CC-213L

Data Structures and Algorithms

Laboratory 01

Algorithms Performance Analysis and Measurement

Version: 1.0.0

Release Date: 21-09-2023

Department of Information Technology

University of the Punjab

Lahore, Pakistan

Contents:

Learning Objectives:

- Defining Algorithm and Properties of Algorithm
- Performance Analysis
- Step Count for an Algorithm (Programming Statements)
- Measurement of Time and Space Complexity for an algorithm
- Formulation of Time Equation
- Representation of Performance in Asymptotic Notations

Resources Required:

- Desktop Computer or Laptop
- Microsoft ® Visual Studio 2022

General Instructions:

- In this Lab, you are **NOT** allowed to discuss your solution with your colleagues, even not allowed to ask how is s/he doing, this may result in negative marking. You can **ONLY** discuss with your Teaching Assistants (TAs) or Lab Instructor.
- Your TAs will be available in the Lab for your help. Alternatively, you can send your queries via email to one of the followings.

Teachers:		
Course Instructor	Prof. Dr. Syed Waqar ul Qounain	swjaffry@pucit.edu.pk
Lab Instructor	Madiha Khalid	madiha.khalid @pucit.edu.pk
Teacher Assistants	Muhammad Nabeel	bitf20m009@pucit.edu.pk
	Muhammad Amir	bsef21m013@pucit.edu.pk

Background and Overview:

Algorithm:

In computer science, an algorithm is a well-defined, step-by-step procedure or set of instructions that are designed to solve a specific problem or perform a particular task. Algorithms are fundamental to computing and play a crucial role in various areas such as programming, data processing, artificial intelligence, and more. They provide a systematic way to solve problems and automate tasks in a way that computers can understand and execute.

In Programming Fundamentals Course, We have discussed about different problems and solved those problems. The solution to a problem is basically an algorithm. We shall discuss about it further.

This is a Simple Algorithm that you have designed in your programming Fundamentals Course using **LARP** that prints the odd numbers from 1 up to 10.

