CS111- Design and Analysis of Algorithms

Programming Project

Project Documentation

I. Project Description

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.

II. Code Documentation

A. Header Files

a. Uses standard libraries (stdio.h, stdlib.h, etc.)
b. Uses OS-specific timing functions for precise timing
c. Uses specialized headers (limits.h, stdint.h) for data types

B. Data Structure

a. SortingAlgorithm Struct
   - const char *name: algorithm name
   - const char *outputFile: output file
   - double time: timing results
   - void (*function)(unsigned long int*, int): function pointer to its corresponding algorithm

C. Global Variables

a. SortingAlgorithm algorithms[]
   - array of SortingAlgorithm structs
   - Made global as multiple functions access it to avoid parameter drilling.
b. algorithmsSize
   - calculated size of algorithms array

D. Main Function

a. Uses a while-loop for the menu and user input
b. Handles user input for the number of integers (N) and data generation method (random or increasing sequence)
c. Coordinates benchmarking process for the selected data generation method
d. Displays the results in ranked order

E. Data Generation Functions

a. unsigned long int *generateRandomIntegers(int n);
   - Generates random integers ranging from [0,MAX_RANGE]
   - Returns the dynamically allocated array
b. unsigned long int *generateIncreasingSequence(int n)
   - Creates an ordered increasing sequence starting from given value
   - Returns the dynamically allocated array

F. Benchmarking System

a. void runBenchmark(unsigned long int *array, int n);
   - Uses for-loop to run each sorting algorithm
   - Duplicates the input array
   - Measures the execution time
   - Saves sorted output to file
   - Records timing results by outputting them to a .csv file for safer extraction and easier data processing

G. Sorting Algorithms

a. void selectionSort(unsigned long int *array, int n);
b. void bubbleSort(unsigned long int *array, int n);
c. void insertionSort(unsigned long int *array, int n);
d. void mergeSort(unsigned long int *array, int n);
e. void quickSort(unsigned long int *array, int n);
   - uses the median-of-three strategy to choose a pivot
f. void heapSort(unsigned long int *array, int n);

H. Utility Functions

a. File I/O operations
   1. void clearFile(void);
   2. void appendStringToFile(const char *filename, const char *format, …);
   3. void appendArrayToFile(const char *filename, unsigned long int *array);

b. Array operations
   1. unsigned long int *duplicateArray(unsigned long int *array, int n);
   2. SortingAlgorithms *duplicateAlgorithmsArray(SortingAlgorithms *array, int n);

c. Display functions
   1. void clearScreen(void);
   2. void displayHeader(void);
   3. void displayConfirmExit(void);

d. Other functions
   1. int compareByTime(const void *a, const void *b);
      - Callback function used in qsort in duplicateAlgorithmsArray()
   2. double getTimeInSeconds(void);
   3. void sleepProgram(int milliseconds);

III. Test Run Documentation

Github Repository: https://github.com/michaelcanonizado/sorting-algorithm-analysis.git

```
Sorting Algorithm Analysis

Number of Elements (N): 10
Generation Method: Random Integers

| Rank |      Algorithm      |      Time      |
|    1 |      Insertion Sort |    0.000000800|
|    2 |      Selection Sort |    0.000000800|
|    3 |           Heap Sort |    0.000001700|
|    4 |           Quick Sort |    0.000001700|
|    5 |          Bubble Sort |    0.000001900|
|    6 |          Merge Sort |    0.000007300|

Enter Y/y to exit:
```

Figure 1.

```
Sorting Algorithm Analysis

Number of Elements (N): 10
Generation Method: Random Integers

| Rank |      Algorithm      |      Time      |
|    1 |      Insertion Sort |    0.000000700|
|    2 |      Selection Sort |    0.000000800|
|    3 |          Bubble Sort |    0.000001000|
|    4 |           Quick Sort |    0.000001600|
|    5 |           Heap Sort |    0.000002100|
|    6 |          Merge Sort |    0.000006300|

Enter Y/y to exit:
```

Figure 2.

```
Sorting Algorithm Analysis

Number of Elements (N): 10
Generation Method: Random Integers

| Rank |      Algorithm      |      Time      |
|    1 |      Insertion Sort |    0.000001000|
|    2 |           Heap Sort |    0.000001900|
|    3 |          Bubble Sort |    0.000002100|
|    4 |           Quick Sort |    0.000002200|
|    5 |      Selection Sort |    0.000002200|
|    6 |          Merge Sort |    0.000011800|

Enter Y/y to exit:
```

Figure 3.

```
Sorting Algorithm Analysis

Number of Elements (N): 10
Generation Method: Random Integers

| Rank |      Algorithm      |      Time      |
|    1 |          Bubble Sort |    0.000000600|
|    2 |      Insertion Sort |    0.000001100|
|    3 |           Heap Sort |    0.000001500|
|    4 |           Quick Sort |    0.000001800|
|    5 |      Selection Sort |    0.000002500|
|    6 |          Merge Sort |    0.000005900|

Enter Y/y to exit:
```

Figure 4.

```
Sorting Algorithm Analysis

Number of Elements (N): 10
Generation Method: Random Integers

| Rank |      Algorithm      |      Time      |
|    1 |          Bubble Sort |    0.000000700|
|    2 |      Insertion Sort |    0.000000700|
|    3 |           Quick Sort |    0.000001300|
|    4 |      Selection Sort |    0.000001300|
|    5 |           Heap Sort |    0.000001800|
|    6 |          Merge Sort |    0.000012500|

Enter Y/y to exit:
```

Figure 5.

Figures 1–5. Execution Time of Sorting Algorithms on Random Integer Inputs (N = 10)

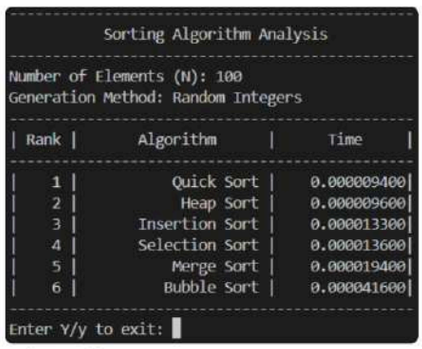


Figure 6.

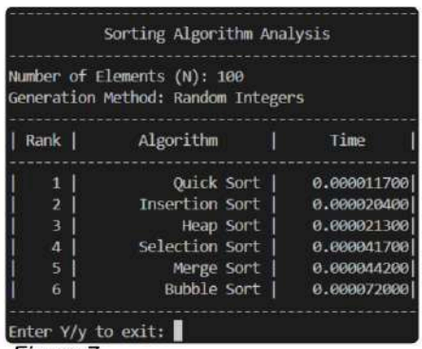


Figure 7.

