CS111-Design and Analysis of Algorithms
2nd Semester, 2024-2025
Programming Project 1 - Empirical Analysis of Sorting Algorithms
Group Members:

1. Gerik Jed L. Abion
2. Raymond C. Balingbing
3. Renz Kirby G. Onia
4. Jayson D. Tripulca
5. Juan Miguel B. Zurbito

This project involves conducting an empirical analysis of six sorting algorithms—Selection Sort, Bubble Sort, Insertion Sort, Mergesort, Quicksort, and Heapsort—by sorting randomly generated integers and evaluating their performance.

1. **Program and Code Documentation**
    a. **Sorting Headers**
    'Sorting Headers' is a folder that contains all the sorting functions used in the experiment. Each header file includes the implementation of a specific algorithm.
    - Selection-Insertion.h file includes both the Selection sort and Insertion sort functions.
    - QuickSort.h file implements the Quicksort algorithm and includes its utility functions such as medianOfThree(), which selects the pivot by finding the median of the first, middle, and last elements to optimize the partitioning, and partition(), which rearranges the array.
    - MergeSort.h file contains the Merge sort function along with its utility function merge().
    - HeapSort.h file includes the Heap sort function along with its utility functions such as maxHeapify(), and buildMaxHeap().
    - BubbleSort.h file contains the Bubble sort function.

    b. **runtime.c**
    - void **generate_n_randoms**(int arr[], unsigned long int n, unsigned long int max_range)
        - Generates n random values to store in arr[]. Uses the rand() function in stdlib.h, a pseudo RNG that needs to be seeded with srand() for each new input.

    - void **generate_n_sequence**(int arr[], unsigned long int n, int X)
        - Generates n values in a sequence starting from X, X+1, X+2… and so on.

    - extern int **selection, insertion, bubble, merges, heap, quick**;
    extern int **output_to_file, output_arr_to_file**;
        - Global variables declared from main. Used as boolean values.

- void **generate_random_runtimes**(int arr[], unsigned long int n, int max_range, FILE* outfile, char* arr_out_name, int num_of_runs)
    - Executes all of the six sorting algorithms (or only some based on user filter) on arrays with randomized values and computes the average from the total num_of_runs.
    - Start timer for runtime is started after declaring variables and printing, and ends immediately after the sorting algorithm finishes.
    - Prints the output in table form in the terminal, and also in an output file specified by the user.

- void **generate_sequenced_runtimes**(int arr[], unsigned long int n, int seq_start, FILE* outfile, char* arr_out_name, int num_of_runs) {
    - Does the same as the function above but performs the sorting on arrays with sequenced/sorted values. All arrays start with seq_start as their first element.

c. main.c

- **Flow of the program**
    - First, the user calls the program on a terminal and supplies it with appropriate arguments. If the arguments were formatted wrong, **print_usage** will be called and the program will exit.
    - The program reads the values passed by the user through argv[].
    - Based on if the user wanted to test with random or sorted values, either **generate_random_runtimes** or **generate_sequenced_runtimes** will be called.
    - A result of the runtimes will be printed on the terminal, and it is optional to the user if they want it to be outputted to a file as well.

- void **print_usage**(char *prog_name)
    - Will print the info on how to properly use the program on the terminal. Immediately ends the program thereafter.

2. Program Execution & Experiment Methodology

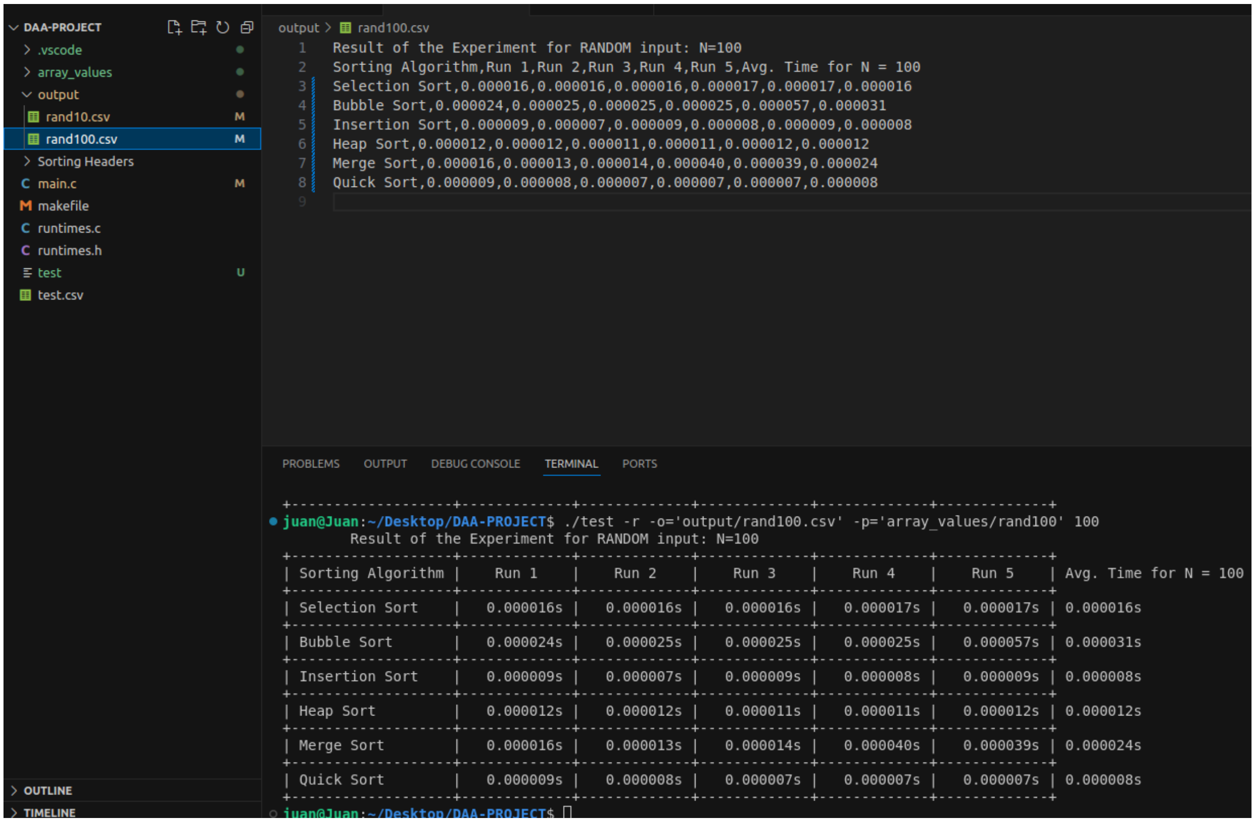
The usage information for the program can be seen by either calling it in the terminal with no other arguments or with incorrect arguments:

