Hands-on Activity 11.1	
Basic Algorithm Analysis	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 10 - 21 - 25
Section: CPE21S4	**Date Submitted:** 10 - 21 - 25
Name(s): Santiago, David Owen A.	**Instructor:** Engr. Jimlord Quejado

6. Output

ILO A: Measure the runtime of algorithms using theoretical analysis. 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;
}
```

Performing worst case analysis:

Let m = the length of array x.

Let n = the length of array y.

The loop in diff repeats n times.

Each call to search requires m comparisons.

Given the above information, the total number of comparisons in the worst case is: **$n^2 + n$**

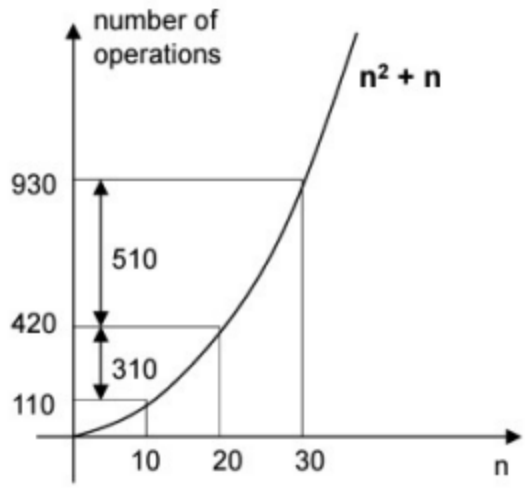
Assuming the worst case, where there are no common elements in both arrays:
x[] = {1, 2, 3}
y[] = {4, 5, 6}

n = 3
3 * 3 = 9 comparisons + 3 calls of search before returning true = 12 comparisons

Therefore, the worst case is that the algorithm will perform $n^2 + n$ times.
So the time complexity of this algorithm is **$O(n^2 + n)$**

Assume that m = n. (The arrays are the same size)

Provide an analysis of the graph for the worst case of the algorithm.



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

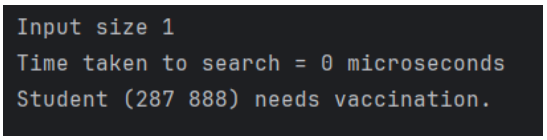
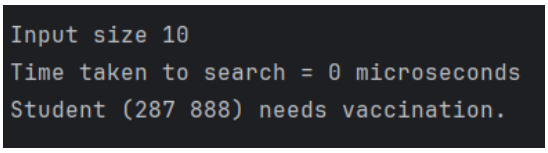
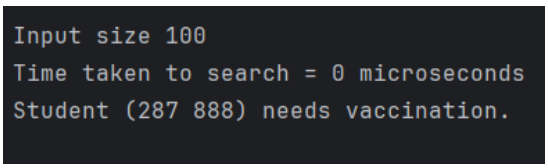
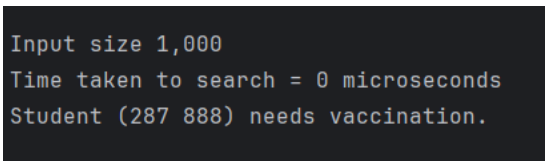
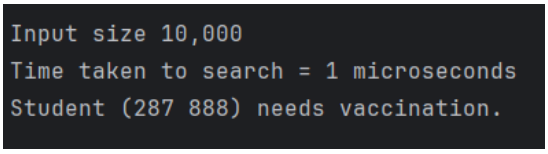
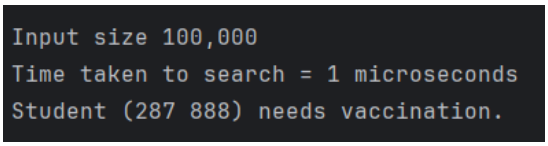
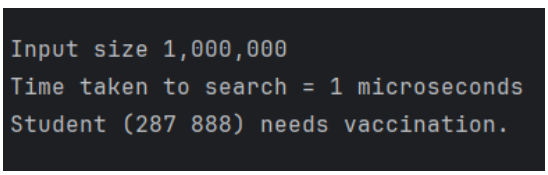
In the worst case scenario of the algorithm, where there are no common elements in the array, each call of the search algorithm will perform n^2 comparisons, since each element is compared to all elements in the array, and since they're the same size, it will be n * n. Then, we add the amount of times we call the search function, which is n, which means we add it to n^2. The graph would then rise rapidly as the size of the array increases, a little faster than if it were only quadratic. Thus, the best case is that the arrays have the exact same elements in the exact same order, but if they were completely different with no common elements, then the runtime of the algorithm would rapidly rise as seen in the graph.

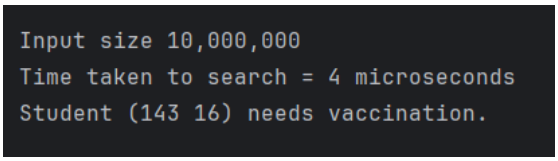
ILO B: Analyze the results of runtime performance by comparing experimental and theoretical analysis

Imagine that it is flu season and health department officials are planning to visit a school to ensure that all the enrolled children are administered their flu shot. However, there is a problem: a few children have already taken their flu shots but do not remember if they have been vaccinated against the specific category of flu that the health officials plan to vaccinate all the students against. Official records are sought out and the department is able to find a list of students that have already been administered the vaccine. A small excerpt of the list is shown here:

The searching algorithm uses a binary search algorithm, hence the average time complexity is **O(log N).** Meanwhile, the sorting algorithm would generally use the Heapsort algorithm, especially with our larger inputs, which has an **O(N log N)** time complexity.

Theoretically, searching would have minimal effects on the run time since we're using the faster, binary search algorithm on a sorted list. The graph of a log N complexity steadies off after a certain input, which means once it hits a certain input value, it wouldn't increase as much or at all. The biggest impact to the runtime would be the sorting function, since the additional N would mean the graph would grow a little faster than a linear time complexity, thus affecting the run time much more heavily.

Input Size (N)	Execution Speed	Screenshot	Observation(s)
1	0 µs		Executes immediately, which is the best case since there is only one element to be sorted and searched.
