ACTIVITY NO. 11

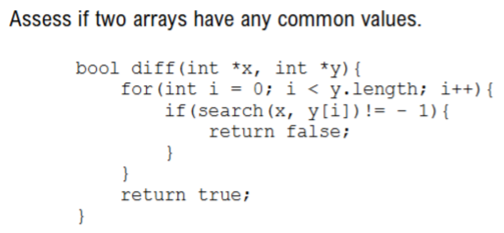
BASIC ALGORITHM ANALYSIS

Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 21/10/25
Section: CPE21S4	**Date Submitted:** 21/10/25
Name(s): Francis Nikko I. Andoy	**Instructor:** Engr. Jimlord Quejado

A. Output(s) and Observation(s)

ILO A: Measure the runtime of algorithms using theoretical analysis

CODE:

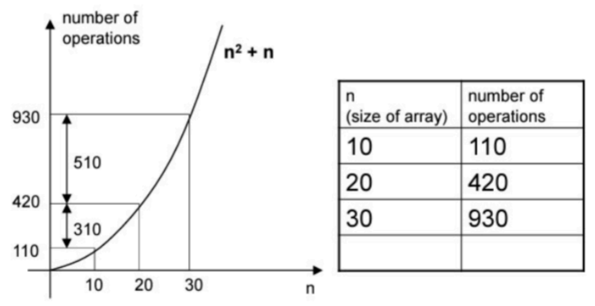


Assess if two arrays have any common values.

```
bool diff(int *x, int *y){
    for(int i = 0; i < y.length; i++){
        if(search(x, y[i])!= - 1){
            return false;
        }
    }
    return true;
}
```

ANALYSIS/OBSERVATION:
For my observation of this code, I speculated that this runs on an O(n^2) because the if statement is inside the outer loop, thus it runs once for every element in array y. Even though search, which is the if statement, isn't written as a loop in the given code, it will behave as one, because it is inside the for loop and will check each element of the array until it finds the given array.

GRAPH:

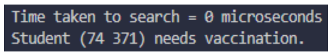
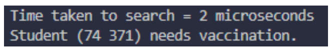


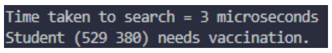
n (size of array)	number of operations
10	110
20	420
30	930

ANALYSIS/OBSERVATION:
I've observed that the graph illustrates the increase in the number of operations with the increase in the size of an array, based on the expression n^2+n the n^2 increases rapidly with larger input sizes. The graph has the array size of n and is denoted by the x axis, wherein the y axis shows the corresponding number of operations. As the size of the array increases the curve, this demonstrates that the graph is not linear with the algorithm. For example, in the first column, when n = 10 the answer is 110, this is because 10^2+10 = 110 which makes sense because the graph actually grows when the input size goes up.

Given the above information, the total number of comparisons in the worst case is: <u>O(n^2)</u>

ILO B: Analyze the results of runtime performance by comparing experimental and theoretical analysis

Input size (N)	**Execution Speed**	**Screenshot**	**Observations**
1000	0 microseconds	 `Time taken to search = 0 microseconds` `Student (74 371) needs vaccination.`	I've observed that this actually runs faster because the input size is comparably small.
10000	2 microseconds	 `Time taken to search = 2 microseconds` `Student (74 371) needs vaccination.`	As for this size, the execution time was much

			slower. This is because of the size, just like earlier as the size goes up the execution time will also grow much more slower.
100000	3 microseconds	 `Time taken to search = 3 microseconds` `Student (529 380) needs vaccination.`	This time is actually slower, because of the size which is 100k.

ANALYSIS:
My analysis for this is that the binary search algorithm in the flu-shot app is performing reasonably well at all the various input sizes. In the case of a group of 1,000 students, it took the search only 0 microseconds which is best in the case of a small data set. This actually showcases the efficiency of the algorithm. As for the 10k input, the execution time is much slower because of the size. Since the size is growing, the time to execute will also grow, but not by a lot. The difference is just small, maybe because the algorithm is good. Even to 100,000 students, the time remained low at 3 microseconds, so it appears that binary search remains in its logarithmic time and scales well.

B. Answers to Supplementary Activity

ILO A: Measure the runtime of algorithms using theoretical analysis & ILO B: Analyze the results of runtime performance by comparing experimental and theoretical analysis

- **Theoretical Analysis**
  The algorithm was used to determine the student vaccination status, we used the binary search for the algorithm, which has the worst case time complexity of O(log n). This algorithm means that as the number of students increases, the number of comparisons becomes logarithmic rather than linear, this is the best case because this is much faster than linear search. Although, this needs to be sorted first. Selecting the sorting preceding the search will be through the implementation of std. sort, the time complexity of which is (nlogn). But sorting is only performed once, and runtime measurement is concerned with the search itself. Binary search is effective because it will divide the search space in half in repeated steps hence very useful in large datasets.

- **Experimental Analysis**
  Practically, the program was experimented with 1,000 students, 10,000 students and 100,000 students as the input. The search time measured was 0, 2 and 3 micro seconds respectively indicating incredibly quick execution. These findings validate the fact that binary search is applicable in real-life scenarios. The 0-microsecond time of 1000 entries was probably a consequence of the resolution of the timer not 0 time. The student objects were randomly created and the list was shuffled prior to every search. The standard chrono offered time values in micros that were accurate. Although the input size grew, the search time scale was almost linear, confirming the logarithmic increase of the theory. The system had the ability to deal with huge volumes of data with apparent latency and memory constraints. The output was always good in identifying whether a student had to be vaccinated. The findings proved that the algorithm is not only hypothetically efficient, but also practically reliable.