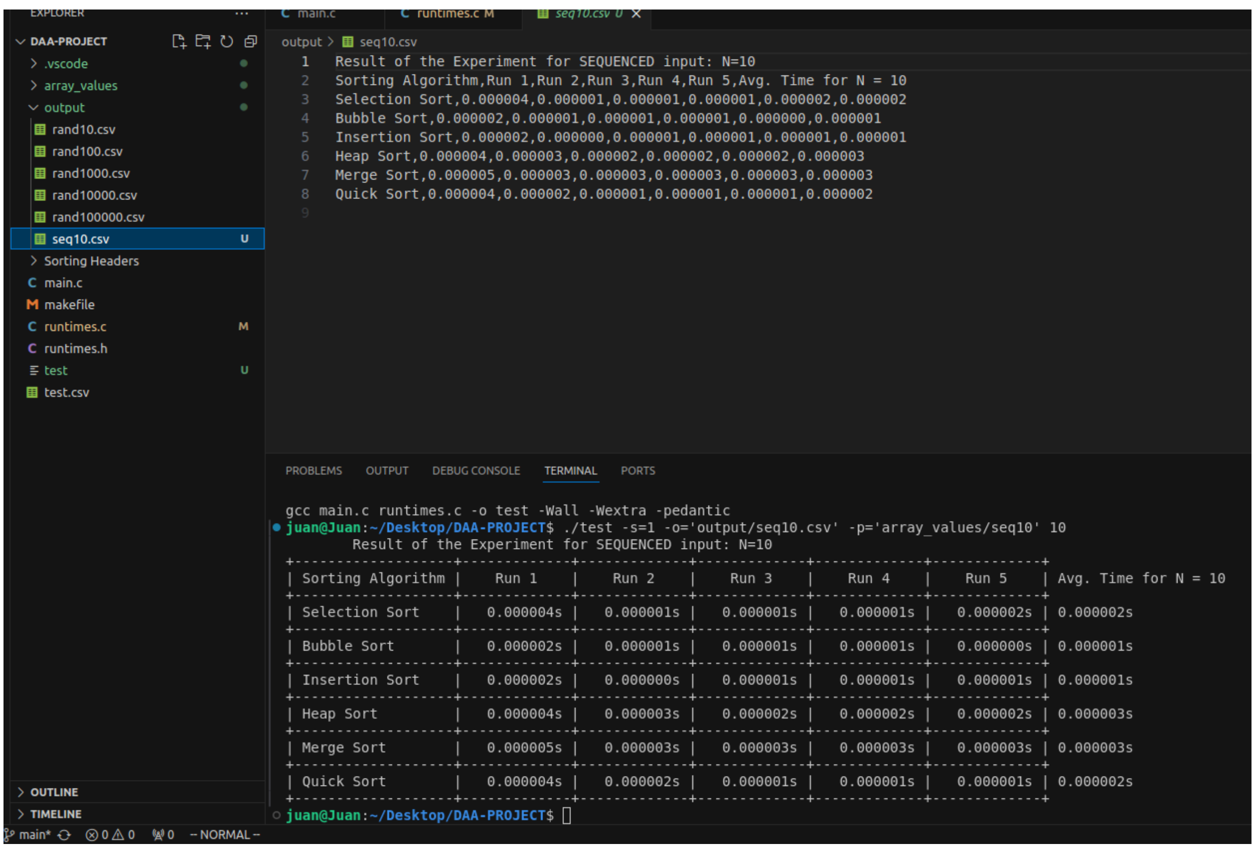
```
juan@Juan:~/Desktop/DAA-PROJECT$ ./test
Usage: ./test (-r | -s=<X>) [-m=<MAX_RANGE>] [-o='file-name'] [-p='file-name2'] [-i=<NUM_OF_RUNS>] [-f=<s|i|b|m|h|q>]<N>
Gives the avg. execution time for Selection, Bubble, Insertion, Merge, Heap, and Quick sort on array specified by user.
Examples: ./test -r -m=32768 -p='tests' 100
          ./test -s=20 -o='test.csv' -i=3 -f=hq 1000

<N> = Number of integers to be sorted, must be positive.
<X> = The number the sequence starts from, must be positive.
<MAX_RANGE> = Specified max_range for array values, can be any positive number from 1 to INT_MAX.
'file-name' = Output file destination, must contain no whitespace and can include an extension (e.g. test.csv).
<NUM_OF_RUNS> = The number of runs the program will calculate the average runtime for, default is 5.

(OPTIONS)
        -r Assigns random values to array from the range [0, <MAX_RANGE>], by default MAX_RANGE=INT_MAX.
        -s Assigns the array with asequence of values starting from <X>.
        -m Sets the limit for the array values.
        -o Outputs the result into a desired file type, .csv recommended for convenient table formatting.
        -p Output array values before and after sorting, file-name should not include extensions for this.
        -i Set the number of runs to calculate average for.
        -f Filter out the given sorting algorithms, s for selection, i for insertion and so on (can input multiple e.g. -f=ibh).
juan@Juan:~/Desktop/DAA-PROJECT$
```

