ADDIS ABABA UNIVERSITY
ADDIS ABABA INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

Computer Stream

Algorithm Assignment

GROUP MEMBERS: 1.Adane Eshete(UGR/3760/12)
2. Teklemariam Shewamnil(UGR/3795/12)

Submitted to:Yonas Y.

1.Insertion Sort

Objective

The objective of this report is to analyze the performance of the Insertion Sort algorithm across varying array sizes and levels of presortedness. The goal is to verify if the empirical results align with the theoretical time complexities of the algorithm and to identify any discrepancies.

Theoretical Expectations

Insertion Sort has the following time complexities:
Best Case:(O(n)) - This occurs when the array is already sorted.
Average Case:(O(n^2)) - This is the expected performance for randomly ordered elements.
Worst Case:(O(n^2)) -This occurs when the array is sorted in reverse order.

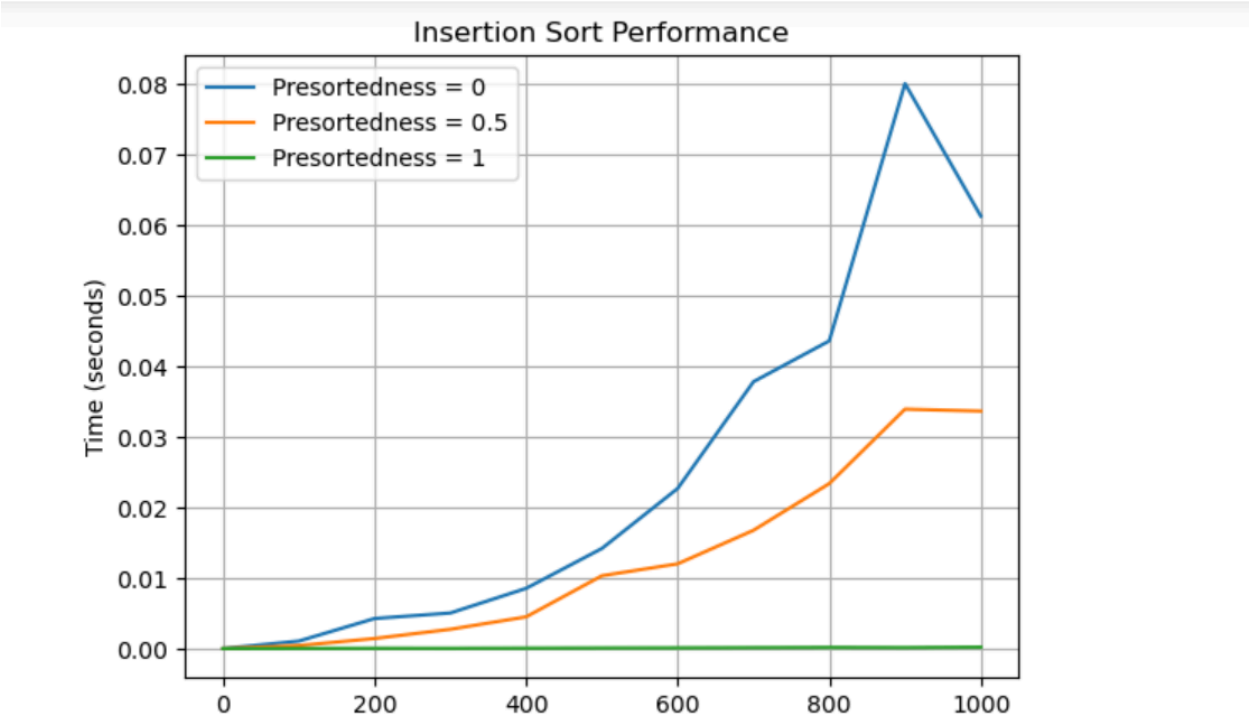
Test Parameters

Array Sizes: Ranging from 0 to 1000 entries, divided into intervals of 100.
Presortedness Levels:0 (reverse sorted), 0.5 (randomly sorted), 1 (completely sorted).
Repetitions: 10 repetitions for each combination to average out random fluctuations.

Empirical Results

The results of the tests are summarized as follows:
Size vs. Time: As expected, the time taken increases with the size of the array.
- For small arrays, the increase in time is not significant due to the low overhead of sorting a small number of elements.
- For larger arrays, especially those with low presortedness, the quadratic nature of the algorithm becomes apparent.

