ACTIVITY NO. 11

BASIC ALGORITHM ANALYSIS	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:**
Section:	**Date Submitted:**
Name:	**Instructor:** Engr. Roman M. Richard

1. Objective(s)

Create an experimental and theoretical analysis of algorithms using different techniques and tools.

2. Intended Learning Outcomes (ILOs)

After this activity, the student should be able to:

- Measure the runtime of algorithms using theoretical analysis
- Analyze the results of runtime performance by comparing experimental and theoretical analysis

3. Discussion

INTRODUCTION

Designing "excellent" data structures and algorithms is something that interests us. An algorithm is a step-by-step process for carrying out some tasks in a finite amount of time, whereas a data structure is a methodical way of organizing and accessing data. We need accurate methods of examining certain data structures and algorithms before we can label them as "excellent." Characterizing the running time and space utilization of algorithms and data structure operations is a necessary step in analyzing a program's efficiency. Since time is valuable, the running time is a natural indicator of goodness.

Running Time Analysis

Most algorithms transform input objects into output objects. The running time of an algorithm or a data structure method typically grows with the input size, although it may also vary for different inputs of the same size. Also, the running time is affected by a lot of factors, such as the hardware environment and the software environment. In spite of the possible variations that come from different environmental factors, we would like to focus on the relationship between the running time of an algorithm and the size of its input. In the end, we would like to characterize an algorithm's running time as a function f(n) of the input size n.

But what is the proper way of measuring it?

- Experimental (or empirical) analysis
- Theoretical analysis

Common Functions used in Analysis

- **The Constant Function $f(n) = C$**
  Constant algorithm does not depend on the input size.
  Examples: arithmetic calculation, comparison, variable declaration, assignment statement, invoking a method or function.

- **The Logarithm Function $f(n) = logn$**
  Logarithm function gets slightly slower as n grows. Whenever n doubles, the running time increases by a constant.

Examples: Binary Search.

- **The Linear Function $f(n) = n$**
  Whenever n doubles, so does the running time.
  Example: print out the elements of an array of size n.

- **The N-Log-N Function $f(n) = nlogn$**
  This function grows a little faster than the linear function and a lot slower than the quadratic function (n2).
  Example: Merge sort.

- **The Quadratic Function $f(n) = n^2$**
  Quadratic algorithms are practical for relatively small problems. Whenever n doubles, the running time increases fourfold.
  Example: $n \times n$ array manipulation.

- **The Cubic Function and Other Polynomials**
  Whenever n doubles, the running time increases eightfold.
  Example: $n \times n$ matrix multiplication.

- **The Exponential Function $f(n) = b^n$**
  Exponential algorithm is usually not appropriate for practical use.
  Example: Towers of the Hanoi.

- **The Factorial Function $f(n) = n!$**
  Factorial function is even worse than the exponential function.
  For example, permutations of n elements.

Comparing Growth Rates

Our algorithms should ideally run in linear or n-log-n time, whereas data structure operations should execute in times proportional to the constant or logarithm function. While algorithms with exponential running times are impossible for any but the smallest sized inputs, those with quadratic or cubic running times are less useful.

	Constant	Logarithmic	Linear	N-Log-N	Quadratic	Cubic	Exponential
N	O(1)	O(logN)	O(N)	O(NlogN)	$O(N^2)$	$O(N^3)$	$O(2^n)$
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65,536
32	1	5	32	160	1,024	32,768	4,294,967,206
64	1	6	64	384	4,069	262,144	1.84×10^{19}

THEORETICAL ANALYSIS

A computer program should be totally correct, but it should also execute as quickly as possible (time-efficiency) and use memory wisely (storage-efficiency). How do we compare programs (or algorithms in general) with respect to execution time? Various computers run at different speeds due to different processors; compilers optimize code before execution; the same algorithm can be written differently depending on the programming paradigm.