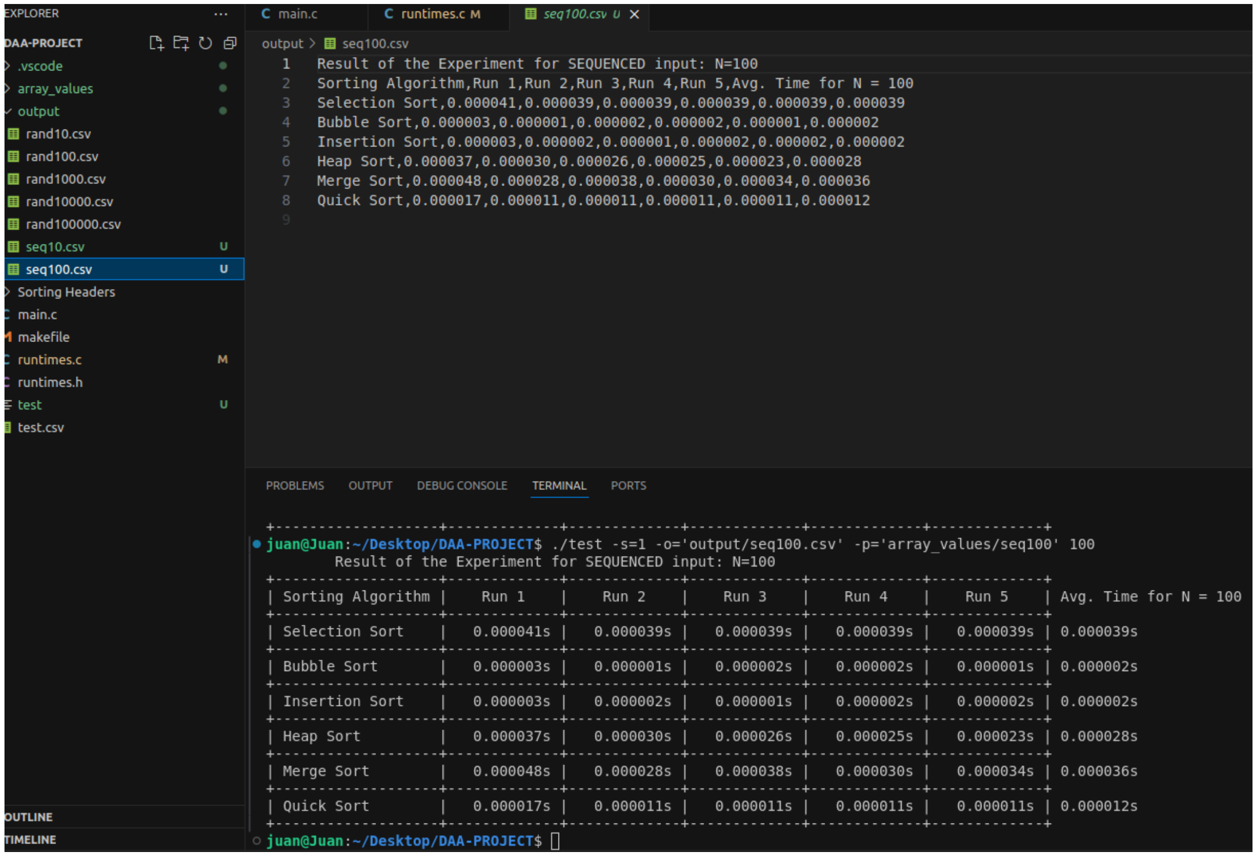
Before running the experiment, it was made sure that the host computer for running the program had no other non-essential background apps running, as sharing processing power and memory with these apps could cause the runtime of the program to fluctuate.

The experiment was done by calling the following arguments on the terminal and yielding the following results:

