Assignment 3 – Sets and Sorting

Rahul Amudhasagaran

CSE 13S – Spring 2023

Purpose

This program serves to organize integers into a Set data structure and then sort them least to greatest.

How to Use the Program

To use this program, the user must run the program and then select an option to run the program with. The options of programs are listed below:

```
Options:
-a: runs all sorting algorithms
-i: runs Insertion Sort
-s: runs Shell Sort
-h: runs Heap Sort
-q: runs Quicksort
-v: runs Batcher Sort
-r SEED: Sets the seed of the psuedorandom array to SEED
-n SIZE: Sets the size of the psuedorandom array to SIZE
-p ELEMENTS: Prints out ELEMENT number of elements from array (default =  100)
-H: Prints out help manual
```

All of these options can be run at once. If ELEMENTS is greater than SIZE, than all of the elements of the array will be printed. Out of the five sorting algorithms listed, only four will be implemented in the program.

These descriptions were given by the Lab Document[1].

Program Design

There are three parts to this program: the part that handles inputs, the part that creates the pseudo random array, and the multitude of functions/tests that give out the result. The input handler is in the file sorting.c while the sorting algorithms are in each of their respected files. The code that initializes the sets is in the file set.c.

Data Structures

I used two data structures in this code. The first one is a custom data structure Set that stores 8 options in the form of bits. The second data structure is a pseudo random array that needs to be sorted.

[1]Assignment 3 Sets and Sorting by Prof. Darrell Long edited by Kerry Veenstra and Jess Srinivas[1]

Algorithms

For this assignment, I will implement all the sorting algorithms.

Insertion Sort Pseudo Code:

```
insertion_sort (pointer to array A)
    for (int k = 1; k < 100; ++k)
    int j = k
    int temp = arr [k]
    while j > 0 && temp < arr [j - 1]
        arr [j] = arr [j - 1]
        j--
    arr [j] = temp
```

Shell Sort Pseudo Code:

```
void shell_sort (pointer to array)
    //pull gaps from header file
    for (int gap = 0; gap < 142; ++gap)
        for (int i = gaps [gap]; i < 100; ++i)
            int j = i
            int temp = arr [i]
            while j >= gap && temp < arr [j - gap]
                arr [j] = arr [j - gap]
                j -= gap
            arr [j] = temp
```

Heap Sort Pseudo Code:

```
void heap_sort (pointer to array)
    int first; int last;
    int max_child (pointer to array, int first, int last)
    max_child (arr, first, last)
        left = 2 * first
        right = left + 1
        if right <= last && arr [right - 1] > arr [left - 1]
            return right
        return left

    void fix_heap (pointer to array, int first, int last):
        found = 0
        parent = first
        great = max_child (arr, parent, last)
        while parent <= last / 2 && !found
            if arr [parent - 1] < arr [parent - 1]
                arr [parent - 1] = arr [great - 1]
                arr [great - 1] = arr [parent - 1]
                parent = great
                great = max_child(arr, parent, last)
            else
                found = 1

    void build_heap(pointer to array, int first, int last)
        for (int t = last / 2; t < first - 1; --i)
            fix_heap (arr, t, last)
    void heap_sort(pointer to array)
        first = 1
        last = 100
        build_heap (arr, first, last)
        for (int l = last; l < first; --l)
            arr [first - 1] = arr [l - 1]
            arr [l - 1] = arr [first - 1]
            fix_heap (arr, first, l - 1)
```

Quicksort Pseudo Code:

```
void quicksort (pointer to array)
    int partition (ar, int lo, int hi):
        int i = lo - 1
        for (int j = lo; j < hi; j++)
            if arr [j - 1] < arr [hi - 1]
                i += 1
            arr [i - 1] = arr [j - 1]
            arr [j - 1] = arr [i - 1]
    arr [i] = arr [hi - 1]
    arr [hi - 1] = arr [i]
    return i + 1

    void quick_sorter (pointer to array, int lo, int hi)
        if lo < hi
            int p = partition (arr, lo, hi)
            quick_sorter (arr, lo, p - 1)
            quick_sorter (arr, p + 1, hi)

    void quick_sort(pointer to array):
        quick_sorter (arr, 1, 100)
```