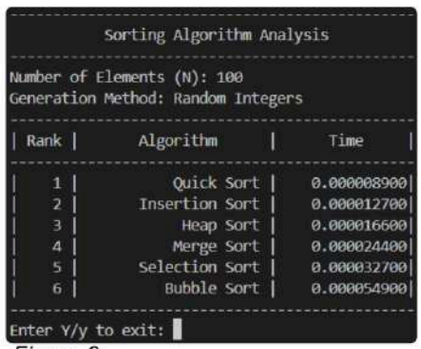


Figure 8.

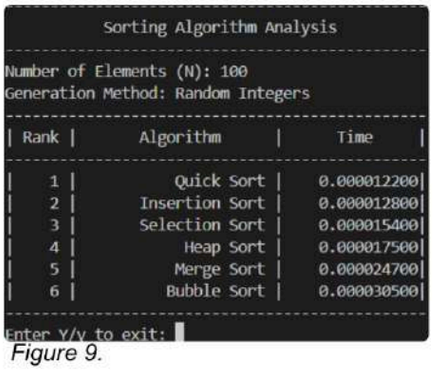


Figure 9.

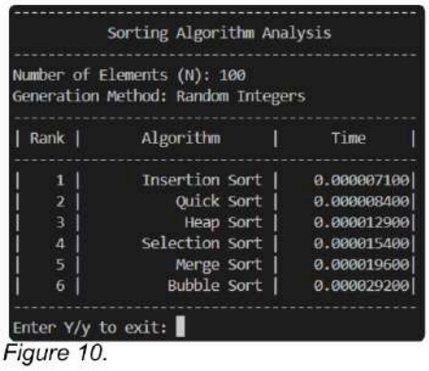


Figure 10.

Figures 6–10. Execution Time of Sorting Algorithms on Random Integer Inputs (N = 100)

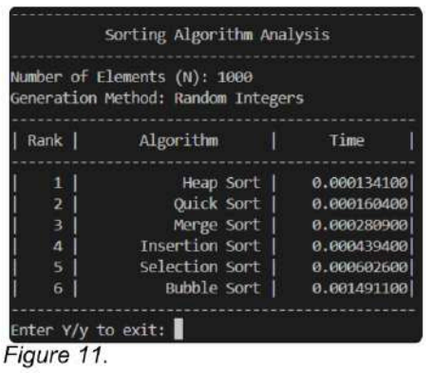


Figure 11.

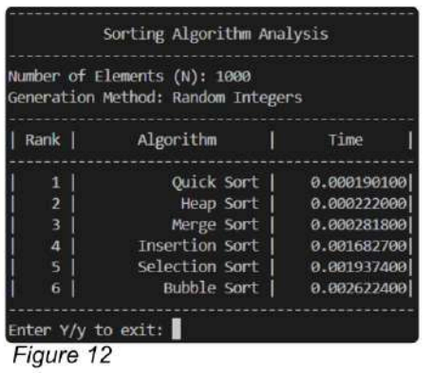


Figure 12

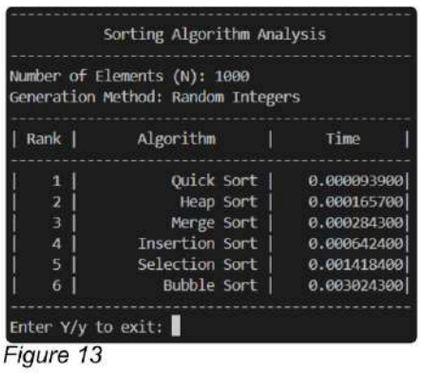


Figure 13

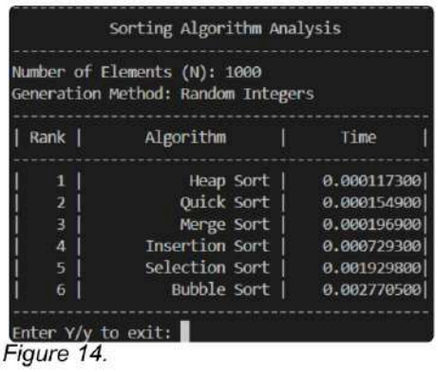


Figure 14.

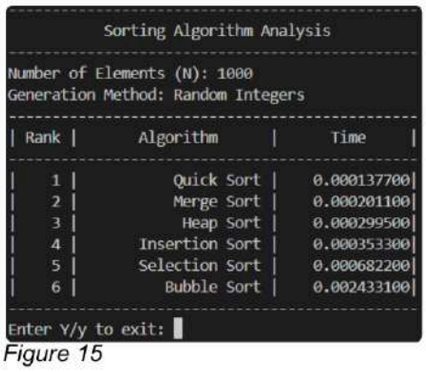


Figure 15

Figures 11–15. Execution Time of Sorting Algorithms on Random Integer Inputs (N = 1000)

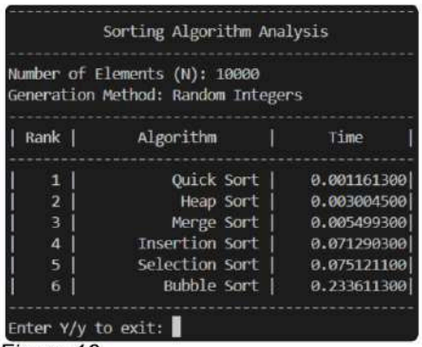


Figure 16.

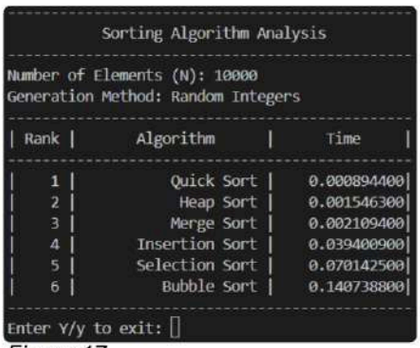


Figure 17.

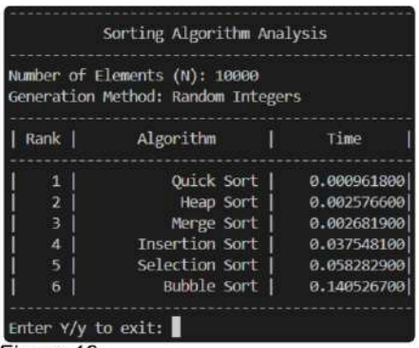


Figure 18.

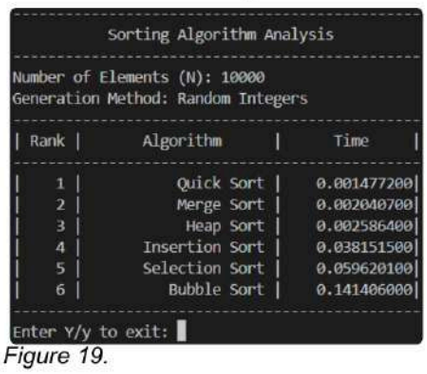


Figure 19.