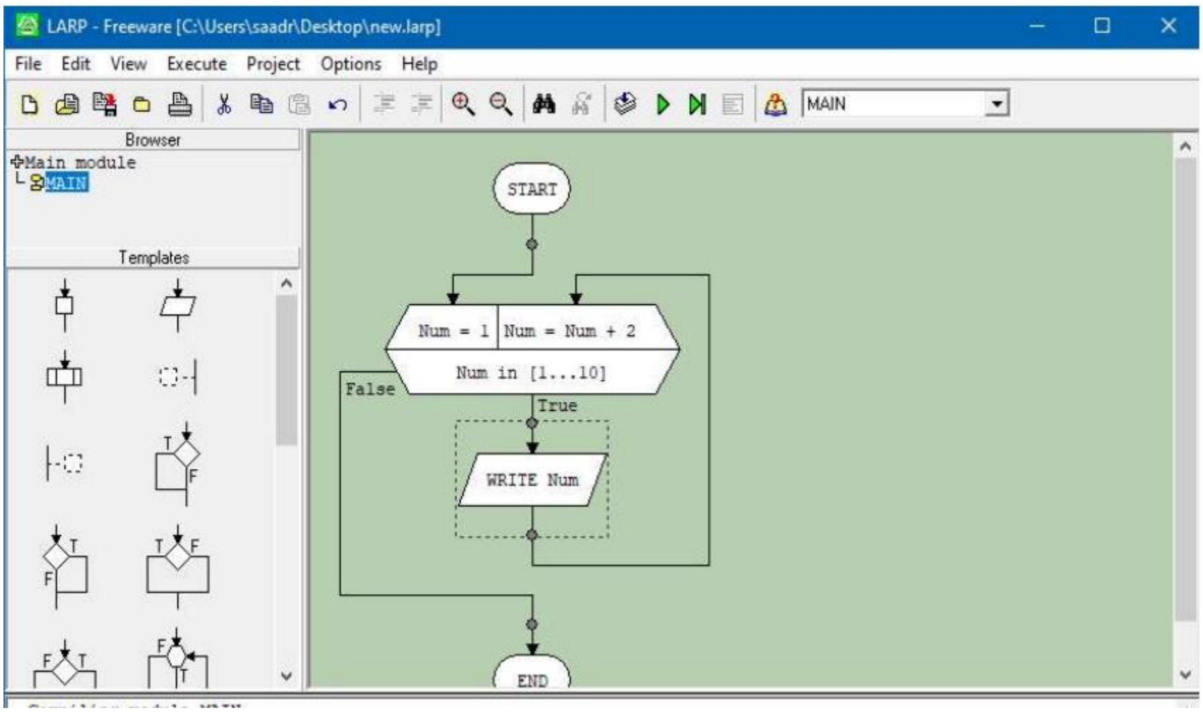

Figure. 1 (Diagrammatic Representation)

The programmatic implementation of this algorithm is follows.

```
#include <iostream>
using namespace std;
int main()
{
    for(int i=1;i<=10;i++)
    {
        if(i%2!=0)
            cout <<i <<endl;
    }
}
```

Figure. 2 (Programmatic Representation)

Output of the Algorithm is

```
1
3
5
7
9
```

Figure. 3 (Output)

This was a very simple algorithm that we discussed. Algorithms can also be very complex and their implementations may not be simple as above.

Properties of Algorithm:

Properties of an algorithm are essential characteristics or qualities that define a well-structured and effective algorithm. Here are some key properties of algorithms along with examples:

1. **Finiteness:** An algorithm must terminate after a finite number of steps.
   Example: In a sorting algorithm like Bubble Sort, the process continues until all elements are in their correct positions, which is a finite number of steps.

2. **Definiteness:** Each step of the algorithm must be precisely defined and unambiguous, leaving no room for interpretation.
   Example: In a pseudocode or programming language, statements like "increment a counter variable" or "swap two values" should be unambiguous.

3. **Input:** An algorithm should have zero or more inputs, which are values or data provided to it before execution.
   Example: In a simple addition algorithm, the input would be two numbers to be added.

4. **Output:** An algorithm should produce one or more outputs, which are the results or values generated after its execution.
   Example: In a multiplication algorithm, the output would be the product of two numbers.

5. **Effectiveness:** An algorithm should solve a specific problem or perform a particular task effectively. It should produce the correct result for all valid inputs.
   Example: A prime number checking algorithm should accurately determine whether a given number is prime or not.

6. **Determinism:** An algorithm's behavior at any step should be entirely determined by the inputs and its current state, ensuring repeatability.
   Example: A simple loop that iterates over an array and performs the same operation on each element is deterministic.

7. **Feasibility:** Algorithms should be implementable using available resources (time, memory, and computing power).
   Example: An algorithm to solve a complex mathematical problem may be theoretically sound but not feasible due to the time it would take to execute.

8. **Precision:** Each step of the algorithm should be well-defined and precise, without any ambiguity.
   Example: In a search algorithm, specifying the conditions for terminating the search precisely is essential.

9. **Clarity:** The algorithm should be expressed clearly and understandably, using a formal or semi-formal notation.

Example: Pseudocode or flowcharts can be used to represent algorithms in a clear and understandable manner.

10. **Modularity:** Algorithms can be divided into smaller, manageable modules or subroutines, which enhances readability and maintainability.
    Example: Breaking down a complex sorting algorithm into smaller functions for comparison and swapping.

11. **Optimality:** An optimal algorithm should produce the correct output while minimizing resource usage (e.g., time, memory).
    Example: Some sorting algorithms, like QuickSort or MergeSort, aim for optimal time complexity.

12. **Generality:** Algorithms can be designed to solve a broad class of problems, not just a specific instance.
    Example: The binary search algorithm can be used to find elements in any sorted list, not just one particular list.

Above properties ensure that an algorithm is well-structured, efficient, and reliable for solving computational problems. When designing or evaluating algorithms, it's crucial to consider these properties to achieve desired outcomes.