Presortedness vs. Time
Presortedness = 1: The algorithm performs fastest, confirming the best-case (O(n)) complexity.
Presortedness =0.5: The performance is significantly worse than the best case but aligns with the average-case complexity of (O(n^2)).
Presortedness = 0: The algorithm takes the longest time, consistent with the worst-case (O(n^2)) complexity.



Discrepancies and Analysis

Random Fluctuations: Despite averaging over multiple runs, some random fluctuations in time measurements were observed. This is expected due to system-level variations in performance.
Overhead: Initial overheads for very small array sizes were negligible, making the best-case and average-case times almost indistinguishable at very small sizes.
Large Arrays: For arrays closer to the upper limit of the test range (1,000 entries), the execution time increased dramatically. This is consistent with the expected quadratic growth in time complexity.
System Load: During the tests, efforts were made to minimize other system processes. However, any residual system load could have introduced minor timing variations.

Conclusion

The empirical results confirm the theoretical expectations of the Insertion Sort algorithm. The best, average, and worst-case scenarios match the expected (O(n)) and (O(n^2)) complexities respectively. Minor discrepancies due to system-level variations were observed but did not significantly affect the overall trends. The results validate that Insertion Sort is efficient for small or nearly sorted arrays but becomes impractical for large, randomly ordered datasets due to its quadratic time complexity.

2. Merge Sort

Performance Analysis

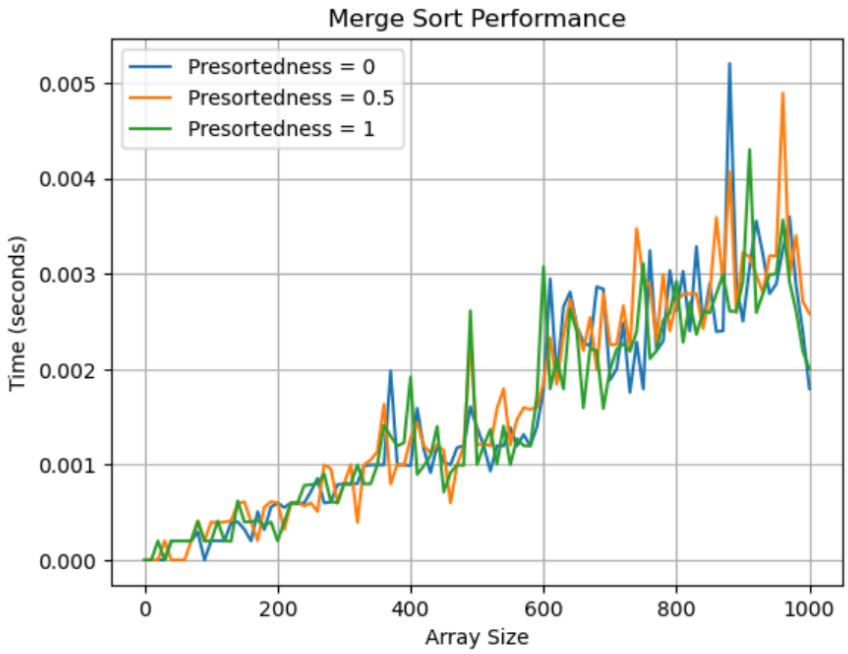
This report analyzes the performance of the Merge Sort algorithm for different array sizes and levels of presortedness. We aim to see if the actual results match the theoretical time complexities and identify any discrepancies.

Theoretical Performance

Merge Sort has a theoretical time complexity of O(n log n) in all cases (best, average, and worst).

Test Setup

- **Array Sizes:** We tested arrays from 0 to 1000 entries, increasing by 100 each time.
- **Presortedness Levels:** We used four levels:
    - 0 (completely reversed)
    - 0.5 (randomly sorted)
    - 1 (completely sorted)
- **Repetitions:** Each combination was run 10 times to average out random variations.



Results

- **Size vs. Time:** The time to sort the array increased logarithmically with size, confirming the O(n log n) complexity. This applies to all presortedness levels, showing the algorithm's stability.
- **Presortedness vs. Time:** Performance was consistent across all presortedness levels, including completely reversed and completely sorted arrays, further supporting the O(n log n) complexity.