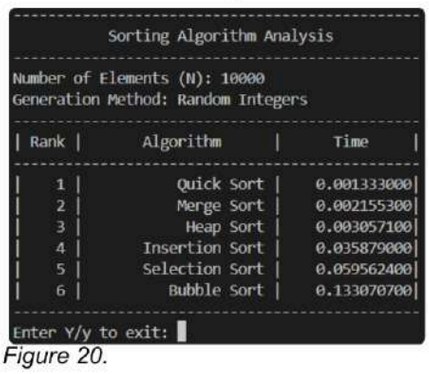


Figure 20.

Figures 16–20. Execution Time of Sorting Algorithms on Random Integer Inputs (N = 10000)

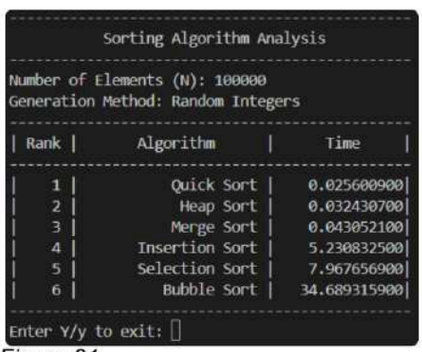


Figure 21.

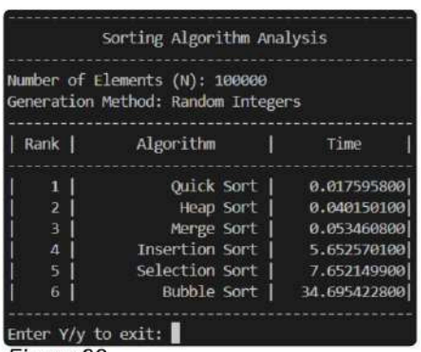


Figure 22.

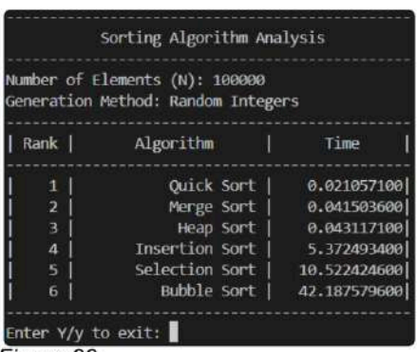


Figure 23.

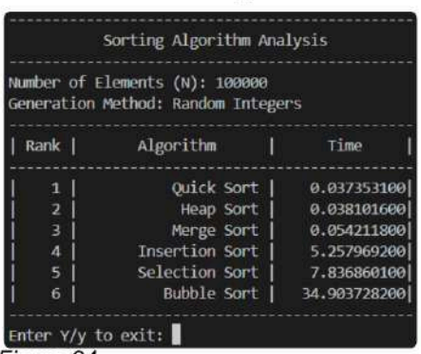


Figure 24.

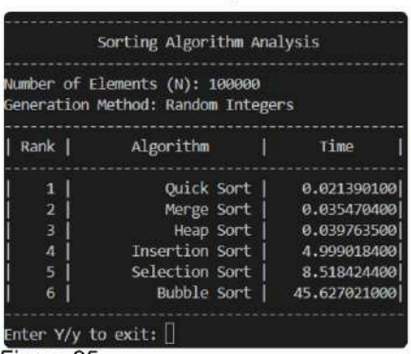


Figure 25.

Figures 21–25. Execution Time of Sorting Algorithms on Random Integer Inputs (N = 100000)

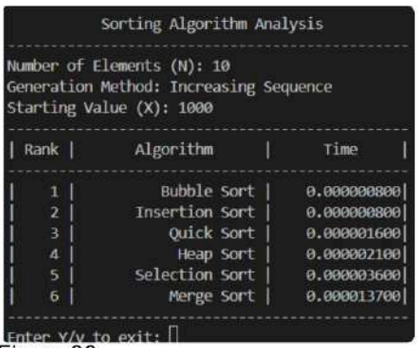


Figure 26.

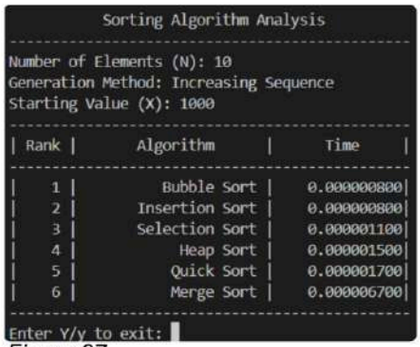


Figure 27.

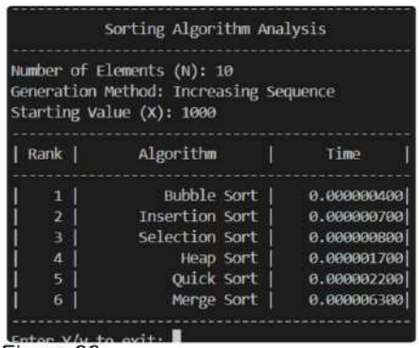


Figure 28.

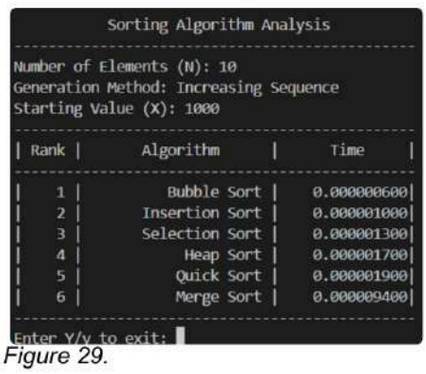


Figure 29.

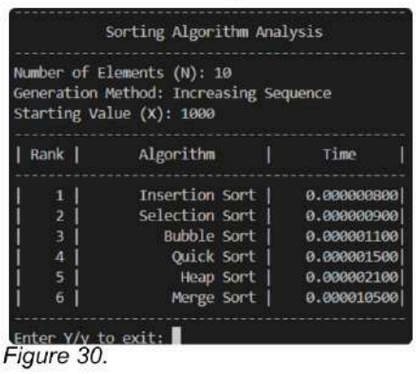


Figure 30.

Figures 26–30. Execution Time of Sorting Algorithms on Increasing Sequence Inputs (N = 10)

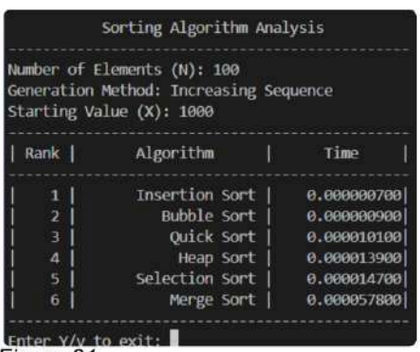


Figure 31.

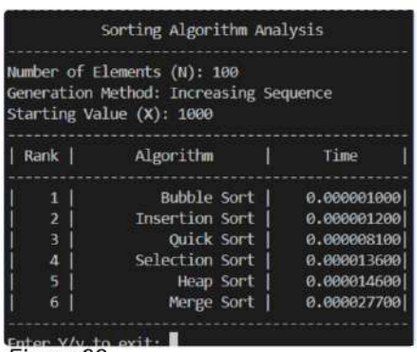


Figure 32.

