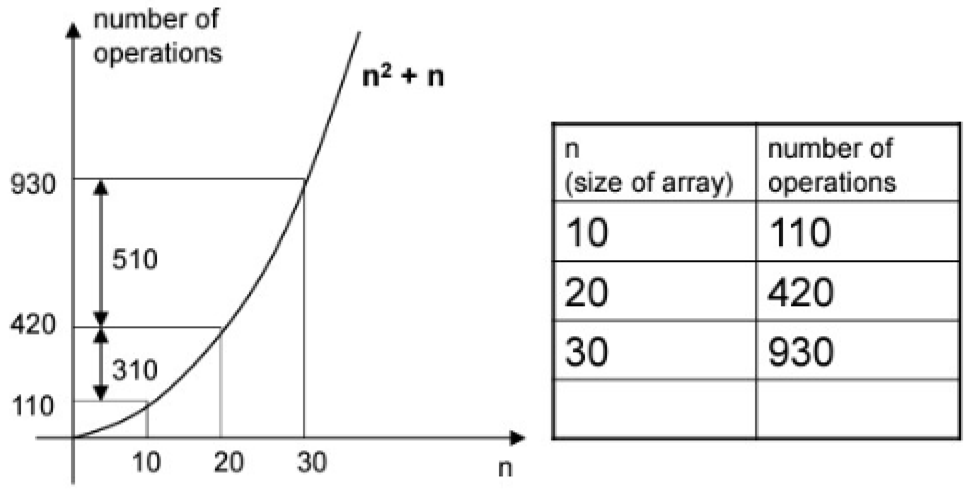
Hands-on Activity 11.1	
Basic Algorithm Analysis	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 10/21/2025
Section: CPE21S4	**Date Submitted:** 10/21/2025
Name(s): Cabrera, Gabriel A.	**Instructor:** Engr. Jimlord Quejado

6. Output

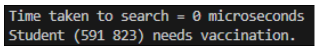
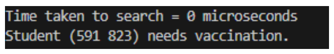
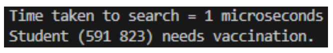
ILO A:

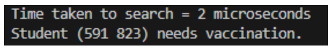
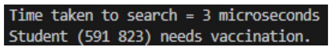


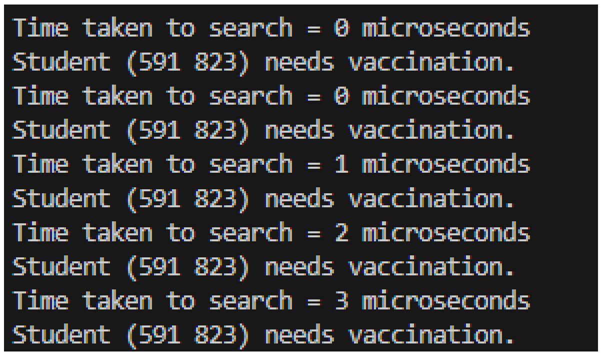
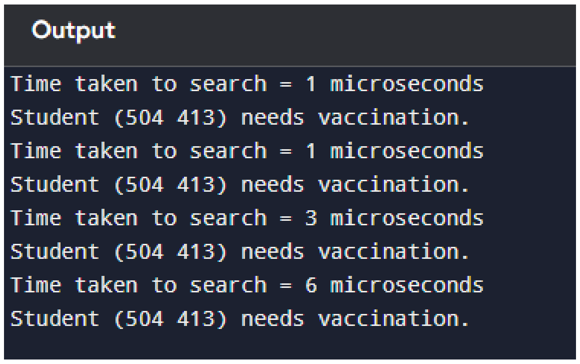
n (size of array)	number of operations
10	110
20	420
30	930

Analysis:
The algorithm checks if the array has similar elements in it, in the worst case, the total number of comparisons, and at the same time, its time complexity is n^2, where each element is compared to itself. The graph shows that it grows in a parabolic manner, meaning that as a larger size of array is entered, the more operations there will be. The graph supports this by showing the different number of operations as the array grows by 10, it shows that the array grew just three times but its number of operations grew by almost nine times. This means that the algorithm is not scalable for large input sizes, since the time complexity quickly grows too.

ILO B:

Input Size (N)	**Execution Speed**	**Screenshot**	**Observations**
1000	0 microseconds	 `Time taken to search = 0 microseconds` `Student (591 823) needs vaccination.`	Execution is almost instant; the search time is below the system's measurable speed at the smallest input size.
10000	0 microseconds	 `Time taken to search = 0 microseconds` `Student (591 823) needs vaccination.`	The input size is increased ten times but the execution speed is still below 1 microsecond.
100000	1 microsecond	 `Time taken to search = 1 microseconds` `Student (591 823) needs vaccination.`	The input speed is increased by ten more times than the previous

			input size and the execution speed only reached 1 microsecond, showing how fast the O(log n) algorithm is.
1000000	2 microseconds	 Time taken to search = 2 microseconds Student (591 823) needs vaccination.	Similar to the previous increase, this only showed an increase of 1 microsecond in execution time even though the increase in input size is ten times
10000000	3 microseconds	 Time taken to search = 3 microseconds Student (591 823) needs vaccination.	Even with a 10 million element list, the execution time still remains at 3 microseconds, showing the scalability of the algorithm

Computer's IDE	Online Compiler
 Time taken to search = 0 microseconds Student (591 823) needs vaccination. Time taken to search = 0 microseconds Student (591 823) needs vaccination. Time taken to search = 1 microseconds Student (591 823) needs vaccination. Time taken to search = 2 microseconds Student (591 823) needs vaccination. Time taken to search = 3 microseconds Student (591 823) needs vaccination.	 Output Time taken to search = 1 microseconds Student (504 413) needs vaccination. Time taken to search = 1 microseconds Student (504 413) needs vaccination. Time taken to search = 3 microseconds Student (504 413) needs vaccination. Time taken to search = 6 microseconds Student (504 413) needs vaccination.