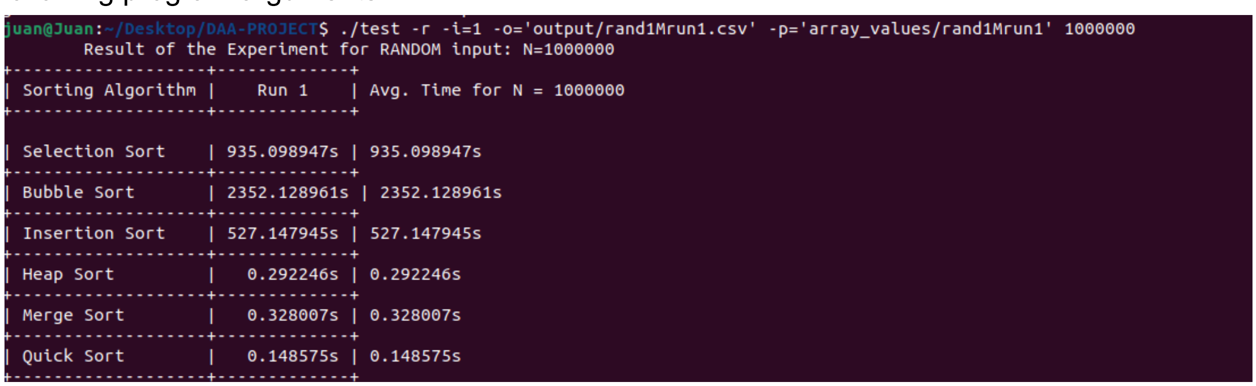
```
juan@Juan:~/Desktop/DAA-PROJECT$ ./test -r -o='output/rand10.csv' -p='array_values/rand10' 10
          Result of the Experiment for RANDOM input: N=10
+-------------------+------------+------------+------------+------------+------------+
| Sorting Algorithm |    Run 1   |    Run 2   |    Run 3   |    Run 4   |    Run 5   | Avg. Time for N = 10
+-------------------+------------+------------+------------+------------+------------+
| Selection Sort    |  0.000005s |  0.000003s |  0.000002s |  0.000003s |  0.000002s | 0.000003s
+-------------------+------------+------------+------------+------------+------------+
| Bubble Sort       |  0.000005s |  0.000003s |  0.000003s |  0.000003s |  0.000002s | 0.000003s
+-------------------+------------+------------+------------+------------+------------+
| Insertion Sort    |  0.000003s |  0.000002s |  0.000002s |  0.000002s |  0.000002s | 0.000002s
+-------------------+------------+------------+------------+------------+------------+
| Heap Sort         |  0.000006s |  0.000004s |  0.000003s |  0.000004s |  0.000004s | 0.000004s
+-------------------+------------+------------+------------+------------+------------+
| Merge Sort        |  0.000009s |  0.000006s |  0.000005s |  0.000005s |  0.000005s | 0.000006s
+-------------------+------------+------------+------------+------------+------------+
| Quick Sort        |  0.000006s |  0.000003s |  0.000002s |  0.000003s |  0.000002s | 0.000003s
+-------------------+------------+------------+------------+------------+------------+
juan@Juan:~/Desktop/DAA-PROJECT$
```


DAA-PROJECT
.vscode
array_values
output
rand10.csv
rand100.csv
Sorting Headers
main.c
makefile
runtimes.c
runtimes.h
test
test.csv
OUTLINE
TIMELINE
output > rand100.csv
Result of the Experiment for RANDOM input: N=100
Sorting Algorithm,Run 1,Run 2,Run 3,Run 4,Run 5,Avg. Time for N = 100
Selection Sort,0.000016,0.000016,0.000016,0.000017,0.000017,0.000016
Bubble Sort,0.000024,0.000025,0.000025,0.000025,0.000057,0.000031
Insertion Sort,0.000009,0.000007,0.000009,0.000008,0.000009,0.000008
Heap Sort,0.000012,0.000012,0.000011,0.000011,0.000012,0.000012
Merge Sort,0.000016,0.000013,0.000014,0.000040,0.000039,0.000024
Quick Sort,0.000009,0.000008,0.000007,0.000007,0.000007,0.000008
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
juan@Juan:~/Desktop/DAA-PROJECT$ ./test -r -o='output/rand100.csv' -p='array_values/rand100' 100
Result of the Experiment for RANDOM input: N=100
| Sorting Algorithm | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Avg. Time for N = 100
| Selection Sort | 0.000016s | 0.000016s | 0.000016s | 0.000017s | 0.000017s | 0.000016s
| Bubble Sort | 0.000024s | 0.000025s | 0.000025s | 0.000025s | 0.000057s | 0.000031s
| Insertion Sort | 0.000009s | 0.000007s | 0.000009s | 0.000008s | 0.000009s | 0.000008s
| Heap Sort | 0.000012s | 0.000012s | 0.000011s | 0.000011s | 0.000012s | 0.000012s
| Merge Sort | 0.000016s | 0.000013s | 0.000014s | 0.000040s | 0.000039s | 0.000024s
| Quick Sort | 0.000009s | 0.000008s | 0.000007s | 0.000007s | 0.000007s | 0.000008s
juan@Juan:~/Desktop/DAA-PROJECT$


EXPLORER
main.c
runtimes.c M
seq10.csv U
DAA-PROJECT
.vscode
array_values
output
rand10.csv
rand100.csv
rand1000.csv
rand10000.csv
rand100000.csv
seq10.csv
Sorting Headers
main.c
makefile
runtimes.c
runtimes.h
test
test.csv
OUTLINE
TIMELINE
output > seq10.csv
Result of the Experiment for SEQUENCED input: N=10
Sorting Algorithm,Run 1,Run 2,Run 3,Run 4,Run 5,Avg. Time for N = 10
Selection Sort,0.000004,0.000001,0.000001,0.000001,0.000002,0.000002
Bubble Sort,0.000002,0.000001,0.000001,0.000001,0.000000,0.000001
Insertion Sort,0.000002,0.000000,0.000001,0.000001,0.000001,0.000001
Heap Sort,0.000004,0.000003,0.000002,0.000002,0.000002,0.000003
Merge Sort,0.000005,0.000003,0.000003,0.000003,0.000003,0.000003
Quick Sort,0.000004,0.000002,0.000001,0.000001,0.000001,0.000002
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
gcc main.c runtimes.c -o test -Wall -Wextra -pedantic
juan@Juan:~/Desktop/DAA-PROJECT$ ./test -s=1 -o='output/seq10.csv' -p='array_values/seq10' 10
Result of the Experiment for SEQUENCED input: N=10
| Sorting Algorithm | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Avg. Time for N = 10
| Selection Sort | 0.000004s | 0.000001s | 0.000001s | 0.000001s | 0.000002s | 0.000002s
| Bubble Sort | 0.000002s | 0.000001s | 0.000001s | 0.000001s | 0.000000s | 0.000001s
| Insertion Sort | 0.000002s | 0.000000s | 0.000001s | 0.000001s | 0.000001s | 0.000001s
| Heap Sort | 0.000004s | 0.000003s | 0.000002s | 0.000002s | 0.000002s | 0.000003s
| Merge Sort | 0.000005s | 0.000003s | 0.000003s | 0.000003s | 0.000003s | 0.000003s
| Quick Sort | 0.000004s | 0.000002s | 0.000001s | 0.000001s | 0.000001s | 0.000002s
juan@Juan:~/Desktop/DAA-PROJECT$
main* -- NORMAL --



This was repeated to include all the results from n=10, 100, …, 100000; for both random and sorted array values. The exception was for n=1000000, which had the following program arguments.