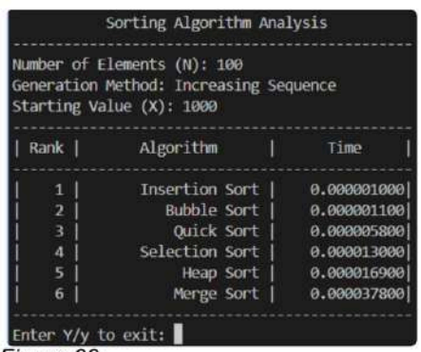


Figure 33.

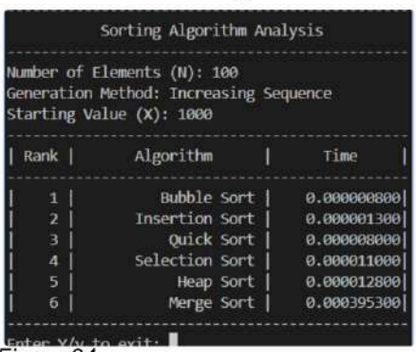


Figure 34.

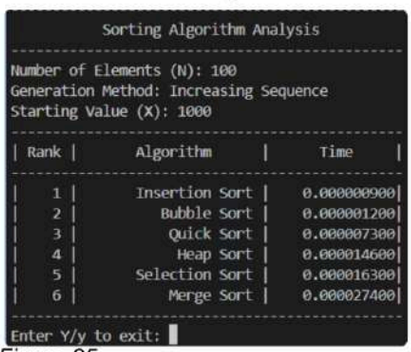


Figure 35.

Figures 31–35. Execution Time of Sorting Algorithms on Increasing Sequence Inputs (N = 100)

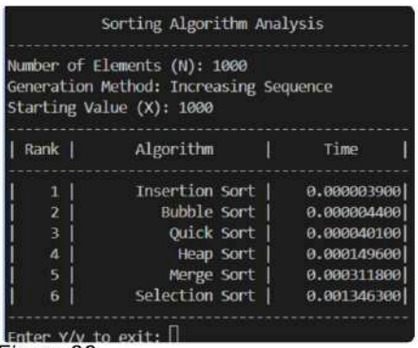


Figure 36.

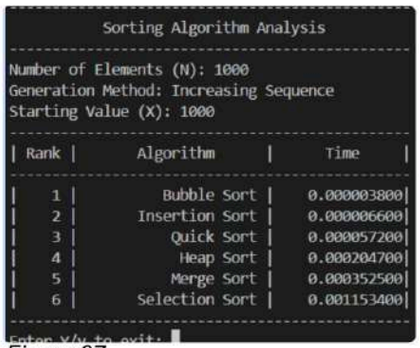


Figure 37.

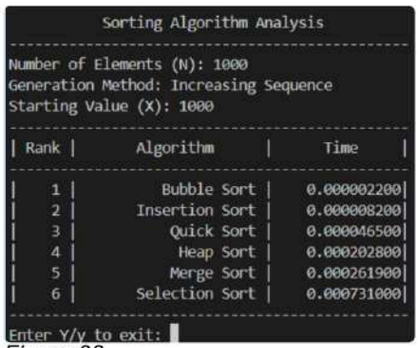


Figure 38.

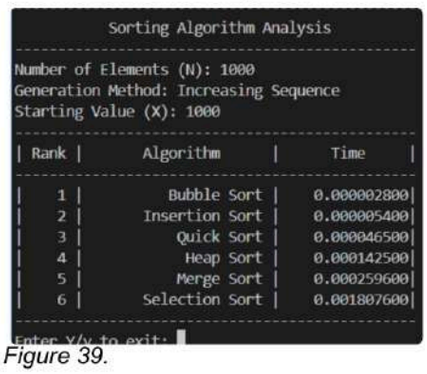


Figure 39.

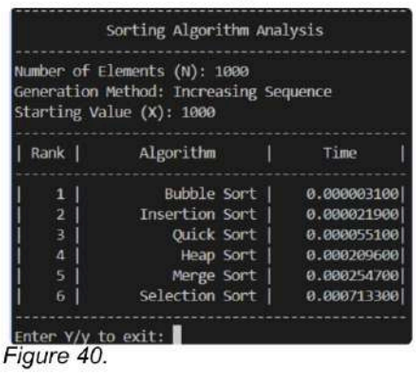


Figure 40.

Figures 36–40. Execution Time of Sorting Algorithms on Increasing Sequence Inputs (N = 1000)

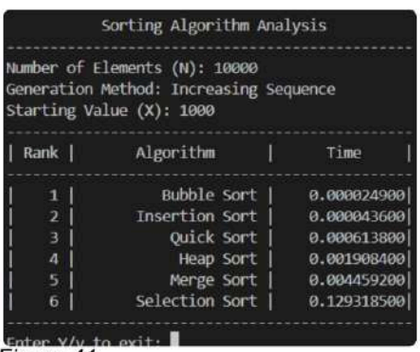


Figure 41.

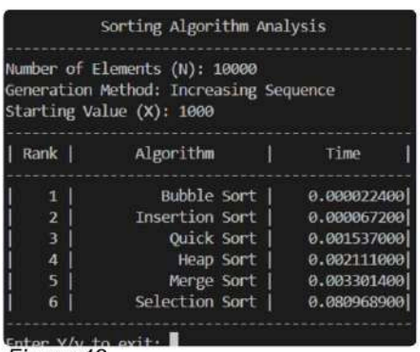


Figure 42.

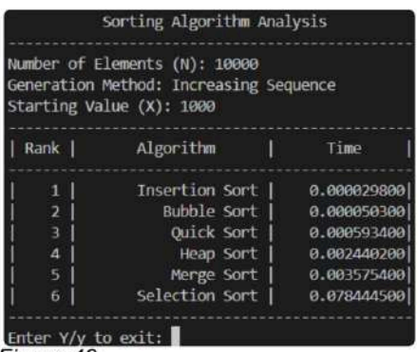


Figure 43.

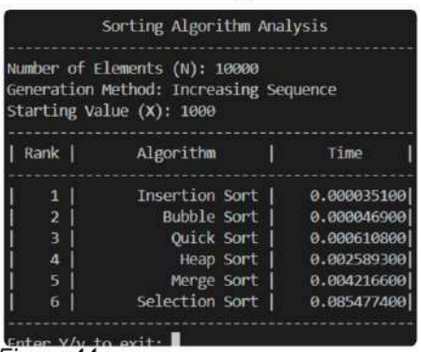


Figure 44.

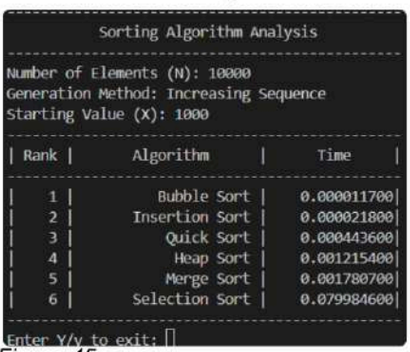


Figure 45.