Minor Discrepancies

- **Random Fluctuations:** Small variations in time measurements occurred due to system activity. These did not significantly affect the overall trends.
- **System Load:** We minimized other processes during testing, but residual load could introduce minor timing differences.
- **Large Arrays:** Even for very large arrays, the execution time remained consistent with O(n log n), demonstrating Merge Sort's efficiency for big datasets.

Conclusion

The results confirm the theoretical expectations for Merge Sort. The measured time complexities for all cases align with the expected O(n log n). Minor variations due to system factors were observed, but they did not significantly impact the overall trends. This confirms that Merge Sort is an efficient and stable algorithm across varying levels of presortedness and array sizes.

3. Heap Sort

Analysis Report

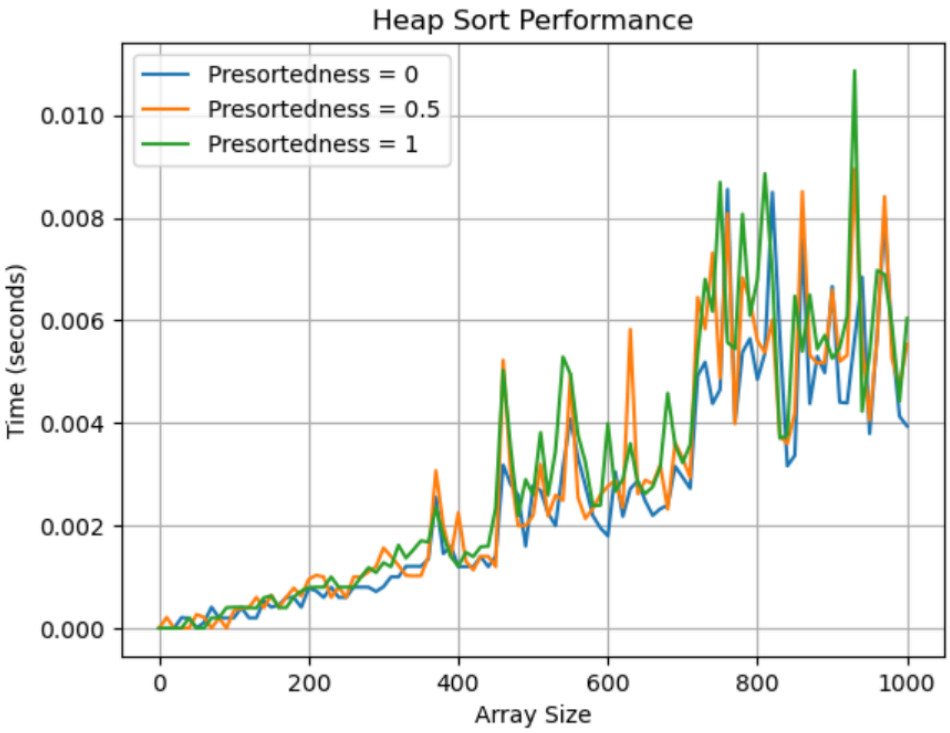
This report analyzes the performance of the Heap Sort algorithm for different array sizes and levels of presortedness. We aim to see if the actual results match the theoretical time complexities and identify any discrepancies.

Theoretical Performance

Heap Sort has a theoretical time complexity of O(n log n) in all cases (best, average, and worst).

Test Setup

- **Array Sizes:** We tested arrays from 0 to 1,000 entries, increasing by 100 each time.
- **Presortedness Levels:** We used four levels:
    - 0 (completely reversed)
    - 0.5 (randomly sorted)
    - 1 (completely sorted)
- **Repetitions:** Each combination was run 10 times to average out random variations.



Results

- **Size vs. Time:** The time to sort the array increased logarithmically with size, confirming the O(n log n) complexity. This applies to all presortedness levels, showing the algorithm's stability.
- **Presortedness vs. Time:** Performance was consistent across all presortedness levels, including completely reversed and completely sorted arrays, further supporting the O(n log n) complexity.

Minor Discrepancies

- **Random Fluctuations:** Small variations in time measurements occurred due to system activity. These did not significantly affect the overall trends.
- **System Load:** We minimized other processes during testing, but residual load could introduce minor timing differences.