SEARCHING ALGORITHM

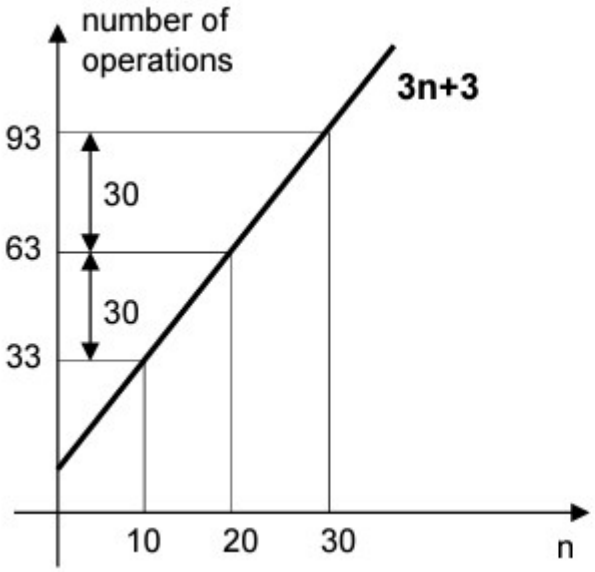
```
int search(int *x, int target){
    for(int i = 0; i < x.length; i++){
        if (x[i] == target) return i;
    }
    return -1;
}
```

```
int search(int *x, int target){
    for(int i = 0; i < x.length; i++){
        if (x[i] == target) return i;
    }
    return -1;
}
```

Let x.length = n	
How many times is operation 1 executed?	1
How many times is operation 5 and 6 executed in total?	1
Total so far:	2

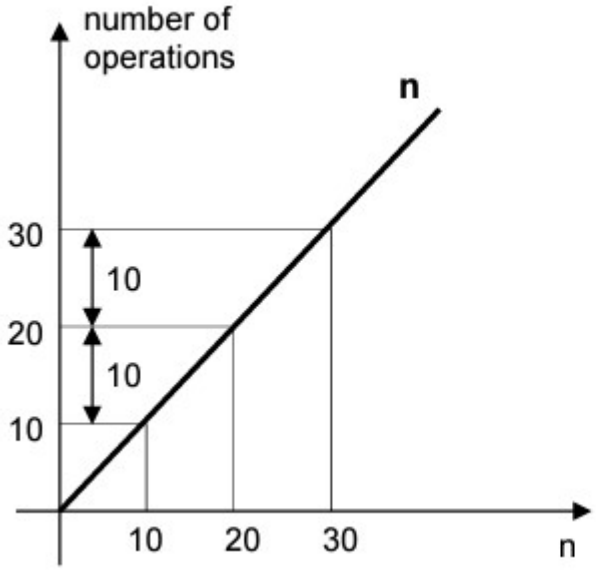
Worst case: The target is not the array.

How many times is operation 2 executed?	n+1
How many times is operation 3 executed?	N
How many times is operation 4 executed?	N
Total number of operations:	3n + 3



n (size of array)	number of operations
10	33
20	63
30	93

What if we just counted the comparison? (Operation 4)
How many times is the comparison executed? N times.

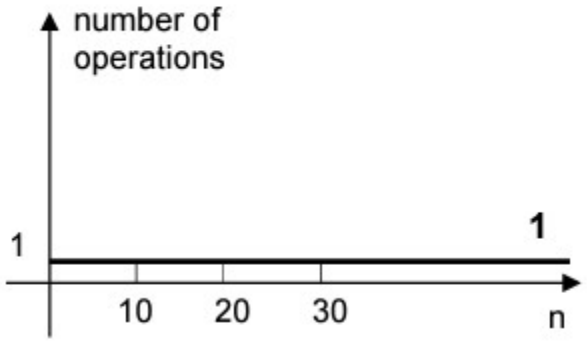


n (size of array)	number of operations
10	10
20	20
30	30

In both cases, the amount of work we do is linearly proportional to the number of data values in the array. If we have n data values in the array, and we double the size of the array, how much work will we do searching the new array in the worst case?

In general, it doesn't matter what we count as operations, as long as we are consistent. If we want to compare two algorithms that perform the same overall function, as long as we count the same type of operations in both, we can compare them for efficiency.