The program took too long in computing the runtimes for the sorting algorithm (mainly for bubble and selection sort), so it was decided that each run would be performed separately just in case a problem arises in the computer environment and all progress would have been lost.

The results in output/*.csv files were then imported to google sheets by doing the following: File > Import > Upload > Append to current sheet. The raw results can be seen from the link: https://docs.google.com/spreadsheets/d/14G79f53YTWPvP0Sz406DUxAarVmX6L4-TEttr4MYTDA/edit?usp=sharing.

3. **Results and Analysis**
   a. System Specification
      System (Oracle VM VirtualBox Manager)
         OS: Ubuntu (64-bit)
         Base Memory: 8192MB
         Processors: 4
   b. Results

Result of the experiment for RANDOM N = 10						
Sorting Algorithm	**Run 1**	**Run 2**	**Run 3**	**Run 4**	**Run 5**	**Avg. Time**
Selection Sort	0.000005	0.000003	0.000002	0.000003	0.000002	0.000003
Bubble Sort	0.000005	0.000003	0.000003	0.000003	0.000002	0.000003
Insertion Sort	0.000003	0.000002	0.000002	0.000002	0.000002	0.000002
Heap Sort	0.000006	0.000004	0.000003	0.000004	0.000004	0.000004
Merge Sort	0.000009	0.000006	0.000005	0.000005	0.000005	0.000006
Quick Sort	0.000006	0.000003	0.000002	0.000003	0.000002	0.000003

Result of the experiment for RANDOM N = 100						
Sorting Algorithm	**Run 1**	**Run 2**	**Run 3**	**Run 4**	**Run 5**	**Avg. Time**
Selection Sort	0.000016	0.000016	0.000016	0.000017	0.000017	0.000016
Bubble Sort	0.000024	0.000025	0.000025	0.000025	0.000057	0.000031
Insertion Sort	0.000009	0.000007	0.000009	0.000008	0.000009	0.000008
Heap Sort	0.000012	0.000012	0.000011	0.000011	0.000012	0.000012
Merge Sort	0.000016	0.000013	0.000014	0.00004	0.000039	0.000024
Quick Sort	0.000009	0.000008	0.000007	0.000007	0.000007	0.000008

Result of the experiment for RANDOM N = 1 000						
Sorting Algorithm	**Run 1**	**Run 2**	**Run 3**	**Run 4**	**Run 5**	**Avg. Time**
Selection Sort	0.001092	0.00136	0.001043	0.001155	0.001114	0.001153
Bubble Sort	0.001564	0.001923	0.005412	0.002474	0.00477	0.003229
Insertion Sort	0.000564	0.000668	0.000555	0.000609	0.000678	0.000615
Heap Sort	0.000134	0.000132	0.000133	0.000131	0.000133	0.000133
Merge Sort	0.000138	0.000144	0.000135	0.000143	0.000137	0.000139
Quick Sort	0.000081	0.000082	0.000081	0.00008	0.000187	0.000102

Result of the experiment for RANDOM N = 10 000						
Sorting Algorithm	**Run 1**	**Run 2**	**Run 3**	**Run 4**	**Run 5**	**Avg. Time**
Selection Sort	0.186278	0.096572	0.096741	0.096888	0.095748	0.114445
Bubble Sort	0.225802	0.196132	0.205495	0.210652	0.197679	0.207152
Insertion Sort	0.061771	0.057683	0.056974	0.054652	0.066415	0.059499
Heap Sort	0.001858	0.001847	0.001992	0.001868	0.001754	0.001864
Merge Sort	0.001845	0.002001	0.002084	0.001654	0.002517	0.00202
Quick Sort	0.002697	0.001632	0.001121	0.001031	0.001246	0.001545

Result of the experiment for RANDOM N = 100 000						
Sorting Algorithm	**Run 1**	**Run 2**	**Run 3**	**Run 4**	**Run 5**	**Avg. Time**
Selection Sort	9.346858	9.224904	9.254818	9.263143	9.240119	9.265968
Bubble Sort	22.614967	22.773977	22.736748	22.631775	22.663411	22.684176
Insertion Sort	5.24579	5.394252	5.400657	5.384738	5.342609	5.353609
Heap Sort	0.025432	0.032659	0.031213	0.025219	0.023949	0.027694
Merge Sort	0.027336	0.032359	0.022259	0.022132	0.022706	0.025358
Quick Sort	0.013557	0.025006	0.013662	0.022425	0.012846	0.017499

Result of the experiment for RANDOM N = 1 000 000						
Sorting Algorithm	**Run 1**	**Run 2**	**Run 3**	**Run 4**	**Run 5**	**Avg. Time**
Selection	935.098947	945.07975	952.040744	1001.646494	997.521166	966.2774202
Bubble	2352.128961	2349.700774	2355.738694	2478.15056	2483.83615	2403.911028
Insertion	527.147945	535.271002	538.242525	563.652755	578.98706	548.6602574
Heap Sort	0.292246	0.30272	0.296546	0.311755	0.363257	0.3133048
Merge Sort	0.328007	0.72895	0.261541	0.299041	0.291257	0.3817592
Quick Sort	0.148575	0.153774	0.151448	0.151535	0.157244	0.1525152