Figures 41–45. Execution Time of Sorting Algorithms on Increasing Sequence Inputs (N = 10000)

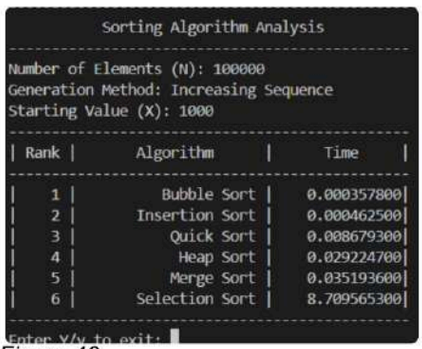


Figure 46.

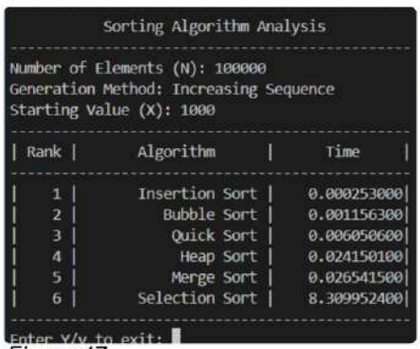


Figure 47.

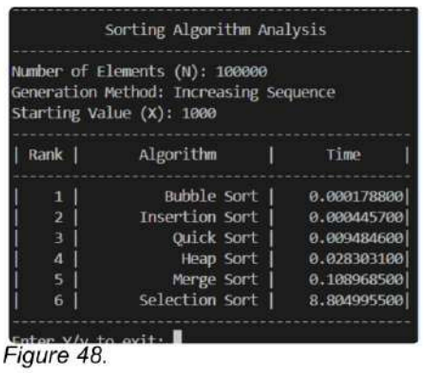


Figure 48.

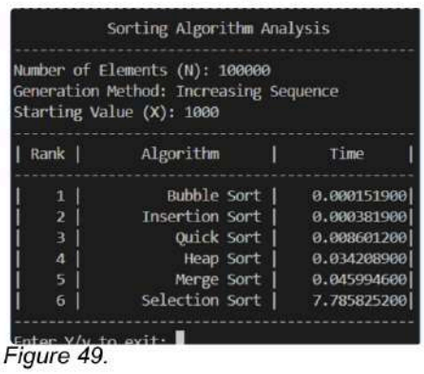


Figure 49.

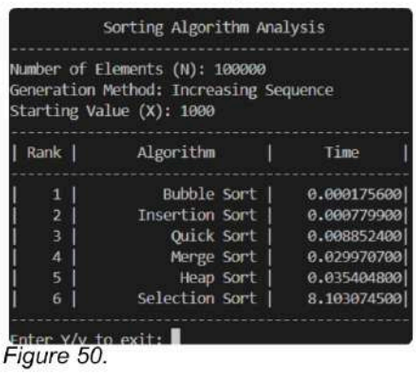


Figure 50.

Figures 46–50. Execution Time of Sorting Algorithms on Increasing Sequence Inputs (N = 100000)

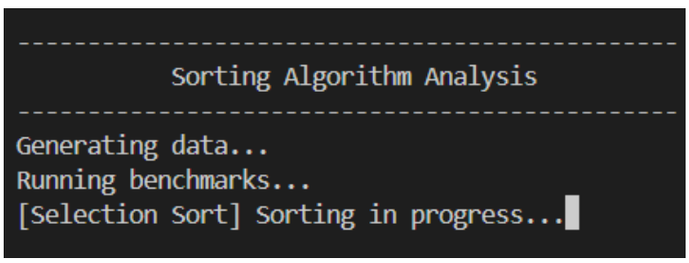


Figure 51. Screenshot of sorting algorithms in progress

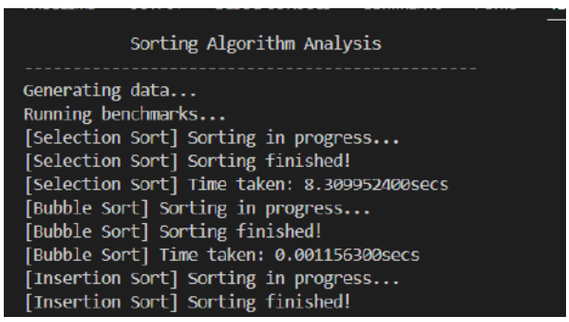


Figure 52. Screenshot of sorting statuses and results

IV. Results and Analysis

Specifications of the Computer Used to Run Tests

Device Name:	Lenovo Ideapad Slim 3i 15 (2022)	On-board RAM:	16 GB
System type:	64-bit OS, x64-based processor	Speed:	3200 MT/s
Processor:	12th Gen Intel Core i5-1235U	Hardware reserved:	276 MB
Base speed:	1.30 GHz		
Cores:	10		
Logical processors:	12		

Testing was performed on a Lenovo IdeaPad (i5-1235U, 16GB RAM) running Windows 11. Benchmarks were executed in VS Code with typical background apps, such as Microsoft Edge and Discord.

Result of the Experiment for Random Input, N = 10

Sorting Algorithm	Run 1	Run 2	Run 3	Run 4	Run 5	Average Time
Selection Sort	0.0000008	0.0000008	0.0000022	0.0000025	0.0000013	0.00000152
Bubble Sort	0.0000019	0.000001	0.0000021	0.0000006	0.0000007	0.00000126
Insertion Sort	0.0000008	0.0000007	0.000001	0.0000011	0.0000007	0.00000086
Merge Sort	0.0000073	0.0000063	0.0000118	0.0000059	0.0000125	0.00000876
Quick Sort	0.0000017	0.0000016	0.0000022	0.0000018	0.0000013	0.00000172
Heap Sort	0.0000017	0.0000021	0.0000019	0.0000015	0.0000018	0.0000018

For N = 10, Insertion Sort achieved the fastest average time, followed by Bubble Sort, Selection Sort, QuickSort, and Heap Sort, while Merge Sort was significantly slower. Merge Sort's worst run was twice as slow as its best, and Bubble/Selection Sort showed 3-4x variance between their fastest and slowest runs.

Result of the Experiment for Random Input, N = 100

Sorting Algorithm	Run 1	Run 2	Run 3	Run 4	Run 5	Average Time
Selection Sort	0.0000136	0.0000417	0.0000327	0.0000154	0.0000154	0.00002376
Bubble Sort	0.0000416	0.000072	0.0000549	0.0000305	0.0000292	0.00004564
Insertion Sort	0.0000133	0.0000204	0.0000127	0.0000128	0.0000071	0.00001326
Merge Sort	0.0000194	0.0000442	0.0000244	0.0000247	0.0000196	0.00002646
Quick Sort	0.0000094	0.0000117	0.0000089	0.0000122	0.0000084	0.00001012
Heap Sort	0.0000096	0.0000213	0.0000166	0.0000175	0.0000129	0.00001558

For N = 100 random inputs, Quick Sort demonstrated the fastest average execution time. Insertion Sort remained relatively fast, but slower than Quick Sort. Selection and Bubble Sort's times significantly increased. Merge Sort, while still slower than quick and heap sort, improved relative to the smaller dataset.