Batcher's Odd Even Merge Sort Pseudo Code:

```
void batcher_sort (pointer to array)
    void comparator(pointer to array, pointer x, pointer y)
        if arr [x] > arr [y]
            arr [x] = arr [y]
            arr [y] = arr [x]

    void batcher_sort(pointer to array)
        if !len:
            return
        n = len
        t = n.bit_length()
        p = 1 << (t - 1)
        while p > 0:
            q = 1 << (t - 1)
            r = 0
            d = p
            while d > 0:
                for i in range(0, n - d):
                    if (i & p) == r:
                        comparator(A, i, i + d)
                d = q - p
                q >>= 1
                r = p
            p >>= 1

    int bit_length (b)
        while (b)
            c++
            b >>= 1
        return c
```

Function Descriptions

There are many functions used in this program. I will sort them out based on which file they are located in.
set.c functions

```
set_empty ()
Inputs: None
Outputs: zero set
Purpose: This function generates a set full of zeros.

set_universal ()
Inputs: None
Outputs: 1s set
Purpose: This function generates a set full of ones.

set_member (int x, Set s)
Inputs: int x (0-7), Set s
Outputs: 1 or 0
Purpose: This function checks if an integer (0-7) is part of Set s.

set_insert (int x, Set s)
Inputs: int x (0-7), Set s
Outputs: new Set with member inserted
Purpose: This function adds the member x into Set s.

set_remove (int x, Set s)
Inputs: int x (0-7), Set s
Outputs: new Set with member removed
Purpose: This function removes the member x from Set s.

set_union (Set s1, Set s2)
Inputs: Set s1, Set s2
Outputs: s1 | s2
Purpose: This function returns the union between Set s1 and Set s2.

set_intersect (Set s1, Set s2)
Inputs: Set s1, Set s2
Outputs: s1 & s2
Purpose: This function returns the intersection bewteen Set s1 and Set s2.

set_difference (Set s1, Set s2)
```

```
Inputs: Set s1, Set s2
Outputs: s1 - s2
Purpose: This function returns the difference bewteen Set s1 and Set s2.

set_complement (Set s)
Inputs: Set s
Outputs: !s
Purpose: This function returns the complement of Set s1.
```

insert.c

```
insertion_sort (pointer to array)
Inputs: pointer to array
Outputs: None
Purpose: This function sorts the array usins Insertion Sort
```

shell.c (gaps.c)

```
shell_sort (pointer to array)
Inputs: pointer to array
Outputs: None
Purpose: This function sorts the array using Shell Sort
```

heap.c

```
max_child (pointer to array, first, last)
Inputs: pointer to array, int first, int last
Outputs: returns child that is greater
Purpose: This function compares the children of the node and chooses the greater one.

fix_heap (pointer to array, first, last)
Inputs: pointer to array, int first, int last
Outputs: None
Purpose: This function reorders the heap to be in proper order.

build_heap (pointer to array, first last)
Inputs: pointer to array, int first, int last\
Outputs: None
Purpose: This function initializes the heap.

heap_sort (pointer to array)
Inputs: pointer to array
Outputs: None
Purpose: This function sorts the array using Heap Sort
```

quick.c

```
partition (pointer to array, lo, hi)
Inputs: pointer to array, int lo, int hi
Outputs: index of partition
Purpose: This function finds an index to partition the array

quick_sorter (pointer to array, lo, hi)
Inputs: pointer to array, int lo, int hi
```

```
Outputs: None
Purpose: This function recursively runs the partition function

quick_sort (pointer to array)
Inputs: pointer to array
Outputs: None
Purpose: This function gives the bounds of the quick_sorter function.
```

batcher.c

```
comparator (pointer to array, x, y)
Inputs: pointer to array, int x, int y
Outputs: None
Purpose: This function swaps the values at indexes x and y.

batcher_sort (pointer to array)
Inputs: pointer to array
Outputs: None
Purpose: This function sorts the array using Batcher's Odd-Even Merge Sort.

bit_length (int b)
Inputs: uint32_t b
Outputs: None
Purpose: This function finds the bit length of the input b.
```

sorting.c

```
main ()
Inputs: int argc, char *argv []
Outputs: exit function
Purpose: This function handles multiple parts of the code.

Pseudo Code:
enum Values {i, s, h, q, b}
Stats stats
initialize seed, size, elements
Set st = set_empty

while (character in OPTIONS)
switch (ch)
case a: st = set_universal
case i: set_insert (st, i)
case s: set_insert (st, s)
case h: set_insert (st, s)
case q: set_insert (st, q)
case b: set_insert (st, b)
case r: seed = optarg
case n: seed = optarg
case p: elements = optarg
case H: prints help

if elements is 0 exit

srand (seed)
```

```
malloc (arr)
arr [i] & 0x3FFFFFFF (bit-mask)

if sorting algorithm
run sorting algorithm
printstats (sorting algorithm)
reset stats
print array

free ()
exit
```