Performance Analysis:

The performance of an algorithm refers to how efficiently and effectively it solves a specific computational problem or task. It is a measure of how well an algorithm utilizes computational resources such as time, memory, and processing power to achieve its objective. Performance evaluation is crucial when comparing and selecting algorithms for a given task. Key aspects of algorithm performance include:

1. **Resource Utilization:** It involves evaluating how efficiently the algorithm uses computational resources, such as CPU utilization and memory allocation. Effective resource utilization ensures that the algorithm doesn't waste resources unnecessarily.

    i. **Time Complexity:** Time complexity quantifies the amount of time or the number of basic operations an algorithm requires to complete its execution as a function of the input size. It provides an upper bound on the algorithm's running time.

    ii. **Space Complexity:** Space complexity measures the amount of memory or storage space an algorithm uses to solve a problem, also as a function of the input size. It helps assess the algorithm's memory efficiency.

2. **Scalability:** Algorithms should be scalable, meaning that they continue to perform well as the input size or problem complexity increases. Scalable algorithms maintain reasonable performance even with larger datasets.

3. **Robustness:** A robust algorithm can handle a wide range of inputs and conditions without failing or producing incorrect results. It should gracefully handle edge cases and unexpected inputs.

4. **Parallelization:** In modern computing environments, the ability to parallelize tasks is essential. Algorithms that can be easily parallelized can take advantage of multiple processors or cores, improving performance.

5. **Energy Efficiency:** In mobile and battery-powered devices, minimizing energy consumption is critical. Energy-efficient algorithms aim to complete tasks with minimal energy usage.

Often, there are trade-offs between different aspects of performance of an algorithms. For example, achieving a faster execution time may require more memory usage. Evaluating these trade-offs is a key part of algorithm design. Some algorithms can adapt their behavior or resource usage based on the available resources or system conditions. Adaptive algorithms can optimize performance in dynamic environments.

Measuring and optimizing algorithm performance involves a combination of theoretical analysis, empirical testing, and benchmarking. Performance evaluation helps determine the suitability of an algorithm for a specific task and guides decisions on algorithm selection, implementation, and optimization.

Example 01:

We are aware that when we compile our C++ program, it ultimately results in the creation of an executable file (.exe). This file is then executed by the processor of our laptop or PC. While the program runs, it utilizes memory and consumes processing time. To illustrate this process, let's consider a basic C++ program as an example.

```cpp
#include <iostream>
using namespace std;
int main()
{
    for(int i=1;i<=100;i++)
        cout << i <<endl;
}
```

Figure. 4 (Print numbers)

This program essentially displays numbers ranging from 1 to 100. Let's assume that each statement within our main function takes 1 second to complete. In this case, the loop header will execute 101 times, and the print statement will execute 100 times. This totals 201 steps, implying that the execution will consume 201 seconds.

Example 02:

```cpp
void displayArray(int a[], int N)
{
    for (int i=0; i<N; i++)          1 * (N + 1)
    {
        cout<<a[i]<<":";             1 * N
    }
}
                                     TOTAL : 2N + 1
```

Figure. 5 (Print Array)

The array's size is denoted as N, which causes the loop header to execute N+1 times, including the check for the False condition. However, the loop's body executes one time less than the header, resulting in N executions of the internal statement. The total execution time is the cumulative sum of all steps. Therefore, the time equation can be expressed as follows:

$F(n) = (N + 1) + N = 2N + 1$

Time Complexity:

The time complexity of an algorithm is usually analyzed in terms of the number of basic operations (such as comparisons, assignments, arithmetic operations, etc.) that the algorithm performs as a function of the input size (usually denoted as "n"). The goal of analyzing time complexity is to determine how the algorithm's performance behaves as the input size grows larger. In other words, it quantifies how the algorithm's execution time grows with larger or more complex inputs. Time complexity is often expressed using Big O notation, which provides an upper bound on the growth rate of the algorithm's running time.

For example, an algorithm with a time complexity of O(N) indicates that its execution time grows linearly with the size of the input, while an algorithm with a time complexity of O(N^2) suggests a quadratic growth rate, where the execution time increases much faster as the input size increases. The goal in algorithm design is often to minimize time complexity to achieve faster and more efficient solutions.

