Assignment 3 Report

By Jason Zhang
Prof. Veenstra
CSE 13S

Due May 10, 2023

Purpose

This program uses set functions and sorting algorithms to order arrays of integers. The sorting algorithms include the following: Insertion Sort, Shell Sort, Quicksort, Batcher's Odd-Even Merge Sort. These functions will then be accessed through one .c file sorting.c which takes in command line options and implements a pseudo-random element array test harness for each of the sorting algorithms.

How to Run

Run the program from command line using a Makefile:

```
make
```

The program can then be ran directly:

```
./sorting
```

Or with an input txt file:

```
./sorting < input.txt > output.txt
```

Possible command line options for sorting.c include:

```
./sorting -i -s -q -b -r -n -p -H
```

Of which,

- -a : Employs all four sorting algorithms implemented.
- -i : Enables Insertion Sort.
- -s : Enables Shell Sort
- -q : Enables Quicksort.
- -b : Enables Batcher Sort.
- -r seed : Set the random seed to seed. (default = 13371453)
- -n size : Set the array size to size. (default = 100)
- -p elements : Print out the number of elements from the array. (default = 100)
- -H : Prints program usage.

Program Design

The source code for the sorting program is found in 14 files. Of which, `batcher.c, shell.c, quick.c, insert.c` contain the sorting functions. These functions are mainly composed of loops that index through the input algorithms. The `set.c` file contains a series of set functions that use bit manipulation operations to perform set operations such as membership, union, intersection, and negation. `sorting.c` contains a `main()` function that the program runs with. In addition, the program uses existing libraries from C such as `stdio.h` and `unistd.h`.

Data Structures

The program uses `int array[]` in C to store data, as well as `int` for indexing into the arrays. Each sorting algorithm will take in an integer array and return a sorted array.

Algorithms

Each sorting algorithm utilizes integers to track the index of given integer arrays. In addition, loops are also used, typically for-loops with an integer that ranges from 0 to len(array)-1. The arrays will be processed element by element until it is sorted.

Function Descriptions

The `main()` function in `sorting.c` works as below:

- input: int argc, char *argv[]
- output: int 0, prints out sorting results as specified in arguments
- The function utilizes switch cases combined with optarg() in order to take in and process arguments given when running sorting.c
- For sorting algorithms, a `Set (uint8_t)` variable is used. the right most 5 bits of the 8-bit int `Set` each correlate to one of the five sorting algorithms. The program uses functions provided in `set.h` and `set.c` to perform bit manipulation in order to edit and detect bits to track which sorting algorithms to use. The left three bits are ignored and thus can be any value. The "all" option, `-a`, signs the Set variable to be the universal set in order to enable all sorting algorithms.

Each sorting algorithm performs steps below:

- **Insertion Sort**
    - input: int array[]
    - output: sorts given array
    -
```
def insertion_sort(A: list):
    for i in range(1, len(A)):
        j = i
        temp = A[i]
        while j > 0 and temp < A[j - 1]:
```

```
                A[j] = A[j - 1]
                j -= 1
            A[j] = temp
```

- **Shell Sort**
    - input: int array[]
    - output: sorts given array
    - ```
      def shell_sort(arr):
          for gap in gaps:
              for i in range(gap, len(arr)):
              j = i
              temp = arr[i]
              while j >= gap and temp < arr[j - gap]:
                  arr[j] = arr[j - gap]
                  j -= gap
                  arr[j] = temp
              A[j] = temp
      ```
- **Quick Sort**
    - input: int array[]
    - output: sorts given array
    - ```
      def partition(A: list, lo: int, hi: int):
          i = lo - 1
          for j in range(lo, hi):
              if A[j - 1] < A[hi - 1]:
                  i += 1
                  A[i - 1], A[j - 1] = A[j - 1], A[i - 1]
          A[i], A[hi - 1] = A[hi - 1], A[i]
          return i + 1
      ```
    - ```
      # A recursive helper function for Quicksort.
      ```
    - ```
      def quick_sorter(A: list, lo: int, hi: int):
          if lo < hi:
              p = partition(A, lo, hi)
              quick_sorter(A, lo, p - 1)
              quick_sorter(A, p + 1, hi)
      ```
    - ```
      def quick_sort(A: list):
          quick_sorter(A, 1, len(A))
      ```
- **Batcher's Odd-Even Merge Sort**
    - input: int array[]
    - output: sorts given array
    - ```
      def comparator(A: list, x: int, y: int):
          if A[x] > A[y]:
              A[x], A[y] = A[y], A[x] # Swap A[x] and A[y]
      ```

```
-  def batcher_sort(A: list):
        if len(A) == 0:
            return
        n = len(A)
        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
```

