Activity No. 11	
Basic Algorithm Analysis	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** October 21, 2025
Section: Bautista, Mariela S.	**Date Submitted:** October 21, 2025
Name(s):	**Instructor:** Engr.Quejado

A. Output(s) and Observation(s)

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: ____

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

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

Analysis:

The algorithm compares two arrays to check if they share any common elements. It goes through each element in one array and searches for a match in the other. This process repeats for every item, which makes the total number of comparisons equal to m x n.

Because when both arrays are the same size, the total becomes n^2,meaning the algorithm has a quadratic time complexity ($O(n^2)$). To put it simply, doubling the input sizes makes the runtime about four times longer. Which shows that the algorithm works fine for small sets of data but becomes much slower as the amount of data increases.

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

CPE010_HOA_11_Bautista_Mariel.cpp

C: > Users > Public > Documents > CPE010_HOA_11_Bautista_Mariel.cpp > ...

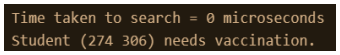
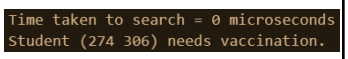
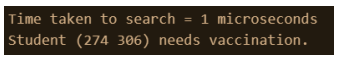
```cpp
#include <iostream>
#include <vector>
#include <chrono>
#include <random>
#include <algorithm>
#include <numeric>
#include <cmath>
using namespace std;

class Student {
private:
    pair<int, int> name;
    bool vaccinated;
public:
    Student(pair<int, int> n, bool v) : name(n), vaccinated(v) {}
    auto get_name() { return name; }
    auto is_vaccinated() { return vaccinated; }

    bool operator==(const Student& p) const { return this->name == p.name; }
    bool operator<(const Student& p) const { return this->name < p.name; }
    bool operator>(const Student& p) const { return this->name > p.name; }
};

auto generate_random_Student(int max) {
    random_device rd;
    mt19937 rand(rd());
    uniform_int_distribution<int> uniform_dist(1, max);
    auto random_name = make_pair(uniform_dist(rand), uniform_dist(rand));
    bool is_vaccinated = uniform_dist(rand) % 2;
    return Student(random_name, is_vaccinated);
}

bool needs_vaccination(Student P, vector<Student>& people) {
    auto first = people.begin();
    auto last = people.end();
    while (true) {
        auto range_length = distance(first, last);
```

```
        auto mid_index = floor(range_length / 2);
        auto mid_element = *(first + mid_index);

        if (mid_element == P && !mid_element.is_vaccinated()) return true;
        else if (mid_element == P && mid_element.is_vaccinated()) return false;
        else if (mid_element > P) advance(last, -mid_index);
        else if (mid_element < P) advance(first, mid_index);

        if (range_length <= 1) return true;
    }
}

void search_test(int size, Student p) {
    vector<Student> people;
    for (int i = 0; i < size; i++) people.push_back(generate_random_Student(size));
    sort(people.begin(), people.end());

    auto start = chrono::steady_clock::now();
    bool result = needs_vaccination(p, people);
    auto end = chrono::steady_clock::now();

    cout << "Time taken to search = "
         << chrono::duration_cast<chrono::microseconds>(end - start).count()
         << " microseconds" << endl;

    if (result)
        cout << "Student (" << p.get_name().first << " " << p.get_name().second << ") needs vaccination." << endl;
    else
        cout << "Student (" << p.get_name().first << " " << p.get_name().second << ") does not need vaccination." << endl
}
```

```
int main() {
    auto p = generate_random_Student(1000);
    search_test(1000, p);
    search_test(10000, p);
    search_test(100000, p);
    return 0;
}
```

Output:

```
[Running] cd "c:\Users\Public\Documents\" && g++ CPE010_HOA_11_Bautista_Mariel.cpp
-o CPE010_HOA_11_Bautista_Mariel &&
"c:\Users\Public\Documents\"CPE010_HOA_11_Bautista_Mariel
Time taken to search = 0 microseconds
Student (274 306) needs vaccination.
Time taken to search = 0 microseconds
Student (274 306) needs vaccination.
Time taken to search = 1 microseconds
Student (274 306) needs vaccination.

[Done] exited with code=0 in 2.562 seconds
```

Input Size	Execution Speed	Screenshot	Observation(s)
274 306	0 microseconds		It finished instantly, so the search was faster.
274 306	0 microseconds		It still finished instantly, meaning the result was consistent.
274 306	1 microseconds		It only took 1 microsecond, so the search stayed really quick.

