Activity No. 14.1	
ALGORITHM COMPLEXITY	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 11/6/2025
Section: CPE21S4	**Date Submitted:** 11/6/2025
Name(s): Dano Alvin John Cruz, Axl Waltz E.	**Instructor:** Engr. Jimlord Quejado

A. Output(s) and Observation(s)

Table 14-1. Best- and Worst-Case Analysis using Theoretical Tools

	Best Case	**Worst Case**	**Analysis**
Algorithm 1: Bubble Sort	O(n)	O(n^2)	I used a bubble sort algorithm, which has a best-case time complexity of **O(n)** and a worst-case of **O(n^2)** (GeeksforGeeks, 2025)**,** the total operation increases rapidly as the input grows. Given by the example N = 100; 1000; 10000, the estimated operation counts in best case are 100; 1000; 10000 work total work units respectively and as for the worse case scenario its 100^2 = 10000, 10000^2 = 1000000; 10000^2 = 100000000 total work units respectively. This shows that the number of operations grows quadratically with N meaning that even a small increase in input size can cause a large increase in runtime. Making bubble sort inefficient on larger datasets because of its repeated pairwire comparisons and swapping.
Algorithm 2: Merge Sort	O(n log n)	O(n log n)	Merge sort algorithm, which consistently performs at O(nlogn) for both best and worst cases (GeeksforGeeks, 2024), the growth rate is much slower than quadratic algorithms. For N = 100; 1000; 10000, the estimated operation counts are log2(100) = 6.64; log2(1000) = 9.97; log2(10000) = 13.29 total work units respectively. This indicates that the total number of operations grows slowly even when the input size increases by 10, Merge sort remains efficient and scalable because of its characteristic of divide-and-conquer, limiting the number of necessary comparisons and merges.

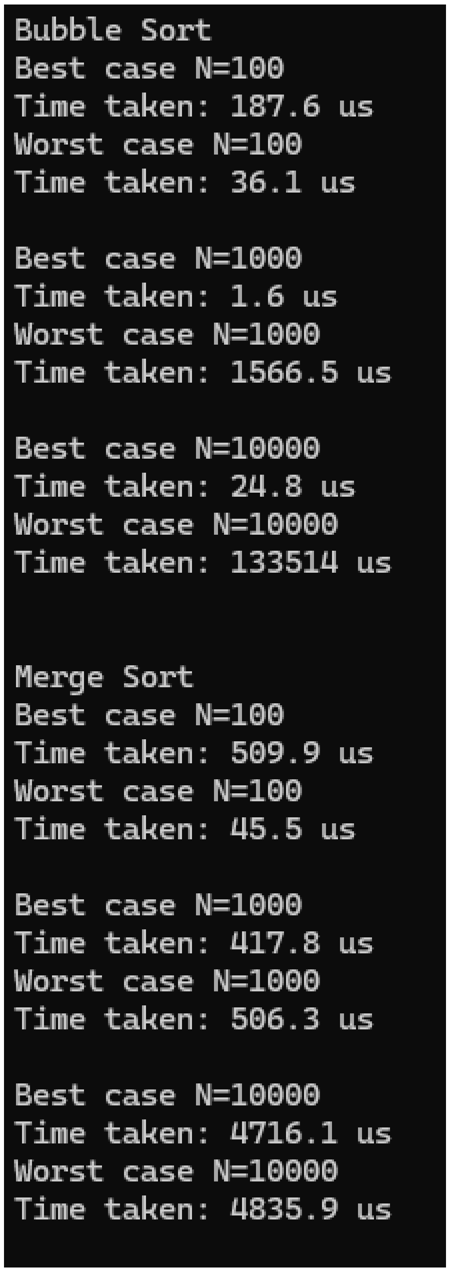
Table 14-2. Best- and Worst-Case Analysis using Experimental Tools

	Algorithm 1: Bubble Sort			**Algorithm 2:** Merge Sort		
n	100	1000	1000	100	1000	1000
Best Case	187.6 us	1.6 us	24.8 us	509.9 us	417.8 us	4716.1 us
Worst Case	36.1 us	1566.5 us	133514 us	45.5 us	506.3 us	4835.9 us

The reason why I used bubble sort and merge sort algorithms is because they represent two constructive approaches to sorting, basically it's the battle between “**simple iterative method vs a divide-conquer strategy**”. When comparing Bubble sort and Merge sort, the main distinction lies in their time complexity behavior and scalability as the input size increases. Bubble sort operates with repeated pairwise comparisons and swapping which causes the performance to deteriorate rapidly on larger datasets. Its growth follows a quadratic pattern O(n) and O(n^2) which means that the total number of operations increases dramatically as n becomes larger. However, Merge Sort utilizes a divide-and-conquer approach that breaks down the array into smaller parts and merges them efficiently, maintaining an consistent O(nlogn) complexity even in its worst case.