Checking out the best case:

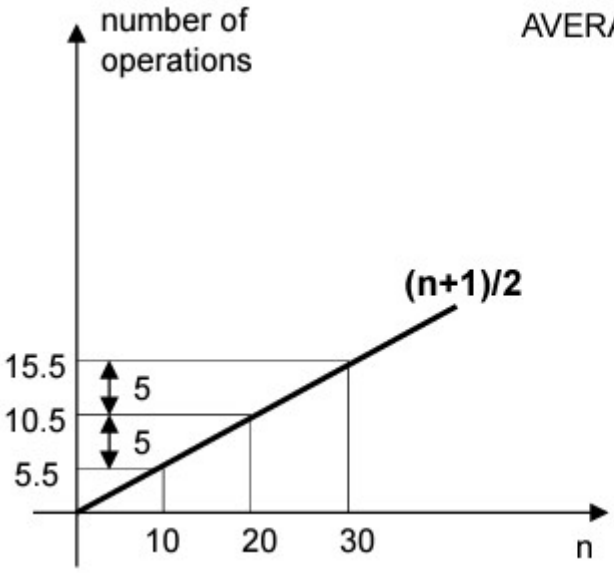


n (size of array)	number of operations
10	1
20	1
30	1

Checking out the average case:

How many comparisons are necessary in the average case for an array of n values? Assuming the target is in the array?

$$\frac{1 + 2 + \cdots + (n-1) + n}{n}$$



n (size of array)	number of operations
10	5.5
20	10.5
30	15.5

4. Materials and Equipment

Personal Computer with C++ IDE

Recommended IDE:

- CLion (must use TIP email to download)
- DevC++ (use the embarcadero fork or configure to **C++20**)

5. Procedure

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;
}
```

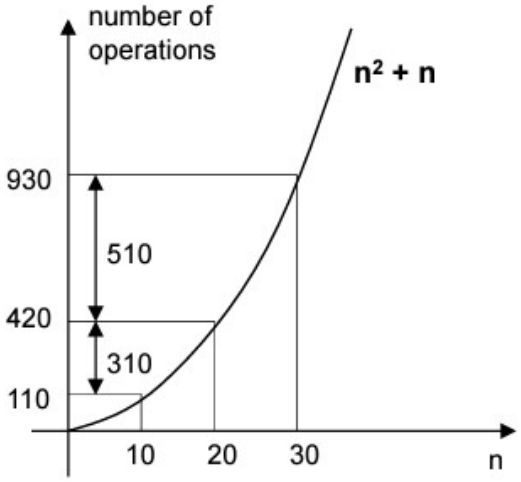
Performing worst case analysis:

- 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 worse case of the algorithm.



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

Analysis:

_____

_____

_____

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:

First Name	Last Name	Flu Shot?
1	3	Yes
2	2	No
2	3	Yes

Assume that all the names are positive integers and that the given list is sorted. Your task is to write a program that can look up the vaccination status of a given student in the list and outputs to the officials whether the student needs to be vaccinated. Students need to be vaccinated in case of two conditions:

- If they are not present in the list
- If they are present in the list but have not been administered a flu shot

Since the list can have a large number of students, your program should be as fast and efficient as possible. The final output of your program should look as follows:

```
Time taken to search = 45 microseconds
Student (836 118) needs vaccination.
Time taken to search = 37 microseconds
Student (836 118) needs vaccination.
Time taken to search = 53 microseconds
Student (836 118) needs vaccination.
```

The high-level steps for this implementation are as follows:

1. Represent each student as an object of the `Student` class.
2. Overload the required operators for the Student class so that a vector of students can be sorted using the STL `std::sort()` function.
3. Use a **binary search** to see if the student is present on the list.
4. If the student isn't present in the list, your function should return `true` since the student needs to be administered the vaccine.
5. Otherwise, if the student is present in the list but has not been administered the vaccine, return true.
6. Else, return false.

Step 0: Begin by including the following headers:

```
#include <iostream>
#include <vector>
#include <chrono>
#include <random>
#include <algorithm>
```

```
#include <numeric>
```

Step 1: Define the STUDENT class as follows:

```
class Student{
    private:
        std::pair<int, int> name;
        bool vaccinated;

    public:
        //constructor
        Student(std::pair<int, int> n, bool v) : name(n), vaccinated(v){
        }

        // Getters
        auto get_name() { return name; }
        auto is_vaccinated() { return vaccinated; }
```

Step 2: Overload the required operators for the Student class so that a vector of students can be sorted using the STL `std::sort()` function.

