Assignment 3
Sets and Sorting

By Jiancheng (Jason) Xiong
CSE 13S Spring 2023

Due May 10th 2023, 11:59 PM

1 Purpose

The purpose of this assignment is to create a program that uses sets in order to store called command line arguments. We will then use a program to call different commands, including;

- Running all sorts
- Insertion sort
- Batcher's merge sort
- Quick sort
- Shell sort
- A command to choose a custom random seed, default is 13371453
- A command to allow different number of elements, default is 100
- A command to choose how many elements to print for the final sorted arrays, default is 100
- A command to print a help message

After each function, a print message with the name of the sort, the number of elements, and the number of comparisons and merges will be printed, with the help of the file stats.c.

2 Using the Program

In order to use the program, compile and run the file sorting.c. In order to run the program, run ./sorting and specify the commands in which you want to run. The available options are;

-a; running all sorting algorithms
-i; runs insertion sort
-b; runs Batcher's merge sort
-q; runs quick sort
-s; runs shell sort
-r seed; sets the srandom seed to seed, default is 13371453
-n size; sets the array size to size, default is 100

-p size; sets the number of printed elements to size, default is 100
-h; displays a help message detailing program usage.

3 Program Design

The main program will be in the file sorting.c, while each of the separate sorting files will be in;

- Insert.c and header file insert.h for insertion sort
- Batcher.c and header file batcher.h for batcher sort
- Quick.c and header file quick.h for quick sort
- Shell.c and head file shell.h for shell sort
- Gaps.h for an array of numbers to use for the gaps in shell sort
- Set.c and header file set.h for an implementation of the set struct to be used for storing of command line options
- Main file sorting.c to be the test file that runs all other functions
- Stats.c and stats.h to keep track of the number of comparisons and swaps each function uses

4 Data Structures

The main data structure is the struct set, which implements a variety of helpful functions that allow us to create and manipulate sets. The various functions allow us to change specific elements in a set, as well as check if a certain element is indeed a member of a set. By giving each argument a specific placement in a set, we can tell the program that they were called if we replace the location in that set with a 1, to indicate membership.

Another main structure are arrays, which are made with random 30 bit long numbers and passed into the sorting algorithms to be sorted.

5 Algorithms

The main function;

creates an empty set 8 elements long, with all numbers set to 0.
Each argument is naturally assigned to a certain index within that set
Parses through the command line with getopt() from stdlib.h, and if an element is present its index is set to 1 to indicate that it has been asked for.

After parsing the entire argument, the set is iterated through, and for elements that have a 1, indicating membership, the corresponding algorithm or function is run.

For the sorting algorithms;

Insertion sort

1. Start by iterating each element in the unsorted array, starting with the second element
2. For each element in the loop, compare it to the elements before it, starting with the previous element
3. If the current element is smaller than the previous element, swap their positions
4. Repeat step 3 until the current element is larger than the previous element
5. Repeat for all unsorted elements

Batcher sort

1. First divide the unsorted array into 2 parts, and recursively sort each part
2. Once the two halves are sorted, merge them using;
    a. Compare the elements at the same indice of the two halves
    b. If the element in the second half is smaller than the first, swap their positions
    c. Repeat the comparison for every element in the two halves

3. Repeat the merges recursively until the entire array is remerged, until which it is sorted

Quick sort

1. Choose a pivot element, which can be done randomly or simply selecting the middle element
2. Partition the array into two subarrays: one containing all the elements smaller than the pivot, and the other will all greater elements
3. Recursively apply the quicksort algorithm to the subarrays

Shell sort

1. Start with a gap value of length of array/2

2. Divide the array into smaller subarrays of the length gap
3. Repeat step 2 for values of len/4, len/8, etc.. until gap is equal to 1
4. Perform a final insertion sort on the array

For the set implementation:
All arrays mentioned are of length 8
Set_empty(void):
Create an empty array of 0s

Set_universal(void):
Create an array of 1s

Set_insert(set, x):
Takes the input set and logic ors it with 0x01 shifted left x times

Set_remove(set,x):
Takes the input set and logic ands it with not 0x01 shifted left x times

Set_member(set, x):
Returns true if (set & 0x01 shifted left x times) is a 1, otherwise returns false

Set_union(set1, set2):
Returns all elements in set1 or set2

Set_intersect(set1,set2):
Returns all elements in both set1 and set2

Set_difference(set1,set2):
Returns all elements in set1 but not in set2

Set_complement(set):
Returns the opposite of set (not set)