In the best case, Bubble Sort can perform in O(n) tiem when the dataset is **ALREADY SORTED,** while Merge Sort **REMAINS** O(nlogn) regardless of the input arrangement. However, in more practical scenarios, Merge Sort predictable performance is more advantageous especially for larger scale data, The additional memory usage in Merge sort is a small trade off for its significantly reduced runtime.

Overall, the experimental and theoretical analyses both confirm that Merge Sort is superior in efficiency, stability, and scalability, while Bubble Sort, though simple and easy to implement, is better suited for small or educational datasets. As the input size increases (N = 100, 1,000, 10,000), Bubble Sort’s execution time grows exponentially, while Merge Sort’s time increases gradually and remains manageable. Therefore, from both a computational and practical standpoint, Merge Sort outperforms Bubble Sort in nearly all aspects of performance and complexity.



```
Bubble Sort
Best case N=100
Time taken: 187.6 us
Worst case N=100
Time taken: 36.1 us

Best case N=1000
Time taken: 1.6 us
Worst case N=1000
Time taken: 1566.5 us

Best case N=10000
Time taken: 24.8 us
Worst case N=10000
Time taken: 133514 us


Merge Sort
Best case N=100
Time taken: 509.9 us
Worst case N=100
Time taken: 45.5 us

Best case N=1000
Time taken: 417.8 us
Worst case N=1000
Time taken: 506.3 us

Best case N=10000
Time taken: 4716.1 us
Worst case N=10000
Time taken: 4835.9 us
```

B. Supplementary Task

Linear Search and Binary Search “sequential traversal vs. divide-and-conquer searching.”

Linear Search

Pseudocode Analysis

```
FUNCTION linearSearch(array, target)
  FOR i = 0 to (length of array) - 1 DO
      IF array[i] = target THEN
          RETURN i
      ELSE
          RETURN -1
END FUNCTION
```

Linear Search is a straightforward searching technique that scans each element in the array until the target value is found or the entire list has been traversed. The pseudocode has a single loop that iterates through all elements and compares each one to the target value. If a match is found, the index is returned, else, the algorithm proceeds until the end. The time complexity of Linear Search is **O(1)** in the best case, when the element is found immediately at the start of the list, and **O(n)** in the worst case, when the target is located at the end or not present at all. In terms of space complexity, Linear Search uses a constant amount of memory, resulting in **O(1)** space usage since it only requires variables for indexing and comparison.

Time and Space Complexities

Time Complexity (GeeksforGeeks ,2025b):
  Best case: O(1) - this happens when the target element is found at the first position.

  Worse case: O(n) - this happens when the element is at the end or not inside the list.

Space Complexity (GeeksforGeeks ,2025b):
  O(1) - it uses only a fixed amount regardless of input size.

Binary Search

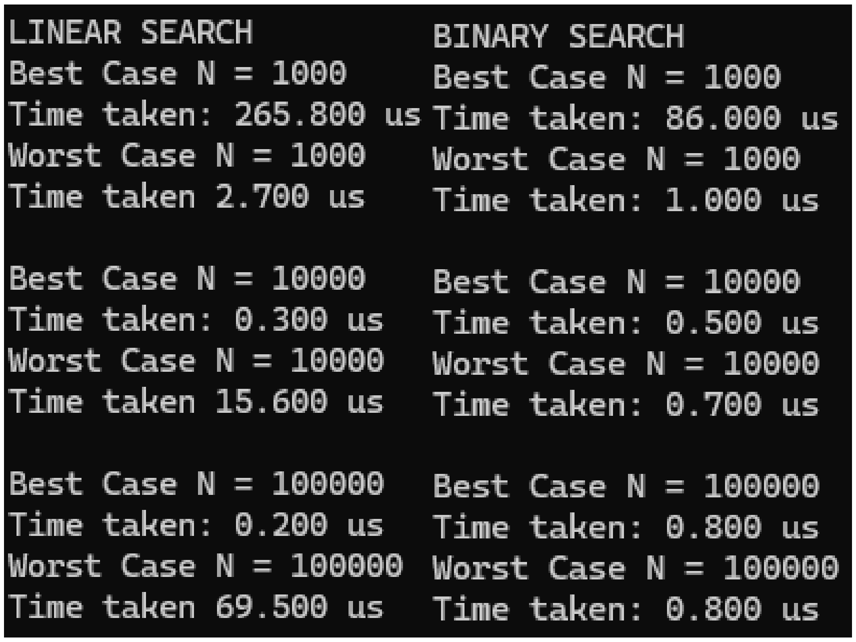
Pseudocode Analysis

```
FUNCTION binarySearch(array, target)
  int left = 0
  int right = (length of array) - 1
    WHILE left IS GREATER THAN right DO
      int mid = (left + right).5
      IF array[mid] = target THEN
        RETURN mid
      ELSE IF array[mid] IS LESS THAN target THEN
        left = mid + 1
      ELSE
        right = mid - 1
   RETURN -1
END FUNCTION
```