Error Handling

`getopt()` includes error handling where in the event that the user's input option does not exist, a help message will be displayed. Additionally, if no sorting algorithm is given in the arguments, a help message will also be displayed.

Result

The program runs as intended. It successfully achieves everything it should. It is clang-formatted, and has intended, working functionality in both command-line options and sorting as well as printing. In terms of improvement, sorting may benefit from more clear variable naming.

scan-build reports no false positives. The program appropriately frees any heap memory allocated during runtime and sets the pointers to NULL. There are no known errors or bugs.

From the different sorting algorithm, I learned that each has its strengths and weaknesses, and that their performances can vary depending on the data set being sorted. Insertion sort, for example, seems to perform well on small data sets or nearly sorted data while Batcher's sort and heap sort are both efficient for sorting larger data sets. Insertion sort took the longest when testing out different element sizes for the graph below. Quick sort's speed seems to depend on how the pivots are chosen as given a good set of pivot points, the sort is $nlogn$, otherwise it would be n^2.

The following graphs compare performances of each sorting algorithm versus number of elements. Note the graphs are log-scaled.

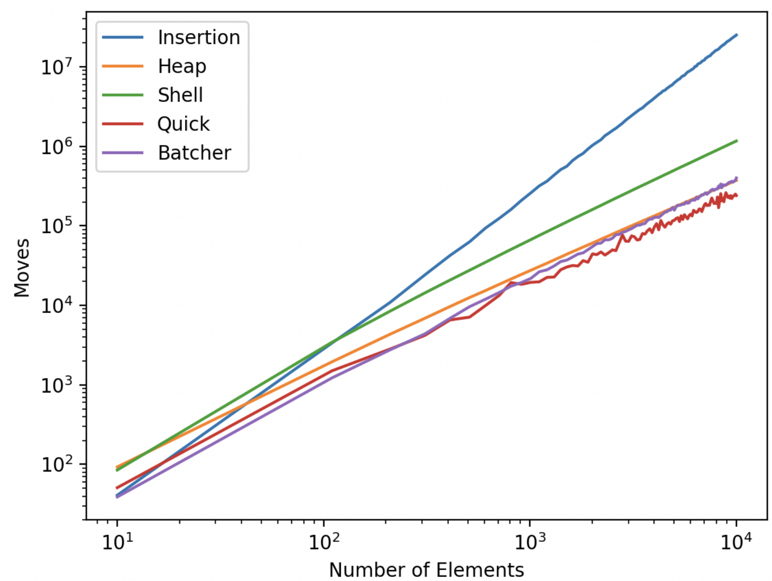

Figure 1. Number of Moves v.s. Number of Elements for each sorting algorithm.