Asymptotic Notations:

Asymptotic notations are mathematical tools used in computer science and mathematics to describe and analyze the performance or efficiency of algorithms in relation to their input size. They provide a concise way to express how the running time or space requirements of an algorithm grow as the input size becomes very large. There are three commonly used asymptotic notations: Big O notation, Omega notation, and Theta notation. Here's a brief explanation of each with examples:

1. **Big O Notation (O-notation):**

Big O notation provides an upper bound on the growth rate of an algorithm. It describes the worst-case scenario in terms of time or space complexity. O(f(n)) represents that the algorithm's resource usage grows at most as fast as the function f(n) as n (the input size) becomes large.

Example 01: If an algorithm's time complexity is O(log n), it means that the running time grows logarithmically with the input size. For instance, binary search has a time complexity of O(log n).

Example 02: This notation denotes the maximum time an algorithm may consume, considering the most unfavorable input scenario. It establishes an upper threshold for the algorithm's completion time. This measure is frequently the most pertinent in assessing algorithm performance because it guarantees that the algorithm will not exceed this limit for any input. For instance, when searching for an element in an array that is not present in the array, this situation represents a worst-case scenario for the algorithm's execution time.

```cpp
#include <iostream>
using namespace std;
bool findElement(const int * arr, int size, int element=0)
{
    for(int i=0;i<size;i++)
    {
        if(element==arr[i])
            return true;
    }
    return false;
}
int main()
{

    const int N=10;  // size of Array
    int arr[N]= { 2, 4, 12, 45, 6, 23, 9, 68, 90, 100}; // static Array
    int find=10; // best case element is in first index
    findElement(arr, N, find)?cout << find <<" is present in Array ": cout << find << " is not present in Array" <<endl;
    return 0;

```

Figure. 6 (Programmatic Representation)

In this scenario, you must iterate through the entire array even though the element you're searching for is not present. The worst-case complexity, in this case, would be denoted as O(size) or O(N).

```
10 is not present in Array
```

Figure. 7 (Output)

2. Omega Notation (Ω-notation):

Omega notation provides a lower bound on the growth rate of an algorithm. It describes the best-case scenario in terms of time or space complexity. Ω(f(n)) represents that the algorithm's resource usage grows at least as fast as the function f(n) as n becomes large.

Example 01: If an algorithm's time complexity is Ω(n^2), it means that the running time will not grow slower than n^2. This can represent the best-case scenario for an algorithm like quicksort in some situations.

Example 02: This notation signifies the lowest possible time requirement for an algorithm, considering the most favorable input scenario. It specifies the minimum time an algorithm will necessitate to finish a task. For instance, when searching for an element in an array, and the element happens to be located at the first index of the array, the best-case scenario would be denoted as Ω(1).

```
#include <iostream>
using namespace std;
bool findElement(const int * arr, int size, int element=0)
{
    for(int i=0;i<size;i++)
    {
        if(element==arr[i])
            return true;
    }
    return false;
}
int main()
{

    const int N=10;  // size of Array
    int arr[N]= { [0]: 2, [1]: 4, [2]: 12, [3]: 45, [4]: 6, [5]: 23, [6]: 9, [7]: 68, [8]: 90, [9]: 100}; // static Array
    int find=2; // best case element is in first index
    findElement(arr,  size: N,  element: find)?cout << find <<" is present in Array ": cout << find << " is not present in Array" <<endl;
    return 0;

}
```

Figure. 8 (find Element)

Output of Code:

```
2 is present in Array
```

Figure. 9 (Output)

3. Theta Notation (Θ-notation):

Theta notation provides a tight bound on the growth rate of an algorithm. It describes both the upper and lower bounds, indicating that the resource usage grows at the same rate as a given function. Θ(f(n)) represents that the algorithm's resource usage grows at the same rate as the function f(n) for large n.

Example 01: If an algorithm's time complexity is Θ(n), it means that the running time grows linearly with the input size. A simple linear search in an unsorted array has a time complexity of Θ(n).