Result of the experiment for SEQUENCED N = 10						
Sorting Algorithm	**Run 1**	**Run 2**	**Run 3**	**Run 4**	**Run 5**	**Avg. Time**
Selection Sort	0.000004	0.000001	0.000001	0.000001	0.000002	0.000002
Bubble Sort	0.000002	0.000001	0.000001	0.000001	0	0.000001
Insertion Sort	0.000002	0	0.000001	0.000001	0.000001	0.000001
Heap Sort	0.000004	0.000003	0.000002	0.000002	0.000002	0.000003
Merge Sort	0.000005	0.000003	0.000003	0.000003	0.000003	0.000003
Quick Sort	0.000004	0.000002	0.000001	0.000001	0.000001	0.000002

Result of the experiment for SEQUENCED N = 100						
Sorting Algorithm	**Run 1**	**Run 2**	**Run 3**	**Run 4**	**Run 5**	**Avg. Time**
Selection Sort	0.000041	0.000039	0.000039	0.000039	0.000039	0.000039
Bubble Sort	0.000003	0.000001	0.000002	0.000002	0.000001	0.000002
Insertion Sort	0.000003	0.000002	0.000001	0.000002	0.000002	0.000002
Heap Sort	0.000037	0.00003	0.000026	0.000025	0.000023	0.000028
Merge Sort	0.000048	0.000028	0.000038	0.00003	0.000034	0.000036
Quick Sort	0.000017	0.000011	0.000011	0.000011	0.000011	0.000012

Result of the experiment for SEQUENCED N = 1 000						
Sorting Algorithm	**Run 1**	**Run 2**	**Run 3**	**Run 4**	**Run 5**	**Avg. Time**
Selection Sort	0.004233	0.004055	0.006781	0.003958	0.009006	0.005607
Bubble Sort	0.000002	0.000002	0.000002	0.000002	0.000002	0.000002
Insertion Sort	0.000003	0.000004	0.000003	0.000003	0.000003	0.000003
Heap Sort	0.000116	0.000116	0.000116	0.00041	0.000159	0.000183
Merge Sort	0.000107	0.000117	0.000099	0.000097	0.000116	0.000107
Quick Sort	0.000038	0.000037	0.000037	0.000036	0.000036	0.000037

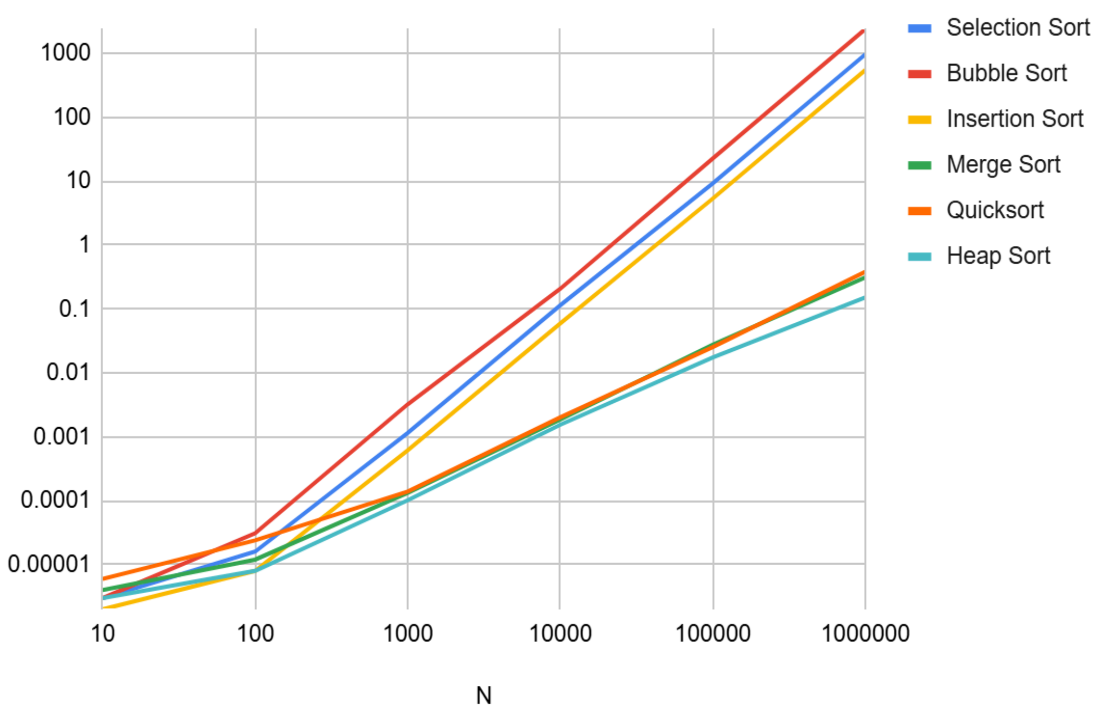
Result of the experiment for SEQUENCED N = 10 000						
Sorting Algorithm	**Run 1**	**Run 2**	**Run 3**	**Run 4**	**Run 5**	**Avg. Time**
Selection Sort	0.138308	0.12332	0.135592	0.157248	0.11696	0.134286
Bubble Sort	0.000019	0.00002	0.000023	0.000021	0.000018	0.00002
Insertion Sort	0.000029	0.000029	0.000029	0.000029	0.000029	0.000029
Heap Sort	0.001473	0.00178	0.001865	0.001512	0.001453	0.001617
Merge Sort	0.001182	0.001182	0.001223	0.001262	0.001258	0.001221
Quick Sort	0.000561	0.000471	0.000642	0.000542	0.000495	0.000542

