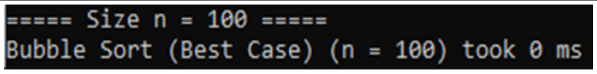
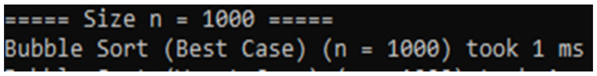
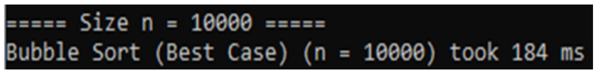
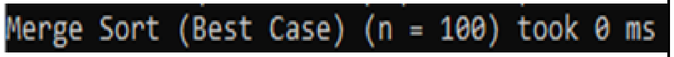
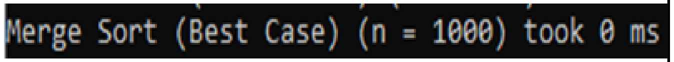
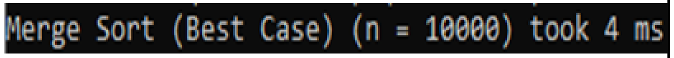
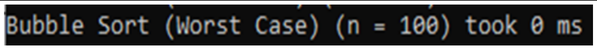
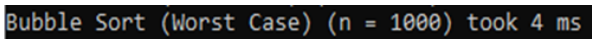
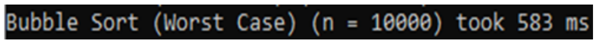
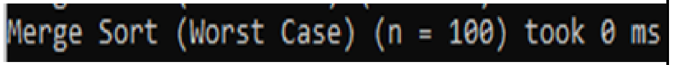
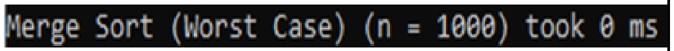
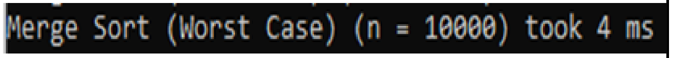
Hands-on Activity 14.1	
Algorithm Complexity	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 11/06/25
Section: CPE 010-CPE21S4	**Date Submitted:** 11/06/25
Name(s): **Jason Alcantara** **Lester arvid anastacio** **Vince Gabriel V. Avila** **Kerwin Jan B. Catungal**	**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) — matches B = n; stops after one pass if already sorted.	$O(n^2)$ — matches D = n·n; worst case needs ≈ n(n−1)/2 comparisons.	The best case scenario for Bubble Sort occurs when the list is already sorted. It will only make one pass. However, in the worst case scenario each element has to be compared and in the end swapped which makes the time complexity grow quadratically (proportional to n^2).
Algorithm 2 Merge Sort	O(n log n) — matches C = $n \cdot \log_2 n$; split-and-merge cost.	O(n log n) — matches C = $n \cdot \log_2 n$; same dominating cost in worst case.	Merge Sort works by recursively partitioning the list and merging the sublists back together, needing around $n \log_2 n$ operations to do so. Merge Sort maintains a steady runtime because partitioning and merging occurs in every scenario, even the best-case one.

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

	Algorithm 1 (Bubble Sort)	Algorithm 2 (merge sort)
Best Case	 `===== Size n = 100 =====` `Bubble Sort (Best Case) (n = 100) took 0 ms`  `===== Size n = 1000 =====` `Bubble Sort (Best Case) (n = 1000) took 1 ms`  `===== Size n = 10000 =====` `Bubble Sort (Best Case) (n = 10000) took 184 ms`	 `Merge Sort (Best Case) (n = 100) took 0 ms`  `Merge Sort (Best Case) (n = 1000) took 0 ms`  `Merge Sort (Best Case) (n = 10000) took 4 ms`
Worst Case	 `Bubble Sort (Worst Case) (n = 100) took 0 ms`  `Bubble Sort (Worst Case) (n = 1000) took 4 ms`  `Bubble Sort (Worst Case) (n = 10000) took 583 ms`	 `Merge Sort (Worst Case) (n = 100) took 0 ms`  `Merge Sort (Worst Case) (n = 1000) took 0 ms`  `Merge Sort (Worst Case) (n = 10000) took 4 ms`

Analysis:
Based on my results, Bubble Sort got slower when the input size got bigger especially in the worst case. It took way longer compared to Merge Sort which stayed fast and consistent. This shows that Bubble Sort isn't good for large data while Merge Sort handles it better.