Example 02: This notation characterizes the mean or expected running time of an algorithm, encompassing all conceivable inputs. It offers a more precise estimate of the algorithm's performance on typical inputs, although it can pose greater analytical challenges. For instance, in the average case, assuming the element resides at the center of the array, at index size/2, the time complexity is represented as Θ(size/n).

```cpp
#include <iostream>
using namespace std;
bool findElement(const int * arr, int size, int element=0)
{
    for(int i=0;i<size;i++)
    {
        if(element==arr[i])
            return true;
    }
    return false;
}
int main()
{

    const int N=10;  // size of Array
    int arr[N]= { 2, 4, 12, 45, 6, 23, 9, 68, 90, 100}; // static Array
    int find=45; // best case element is in first index
    findElement(arr, N, find)?cout << find <<" is present in Array ": cout << find << " is not present in Array" <<endl;
    return 0;

}
```

Figure. 10 (Algorithm)

These notations are essential for analyzing and comparing algorithms, helping us understand how efficiently they perform as the input size becomes very large and aiding in algorithm selection for specific tasks.

Activities:

Pre-Lab Activities:

Algorithms:

You have previously delved into algorithms extensively during your Programming Fundamentals course. This is a straightforward algorithm designed to convert a binary string, which represents a binary number, into its decimal (denary) equivalent.

```cpp
#include <iostream>
#include <string>
#include <cmath>

int binaryToDecimal(const std::string& binary) {
    int decimal = 0;
    int size = binary.size();

    for (int i = 0; i < size; ++i) {
        if (binary[size - i - 1] == '1') {
            decimal += std::pow( x: 2,  y: i);
        }
    }

    return decimal;
}
```

Figure. 11 (Binary to Decimal)

Now, attempt to formulate the time equation for this algorithm.

Before delving directly into the performance analysis of algorithms, it's essential to have a solid grasp of the big-O notation and its concepts. In this context, you may find it beneficial to refer to the recommended book below as a starting point, and afterward, proceed to work on solving practice questions.

Helping Material:

Book B - Data Structures and Algorithms in C++ (2nd Ed. by Adam Drozdek) 2.1 to 2.4

If you encounter difficulty understanding the material in this book, you have the option to turn to YouTube for additional resources. It's much better to comprehend through visualization. There's a well-explained video by an individual who has done an excellent job illustrating this concept:

https://youtu.be/Q_1M2JaijjQ?si=iV-eFvVHmUz30Xyp

Practice Questions:

Now that you have a clear grasp of what big-O notation entails, here are several equations for which you need to determine their respective big-O notations.

Step Count Complexity Equation:		Big O
4n^2 + 2n + 5	=	
(n^2 + 3) * log (n)	=	
(n + 1) * log (n^2 + 1)	=	
2^n + n ^ 10 + log (n)	=	
$log2(n)$ + 10	=	
2^n + n! + 9	=	

Figure. 12 (O Equations)

Note: Additionally, ensure that you have a firm grasp of the concepts related to arithmetic series and logarithms, as this understanding will prove to be beneficial.

Step counting of an algorithm:

It is a method used to analyze and understand the performance of an algorithm by counting the number of basic operations or steps executed by the algorithm as a function of the input size. These basic operations include arithmetic operations, comparisons, assignments, and other fundamental actions performed by the algorithm. The goal of step counting is to determine the algorithm's time complexity, which describes how the number of operations scales with respect to the input size.

Formulation of a time complexity equation involves expressing the algorithm's performance in terms of big O notation. It provides an upper bound on the growth rate of the algorithm's running time as a function of the input size. In other words, it describes how the algorithm's execution time increases as the input size becomes larger.