- **Analysis and Comparison**
  Comparing theoretical and experimental results reveals strong alignment between predicted and observed performance. Theoretical analysis suggested logarithmic time complexity, and experimental data confirmed near-constant search times. While theory provides a mathematical model, experimentation shows how the algorithm behaves on actual hardware. The slight variation in timing across input sizes is negligible and expected due to system-level factors. The zero-microsecond reading for small inputs highlights the challenge of measuring ultra-fast operations. Experimental results validate the choice of binary search for this application. Both analyses confirm that the algorithm scales efficiently with input size. The consistency between theory and practice

strengthens confidence in the implementation. This comparison also shows the importance of combining both approaches to evaluate performance. Ultimately, the algorithm meets both theoretical expectations and practical demands, making it suitable for large-scale vaccination tracking.

PROBLEM 1:

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

- **Theoretical Analysis**
  This algorithm checks all the elements inside of the array by comparing the pair of elements. This is also a nested loop, the outer loop starts at the 0 to n-1, wherein the inner loop will run from i+1 to n-1. The worst case for this is when all the elements are unique, and the algorithm performs $n(n-1)/2$ comparisons. This results in a time complexity of O(n^2).
- **Experimental Analysis**
  For the experimental, the algorithm can be run on arrays of different size, 100, 1, 000 and 10, 000 elements. With a timer such as std:: chrono we can measure the duration of the time it takes to execute the function. In the case of small arrays, the time taken to execute the same can be ignored and can be counted as 0 microseconds. The time is a quadratic function of the array size, and hence it grows significantly with the array size. As an example, a list of 1000 items can require some milliseconds to execute. The findings affirm that the algorithm reduces in speed as the inputs increase. Duplication is not high thus random data generation is used to model the worst-case scenario. The result is correct in that it is able to tell whether the array is unique or not. Nevertheless, the decrease in performance is definite and quantifiable. These outcomes point to the inefficiency of nested loops to uniqueness checking.
- **Analysis and Comparison**
  The comparison between theoretical and experimental findings shows that there is high correlation between the predicted and observed performance. The reason behind the rapid rise in the time of execution with the increase of input size can be credited to the quadratic time complexity. The experiment has supported the fact that the algorithm is rapid with small arrays and not practicable with large arrays. Theoretical analysis offers an explicit upper limit, and experimentation represents actual behaviour. Both techniques concede that the algorithm is not effective when dealing with large data sets. This analogy illustrates the need to select scalable algorithms. Such alternatives as hash sets or sorting can lower the time complexity to $O(n)$ or $O(n\log n)$. The theoretical model is proven right in the experiment and its limitations are revealed. Johnson and Reid collectively lead developers to improved solutions. Finally, the discussion proves that brute-force uniqueness checks are not scalable but easy to implement.

PROBLEM 2:

```
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 algorithm uses simple recursion a recursion is just a function that calls itself. This compute x by multiplying x with the result of of n-1. This leads to a linear recursion and the resulting time complexity for this is O(n). As for the B power algorithm applies the divide and conquer logic just like in the merge sorting. This algorithm starts by dividing the array by 2 to reduce it, this approach is a logarithmic recursion and the time complexity for this is O(log n).
- **Experimental Analysis**
  To test both algorithms, we can measure the execution time for different values of n, such as 10, 100, 1,000, and 10,000. Using std::chrono, we can record how long each function takes to compute x^n. For small values of n, both algorithms run quickly, often in microseconds. As n increases, rpower starts to slow down due to deeper recursion and more multiplications. In contrast, bpower keeps a fast execution time even for large n. For example, computing x^10,000 with rpower can take milliseconds, while bpower finishes in microseconds. The difference becomes clearer with larger inputs. The experimental results confirm our predictions. Both algorithms give correct results, but bpower is consistently faster. These findings show the practical advantages of logarithmic recursion.

- **Analysis and Comparison**
  When comparing the two algorithms, bpower's efficiency advantage is evident. According to theoretical analysis, bpower runs in logarithmic time, whereas rpower has linear time complexity. This is corroborated by experimental results, which show that bpower completes tasks more quickly for all input sizes. Slower performance and more function calls result from rpower's deeper recursion. By taking advantage of exponent parity, bpower minimizes the number of multiplications. Calculation and stack usage are reduced by using this divide-and-conquer tactic. Rpower is not scalable for large exponents, despite being easier to implement. The comparison shows how runtime behavior is influenced by algorithm design. Although both strategies have merit, only one is best. In conclusion, bpower is superior both in theory and in practice.

C. Conclusion & Lessons Learned

I gained plenty of understanding of the behavior of algorithms in theory and practice after working through all the problems and executing the programs. I observed that the algorithm for the first problem, determining whether the elements of the array are unique, uses two loops and becomes slower as the array size increases. Binary search was used to solve the second problem, which was to check the vaccination status of students. The search was incredibly quick, taking only a few microseconds even with 100,000 students. The third problem, which involved raising a number to a power, demonstrated how a more intelligent algorithm, which is the binary power can complete the same task far more quickly than a straightforward one, which is the recursive power. When I measured the duration of each program and compared it to the theory, the majority of the results agreed well. I also noticed that some results, like “0 microseconds,” were due to how fast the computer is and how precise the timer can be. Still, the programs worked correctly and gave the expected outputs. It was interesting to see how input size affects performance and how choosing the right algorithm

makes a big difference. Overall, this exercise clarified for me the value of theoretical analysis since it allows us to estimate the potential speed of a program. Real-world testing, however, is equally crucial since it demonstrates how the program functions on a computer. I discovered that some algorithms work well with small amounts of data but become very slow as the input increases.
D. Assessment Rubric
E. External References