Binary Search is a more efficient method that requires the input array to be sorted beforehand. It follows a divide-and-conquer approach by repeatedly halving the search range: comparing the target value to the middle element and discarding the half where the target cannot reside. This process continues until the element is found or the search interval becomes empty. The time complexity of Binary Search is O(1) in the best case, when the target happens to be at the midpoint on the first comparison, and O(log n) in the

	worst case, when multiple divisions of the array are needed before reaching the result. For space complexity, the iterative version of Binary Search uses O(1) space, while the recursive version consumes O(log n) due to its recursive stack calls.
Time and Space Complexities	Time Complexity (GeeksforGeeks ,2025b): Best case: O(1) - this happens when the target element is in the midpoint of the first comparison. Worse case: O(log n) - this happens when search continues halving the array until the range is reduced to single element Space Complexity (GeeksforGeeks ,2025b): O(1) - it uses a iterative version in which uses a few variable (left, right, mid) for indexing tracking. O(logn) if its implemented recursively used in stack depth.

Input Size (N)	**Elapsed Time for Binary Search (Worst-Case)**	**Elapsed Time for Linear Search (Worst-Case)**
1,000	1.000 us	2.700 us
10,000	0.700 us	15.600 us
100,000	0.800 us	69.500 us



```
LINEAR SEARCH               BINARY SEARCH
Best Case N = 1000          Best Case N = 1000
Time taken: 265.800 us      Time taken: 86.000 us
Worst Case N = 1000         Worst Case N = 1000
Time taken 2.700 us         Time taken: 1.000 us

Best Case N = 10000         Best Case N = 10000
Time taken: 0.300 us        Time taken: 0.500 us
Worst Case N = 10000        Worst Case N = 10000
Time taken 15.600 us        Time taken: 0.700 us

Best Case N = 100000        Best Case N = 100000
Time taken: 0.200 us        Time taken: 0.800 us
Worst Case N = 100000       Worst Case N = 100000
Time taken 69.500 us        Time taken: 0.800 us
```

The experimental results demonstrate a clear performance between binary Search and Linear Search when the input size increases. Linear Search Worst times were 2.7us, 15.6 us, and 69.5 us for input sizes of 1,000, 10,000, and 100,000 respectively. This consistent increase in runtime reflects its O(n) growth pattern, where execution time rises proportionally with the number of elements. The linear nature of this algorithm makes it inefficient for large datasets since it must traverse each element sequentially until the target is found or the list is exhausted.

However, Binary Search recorded much faster runtimes of 1.0us, 0.7us, 0.8us for input sizes of **1,000, 10,000, and 100,000** respectively.The minimal change in execution time despite a hundredfold increase in input size demonstrates its **O(log n)** time complexity, which grows very slowly as input size expands. This efficiency comes from its divide-and-conquer method of repeatedly halving the search range, allowing it to locate elements in a fraction of the time compared to Linear Search.

Overall, the experimental data confirms the theoretical expectations. Binary Search is exponentially faster than Linear Search for larger datasets, provided that the data is sorted beforehand. Despite the difference, Linear Search remains useful for unsorted lists or smaller datasets due to its simplicity and constant space requirements. The comparison highlights the scalability advantage of Binary Search.

C. Conclusion & Lessons Learned

dano

- This activity emphasizes the importance of choosing algorithms not just for functionality, but for scalability and performance in real-world applications.

cruz

- This activity shows me the algorithm efficiency greatly depends on the approach used and the data size. Divide and conquer methods like merge sort and binary search perform much better than simple iterative ones. This shows the importance of choosing the right algorithms for faster and more scalable results.

D. Assessment Rubric

E. External References

- GeeksforGeeks. (2025, July 23). *Time and Space complexity Analysis of bubble sort*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/time-and-space-complexity-analysis-of-bubble-sort
- GeeksforGeeks. (2024, March 14). *Time and space complexity analysis of Merge sort*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/time-and-space-complexity-analysis-of-merge-sort
- GeeksforGeeks. (2025b, October 22). *Linear Search Algorithm*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/linear-search
- GeeksforGeeks. (2025b, September 10). *Binary search*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/binary-search