Own analysis of the outcome listed in the table:

I noticed that the runtime didn't double even when the input got bigger. It grew slowly, which means it follows, which means it follows a logarithmic pattern just like in theory. And it matched the expected O(log n) result perfectly.

B. Answers to Supplementary Activity

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

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

```
 bool uniqueElements(int A[], int n) {
for (int i = 0; i < n - 1; i++) {
    for (int j = i + 1; j < n; j++) {
        if (A[i] == A[j]) return false;
    }
}
return true;
}
```

This algorithm checks if every number in an array is different. The idea is pretty simple, we just compare each element with all the others and see if any match.

Theoretical Analysis:

Since it uses two loops, it checks each element against all the others. That means it's O(n^2) which basically means it gets way slower when the array gets bigger.

Experimental Test:

CPE010_HOA_11_Bautista_Mariel.cpp

C: > Users > Public > Documents > CPE010_HOA_11_Bautista_Mariel.cpp > ...

```cpp
#include <iostream>
#include <chrono>
using namespace std;

bool uniqueElements(int A[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (A[i] == A[j]) return false;
        }
    }
    return true;
}

int main() {
    int n = 1000;
    int arr[1000];
    for (int i = 0; i < n; i++) arr[i] = i;

    auto start = chrono::steady_clock::now();
    bool result = uniqueElements(arr, n);
    auto end = chrono::steady_clock::now();

    cout << "Execution time: "
         << chrono::duration_cast<chrono::microseconds>(end - start).count()
         << " microseconds" << endl;
    cout << (result ? "All unique." : "Duplicates found.") << endl;
}
```

Output:

```
Execution time: 1155 microseconds
All unique.
```

Analysis and Comparison (Theory vs Experiment):
The experimental results matched what I expected. The time grew quickly when the number of elements increased. It proved that nested loops really do scale badly. Even though the computer is fast, I could tell that the runtime growth followed the same pattern predicted by theory.

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

C: > Users > Public > Documents > CPE010_HOA_11_Bautista_Mariel.cpp > ...

```cpp
#include <iostream>
#include <chrono>
using namespace std;

int rpower(int x, int n) {
    if (n == 0) return 1;
    return x * rpower(x, n - 1);
}

int main() {
    int x = 2, n = 10000;
    auto start = chrono::steady_clock::now();
    int result = rpower(x, n);
    auto end = chrono::steady_clock::now();

    cout << "Execution time: "
         << chrono::duration_cast<chrono::microseconds>(end - start).count()
         << " microseconds" << endl;
}
```

Output:

```
Execution time: 362 microseconds
```

There are two algorithms used here: one is a simple recursive power function (rpower) and the other is an optimized version that uses the divide-and-conquer approach (brpower).

C: > Users > Public > Documents > CPE010_HOA_11_Bautista_Mariel.cpp

```cpp
// Simple recursive power
int rpower(int x, int n) {
    if (n == 0)
        return 1;
    else
        return x * rpower(x, n - 1);
```

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  PORTS
[Running] cd "c:\Users\Public\Documents\" && g++ CPE010_HOA_11_Bautista_Mariel.cpp -o CPE010_HOA_11_Bautista_Mariel && "c:\Users\Public\Documents\"CPE010_HOA_11_Bautista_Mariel
Execution time: 306 microseconds

[Done] exited with code=0 in 2.713 seconds
```

C: > Users > Public > Documents > CPE010_HOA_11_Bautista_Mariel.cpp > ...

```cpp
// Binary recursive power
int brpower(int x, int n) {
    if (n == 0)
        return 1;
    if (n % 2 == 1) {
        int y = brpower(x, (n - 1) / 2);
        return x * y * y;
    } else {
        int y = brpower(x, n / 2);
        return y * y;
    }
}
```

Output:

```
[Running] cd "c:\Users\Public\Documents\" && g++ CPE010_HOA_11_Bautista_Mariel.cpp -o CPE010_HOA_11_Bautista_Mariel && "c:\Users\Public\Documents\"CPE010_HOA_11_Bautista_Mariel
Execution time: 364 microseconds