Result of the Experiment for Random Input, N = 1,000

Sorting Algorithm	Run 1	Run 2	Run 3	Run 4	Run 5	Average Time
Selection Sort	0.0006026	0.0019374	0.0014184	0.0019298	0.0006822	0.00131408
Bubble Sort	0.0014911	0.0026224	0.0030243	0.0027705	0.0024331	0.00246828
Insertion Sort	0.0004394	0.0016827	0.0006424	0.0007293	0.0003533	0.00076942
Merge Sort	0.0002809	0.0002818	0.0002843	0.0001969	0.0002011	0.00024900
Quick Sort	0.0001604	0.0001901	0.0000939	0.0001549	0.0001377	0.00014740
Heap Sort	0.0001341	0.0002220	0.0001657	0.0001173	0.0002995	0.00018772

For N = 1 000, Quick Sort showed the fastest average time, followed by Heap Sort and Merge Sort. These results show that recursive sorting algorithms outperform non recursive algorithms, hence, their efficiency for larger numbers of inputs. Bubble Sort achieved the slowest average time, followed by Selection Sort and Insertion Sort.

Result of the Experiment for Random Input, N = 10,000

Sorting Algorithm	Run 1	Run 2	Run 3	Run 4	Run 5	Average Time
Selection Sort	0.0751211	0.0701425	0.0582829	0.0596201	0.0595624	0.06454580
Bubble Sort	0.2336113	0.1407388	0.1405267	0.1414060	0.1330707	0.15787070
Insertion Sort	0.0712903	0.0394009	0.0375481	0.0381515	0.0358790	0.04445396
Merge Sort	0.0054993	0.0021094	0.0026819	0.0020407	0.0021553	0.00289732
Quick Sort	0.0011613	0.0008944	0.0009618	0.0014772	0.0013330	0.00116554
Heap Sort	0.0030045	0.0015463	0.0025766	0.0025864	0.0030571	0.00255418

Similar to N = 1 000, the table shows that Bubble, Selection, and Insertion Sort are significantly slower, while Quick Sort, Merge Sort, and Heap Sort perform much better. In this dataset, the average execution time of simpler sorting algorithms became slightly higher compared to the previous dataset, manifesting its inefficiency for a larger number of inputs.

Result of the Experiment for Random Input, N = 100,000

Sorting Algorithm	Run 1	Run 2	Run 3	Run 4	Run 5	Average Time
Selection Sort	7.9676569	7.6521499	10.5224246	7.8368601	8.5184244	8.49950318
Bubble Sort	34.6893159	34.6954228	42.1875796	34.9037282	45.627021	38.4206135
Insertion Sort	5.2308325	5.6525701	5.3724934	5.2579692	4.9990184	5.30257672
Merge Sort	0.0430521	0.0534608	0.0415036	0.0542118	0.0354704	0.04553974
Quick Sort	0.0256009	0.0175958	0.0210571	0.0373531	0.0213901	0.02459940
Heap Sort	0.0324307	0.0401501	0.0431171	0.0381016	0.0397635	0.03871260

For an input size of 100,000, Bubble, Selection, and Insertion Sort perform the worst, with Bubble Sort being the least efficient, showing significant differences from the recursive sorting algorithms. In contrast, Merge, Quick, and Heap Sort handle the data more effectively, with Quick Sort being the fastest.

Result of the Experiment for Sequenced Input, N = 10

Sorting Algorithm	Run 1	Run 2	Run 3	Run 4	Run 5	Average Time
Selection Sort	0.0000036	0.0000011	0.0000008	0.0000013	0.0000009	0.00000154
Bubble Sort	0.0000008	0.0000008	0.0000004	0.0000006	0.0000011	0.00000074
Insertion Sort	0.0000008	0.0000008	0.0000007	0.0000010	0.0000008	0.00000082
Merge Sort	0.0000137	0.0000067	0.0000063	0.0000094	0.0000105	0.00000932
Quick Sort	0.0000016	0.0000017	0.0000022	0.0000019	0.0000015	0.00000178
Heap Sort	0.0000021	0.0000015	0.0000017	0.0000017	0.0000021	0.00000182

The table shows the execution times of various sorting algorithms for a sequenced input of N=10. Bubble Sort is the fastest, while Heap Sort and Quick Sort take slightly longer. Differences are minimal due to the small input size.

Result of the Experiment for Sequenced Input, N = 100

Sorting Algorithm	Run 1	Run 2	Run 3	Run 4	Run 5	Average Time
Selection Sort	0.0000147	0.0000136	0.0000130	0.0000110	0.0000163	0.00001372
Bubble Sort	0.0000009	0.0000010	0.0000011	0.0000008	0.0000012	0.00000100
Insertion Sort	0.0000007	0.0000012	0.0000010	0.0000013	0.0000009	0.00000102
Merge Sort	0.0000578	0.0000277	0.0000378	0.0003953	0.0000274	0.00010920
Quick Sort	0.0000101	0.0000081	0.0000058	0.0000080	0.0000073	0.00000786
Heap Sort	0.0000139	0.0000146	0.0000169	0.0000128	0.0000146	0.00001456

For N = 100 and a sorted input, Bubble Sort and Insertion Sort were significantly faster, demonstrating their linear time complexity. Selection Sort's quadratic time complexity resulted in a much higher average time. Merge Sort exhibited the highest overhead, reflected in its slower average time and larger run-time variance. Quick Sort and Heap Sort performed better than Selection Sort, but were slower than Bubble and Insertion Sort.

Result of the Experiment for Sequenced Input, N = 1,000

Sorting Algorithm	Run 1	Run 2	Run 3	Run 4	Run 5	Average Time
Selection Sort	0.0013463	0.0011534	0.0007310	0.0018076	0.0007133	0.00115032
Bubble Sort	0.0000044	0.0000038	0.0000022	0.0000028	0.0000031	0.00000326
Insertion Sort	0.0000039	0.0000066	0.0000082	0.0000054	0.0000219	0.00000920
Merge Sort	0.0003118	0.0003525	0.0002619	0.0002596	0.0002547	0.00028810
Quick Sort	0.0000401	0.0000572	0.0000465	0.0000465	0.0000551	0.00004908
Heap Sort	0.0001496	0.0002047	0.0002028	0.0001425	0.0002096	0.00018184

For N = 1,000, Bubble Sort was the fastest, confirming its linear time for sorted data. Insertion Sort also performed well, though with some run-time variability. Selection Sort was significantly slower, due to its quadratic time complexity. Merge, Quick, and Heap sorts had higher average times. Quick Sort exhibited the most consistent run-times amongst the more complex algorithms.