Result of the experiment for SEQUENCED N = 100 000						
Sorting Algorithm	**Run 1**	**Run 2**	**Run 3**	**Run 4**	**Run 5**	**Avg. Time**
Selection Sort	9.452802	9.322537	9.275455	9.26268	9.253998	9.313494
Bubble Sort	0.000191	0.000206	0.000239	0.000188	0.000193	0.000203
Insertion Sort	0.000297	0.00031	0.000295	0.000302	0.000302	0.000301
Heap Sort	0.019228	0.022516	0.017898	0.017642	0.018221	0.019101
Merge Sort	0.016794	0.016426	0.013451	0.01385	0.013842	0.014873
Quick Sort	0.006055	0.006315	0.006107	0.006636	0.005925	0.006208

Result of the experiment for SEQUENCED N = 1 000 000						
Sorting Algorithm	**Run 1**	**Run 2**	**Run 3**	**Run 4**	**Run 5**	**Avg. Time**
Selection Sort	967.150653	942.296004	952.652591	952.96163	951.613835	953.3349426
Bubble Sort	0.001875	0.002078	0.001931	0.002037	0.00206	0.0019962
Insertion Sort	0.002895	0.002974	0.003151	0.00289	0.003267	0.0030354
Heap Sort	0.215287	0.220629	0.219291	0.229867	0.223469	0.2217086
Merge Sort	0.178892	0.63852	0.69768	0.738449	0.637546	0.5782174
Quick Sort	0.066537	0.068741	0.067741	0.068276	0.067827	0.0678244

c. Analysis

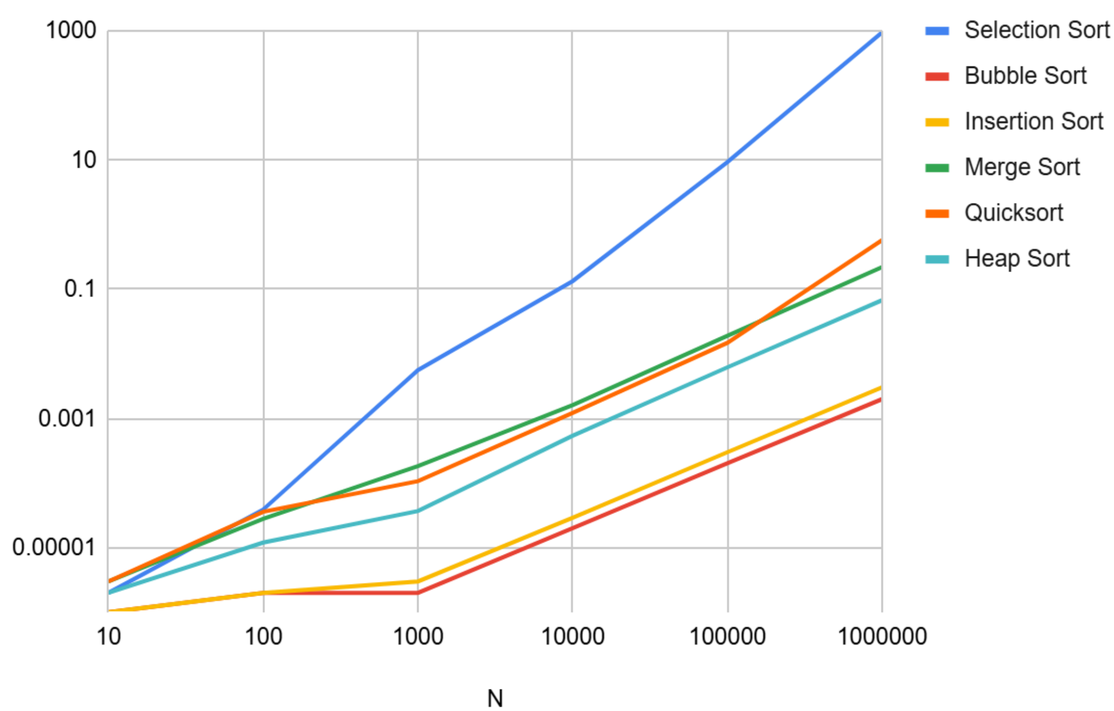
Average Running Time for an Input Array that is Random						
N	Selection Sort	Bubble Sort	Insertion Sort	Merge Sort	Quicksort	Heap Sort
10	0.000003	0.000003	0.000002	0.000004	0.000006	0.000003
100	0.000016	0.000031	0.000008	0.000012	0.000024	0.000008
1000	0.001153	0.003229	0.000615	0.000133	0.000139	0.000102
10000	0.114445	0.207152	0.059499	0.001864	0.00202	0.001545
100000	9.265968	22.684176	5.353609	0.027694	0.025358	0.017499
1000000	966.2774202	2403.911028	548.6602574	0.3133048	0.3817592	0.1525152



Based on the data that had been gathered for all the sorting algorithms, starting from an input array that is random. **Quadratic sorting algorithms** performed at minimal speed as expected when the input size is small, particularly at 10,100, and 1000. However, starting from the 1000th input size, it is where linear sorting algorithms' speed started to **gradually decline as more swapping and searching operations are needed to be done** when there are an increasing number of inputs, hence the reason that they grow quadratically because of these excessive data movements that occur on larger inputs. On the other hand,

recursive algorithms throughout their runs and average time had consistent efficient speed even at **larger inputs**. Their **divide-and-conquer** mechanisms allowed them to perform efficiently and more faster unlike the quadratic sorting algorithms due to minimizing the redundancy of the swapping and searching operations. Instead of the quadratic operations, they divide the array into smaller sub-arrays repeatedly that explain the log n division where they are then multiplied to the linear operations such as merging, partitioning, and comparisons—producing an n log n time complexity.