Now, let's explore step counting and time complexity formulation with examples in C/C++:

Example 1: A Simple C code

You will receive a code snippet, and your objective is to determine the number of times each line within the code will be executed. To illustrate this concept, here is a straightforward example:

for (int i = 0; i < n; i++) {	1 + (n+1) + n
If (i % 2 == 0)	N
print(“Yes”)	n / 2
else print(“No”)	n / 2
if (check == 1)	N
fun(n);	n * time complexity of fun(n)
}	

Figure.13 (Time Equation)

The line containing for loop have three statements initialization, condition, and increment, these would be executed 1, (n+1) and n times respectively. Keep in mind that you have to consider the worst cases of an algorithm and that’s why in the above example we assumed that the variable check is true and so the function fun () is being called each time. Time equation of the snippet is the sum of all these terms written on the right side against each line. If we assume the time complexity of function fun () to be n, the equation goes as follows:

f(n) = 1 + (n+1) + n + n + n/2 + n/2 + n + (n*n)

f(n) = n^2 + 5n + 2

So, O(f(n)) = O(n^2)

This algorithm runs in quadratic time (O(n^2)).

Space Complexity Analysis:
Similarly, you can conduct an analysis of space complexity in almost the same manner. The primary distinction lies in the fact that you don't need to aggregate all the terms listed alongside each line. Your task is to observe the maximum amount of space utilized by the algorithm during a specific time interval.

When utilizing a static or dynamic array, the program allocates space corresponding to the size of the array. An algorithm can be categorized as having constant space complexity if the maximum space allocated by the algorithm remains constant.

Example 1: Constant Space Complexity
Consider this code snippet as an example. In the space_complexity function, no extra space is allocated; it simply prints the string "Space Complexity" on the console.

void space_complexity () {	0 bytes
for (int i = 0; i < 10; i++) {	2 bytes for variable i
count << "Space Complexity"	0 bytes
}	0 bytes
}	
int main () {	0 bytes
space_complexity();	0 bytes
}	0 bytes

Figure. 14 (Space Equation)

The main() function doesn't declare any variables, resulting in a space complexity of 0 bytes. However, when the main() function calls the space_complexity() function, it declares an integer counter variable that consumes 2 bytes in RAM. Once the space_complexity() function exits, the local variable "i" disappears, and the main program terminates. To calculate the space complexity of the program, we can use the following equation:

f(n) = 0 + 0 + 0 + 2 + 0 + 0 + 0

f(n) = 2

Therefore,

O(f(n)) = O(1).

This program operates with constant space complexity (O(1)).

Example 2: Variable Space Complexity
Here's another example:

Consider this code snippet as an example which dynamically allocate memory in RAM (heap of the program memory area).

for (int i = 0; i < n; i++) {	2 bytes for variable i
bool* arr = new bool [n];	n bytes
delete[] arr;	0 bytes
}	0 bytes

Figure. 15 (Space Complexity)

In this code snippet, we declare an integer counter variable that occupies 2 bytes in RAM. During each iteration of the loop, memory is allocated in RAM for 'n' boolean variables, which is subsequently deallocated using the delete operator. Once the loop exits, the local variable "i" disappears. To determine the worst-case space complexity of this code, we can follow this equation:

$$f(n) = 2 + n + 0 + 0$$

$$f(n) = 2 + n$$

Therefore,

$$O(f(n)) = O(n).$$

This program exhibits linear space complexity in relation to 'n' (O(n)). It's important to note that despite allocating an array of size 'n' 'n' times, memory is deallocated after each allocation. Thus, at any given time, no more than 'n' bytes of memory are in use. Consequently, the space complexity is O(n), and not O(n^2).

Example 3: Linear Search in C
Step Count Analysis:

- Initialization: Initializing **i** to 0 requires one step.
- Loop Initialization: Initializing **i** once requires one step.
- Loop Iteration: In each iteration, we perform a comparison (**arr[i] == target**) and an increment of **i**. These two operations are performed **size** times.
- Return Statement (if the element is found): If the element is found, we execute the return statement, which requires one step.
- Return Statement (if the element is not found): If the element is not found, we execute the return statement with **-1**, which requires one step.

Step Count Total:

- The loop iterates for **size** times, and for each iteration, we have two constant operations (comparison and increment). Therefore, the total step count for the loop is **2 * size** steps.