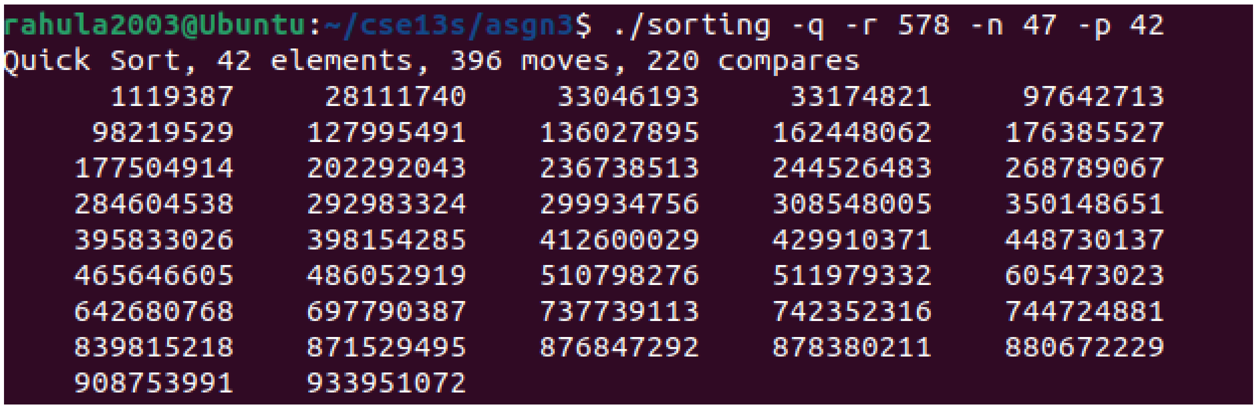
Results

The code works perfectly according to the directions. All of the test inputs worked for the program. There are a number of places where I could have improved upon in this code. I definitely did not code in the most optimized way, nor was my code very organized either. I think putting more helper functions into the main would have helped me solve both problems, which I hope to do more in the future.

Regarding the graphs of these sorting algorithms, there were several things to note. Most importantly, the algorithm that stood out the most was the Quick Sort. The Quick Sort had the least moves and comparisons out of all the functions, but it was also the most unstable, shown by the fluctuations shown at the end of the graphs. This means at x values much much greater, Quick Sort could potentially run worse than the other sorting algorithms. Other than that, the results were pretty self explanatory. Insertion Sort turned out to have the greatest amount of Moves and Comparisons, which would make sense given that it is an O (n squared) time complexity algorithm. Shell Sort did a little bit better but was beaten by both Batcher Sort and Heap Sort, both of which have O (logn) time complexities.