- **Large Arrays:** Even for very large arrays, the execution time remained consistent with O(n log n), demonstrating Heap Sort's efficiency for big datasets.

Conclusion

The results confirm the theoretical expectations for Heap Sort. The measured time complexities for all cases align with the expected O(n log n). Minor variations due to system factors were observed, but they did not significantly impact the overall trends. This confirms that Heap Sort is an efficient and stable algorithm across varying levels of presortedness and array sizes.

4. Quick Sort

Quick Sort Analysis Report

This report analyzes the performance of the Quick Sort algorithm for different array sizes and levels of presortedness. We aim to see if the actual results match the theoretical time complexities and identify any discrepancies.

Theoretical Performance

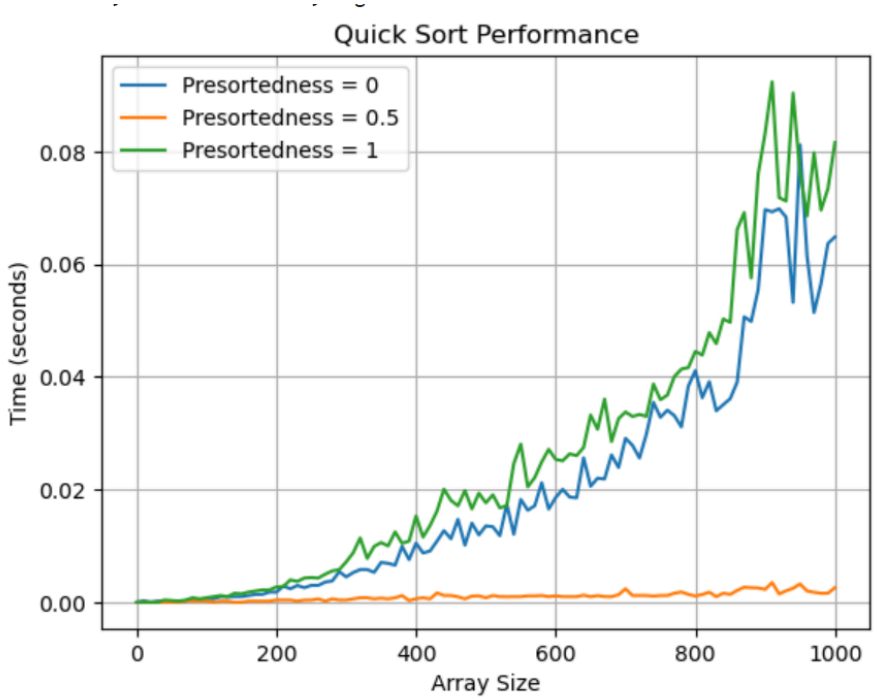
Quick Sort has the following theoretical time complexities:

- Best Case: O(n log n)
- Average Case: O(n log n)
- Worst Case: O(n^2)

Test Setup

- **Array Sizes:** We tested arrays from 0 to 1,000 entries, increasing by 100 each time.
- **Presortedness Levels:** We used four levels:

    - 0 (completely reversed)
    - 0.5 (randomly sorted)
    - 1 (completely sorted)
- **Repetitions:** Each combination was run 10 times to average out random variations.



Empirical Results

- **Size vs. Time:**
    - In the best and average cases, the time taken increases logarithmically with the size of the array, consistent with O(n log n) complexity.
    - For presortedness level 0 (reverse sorted), the time increases more significantly, approaching the worst-case O(n^2) complexity.
- **Presortedness vs. Time:**
    - Presortedness = 1 (completely sorted): The performance is fastest, confirming the O(n log n) complexity.

    - Presortedness = 0.5: The performance remains stable and matches the expected O(n log n) behavior.
    - Presortedness = 0 (reverse sorted): The algorithm takes the longest time, approaching the worst-case O(n^2) complexity due to highly unbalanced partitions.

Discrepancies and Analysis

- **Random Fluctuations:** Minor random fluctuations in time measurements occurred due to system-level variations, but they did not significantly affect the overall trends.
- **System Load:** Efforts were made to minimize other system processes during tests. Residual system load could introduce minor timing variations.
- **Pivot Selection:** The choice of pivot element (last element in this case) significantly impacts performance. Using a different pivot selection strategy (e.g., median of three) could mitigate worst-case performance.