```
        // Two people are same if they have the same name
        bool operator ==(const Student& p) const {
            return this->name == p.name;
        }

        // The ordering of a set of people is defined by their name
        bool operator< (const Student& p) const {
            return this->name < p.name;
        }

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

Step 3: The following function lets us generate a student from random data:

```
auto generate_random_Student(int max) {
    std::random_device rd;
    std::mt19937 rand(rd());

    // the IDs of Student should be in range [1, max]
    std::uniform_int_distribution<std::mt19937::result_type> uniform_dist(1, max);

    // Generate random credentials
    auto random_name = std::make_pair(uniform_dist(rand), uniform_dist(rand));
    bool is_vaccinated = uniform_dist(rand) % 2 ? true : false;
    return Student(random_name, is_vaccinated);
}
```

For the next important function, we will implement the use of **std::chrono for experimental analysis**.

The following code is used to run and test the output of our implementation.

```
void search_test(int size, Student p) {
    std::vector<Student> people;

    // Create a list of random people
    for (auto i = 0; i < size; i++)
        people.push_back(generate_random_Student(size));

    std::sort(people.begin(), people.end());

    // To measure the time taken, start the clock
    std::chrono::steady_clock::time_point                  begin                  =
std::chrono::steady_clock::now();
    bool search_result = needs_vaccination(p, people);

    // Stop the clock
    std::chrono::steady_clock::time_point end = std::chrono::steady_clock::now();
    std::cout << "Time taken to search = " <<
        std::chrono::duration_cast<std::chrono::microseconds>
        (end - begin).count() << " microseconds" << std::endl;
    if (search_result)
        std::cout << "Student (" << p.get_name().first << " " << p.get_name().second
<< ") " << "needs vaccination." << std::endl;
    else
        std::cout << "Student (" << p.get_name().first << " " << p.get_name().second
<< ") " << "does not need vaccination." << std::endl;
}
```

5. The following function implements our logic for whether a vaccination is needed:

```
bool needs_vaccination(Student P, std::vector<Student>& people) {
    auto first = people.begin();
    auto last = people.end();
    while (true)
    {
        auto range_length = std::distance(first, last);
        auto mid_element_index = std::floor(range_length / 2);
        auto mid_element = *(first + mid_element_index);

        // Return true if the Student is found in the sequence and
        // he/she's not vaccinated
        if (mid_element == P && mid_element.is_vaccinated() == false)
            return true;
            else if (mid_element == P && mid_element.is_vaccinated() == true)
            return false;
        else if (mid_element > P)
            std::advance(last, -mid_element_index);
        if (mid_element < P)
            std::advance(first, mid_element_index);

        // Student not found in the sequence and therefore should be vaccinated
        if (range_length == 1)
            return true;
    }
}
```

6. Finally the driver code is implemented as follows:

```
int main() {
    // Generate a Student to search
    auto p = generate_random_Student(1000);
    search_test(1000, p);
    search_test(10000, p);
    search_test(100000, p);
    return 0;
}
```

Fill out the table below and include in section 6 table 11-1.

Input Size (N)	Execution Speed	Screenshot	Observation(s)

Provide your own analysis of the outcome listed in this table.

6. Output

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 to A[j] then
        return false
  return true
```

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

```
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
```

8. Conclusion

Provide the following:

- Summary of lessons learned
- Analysis of the procedure
- Analysis of the supplementary activity
- Concluding statement / Feedback: How well did you think you did in this activity? What are your areas for improvement?

9. Assessment Rubric