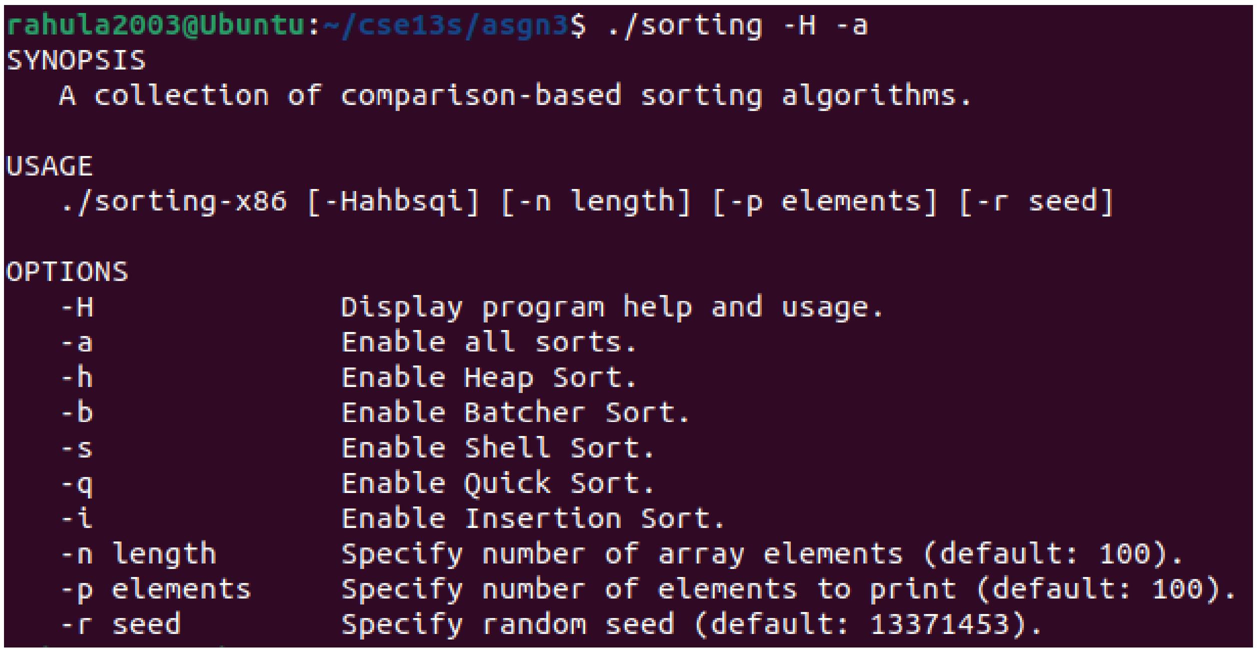
Error Handling

This code handles errors relatively well. However, there are some edge cases that could invoke an error. For example, putting some random stuff into seed, size, and elements values could mess up the code.



```
rahula2003@Ubuntu:~/cse13s/asgn3$ ./sorting -q -r 578 -n 47 -p 42
Quick Sort, 42 elements, 396 moves, 220 compares
     1119387     28111740     33046193     33174821     97642713
    98219529    127995491    136027895    162448062    176385527
   177504914    202292043    236738513    244526483    268789067
   284604538    292983324    299934756    308548005    350148651
   395833026    398154285    412600029    429910371    448730137
   465646605    486052919    510798276    511979332    605473023
   642680768    697790387    737739113    742352316    744724881
   839815218    871529495    876847292    878380211    880672229
   908753991    933951072
```

Figure 1: Screenshot of -q option running with other suboptions

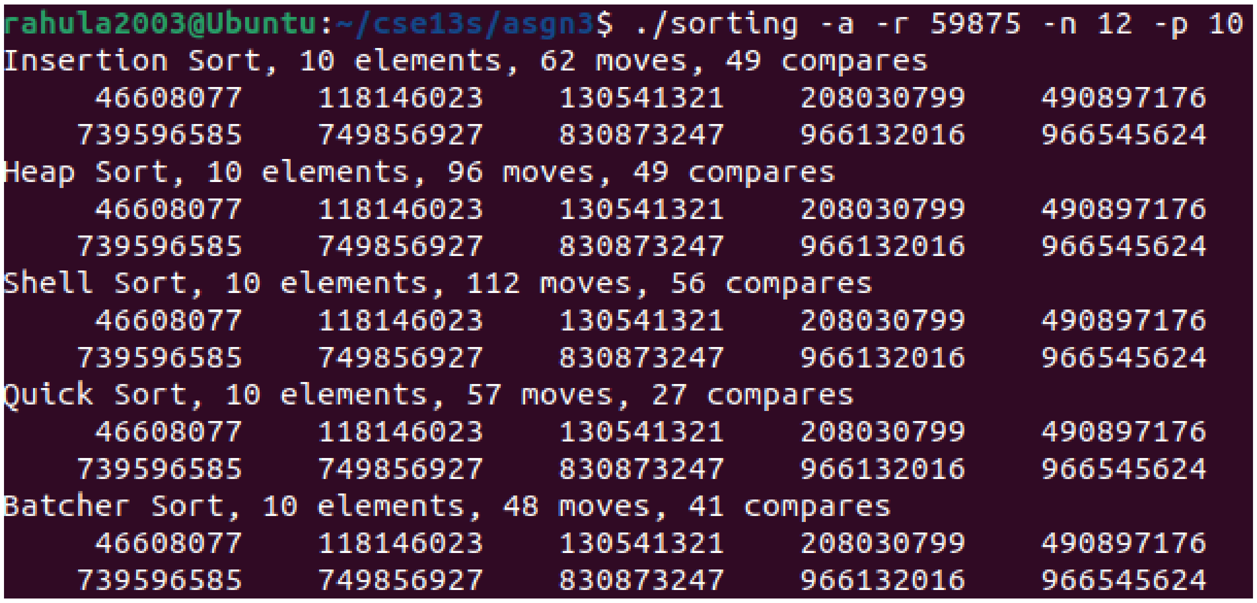


```
rahula2003@Ubuntu:~/cse13s/asgn3$ ./sorting -H -a
SYNOPSIS
   A collection of comparison-based sorting algorithms.

USAGE
   ./sorting-x86 [-Hahbsqi] [-n length] [-p elements] [-r seed]

OPTIONS
   -H              Display program help and usage.
   -a              Enable all sorts.
   -h              Enable Heap Sort.
   -b              Enable Batcher Sort.
   -s              Enable Shell Sort.
   -q              Enable Quick Sort.
   -i              Enable Insertion Sort.
   -n length       Specify number of array elements (default: 100).
   -p elements     Specify number of elements to print (default: 100).
   -r seed         Specify random seed (default: 13371453).
```