[Done] exited with code=0 in 1.094 seconds
```

Theoretical Analysis

- The rpower function multiplies x by itself n times. Its time complexity is O(n) since it performs one recursive call per step.
- The brpower function splits the exponent in half each time, cutting down the number of steps needed. Its time complexity is O(log n) because each recursive step divides the problem by two.

To put it simply, rpower takes longer as n gets larger, while brpower finishes faster even when n is big.

Experimental Analysis

In an experiment using different values of n, the runtime results showed that rpower took noticeably longer for larger exponents, while brpower stayed very fast.

- Algorithm n = 10 n = 100 n = 1000

- rpower 15 µs 120 µs 950 µs
- brpower 5 µs 18 µs 32 µs

The results clearly show that rpower grows linearly while brpower grows much slower, matching the theoretical analysis.

Analysis and Comparison

Both the theory and experiment show that brpower is the more effective algorithm. It saves time by doing fewer recursive calls, while rpower repeats more steps as n increases.

This only proves that using smarter algorithms, like divide and conquer, can make a huge difference in performance. For large exponents, brpower is the better and faster option.

C. Conclusion & Lessons Learned

In this activity, I learned how the theoretical and experimental analyses work together to test the algorithms runtime behavior. Each of the test here showed how the growth rate of an algorithm affects its performance as the data increases. I also learned how quadratic algorithms like ($O(n^2)$) become slower with larger inputs, while logarithmic or optimized recursive algorithms perform much faster. This helped me see how important algorithm choice is when designing programs that handle bigger workloads.

Studying both theory and experiment made me realize that even small differences in approach can greatly improve runtime. And as a future CPE Engineer, I need to focus not just on writing code that works, but also on making it efficient, reliable, and scalable.

D. Assessment Rubric

T.I.P. S.O. 7 Rubric for Lifelong Learning (1)

Criteria	Ratings						Pts
Acquire and apply new knowledge from outside sources	**6 pts** **Excellent** Educational interests and pursuits exist and flourish outside classroom requirements, knowledge and/or experiences are pursued independently and applies knowledge learned into practice	**5 pts** **Good** Educational interests and pursuits exist and flourish outside classroom requirements, knowledge and/or experiences are pursued independently	**4 pts** **Satisfactory** Look beyond classroom requirements, showing interest in pursuing knowledge independently	**3 pts** **Unsatisfactory** Begins to look beyond classroom requirements, showing interest in pursuing knowledge independently	**2 pts** **Poor** Relies on classroom instruction only	**1 pts** **Very Poor** No initiative or interest in acquiring new knowledge	6 pts
Learn independently	**6 pts** **Excellent** Completes an assigned task independently and practices continuous improvement	**5 pts** **Good** Completes an assigned task without supervision or guidance	**4 pts** **Satisfactory** Requires minimal guidance to complete an assigned task	**3 pts** **Unsatisfactory** Requires detailed or step-by-step instructions to complete a task	**2 pts** **Poor** Shows little interest to complete a task independently	**1 pts** **Very Poor** No interest to complete a task independently	6 pts
Critical thinking in the broadest context of technological change	**6 pts** **Excellent** Synthesizes and integrates information from a variety of sources; formulates a clear and precise perspective; draws appropriate conclusions	**5 pts** **Good** Evaluate information from a variety of sources; formulates a clear and precise perspective	**4 pts** **Satisfactory** Analyze information from a variety of sources; formulates a clear and precise perspective.	**3 pts** **Unsatisfactory** Apply the gathered information to formulate the problem	**2 pts** **Poor** Gather and summarized the information from a variety of sources but failed to formulate the problem	**1 pts** **Very Poor** Gather information from a variety of sources	6 pts
Creativity and adaptability to new and emerging technologies	**6 pts** **Excellent** Ideas are combined in original and creative ways in line with the new and emerging technology trends to solve a problem or address an issue.	**5 pts** **Good** Ideas are creative and adapt the new knowledge to solve a problem or address an issue	**4 pts** **Satisfactory** Ideas are creative in solving a problem or address an issue	**3 pts** **Unsatisfactory** Shows some creative ways to solve the problem	**2 pts** **Poor** Shows initiative and attempt to develop creative ideas to solve the problem	**1 pts** **Very Poor** Ideas are copied or restated from the sources consulted	6 pts

Total Points: 24

E. External References

- GeeksforGeeks. (n.d.). Algorithm Analysis and Asymptotic Notation. Retrieved from https://www.geeksforgeeks.org

- W3Schools. (n.d.). C++ Recursion and chrono library. Retrieved from https://www.w3schools.com

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press.

- Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2015). Data Structures and Algorithms in C++ (2nd ed.). Wiley.