6 Function Descriptions

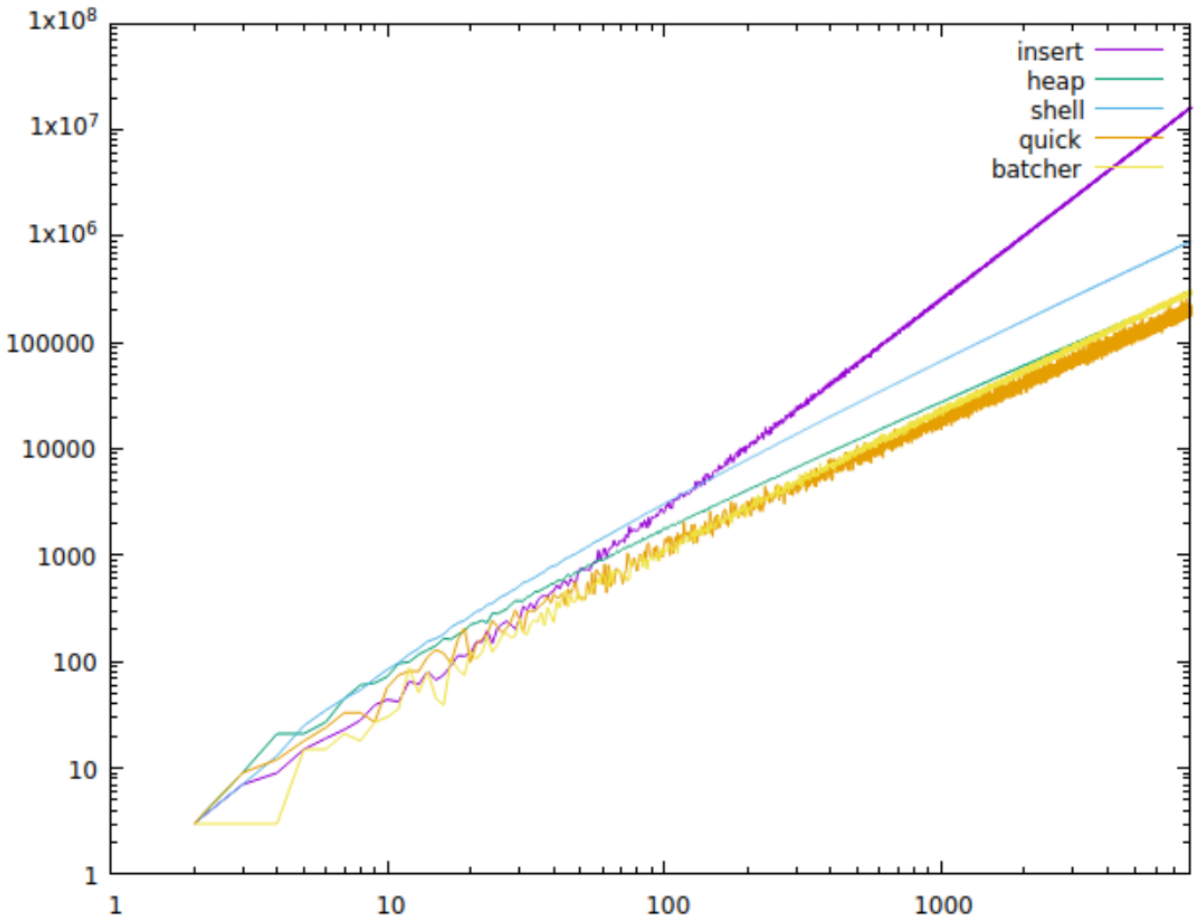
Main function: takes its arguments from the command line, using getopt to know what sorting functions to run. Takes one (command line) input and outputs the sorted arrays requested along with the stats of the sorts

All Sorting Functions: takes the arguments (stats, input array, length of array). The sorting functions then sort the input array and output nothing.

