Assignment 4 – Sets and Sorting

Aiden Trager

CSE 13S – Fall 2023

Purpose

In this assignment, we will be implementing various essential functions for sets and sorting algorithms. First, we focus on set functions, specifically using sets to track which command-line options are specified when running your program. Sets are manipulated with bit-wise operators to record and check command-line options efficiently. Additionally, we'll be implementing four sorting algorithms out of the five available, including Insertion Sort, Shell Sort, Heap Sort, Quicksort, and Batcher's Odd-Even Merge Sort. The sorting algorithms aim to arrange arrays of integers, and this assignment will provide a deep understanding of computational complexity and sorting techniques. We will also track and report the number of operations performed during sorting.

How to Use the Program

In order to run the program first make the program and then run it. These are what the commands should look like in your terminal:

```
make all
```

Followed by:

```
./sorting *Flags will be needed*
```

The flags necessary can be described by the output of the -H flag, which gives:
SYNOPSIS A collection of comparison-based sorting algorithms.
USAGE ./sorting [-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 the number of array elements (default: 100). -p elements Specify the number of elements to print (default: 100). -r seed Specify random seed (default: 13371453).

Program Design

The main program design is as follows:
1. **Main Function:** The entry point of the program is the 'main' function. It parses command-line arguments, initializes options, and performs the sorting tasks based on user input.
2. **Modularity:** The program is structured to accommodate various sorting algorithms. Each algorithm is encapsulated in a separate function, making it easy to add or remove sorting methods in the future.
3. **Command-line Arguments:** The program utilizes the 'getopt' library to handle command-line arguments, allowing users to specify sorting algorithms, array size, random seed, and the number of elements to print.

4. **Random Number Generation:** To ensure repeatability and reproducibility, the program generates pseudo-random numbers using the ‘generatePseudoRandom‘ function. These numbers are masked to fit within 30 bits.

5. **Testing Functions:** The ‘testSortingAlgorithm‘ function is employed to test and evaluate the sorting algorithms. It records statistics on the number of moves and comparisons during sorting.

6. **Statistics:** The program keeps track of statistics such as the number of moves and comparisons made by each sorting algorithm. These statistics are printed to the console.

7. **Error Handling:** Error handling is an integral part of the program. In case of memory allocation failures or invalid user input, appropriate error messages are displayed.

8. **Help and Usage Information:** Users can access help and usage information by using the ‘-H‘ option. The program displays a synopsis and options to guide the user.

9. **Algorithm Selection:** Sorting algorithms can be enabled or disabled through command-line options. The program then performs the selected sorting tasks.

Data Structures

In this program, several key data structures have been employed to manage and organize the data efficiently. Below, I describe the primary data structures used: [1]

1. **Arrays:** Arrays are used for storing collections of similar data types. They are an excellent choice when dealing with fixed-size data sets, such as storing user information or configuration settings. Arrays offer constant-time access to elements, which makes them suitable for scenarios where quick data retrieval is essential.

2. **Enums:** Enums, are utilized to define a set of named integer constants. They are particularly valuable when there is a need to represent a finite set of discrete values. For instance, enums might be employed to represent different states or options within the program.

3. **Strings:** Strings are fundamental data structures for working with textual data. They are extensively used for input/output, text processing, and user interaction. In this program, strings are employed for handling user input, generating log messages, and displaying information to the user.

4. **Structs:** Structs, are used to create custom data types by grouping different data elements together. They are suitable for representing complex entities with multiple attributes. Structs are chosen for defining custom data structures in this program, making it easier to encapsulate related data and functions.

The selection of these data structures has been driven by the specific requirements of the program, considering factors like data size, access patterns, and the overall structure of the code.

Algorithms

1. Insertion Sort:

   - Sorts elements one at a time, placing them in their correct, ordered position.
   - It iterates through the array, comparing each element with the preceding elements, shifting them as needed.
   - Typically, it starts with the second element and compares it with the first, then moves to the third and so on, inserting each element in the right place.
   - This process continues until the entire array is sorted.

2. Shell Sort:

   - A variation of Insertion Sort that starts with larger gaps and gradually reduces the gap size.
   - Elements separated by a certain gap are compared and swapped if necessary.
   - The gap is reduced in each pass until it becomes 1, which effectively performs a final Insertion Sort on nearly sorted data.
   - This approach can improve the efficiency of Insertion Sort for larger arrays.