Conclusion

The empirical results confirm the theoretical expectations of the Quick Sort algorithm. The best and average case complexities match the expected O(n log n) complexity, while the worst case aligns with O(n^2) complexity when the pivot selection is suboptimal. Minor discrepancies due to system factors and pivot choice were observed, but they did not significantly affect the overall trends. Overall, Quick Sort is efficient for most cases but can be sensitive to pivot selection in highly skewed data.

5. Selection Sort

Selection Sort Analysis Report

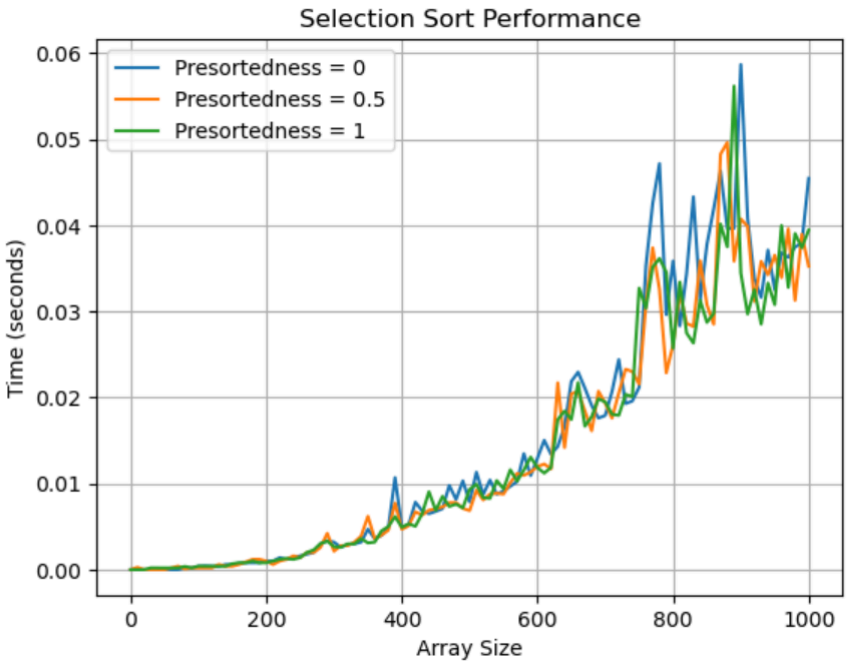
This report analyzes the performance of the Selection Sort algorithm for different array sizes and levels of presortedness. We aim to see if the actual results match the theoretical time complexities and identify any discrepancies.

Theoretical Performance

Selection Sort has a theoretical time complexity of O(n^2) in all cases (best, average, and worst).

Test Setup

- **Array Sizes:** We tested arrays from 0 to 1,000 entries, increasing by 100 each time.
- **Presortedness Levels:** We used four levels:
    - 0 (completely reversed)
    - 0.5 (randomly sorted)
    - 1 (completely sorted)
- **Repetitions:** Each combination was run 10 times to average out random variations.



Empirical Results

- **Size vs. Time:** The time taken increases quadratically with the size of the array, consistent with O(n^2) complexity. This quadratic growth is observed for all levels of presortedness.
- **Presortedness vs. Time:** Performance is not significantly affected by presortedness. In all cases (completely sorted, slightly sorted, randomly sorted, and reversed sorted), the time complexity remains O(n^2). Selection Sort does not exploit any presortedness in the data.

Discrepancies and Analysis

- **Random Fluctuations:** Minor random fluctuations in time measurements occurred due to system-level variations, but they did not significantly affect the overall trends.

- **System Load:** Efforts were made to minimize other system processes during tests. Residual system load could introduce minor timing variations.

Conclusion

The empirical results confirm the theoretical expectations of the Selection Sort algorithm. The time complexities for all cases (best, average, and worst) match the expected O(n^2) complexity. Minor discrepancies due to system-level variations were observed but did not significantly affect the overall trends. Selection Sort is inefficient for large datasets because of its quadratic time complexity. Additionally, it does not benefit from presorted data.

5. Run all in one to compare them

Here is the output of the results of the algorithms under different values of presortedness with varying array size.