10	0 µs		These are also relatively smaller inputs, hence, the run time would be much lower than a microsecond, which rounds down to 0 µs at run time.
100	0 µs		
1,000	0 µs		
10,000	1 µs		Executes immediately, but as seen in the console output, the program now recognizes that an input size of 10,000 takes at least 1 microsecond, which is a reflection of the N log N time complexity, where after a certain input size, the graph would have a small jump in growth.
100,000	1 µs		This now takes a second to print, indicating that the system is now starting to work in order to complete all of the sorting happening. Despite the longer load time, the actual time taken to search as calculated by the algorithm is still about 1 microsecond.
1,000,000	1 µs		Takes about three seconds to print and finish, indicating the system is now consuming more processing power and memory to execute the sorting. The algorithm still calculates a 1 microsecond

10,000,000	4 μs		Takes almost a minute to finish executing, indicating that the system is now starting to struggle in processing the 10 million inputs. However, again, since we're only recording and measuring the time taken to search, which uses Log N, then the time would still be very small, almost negligible all throughout the board.

Overall Analysis:

Since our theoretical analysis shows that the searching time complexity would be O(log N) for a binary search, the program reflects this by showing very small execution times for the actual searching. This is because no matter how large the input size, after a certain point, the number of computations or comparisons happening would plateau. Hence, at very large inputs, the execution time would still be relatively small, almost negligible across the board; and any sudden increases would only show up at lower values.

Despite the low number of comparisons happening, what causes our run time to tank so suddenly is the number of sorting happening from std::sort(), which has an average time complexity of N log N. So despite log N smoothening after a certain value, we still have to multiply it by the input value N, so the graph would be log-linear, where the number of computations or comparisons would increase suddenly after a certain input size. This is evident at 100,000, 1,000,000, and even more at 10,000,000, where the system suddenly takes longer to run and execute. I also tested this algorithm in an online compiler and the 10,000,000 input run took over 10 minutes before printing a result. This proves that optimizing time complexity must be both done theoretically and experimentally to understand and see where the algorithm's strengths and limits are.

7. 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

For each of the problems below, provide the following:

- Theoretical Analysis
- Experimental Analysis
- Analysis and Comparison

Problem 1: Consider a program that returns true if the elements in an array are unique.

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

This is a nested for loop, which means that for each n there will be n comparisons done on the worst case where each element is unique. Thus, the time complexity would be $O(n^2)$ or quadratic. For the comparisons, it can be calculated as the sum of $(n-1)+(n-2)+(n-3)+...+1$, which is approximately equal to $n(n-1)/2$. So, in theory, if n = 10, the comparisons should be $(10^2 - 10) / 2 = 45$. The comparison counter will be the main basis to use in the experimental analysis, since the actual run time would still be too fast and negligible to measure.

Experimental

Code:

```
#ifndef SUPP_H
#define SUPP_H

bool Unique(int A[], int n, int &comparisons) {
   comparisons = 0;
   for (int i = 0; i < n; i++) {
       for (int j = i + 1; j < n; j++) {
           comparisons++;
           if (A[i] == A[j]) {
               return false;
           }
       }
   }
   return true;
}
#endif //SUPP_H
```