Result of the Experiment for Sequenced Input, N = 10,000

Sorting Algorithm	Run 1	Run 2	Run 3	Run 4	Run 5	Average Time
Selection Sort	0.1293185	0.0904698	0.0809689	0.0784445	0.0799846	0.09183726
Bubble Sort	0.0000249	0.0000378	0.0000224	0.0000503	0.0000117	0.00002942
Insertion Sort	0.0000436	0.0000433	0.0000672	0.0000298	0.0000218	0.00004114
Merge Sort	0.0044592	0.0094121	0.0033014	0.0035754	0.0017807	0.00450576
Quick Sort	0.0006138	0.0003443	0.0015370	0.0005934	0.0004436	0.00070642
Heap Sort	0.0019084	0.0019543	0.0021110	0.0024402	0.0012154	0.00192586

For N = 10,000, Bubble Sort and Insertion Sort were significantly faster. Selection Sort was extremely slow due to its quadratic complexity. Merge, Quick, and Heap sorts were slower due to overhead. Bubble and Insertion sorts demonstrate the extreme efficiency of linear runtime on sorted data.

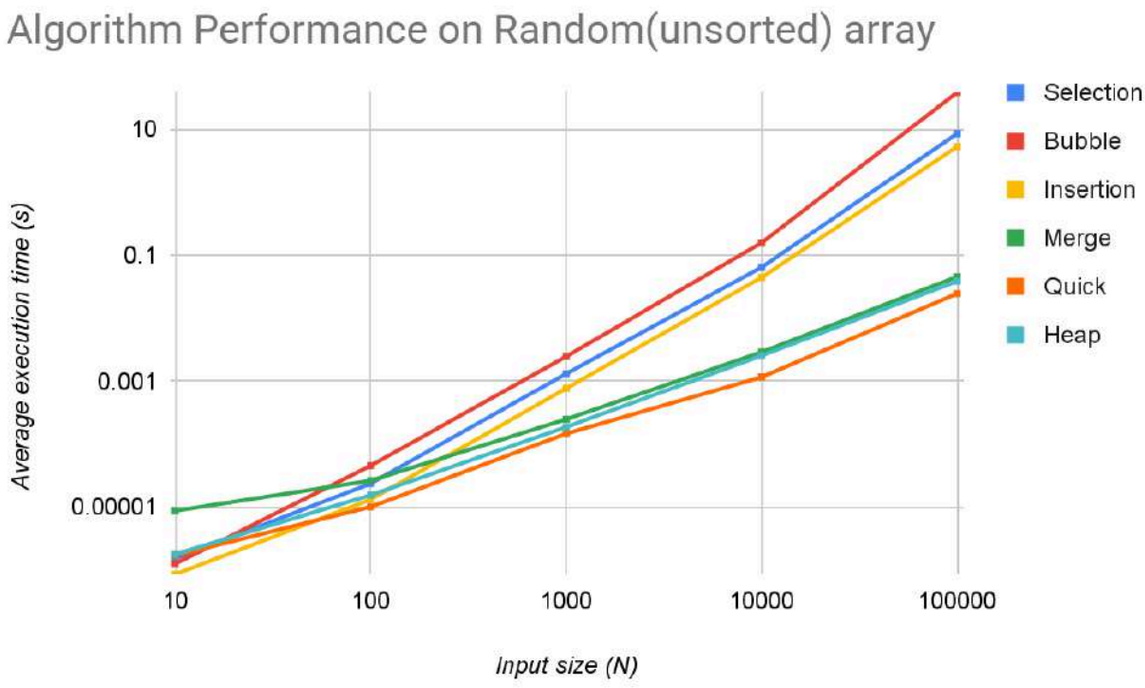
Result of the Experiment for Sequenced Input, N = 100,000

Sorting Algorithm	Run 1	Run 2	Run 3	Run 4	Run 5	Average Time
Selection Sort	8.7095653	8.3099524	8.8049955	7.7858252	8.1030745	8.34268258
Bubble Sort	0.0003578	0.0011563	0.0001788	0.0001519	0.0001756	0.00040408
Insertion Sort	0.0004625	0.0002530	0.0004457	0.0003819	0.0007799	0.00046460
Merge Sort	0.0351936	0.0265415	0.1089685	0.0459946	0.0299707	0.04933378
Quick Sort	0.0086793	0.0060506	0.0094846	0.0086012	0.0088524	0.00833362
Heap Sort	0.0292247	0.0241501	0.0283031	0.0342089	0.0354048	0.03025832

For N = 100,000, Bubble and Insertion sorts were exceptionally fast, due to their linear time efficiency. Selection Sort's quadratic complexity rendered it extremely slow. Merge, Quick, and Heap sorts were significantly slower, showing the impact of their overhead. Bubble and Insertion sort, once again, illustrated the dominance of linear time algorithms for pre-sorted data at a large scale.

Average Running Time for an Input Array that is Random

N	Selection Sort	Bubble Sort	Insertion Sort	Merge Sort	Quick Sort	Heap Sort
10	0.00000152	0.00000126	0.00000086	0.00000876	0.00000172	0.00000180
100	0.00002376	0.00004564	0.00001326	0.00002646	0.00001012	0.00001558
1000	0.00131408	0.00246828	0.00076942	0.00024900	0.00014740	0.00018772
10,000	0.06454580	0.15787070	0.04445396	0.00289732	0.00116554	0.00255418
100,000	8.49950318	38.4206135	5.30257672	0.04553974	0.02459940	0.03871260



