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): Quioyo, Angelo	**Instructor:** Engr. Jimlord Quejado

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

ILO A:

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.
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.
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.

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 checks a list for duplicates, but it is highly inefficient because it has a quadratic time complexity, or O(n^2). This inefficiency stems from its method of comparing every element against every other element in the list. As the number of unique elements increases, the time required to process the list escalates very quickly. While the logic behind this function is straightforward and easy to understand, its lack of efficiency makes it unsuitable for processing large-scale datasets. Essentially, it is a basic but very slow approach to verifying list uniqueness.

Experimental Analysis:

The program's performance was tested using various input sizes. As expected, based on its theoretical O(n^2) growth rate, it processed small data inputs quickly but became significantly slower when presented with very large lists. The clear outcome was that the function's performance rapidly degrades as the input size increases. This confirms that the function operates correctly but is simply not designed or suitable for use with large-scale inputs.

Analysis and comparison:

The comparison between the theoretical analysis and the experimental results showed a consistent trend, confirming the expected rapid increase in the function's running time as the data size grew. Although the actual measured times were slightly faster than the theoretical worst-case analysis it is likely because of the computer system's efficiency of the overall performance pattern still closely matched the predicted quadratic time complexity, O(n^2). This final confirmation proves that while the Unique(A) function is correct, its inherent inefficiency means it should only be used with small inputs and is not appropriate for large datasets.

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 calculates powers using simple recursion to represent repeated multiplication, giving it a linear time complexity of O(n). In stark contrast, the brpower function is significantly faster, achieving a logarithmic time complexity of O(log n) by employing a divide-and-conquer strategy known as exponentiation by squaring. Therefore, especially when calculating large powers, brpower is the superior algorithm, offering a substantial performance gain. This confirms brpower's theoretical status as a highly efficient and practical method for large-scale numerical computation.

Experimental Analysis:

During testing, both power functions produced the correct results. However, the rpower function showed a clear and noticeable increase in its runtime as the exponent's value grew. Conversely, the brpower function consistently executed more quickly and required a significantly lower number of multiplication operations. These empirical findings perfectly aligned with the theoretical predictions, definitively proving that brpower is the more efficient and faster function for calculating powers.

Analysis and comparison:

Both theoretical analysis and experimental results confirmed the same outcome: the rpower function is functionally correct but consistently shows slower completion times when processing large input values. In distinct contrast, the brpower function executed much better due to its significantly smaller execution time. This sharp difference in performance strongly emphasizes the critical impact that algorithm design has on computational efficiency. Ultimately, brpower was conclusively determined to be the more effective and efficient option compared to rpower.

C. Conclusion & Lessons Learned:

- The lab exercise teaches, among other things, the immediate and crucial role of algorithm efficiency in program speed. I found out that naive methods and such as the O(n^2) quadratic approach for uniqueness or the O(n) rpower function one that are slower and more wasteful than optimized ones, like the O(log n) brpower. By conducting experiments, I could see how well the theoretical time complexity corresponded to the program's actual performance. This indicated that selection of an efficient algorithm is important in the context of big data processing. Also, it was a good practice of improved my programming skills and knowledge of execution time measurement and how to choose the appropriate algorithm for each problem. I'd say my performance was alright, bit I need to be able to write better code and keep it clean.

D. Assessment Rubric

E. External References