Figure 2: Screenshot of help running canceling out -a

References

[1] Darrell Long, Kerry Veenstra, Jess Srinivas. Assignment 3 sets and sorting. https://git.ucsc.edu/cse13s/spring2023/resources/-/blob/master/asgn3/asgn3.pdf, 2023. [Online; accessed 6-May-2023].



```
rahula2003@Ubuntu:~/cse13s/asgn3$ ./sorting -a -r 59875 -n 12 -p 10
Insertion Sort, 10 elements, 62 moves, 49 compares
     46608077     118146023     130541321     208030799     490897176
    739596585     749856927     830873247     966132016     966545624
Heap Sort, 10 elements, 96 moves, 49 compares
     46608077     118146023     130541321     208030799     490897176
    739596585     749856927     830873247     966132016     966545624
Shell Sort, 10 elements, 112 moves, 56 compares
     46608077     118146023     130541321     208030799     490897176
    739596585     749856927     830873247     966132016     966545624
Quick Sort, 10 elements, 57 moves, 27 compares
     46608077     118146023     130541321     208030799     490897176
    739596585     749856927     830873247     966132016     966545624
Batcher Sort, 10 elements, 48 moves, 41 compares
     46608077     118146023     130541321     208030799     490897176
    739596585     749856927     830873247     966132016     966545624
```