3. Heapsort:
    - Utilizes a max-heap data structure to sort elements.
    - The array is first transformed into a max-heap, ensuring the largest element is at the root.
    - Then, the root element is removed and placed at the end of the sorted portion of the array.
    - The heap is restored, and this process continues until the entire array is sorted.
4. Quicksort:
    - A divide-and-conquer algorithm that selects a pivot element from the array and partitions the array into two sub-arrays.
    - Elements less than the pivot are placed in one sub-array, while elements greater than or equal to the pivot are placed in the other.
    - Quicksort is then recursively applied to the sub-arrays.
5. Batcher's Odd-Even Merge Sort:
    - Batcher's method is a sorting network that works in rounds, each of which sorts even and odd subsequences of the array.
    - It uses bitwise operations to compare and swap elements in a parallel fashion.
    - The array is k-sorted, where k starts as the nearest power of 2 and decreases in each round until it becomes 1.
    - Batcher's method can sort arrays of any size efficiently but is particularly effective for power-of-2 sized arrays.

These sorting algorithms have different characteristics and are suitable for various scenarios, depending on the size and characteristics of the data to be sorted.

Function Descriptions

These are just the functions in my sorting.c file.

1. int generatePseudoRandom(void)
    - Inputs: None
    - Outputs: An integer representing a pseudorandom number.
    - Purpose: Generates a pseudorandom number and masks it to fit 30 bits.
2. void generateRandomArray(int arr[], int size)
    - Inputs:
        - `arr`: An array where random integers will be stored.
        - `size`: The size of the array.
    - Outputs: None
    - Purpose: Populates the given array with random integers, each masked to 30 bits.
3. **void testSortingAlgorithm(const char *algorithmName, void (*sortFunction)(Stats , int[], int), int arr[], int size, int print_elements)*
    - Inputs:
        - `algorithmName`: A string representing the name of the sorting algorithm.
        - `sortFunction`: A function pointer to the sorting algorithm.

  - `arr`: An array to be sorted.
  - `size`: The size of the array.
  - `print_elements`: The number of elements to print (or -1 to skip printing).
- Outputs: None
- Purpose: Tests a sorting algorithm by sorting an array and printing statistics, including comparisons and movements. Optionally, it prints the sorted array.

4. **int main(int argc, char argv[])*

- Inputs:
  - `argc`: The number of command-line arguments.
  - `argv`: An array of command-line argument strings.
- Outputs: An integer return code.
- Purpose: The main function of the program. It handles command-line options, initializes randomization, generates a random array, and tests different sorting algorithms. It also frees dynamically allocated memory.

Results

My code works correct in implementing 5 different sorting algorithms. All of my functions work as intended. These five sorting algorithms have been implemented and tested: Insertion Sort, Shell Sort, Heap Sort, Quicksort, and Batcher Sort.

The code successfully tracks and reports performance metrics for each sorting algorithm. These metrics include the number of comparisons and movements. The generated statistics provide valuable insights into the behavior of each algorithm under various conditions.

Overall, the code meets the assignment's requirements, and the implementation of sorting algorithms demonstrates my strong understanding of computational complexity and sorting techniques.

The graphs of the number of moves and comparisons of the different sorts are shown below.

References

[1] Wikipedia contributors. List of data structures. https://en.wikipedia.org/wiki/List_of_data_structures, 2023. [Online; accessed 5-Nov-2023].