B. Answers to Supplementary Activity

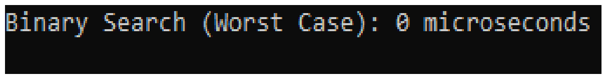
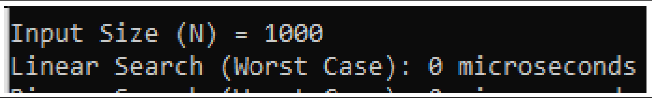
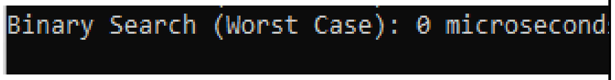
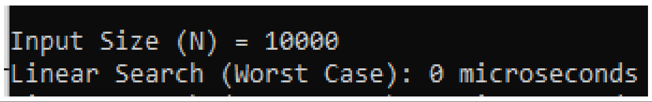
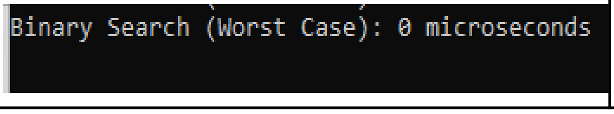
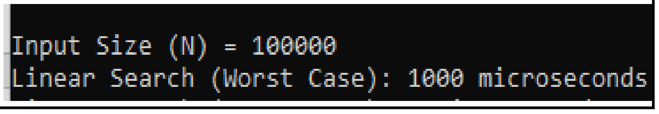
ILO A:

```
LinearSearch(array, key)
for i  0 to length(array) - 1
   if array[i] == key then
      return i
return -1


BinarySearch(sortedArray, key)
low 0
high  length(sortedArray) - 1
while low ≤ high do
   mid  (low + high) / 2
   if sortedArray[mid] == key then
      return mid
   else if sortedArray[mid] < key then
      low  mid + 1
   else
      high  mid - 1
return -1
```

Time and Space Complexities:

Algorithm	Best Case	Average Case	Worst Case	Worst Case
Linear Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)

Input Size (N)	Elapsed Time for Binary Search (Worst-Case)	Elapsed Time for Linear Search (Worst-Case)
1,000	 `Binary Search (Worst Case): 0 microseconds`	 `Input Size (N) = 1000` `Linear Search (Worst Case): 0 microseconds`
10,000	 `Binary Search (Worst Case): 0 microseconds`	 `Input Size (N) = 10000` `Linear Search (Worst Case): 0 microseconds`
100,000	 `Binary Search (Worst Case): 0 microseconds`	 `Input Size (N) = 100000` `Linear Search (Worst Case): 1000 microseconds`

Analysis:
Based on the result in linear search, it became time consuming particularly as the initial size grew. This is expected given that linear search accesses each item individually and with each iteration of a binary search the search space is effectively reduced by half. The results demonstrate that binary search is faster but pre-sorted data is a prerequisite for this algorithm so binary search clearly performs better than linear search in the worst-case scenario.

C. Conclusion & Lessons Learned

Catungal - In this activity, I was able to measure the elapsed time in C++ using the <chrono> library and understand how to apply time-related functions in programming. I learned how to record the start and end times, calculate the duration, and display the result properly. It helped me see how timing can be useful in testing program performance. Although I was able to make it work, I still need to improve my understanding of how time libraries and precision work in C++, especially when handling larger or more complex tasks.

Lester Anastacio - This activity provided me with a hand-on perspective of the performance of algorithms in the real world by recording the elapsed time of two sorting and two searching methods and I found out that while bubble sort may be the simplest to code, its O(n^2) worst-case complexity makes it very inefficient for large data sets, unlike merge sort which is consistently fast and similarly, the searching experiments revealed that binary search is better than linear search in the worst-case scenarios for large inputs, thus giving importance to the need of a low time complexity algorithm for scalability. This exercise strengthened my knowledge that algorithm selection is probably the most important factor in achieving true program efficiency.

Jason Alcantara - This activity helped me understand how algorithms perform by measuring the time it takes for two sorting methods (bubble sort and merge sort) and two searching methods (linear search and binary search). I learned that bubble sort, although simple to code, is not efficient for large data sets because of its O(n^2) time complexity, while merge sort is much faster and more reliable. The searching experiments showed that binary search is more efficient than linear search, especially for large amounts of data. I also used the <chrono> library in C++ to measure the time taken by each algorithm, which taught me how to track program performance. However, I still need to work on understanding how time precision works, especially when dealing with more complex tasks. Overall, this activity showed me how important it is to choose the right algorithm for better program efficiency.

Vince Gabriel V. Avila - Through this activity, I learnt how algorithms truly work in reality through timing two searching methods and two sorting methods. I discovered that bubble sort is simple to implement, but inefficient on larger datasets, while merge sort is very efficient even with large datasets. I discovered that binary search is much better than linear search with larger input sizes. This emphasized the significance of using the most effective algorithm, as it increases the overall performance of the program.

D. Assessment Rubric

E. External References

1. https://www.geeksforgeeks.org/dsa/linear-search
2. https://www.freecodecamp.org/news/binary-search-algorithm-and-time-complexity-explained
3. https://www.geeksforgeeks.org/dsa/complexity-analysis-of-binary-search
4. https://www.geeksforgeeks.org/dsa/worst-average-and-best-case-analysis-of-algorithms
5. https://www.geeksforgeeks.org/dsa/analysis-algorithms-big-o-analysis