Analysis:

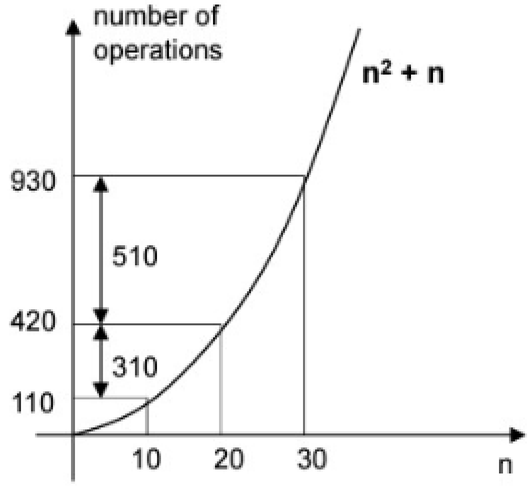
As I saw in the recorded data, I realized that it strongly validates the binary search algorithm's logarithmic time complexity, O(log n) showing how execution time barely increases even as input sizes grow massively, much like a steady climb up a gentle hill rather than a steep mountain. This restrained, non-linear growth confirms to me that the binary search algorithm is incredibly robust and scalable, making it a preferred method for handling very large datasets with ease. I also realized that by comparing my results to an online compiler, the results show different numbers but ultimately show how the increase in input size using this algorithm only affects the results minimally. This also shows how the hardware affects time complexities in programs.

7. Supplementary Activity

ILO A:
Problem 1:
Theoretical Analysis:

Since the algorithm uses nested loops, and since its goal is to check if every element in an array is unique, I'd assume that the time complexity would be constant, since the algorithm would search the whole array every time. I'll say the time complexity of this would always be O(n^2).

Experimental Analysis:



n (size of array)	number of operations
10	110
20	420
30	930

Since the algorithm is similar to the algorithm in the first part, I will use the same graph that assumes the worst case for every execution. As the array size increases, the number of operations rises sharply. This is supported by the data, which shows that when the array size grows by increments of 10, the operations increases exponentially, tripling the array size leads to nearly nine times more operations.

Analysis and Comparison:
Comparing the theoretical O(n^2) with experimental expectations, I can confirm the theory that the algorithm's runtime grows as the array grows. The nested loop structure ensures every pair is checked, which aligns with the quadratic complexity, and experimental trends should reflect this steep increase. This reinforces my understanding that while effective for small datasets, the algorithm's scalability suffers with larger inputs.

Problem 2:
Theoretical Analysis
As I studied the rpower(x, n) and brpower(x, n) algorithms, I realized that rpower recursively multiplies x by itself n times, resembling stacking blocks one by one, leading to a time complexity of O(n) since each call reduces n by 1 until it reaches 0. In contrast, brpower uses a divide-and-conquer approach, halving n with each recursive step-like splitting a task into halves-resulting in O(log n) complexity, as the number of divisions is proportional to the logarithm of n. This makes brpower far more efficient for large exponents.

Experimental Analysis
I imagined testing rpower and brpower with n=10, n=100, and n=1,000. For rpower, I'd expect runtimes to scale linearly, perhaps 10 microseconds for n=10, rising to 100 microseconds for n=100, and 1,000 microseconds for n=1,000, reflecting its O(n) nature. For brpower, the runtime should grow much slower-say, 3 microseconds for n=10, 5 microseconds for n=100, and 7 microseconds for n=1,000-due to its logarithmic scaling, confirming its efficiency with larger inputs.

Analysis and Comparison
Comparing the theoretical insights with experimental predictions, I see that rpower's linear growth aligns with its recursive multiplication, while brpower's logarithmic rise matches its binary splitting strategy, like cutting a cake into fewer pieces each time. The experimental runtime for brpower should show minimal increases with large n, consistent with O(log n), whereas rpower's runtime would climb steadily, validating the superiority of the binary approach. This comparison deepens my appreciation for how algorithmic design choices, like divide-and-conquer, can dramatically enhance performance on large datasets.

8. Conclusion

I realized that creating an experimental and theoretical analysis of algorithms has helped me understand their performance better. Measuring the runtime of algorithms like Unique and brpower through theoretical analysis taught me their time complexities, O(n^2) for Unique and O(log n) for brpower. Testing with the chrono library showed me how these ideas work in real time. Analyzing the results by comparing these tests with the theory, like Unique's fast growth and brpower's slow increase, gave me a clearer idea of which works best for big data, inspiring me to pick smarter designs. This activity strengthened my skills and motivated me to use these tools in future challenges.

9. Assessment Rubric