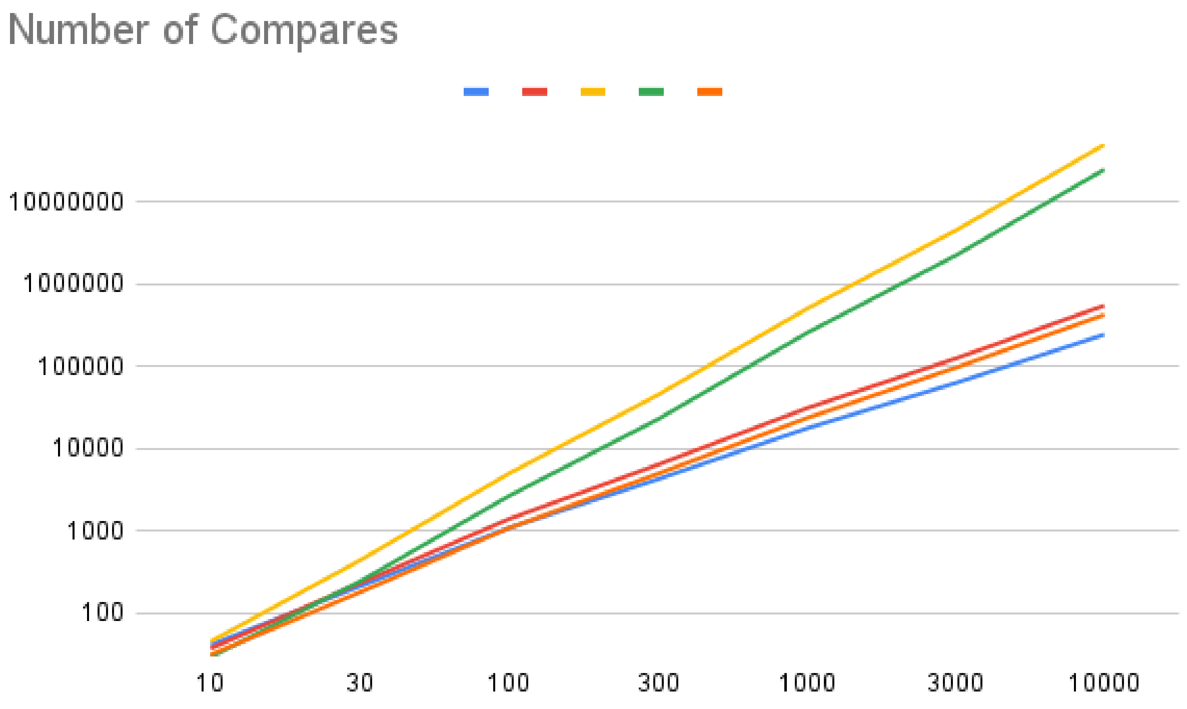

Figure 1: Blue is Heap, Red is Shell, Yellow is Quick, Green is Insertion, and Orange is Batcher.

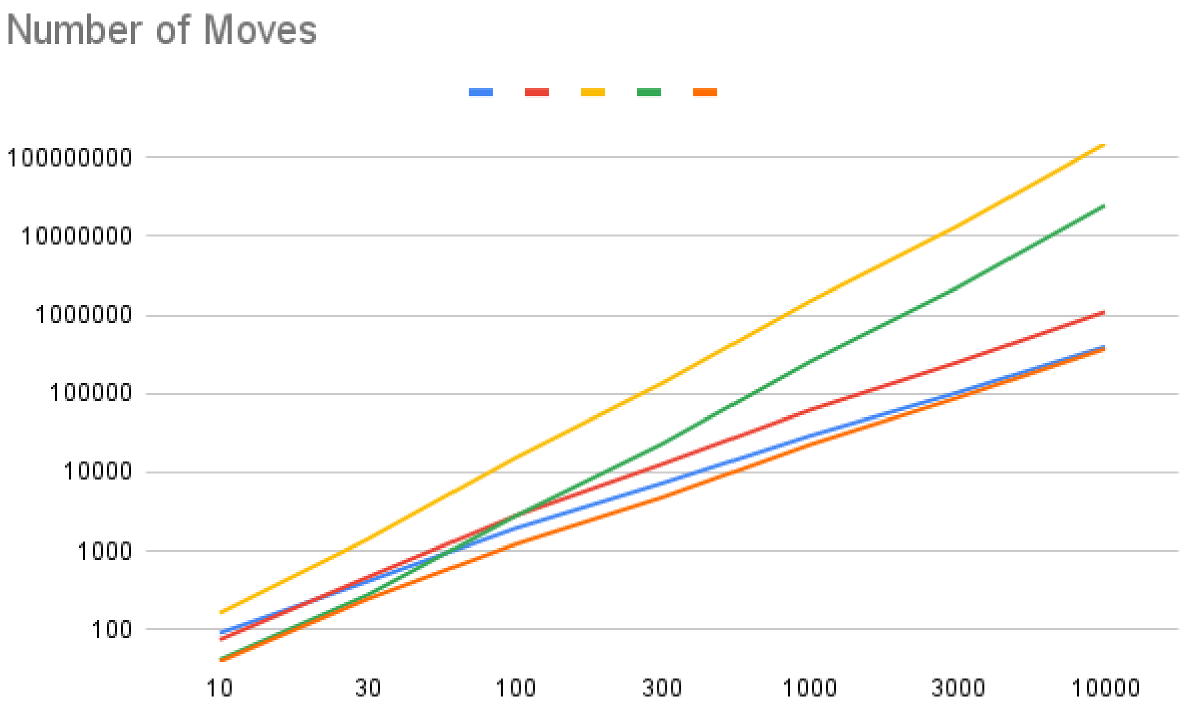

Figure 2: Blue is Heap, Red is Shell, Yellow is Quick, Green is Insertion, and Orange is Batcher.