on itself twice with the low and x-1 going in one and high and x+1 in the other.

Results

The code ran all 5 sorting algorithms perfectly, giving sorted arrays with their statistics. Looking at the graph, I noticed that both insertion and batcher sort took less moves as the lines grew exponentially while shell and heap took more. In terms of compares, the quick sort function took the least amount of comapares by a big difference, while both, insertion and batcher, and, heap and shell, still took around the same compares to sort. The functions performed better with less elements as some needed a lot of moves and compares to perform. I concluded that insertion and batcher takes the least amount of moves than the rest, making it the most efficient. I learned that there are many ways to sort arrays but the most important difference is their efficiency.

Numeric results

```
Insertion Sort, 15 elements, 82 moves, 65 compares
     34732749     42067670     54998264    102476060    104268822
    134750049    182960600    538219612    629948093    783585680
    954916333    966879077    989854347    994582085   1072766566
Heap Sort, 15 elements, 144 moves, 70 compares
     34732749     42067670     54998264    102476060    104268822
    134750049    182960600    538219612    629948093    783585680
    954916333    966879077    989854347    994582085   1072766566
Shell Sort, 15 elements, 170 moves, 87 compares
     34732749     42067670     54998264    102476060    104268822
    134750049    182960600    538219612    629948093    783585680
    954916333    966879077    989854347    994582085   1072766566
Quick Sort, 15 elements, 135 moves, 51 compares
     34732749     42067670     54998264    102476060    104268822
    134750049    182960600    538219612    629948093    783585680
    954916333    966879077    989854347    994582085   1072766566
Batcher Sort, 15 elements, 90 moves, 59 compares
     34732749     42067670     54998264    102476060    104268822
    134750049    182960600    538219612    629948093    783585680
    954916333    966879077    989854347    994582085   1072766566
```

Figure 1: Screenshot of the program running.

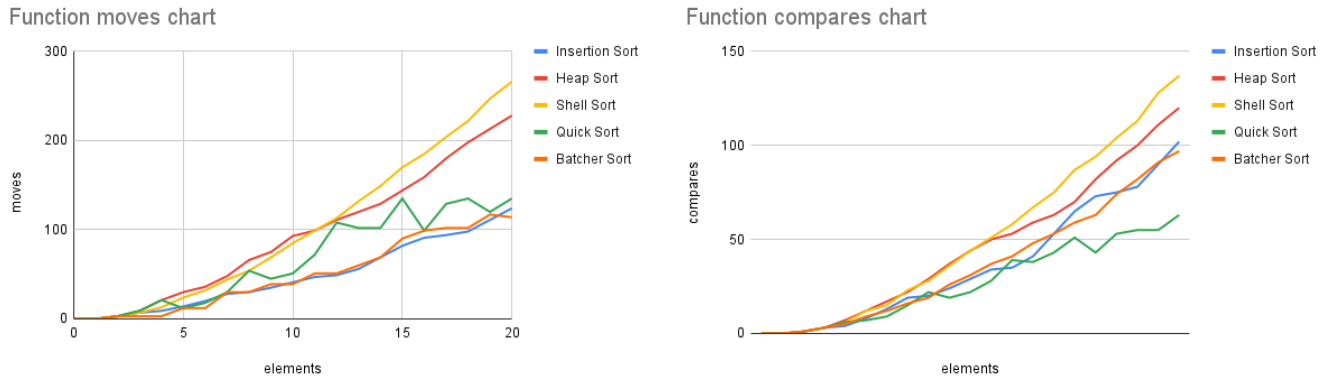

Figure 2: Screenshot of the graphs of both move and compares growth in sorting functions.

Error Handling

The only error i ran into was segmentation fault in batcher and was fixed by changing an == to != for getting the bit length.