```
#include <iostream>
#include "supp.h"


int main() {

   int sizes[] = {1, 5, 10, 50, 100};

   for (int s : sizes) {
       int* arr = new int[s];

       for (int i = 0; i < s; i++) {
           arr[i] = i;
       }

       int comparisons = 0;
```

```
        bool result = Unique(arr, s, comparisons);

        std::cout << "Input size: " << s
                  << "\nComparisons = " << comparisons
                  << (result ? "\nUnique" : "\nNot Unique")
                  << '\n';

        delete[] arr;
    }

    return 0;
}
```

Output:

```
"D:\College\Year 2 CPE\Y2 Sem 1\CPE 010\
Input size: 1
Comparisons = 0
Unique
Input size: 5
Comparisons = 10
Unique
Input size: 10
Comparisons = 45
Unique
Input size: 50
Comparisons = 1225
Unique
Input size: 100
Comparisons = 4950
Unique

Process finished with exit code 0
```

Comparisons are calculated using: $(n^2 - n) / 2$
So for inputs like 10, it would be $(10^2 - 10) / 2 = 45$

Analysis and Comparison

Since I included a comparisons counter when testing this algorithm, I was able to observe that the algorithm behaves exactly like how it would in the theoretical analysis. This algorithm is a nested for loop, where the comparisons are done similar to the selection sort, with $n(n-1)/2$ which results in the time complexity of $O(n^2)$.

Problem 2: Consider another program that raises a number x to an arbitrary nonnegative integer n.

Algorithm rpower(x, n):

1 if n == 0 return 1
2 else return x * rpower(x, n-1)

Algorithm brpower(x, 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

Algorithm rpower(x, n) is a recursive search algorithm that executes n times, which gives a time complexity of O(n). Thus, if we count the number of times it executes by adding a count iterator, we should have the number of operations be equal to the input n.

Algorithm brpower(x, n) is a recursive binary search algorithm that cuts the operation time by dividing the exponent by 2 every recursive call, thus the time complexity is O(log n). For example, if n = 16, the multiplications would be:

brpower(2, 16) → 1 multiplication $2^8 * 2^8$
brpower(2, 8) → 1 multiplication $2^4 * 2^4$
brpower(2, 4) → 1 multiplication $2^2 * 2^2$
brpower(2, 2) → 1 multiplication $2^1 * 2^1$
brpower(2, 1) → 2 multiplications because 2 * 1 * 1

Thus, the total multiplications should be 6

Experimental Analysis

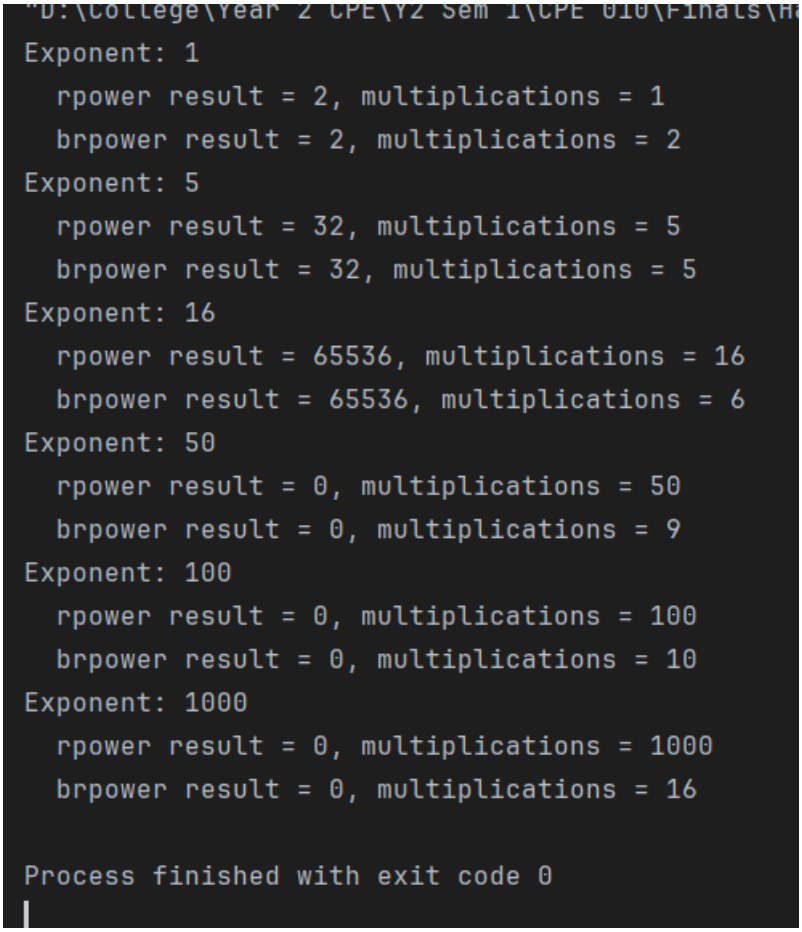
Code:

```
// Recursive power: O(n)
inline int rpower(const int x, const int n, int &count) {
   if (n == 0) return 1; // x^0 = 1
   count++;
   return x * rpower(x, n - 1, count);
}

// Binary recursive power: O(log n)
inline int brpower(const int x, const int n, int &count) {
   if (n == 0) return 1;

   int y = brpower(x, n / 2, count);

   if (n % 2 == 0) {
```

```
        count += 1; // y * y
        return y * y;
    } else {
        count += 2; // x * y * y
        return x * y * y;
    }
}
```

```
#include <iostream>
#include "supp.h"


int main() {
    int x = 2;
    int exponents[] = {1, 5, 16, 50, 100, 1000};

    for (int n : exponents) {
        int count_r = 0, count_br = 0;

        int result_r = rpower(x, n, count_r);
        int result_br = brpower(x, n, count_br);

        std::cout << "Exponent: " << n
             << "\n  rpower result = " << result_r << ", multiplications
= " << count_r
             << "\n  brpower result = " << result_br << ",
multiplications = " << count_br
             << "\n";
    }

    return 0;
}
```

Output:



Analysis:

As seen in the code implementation, as the exponent increases, we can reduce the number of multiplications using the binary recursive implementation, where the exponent is divided by 2 after every recursive call. The results from the theoretical analysis are also reflected in the experimental analysis, showing that using simpler recursive power yields O(n), while binary recursive power yields O(log n). However, since 32 bit integers only go up to 2^31, the results would overflow and result to 0, but the number of operations happening are still recorded and correctly calculated.

References Used:

GeeksforGeeks. (2025a, July 11). *Internal Working of sort() in C++*. GeeksforGeeks.

https://www.geeksforgeeks.org/cpp/internal-details-of-stdsort-in-c/

GeeksforGeeks. (2025d, July 23). *Introsort C++'s Sorting Weapon*. GeeksforGeeks.

https://www.geeksforgeeks.org/dsa/introsort-cs-sorting-weapon/

GeeksforGeeks. (2025a, July 11). *Time and space complexity analysis of binary search algorithm*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/complexity-analysis-of-binary-search/

Verma, P. (2025, May 1). *Understanding time complexity: a key to efficient algorithms - BackendMesh*. Backendmesh. https://www.backendmesh.com/time-complexity-a-key-to-efficient-algorithms/

8. Conclusion

In conclusion, this activity explores the importance and basics of theoretical and experimental algorithm analysis. Using basic algorithms, I was able to analyze the best and worst cases using theoretical analysis. To prove that my theoretical analysis is accurate, I also tested it using experimental analysis by implementing the algorithms in usable C++ code. More importantly, I was able to visualize how different time complexities are calculated using graphs and counters in the C++ code. Additionally, I was able to understand how logarithmic time complexities work due to the constant division happening during each iteration and execution. I was also able to recall the previous algorithms we studied, like binary algorithms, when analyzing the given code. I was able to appreciate how the time complexities were calculated because of the actual testing I did throughout the activity. Along with that, I was also able to understand the importance of time complexity when testing actual run time, where I was able to see in real-time how the compiler takes a minute when processing large input values. Even with my current gen CPU, large input values with bad worst case time complexities, like quadratic and exponential time complexities, still take a few seconds to minutes to execute. Thus, I am able to understand and learn why we should do both theoretical and experimental analysis of algorithms. Theoretical analysis allows us to have a basis for the time complexity regardless of the hardware being used. Meanwhile, experimental analysis allows us to test the theoretical analysis findings using different types of hardware to then test and prove the upper limits of and thresholds of the algorithms.

This activity made me appreciate all of the algorithms used universally, be it from apps, social media, search engines, etc. Handling millions to billions of user data would require very optimized algorithms and systems; knowing computers and servers that can handle them exist in our world made me appreciate them even more. Thus, to improve my own analytical skills, I believe I still need to go back to our previous activities to analyze the algorithms being used. I still struggle to visualize some of the algorithms due to their complexity, thus, utilizing graphs and testing would also help me improve further. Overall, I found this activity to be fun and insightful.

9. Assessment Rubric