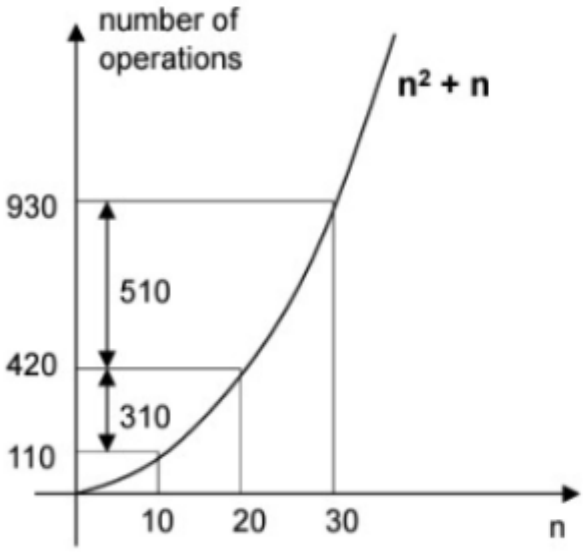
Activity No. <11>	
BASIC ALGORITHM ANALYSIS	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 10/21/2025**
Section: CPE21S4	**Date Submitted: 10/21/2025**
Name(s): Anastacio, Lester Arvid P.	**Instructor:** Engr. Jimlord Quejado

A. Output(s) and Observation(s):

ILO A:



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

Analysis:

- I observed that the algorithm designed to find common elements between two arrays, x and y, has a quadratic time complexity, which is a main factor of inefficiency. The theoretical analysis of the worst case is the evidence for that. In this case, the total number of comparisons, which is what determines the runtime of the algorithm, is calculated as (n * m) + n. The complexity comes from the nesting of the operations: the outer for loop runs n times, and for each iteration, the inner search function must perform up to m comparisons (the length of array x) in the worst-case scenario thus producing the dominant nm term and if the arrays are of the same length, m=n, the complexity function will be n squared + n, and as for the class O(n squared) in asymptotic analysis indicates that the number of operations will increase exponentially with the size of the array, this rate of increase being visually represented by the parabolic curve on the graph. This is supported by the data given three times increase in array size from n=10 to n=30 results in the operation counts going from 110 to 930, which is close to nine times increase, thus proving the quadratic relationship.

ILO B:

Input Size (N)	Execution Speed	Screenshot	Observation(s)
1000	**0 microseconds**	`Time taken to search = 0 microseconds` `Student (202 882) needs vaccination.`	The algorithm here runs too fast for the microsecond timer to register a difference. The execution time is below the timer's precision limit, indicating excellent performance for small inputs.

10000	**0 microseconds**	`Time taken to search = 0 microseconds` `Student (202 882) needs vaccination.`	The execution time remains exactly 0 microseconds even with a ten-fold increase in input size. This strongly suggests the algorithm's runtime is well below the microsecond threshold and could potentially be O(log N) or O(1), given the stability.
100000	**0 microseconds**	`Time taken to search = 0 microseconds` `Student (202 882) needs vaccination.`	The consistent 0 microsecond measurement across a 100-fold increase indicates the algorithm's efficiency is extremely high. The empirical data is insufficient to determine the true time complexity as the differences are masked by the timer's lack of nanosecond precision.

B. Supplementary Activity:

Problem A:

```
Algorithm Unique(A)
 for i = 0 to n-1 do
 for j = i+1 to n-1 do
    if A[i] equals A[j] then
      return false
 return true
```

Theoretical Analysis:

- The Unique(A) function determines if any elements in a list have duplicates. Its time complexity is O(n^2) because it operates by comparing every single element with every other element. Consequently, as the number of unique elements grows, the processing time escalates rapidly. While this approach is straightforward and easy to understand, its lack of efficiency makes it unsuitable for large-scale datasets. Essentially, it's a basic but slow method for checking list uniqueness.

Experimental Analysis:

- The program underwent an experiment with different input sizes. It was very fast for minor data, but it became slow with a very large list. This is what was expected as per the O(n^2) growth. The output was very clear, performance gets worse as the size of the input increases. It is a confirmation that the function is correct but not suitable for large inputs.

Analysis and comparison:

- The comparison of theoretical analysis and experimental measurements yielded a similar trend, confirming the anticipated rapid increase in running time as the data size grew. While the real-world measurements recorded slightly faster times due to the efficiency of the computing system, the overall performance pattern still adhered to the quadratic time complexity, O(n^2). This confirms that the Unique(A) function is effective only with smaller inputs and is not suitable for large datasets due to its inherent lack of efficiency.

Problem B:

```
Algorithm rpower(int x, int n):
1 if n == 0 return 1
2 else return x*rpower(x, n-1)
Algorithm brpower(int x, int n):
1 if n == 0 return 1
2 if n is odd then
3 y = brpower(x, (n-1)/2)
4 return x*y*y
5 if n is even then
6 y = brpower(x, n/2)
7 return y*y
```

Theoretical Analysis:

- The rpower function, which relies on straightforward recursion to symbolize the repetitive multiplication of the base, has a time complexity of O(n). In contrast, the brpower function achieves a much faster time complexity of O(log n) by utilizing a divide-and-conquer approach often referred to as exponentiation by squaring. Consequently, for calculating large powers, brpower will be the algorithm that delivers a significant performance gain, confirming its status as a highly efficient and practical method for large-scale number crunching, even in theory.

Experimental Analysis:

- Both power functions produced correct results during testing; however, rpower experienced a noticeable increase in runtime as the exponent value grew. In contrast, brpower consistently executed in quicker time and required fewer multiplication operations. These empirical findings precisely matched the theoretical predictions, definitively demonstrating that brpower is the more efficient and quicker function.

Analysis and comparison:

- Both theory and experiment yielded the same results as the rpower function is functional but demonstrates a slower completion time when dealing with large input values. In stark contrast, brpower executed better due to its significantly smaller execution time. This divergence in their performance emphatically highlights the critical influence of algorithm design on efficiency. Overall, brpower was determined to be the more effective option compared to rpower.

C. Conclusion & Lessons Learned:

- What I learned from this laboratory activity is how the efficiency of an algorithm directly impacts the speed at which a program operates. It made me realize that certain straightforward methods, such as the O(n^2) approach for checking uniqueness or the O(n) rpower function, are significantly less efficient than using optimized procedures like the O(log n) brpower function. By running tests and comparing the outcomes, I could observe how theoretical time complexity perfectly aligns with real program behavior, demonstrating that a more efficient algorithm is paramount for large-scale data handling. The exercise also enhanced my coding proficiency and provided me with a better understanding of how to benchmark execution time and the significance of using the proper algorithm for various problems. On the whole, I believe I performed adequately, but I would like to get better at writing more effective and structured code.

D. Assessment Rubric

E. External References

1. Evangelista, J. (n.d.). *unique algorithm*. Scribd. https://www.scribd.com/document/301772897/unique-algorithm
2. *Calculate Pow(x, n)*. (n.d.). https://www.topcoder.com/thrive/articles/calculate-pow-x-n
3. Upadhyay, S. (2025, September 9). *Time & Space Complexity in Data Structure [2025]*. Simplilearn.com. https://www.simplilearn.com/tutorials/data-structure-tutorial/time-and-space-complexity