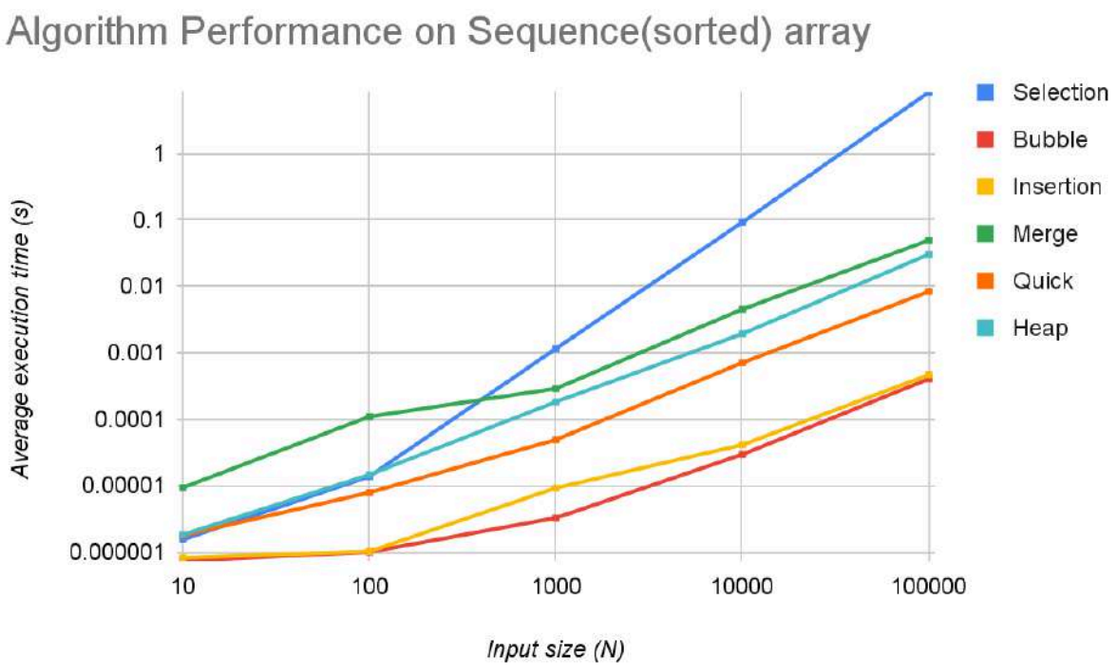
For small 'N' (10-100), Insertion Sort performs well initially due to its simplicity, but Quick Sort becomes faster by N=100. The effect of quadratic time complexity is seen in the increasing slowness of Selection and Bubble Sort. As N increases to 1,000 and beyond, Quick and Merge Sort scale better, with Quick Sort slightly faster. Heap Sort remains reasonably efficient. However, Insertion Sort's performance decreases, while Selection and Bubble Sort become slower, limiting their use for large random datasets. The increasing time difference between efficient and inefficient algorithms shows the importance of algorithm selection for large-scale sorting. Quick Sort and Merge Sort have similar

performance, with Quick Sort slightly faster, and both perform considerably better than the other sorts as N increases.

Average Running Time for an Input Array that is Sorted

N	Selection Sort	Bubble Sort	Insertion Sort	Merge Sort	Quick Sort	Heap Sort
10	0.00000154	0.00000074	0.00000082	0.00000932	0.00000178	0.00000182
100	0.00001372	0.000001	0.00000102	0.0001092	0.00000786	0.00001456
1000	0.00115032	0.00000326	0.0000092	0.0002881	0.00004908	0.00018184
10,000	0.09183726	0.00002942	0.00004114	0.00450576	0.00070642	0.00192586
100,000	8.34268258	0.00040408	0.0004646	0.04933378	0.00833362	0.03025832

For a sorted input array, Bubble Sort and Insertion Sort demonstrate high efficiency, maintaining near-constant time complexity due to their linear nature. Selection Sort, however, exhibits a significant increase in runtime as the input size N grows, clearly illustrating the impact of its quadratic time complexity. Merge Sort, Quick Sort, and Heap Sort, while faster than Selection Sort, are notably slower than Bubble and Insertion Sort, reflecting the overhead associated with their algorithmic structures. The difference in scaling is observed as 'N' increases, with Selection Sort's runtime rising sharply, while Bubble and Insertion Sort remain relatively stable. Quick Sort and Merge Sort show moderate scaling, but are consistently slower than the linear time sorts. The data highlights the profound effect of algorithm choice on performance when dealing with already-sorted data.



V. Conclusion

The experimental results clearly demonstrate that the performance of sorting algorithms is heavily influenced by both input size and input order. While all experiments were conducted on the same hardware, eliminating cross-machine variability, the computer's internal state (background processes and memory allocation patterns) could still introduce minor fluctuations in timing measurements across runs.

When processing randomly ordered data, divide-and-conquer algorithms like Quick Sort dominate for larger datasets (N ≥ 100) due to their superior O(n log n) average-case complexity and memory usage efficiency, with Heap Sort following closely due to its in-place operation advantage over Merge Sort, which uses temporary array. In contrast, for sequenced data, simpler algorithms like Bubble Sort and Insertion Sort consistently outperform more complex ones due to their ability to exploit existing order - Bubble Sort achieves its best-case performance through early termination, while Insertion Sort benefits from reduced swap operations.

Notably, the relative performance of algorithms shifts dramatically based on input size. For very small datasets (N = 10), even with random ordering, Insertion Sort becomes the fastest option as its $O(n^2)$ complexity proves negligible at this scale. However, Selection Sort consistently underperforms in all scenarios due to its rigid implementation that fails to adapt to either favorable input patterns or small sizes. Merge Sort, while stable, suffers from significant overhead in small datasets and only becomes competitive for larger N values.

These findings highlight the importance of selecting sorting algorithms based on specific use cases. Bubble Sort and Insertion Sort are ideal choices for nearly-sorted or very small datasets, while QuickSort should be the default for general-purpose sorting of larger, randomly ordered data. Selection sort should be avoided in practice and consider hybrid approaches that combine multiple algorithms to optimize performance across different input characteristics. The results emphasize that there is no universally superior sorting algorithm - optimal performance requires matching algorithmic strengths to the particular characteristics of the input data.

VI. Challenges

1. **Precision in time measurement.** Measuring execution time accurately is crucial in empirical analysis, especially when sorting algorithms operate within microseconds. The clock() function from <time.h> provides a simple way to measure CPU time, but it has limitations. To improve precision, using clock_gettime(CLOCK_MONOTONIC, &ts) on Linux or QueryPerformanceCounter() on Windows can provide higher resolution timing. These functions measure real-time elapsed duration and are less affected by system scheduling or background processes.

2. **Handling user input robustly.** Validating user input is essential to ensure the program runs smoothly and does not encounter unexpected behavior. If a user mistakenly enters an invalid number, such as a negative value for N or X, the program must detect this and prompt the user to re-enter valid input rather than proceeding with an invalid state.

3. **Premature Program Termination with large values of N**. The program was unable to complete quicksort on an array where N = 100 000 and initial value X = 100 000. It was surmised that this may be caused by the current pivot selection strategy, which takes the last element as the pivot. Therefore, to solve this, the program was refactored to use the median-of-three pivot selection strategy instead and re-run while at same time reducing the initial value X = 1000 for good measure. As expected, the program did not run into any issues this time.

VII. Participation

1. Deanne Clarice Bea
    - Contributed in the documentation of the program describing the code structure
    - Conducted all experiments on one computer to exclude cross-machine variability, including screenshots of each test result
    - Analyzed and discussed some of the results of the experiment
    - Constructed a conclusion based on the results and analysis
2. Michael Xavier Canonizado
    - Programmer
    - Contributed in the documentation of the program describing the code structure
    - Described the challenges encountered during programming
3. John Patrick Drio
    - Assisted in inputting results of the experiment
    - Contributed to observations and analysis of the results
    - Designed the graphical representation of average execution times
    - Described the challenge encountered during experimentation
4. Christian Morga
    - Contributed in adding data in graph analysis.
    - Provided some explanations about the data provided.
    - Computed each sorting algorithm’s average.
5. Simon Angelo Narvaez
    - Contributed by adding the observations and analysis of the results for each test.
    - Assisted in attaching the test run findings.