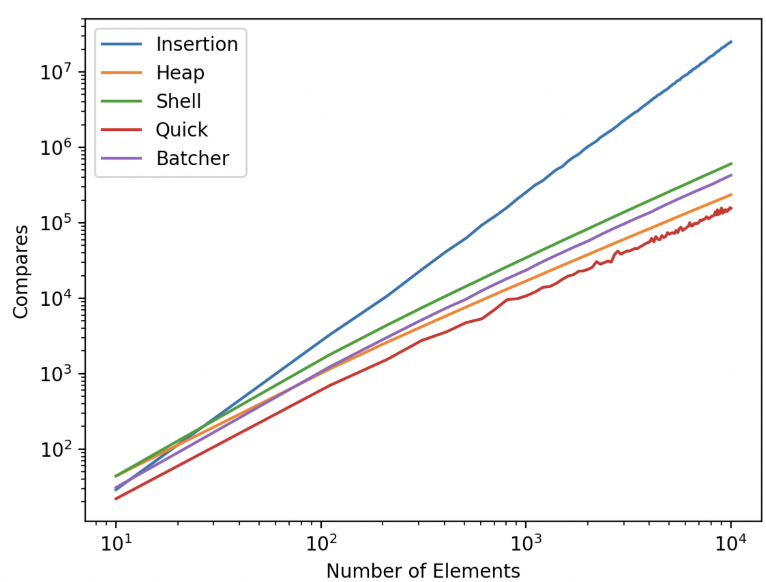

Figure 2. Number of Comparisons v.s. Number of Elements for each sorting algorithm.

```
jasonz@vm-cse13:~/desktop/cse13s/asgn3$ ./sorting -a
Insertion Sort, 100 elements, 2741 moves, 2638 compares
        8032304      34732749      42067670      54998264      56499902
       57831606      62698132      73647806      75442881     102476060
      104268822     111498166     114109178     134750049     135021286
      176917838     182960600     189016396     194989550     200592044
      212246075     243082246     251593342     256731966     261742721
      281272176     282549220     287277356     297461283     331368748
      334122749     343777258     370030967     391223417     398173317
      426152680     433486081     438071796     444703321     447975914
      451764437     455275424     460885430     464871224     473260275
      500293632     510040157     518072461     521864874     522702830
      527207318     530718305     530735134     538219612     573093082
      579453371     587189713     607875172     611422544     616902904
      620182312     629948093     630759321     648567958     689665138
      708948898     738166936     744868500     754364921     782250002
      783550802     783585680     855167780     860725547     868766010
      908068554     910310679     919290914     920038191     923423680
      934604298     935579555     944225142     950136224     954916333
      965680864     966879077     988526615     989854347     994582085
      995796877     999105042    1018598925    1025188081    1037080358
     1037686539    1048807596    1054405046    1057925624    1072766566
Heap Sort, 100 elements, 1755 moves, 1029 compares
        8032304      34732749      42067670      54998264      56499902
       57831606      62698132      73647806      75442881     102476060
      104268822     111498166     114109178     134750049     135021286
      176917838     182960600     189016396     194989550     200592044
      212246075     243082246     251593342     256731966     261742721
      281272176     282549220     287277356     297461283     331368748
      334122749     343777258     370030967     391223417     398173317
      426152680     433486081     438071796     444703321     447975914
      451764437     455275424     460885430     464871224     473260275
      500293632     510040157     518072461     521864874     522702830
      527207318     530718305     530735134     538219612     573093082
      579453371     587189713     607875172     611422544     616902904
      620182312     629948093     630759321     648567958     689665138
      708948898     738166936     744868500     754364921     782250002
      783550802     783585680     855167780     860725547     868766010
      908068554     910310679     919290914     920038191     923423680
      934604298     935579555     944225142     950136224     954916333
      965680864     966879077     988526615     989854347     994582085
      995796877     999105042    1018598925    1025188081    1037080358
     1037686539    1048807596    1054405046    1057925624    1072766566
Shell Sort, 100 elements, 3025 moves, 1575 compares
        8032304      34732749      42067670      54998264      56499902
       57831606      62698132      73647806      75442881     102476060
      104268822     111498166     114109178     134750049     135021286
      176917838     182960600     189016396     194989550     200592044
      212246075     243082246     251593342     256731966     261742721
      281272176     282549220     287277356     297461283     331368748
      334122749     343777258     370030967     391223417     398173317
      426152680     433486081     438071796     444703321     447975914
      451764437     455275424     460885430     464871224     473260275
      500293632     510040157     518072461     521864874     522702830
      527207318     530718305     530735134     538219612     573093082
      579453371     587189713     607875172     611422544     616902904
      620182312     629948093     630759321     648567958     689665138
      708948898     738166936     744868500     754364921     782250002
      783550802     783585680     855167780     860725547     868766010
      908068554     910310679     919290914     920038191     923423680
      934604298     935579555     944225142     950136224     954916333
      965680864     966879077     988526615     989854347     994582085
      995796877     999105042    1018598925    1025188081    1037080358
     1037686539    1048807596    1054405046    1057925624    1072766566
```

Figure 3. Screenshot of program output