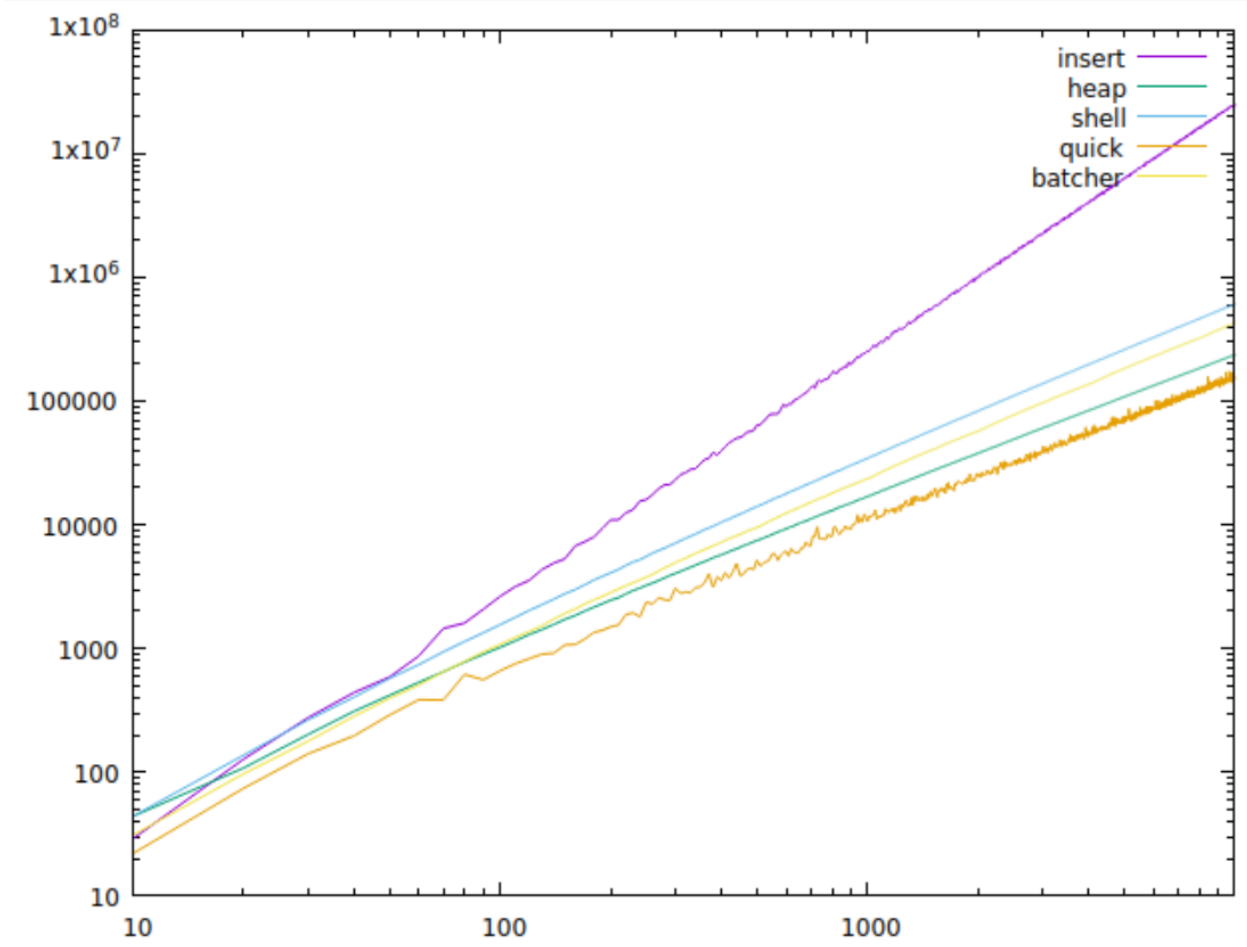
7 Results

Through studying the different sorting algorithms, I've learned that each algorithm has its own strengths and weaknesses, which can depend on factors such as input size and initial order of the elements.

1. Insertion sort: insertion sort performs well when the input data is already partially sorted or when new elements are added to an already sorted list. It is efficient for small lists. However, it performs poorly when dealing with large input sizes, as it has a large time complexity of $O(n^2)$.
2. Heap sort: Heap sort performs well for large input sizes and has a time complexity of O(n log n). It however tends to perform slowly for small input sizes.
3. Shell sort: Shell sort performs well for large input sizes and if the data is partially sorted. It also handles non-uniform data distribution better than other sorting algorithms. It however performs badly when the input sizes are small and if the data is already sorted, as it will make unnecessary passes.
4. Quick sort: quick sort is efficient and performs well for large input sizes. It has an average time complexity of O(n log n) but in its worst case can be as slow as $O(n^2)$.
5. Batcher sort: Batcher sort performs well when parallel processing is possible, as it was designed to be capable of being executed concurrently. It works well for large input sizes, but is slower for small input sizes.



The above graph is a graphical representation of the number of moves each sorting algorithm took to finish sorting an array of various random numbers. We can see that insert and batcher are very efficient at small numbers of input, but insert quickly becomes the slowest and least efficient sorting method as the number of inputs passes 100. Quick and heap on the other hand are great for large numbers of inputs, taking less moves than the other algorithms. Shell and heap seem to excel at mid-range number of inputs, being efficient at 30-40 inputs.



The above graph is another representation of the efficiencies of the sorting algorithms, this time of comparisons rather than moves. It follows a similar trend of the previous graph, with insert being efficient at smaller sample sizes but quickly becoming worse than the other methods. Quick is consistently the sorting algorithm with the lowest comparisons though, maintaining a consistent lead over the other algorithms.

The program running:

```
jasonx@cse13s:~/cse13s/asgn3$ ./sorting -a -n 100 -p 10
Insertion Sort, 100 elements, 2741 moves, 2638 compares
      8032304     34732749     42067670     54998264     56499902
     57831606     62698132     73647806     75442881    102476060
Heap Sort, 100 elements, 1755 moves, 1029 compares
      8032304     34732749     42067670     54998264     56499902
     57831606     62698132     73647806     75442881    102476060
Shell Sort, 100 elements, 3025 moves, 1575 compares
      8032304     34732749     42067670     54998264     56499902
     57831606     62698132     73647806     75442881    102476060
Quick Sort, 100 elements, 1053 moves, 640 compares
      8032304     34732749     42067670     54998264     56499902
     57831606     62698132     73647806     75442881    102476060
Batcher Sort, 100 elements, 1209 moves, 1077 compares
      8032304     34732749     42067670     54998264     56499902
     57831606     62698132     73647806     75442881    102476060
```

8 References

"Official Gnuplot Documentation." *Gnuplot Documentation*, Feb. 2022, www.gnuplot.info/documentation.html.

"Getopt() Function in C to Parse Command Line Arguments." *GeeksforGeeks*, 10 Sept. 2018, www.geeksforgeeks.org/getopt-function-in-c-to-parse-command-line-arguments/.