Average Running Time for an Input Array that is Sorted						
N	**Selection Sort**	**Bubble Sort**	**Insertion Sort**	**Merge Sort**	**Quicksort**	**Heap Sort**
10	0.000002	0.000001	0.000001	0.000003	0.000003	0.000002
100	0.000039	0.000002	0.000002	0.000028	0.000036	0.000012
1000	0.005607	0.000002	0.000003	0.000183	0.000107	0.000037
10000	0.134286	0.00002	0.000029	0.001617	0.001221	0.000542
100000	9.313494	0.000203	0.000301	0.019101	0.014873	0.006208
1000000	953.3349426	0.0019962	0.0030354	0.2217086	0.5782174	0.0678244



Following on the results for an input array that is sorted. It is observed that the **quadratic sorting algorithms** had an improvement regarding their running time. The principle behind this development was caused by the **reduction of comparing adjacent elements** to perform swaps, shifts, and search operations. Mainly because the array is already sorted as mentioned earlier, thus if it is in the right order already. **Insertion sort and (Modified) Bubble sort** in particular, would know that there are lesser comparisons needed to be done resulting in n time complexity. This is also the reason why compared to the selection sort amongst quadratic algorithms, insertion and modified bubble sort had significant improvement speeds because of flags and insertion's left sorted array. While selection sort **needs to search in the entire N array** even if it is sorted already, hence having miniscule differences for better or worse on its running time. **Recursive algorithms** in this scenario had also **little to no differences on their running time** at all. There are no flags or any indicators on how they are programmed to know if the array is already sorted, hence despite the array's sorted order. A recursive algorithm will still divide the array into subarrays just like in a random array, making their n log n time complexity as the **same as before**

4. Conclusion

The main variables for quadratic sorting algorithms are the N size of the array and an array's ordered disposition (whether it is sorted or not). It is evident in the results that these are the factors that mainly affect their running times. In contrast, recursive algorithms stay almost consistent and efficient throughout their running times primarily because it is independent of an array's initial order of elements. Moreover, their divide-and-conquer systematic process allowed their algorithms to be optimized and scalable in larger data inputs as shown in the tables compared to the quadratic algorithms.

Nevertheless there is an exception in selecting quicksort's pivot element, in the initial testing of its running time. Quicksort was unable to sort an array of 100,000 elements in sequence order, because of its pivot being the last element (a very large X element) resulting in the worst case time complexity O(n^2). It was inferred to have caused a memory stack overflow due to the recursive function calls. Changing its pivot to median of three successfully allowed us to sort the array due to having a balanced partition in the even distribution of elements, leading to fewer recursive calls. That is why it is important to take note of your situation and a computer's hardware capabilities, as this test solely focuses on the behaviors of each sorting algorithm. This gives us the insight to take in account on what proper sorting algorithm we are going to use in a certain problem. The running time is one of the many aspects to consider in implementing a sorting algorithm, as there are also space complexities and cpu efficiency that plays a role in determining their best performances—that could vary on different environments.

5. Challenges & Contributions

a. Challenges

- **Program Structure**
  - Deciding on how the user would have to instruct the program either with a dynamic user interface or a static Command Line Interface (CLI), similar to programs like git, grep, make, etc. Ultimately, it was decided that the former was too cumbersome and takes longer as inputs would be in a sequence, as with the current interface, it's all in one go.

  - Structuring the runtime outputs was also hard as just copying the prints from the terminal would just prove to be useless and redundant. The solution was the format, Comma-separated values(csv), as it was easy to produce and is convenient for importing the table to spreadsheet programs like Excel and Google Sheets.

- **Coding**
  - The code written in runtime.c could certainly have been more simple and cleaner. The file got long very quickly because it had to repeat all the computations, all the printing and everything for all six sorting algorithms. It could have been solved simply by including the sorting algorithms as arguments to the functions computing the runtime, but this realization had arrived too late as most of the program was already done.

- **Data Gathering**
  - There were many problems in getting the output results for very high values for n (n=100k and n=1M). First, was that for the functions that use recursion a lot like merge sort, and quicksort that chooses the first or last element as pivot, the memory stack would overflow thus causing the program to end prematurely. The case for merge sort was solved by allocating the auxiliary array values to the heap, and the case for quicksort was solved by using the median of three in choosing the pivot.

  - The second problem for very high values of n, was that the algorithms with quadratic time complexity can take as long as 40 minutes making each run long for n=1M. No alternative solution was found other than plainly waiting.

- **Documentation**
  - The file sizes for the values of the original and sorted arrays used in each test grew too big to be placed in a project repository like Github. No solution was found other than removing all array value files from the repository and just keeping them in the host computer.

b. Contributions

- Juan Miguel B. Zurbito
    - Authored ‘main.c’, ‘runtimes.c’ and ‘runtime.h’, which included the CLI, I/O, array generation and runtime computation.
    - Performed the experiment on my laptop.
    - Formatted the results to the tables.
    - Code documentation and report writing.

- Raymond C. Balingbing
    - Coded the Bubble Sort algorithm in C.
    - Formatted the results in the tables into graphical representation.
    - Data documentation and report writing of the ‘Results and Analysis’ and ‘Sorting Headers’ section.

- Gerik Jed L. Abion
    - Coded the Quick Sort and Heap Sort.
    - Writing of data analysis and conclusion.

- Jayson Tripulca
    - Coded the Merge Sort.

- Renz Kirby Onia
    - Coded the Selection and Insertion Sort.