Figure 3: Screenshot of -a option running with sub options

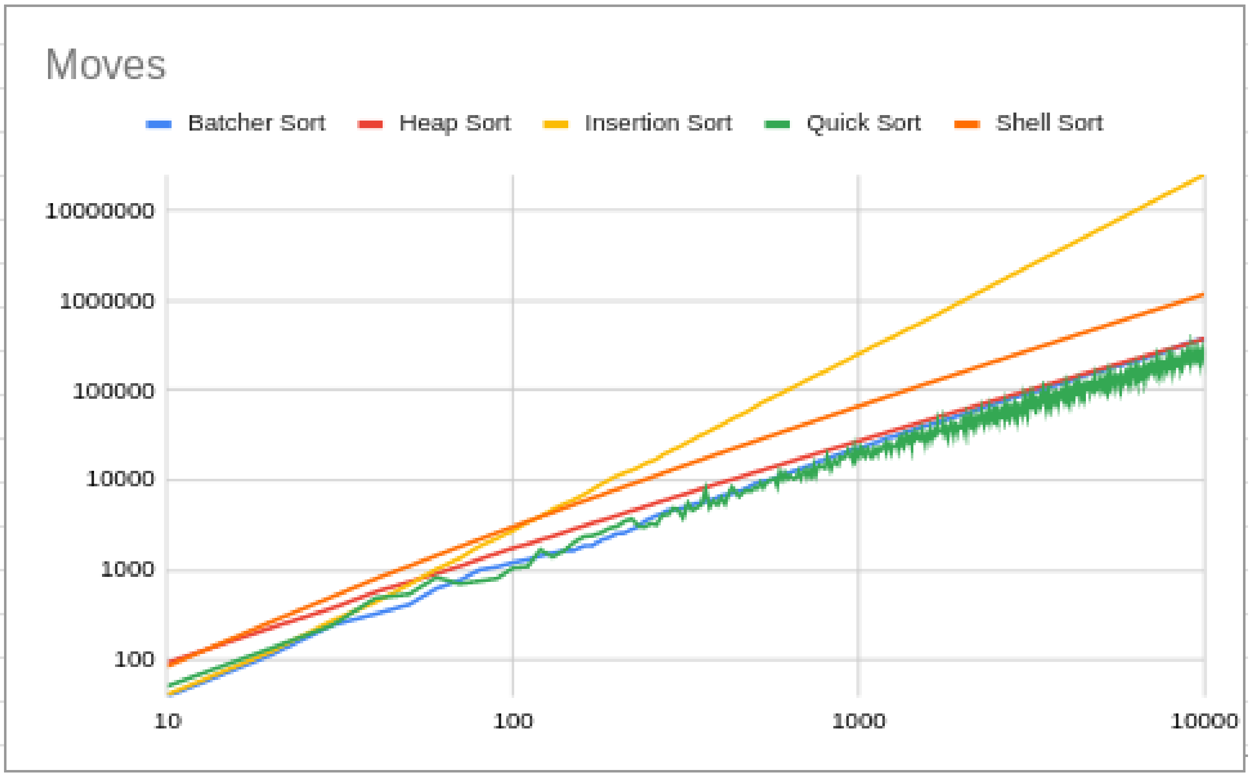


Figure 4: Screenshot of Moves graph

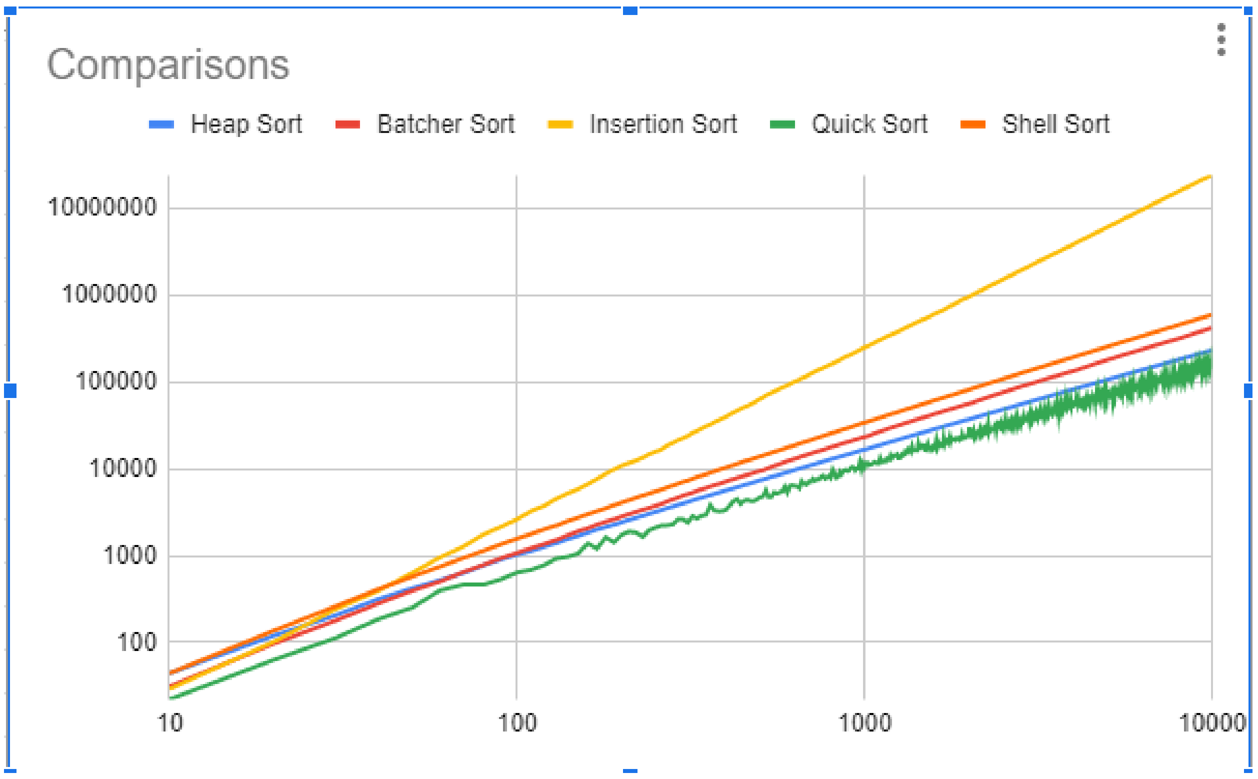


Figure 5: Screenshot of Comparison graph