Time Complexity Equation: The time complexity of the linear search algorithm in this case is expressed as follows:

$$T(n) = O(n)$$

Here, **n** represents the size of the array, and the algorithm's time complexity is linearly proportional to the size of the array.

```c
#include <stdio.h>

int linearSearch(int arr[], int size, int target) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {
            return i;
        }
    }
    return -1;
}

int main() {
    int arr[] = {2, 4, 6, 8, 10};
    int target = 6;
    int size = sizeof(arr) / sizeof(arr[0]);

    int result = linearSearch(arr, size, target);

    if (result != -1) {
        printf("Element found at index %d\n", result);
    } else {
        printf("Element not found\n");
    }

    return 0;
}
```

Figure. 16 (Binary Search)

Example 3: Bubble Sort in C++

Step Count Analysis:

- Initialization: Initializing **i** to 0 and **j** to 0 requires two steps.
- Outer Loop: The outer loop (**for i**) iterates for **(size - 1)** times, where **size** is the size of the array.
- Inner Loop: The inner loop (**for j**) iterates for **(size - i - 1)** times in each iteration of the outer loop.
- Comparison and Swap: In each iteration of the inner loop, we perform a comparison (**arr[j] > arr[j + 1]**) and possibly a swap operation.
- Return Statement: The main function doesn't have a return statement.

Step Count Total:
The total step count for the bubble sort algorithm depends on the number of comparisons and swaps performed. Bubble sort has a time complexity of O(n^2), where **n** is the size of the array.

Time Complexity Equation:
The time complexity of bubble sort is expressed as follows:

T(n) = O(n^2)

Here, **n** represents the size of the array, and the algorithm's time complexity is quadratic in terms of the size of the array.

```
#include <iostream>

void bubbleSort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap arr[j] and arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int size = sizeof(arr) / sizeof(arr[0]);

    bubbleSort(arr, size);

    std::cout << "Sorted array: ";
    for (int i = 0; i < size; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Figure. 17 (Bubble Sort)

Task 01: Step Count and construction of a Time Equation

Time Equation:

Typically, we evaluate the number of steps associated with each programming statement and then construct an equation that informs us about the time complexity of a given algorithm. This equation provides insight into the time spent by the algorithm for a specific problem size during the given iteration.

Derive the time equation and find its big-O notation for the following snippets:

Code 01:

```
sum = 0;
for( i = 0; i < n; ++i )
    for( j = 0; j < n; ++j )
        ++sum;
sum = 0;
for( i = 0; i < n; ++i )
    for( j = 0; j < n * n; ++j )
        ++sum;
sum = 0;
for( i = 0; i < n; ++i )
    for( j = 0; j < i; ++j )
        ++sum;
sum = 0;
for( i = 0; i < n; ++i )
    for( j = 0; j < i * i; ++j )
        for( k = 0; k < j; ++k )
            ++sum;
```

Figure. 18 (Code)

Code 02:

```
count = 0
for (int i = N; i > 0; i /= 2)
  for (int j = 0; j < i; j++)
    count++;
```

Figure. 19 (Code)

Task 02: Space Complexity Analysis

Find the space equation as well as its big-O notation for the following snippets:

Code 01:

```
void fun(int n)
{
    for (int i = 1; i <= n; i *= 2)
    {
        bool* arr_2 = new bool[n];
        bool* arr_1 = new bool[2*n];
    }
}
```

Figure. 20 (Code)

Code 02:

```
void fun(int n)
{
    for (int i = 0; i <= pow(2, n); i++)
    {
        bool* arr = new bool[5];
    }
}
```

Figure. 21 (Code)

Code 03:

```
void fun(int n)
{
    for (int i = 0; i <= pow(2, n); i++)
    {
        bool* arr = new bool[5];
    }
}
```

Figure. 22 (Code)

Code 04:

```
void fun(int n)
{
    for (int i = 0; i < 10; i++)
    {
        bool* arr_1 = new bool[n];
    }
}
```

Figure. 23 (Code)

Code 05:

```
void fun(int n)
{
    for (int i = n; i >= 1; i--)
    {
        bool* arr_1 = new bool[i];
    }
}
```

Figure. 24 (Code)

In-Lab Activities:

Code the following function

1. int fnLinearSearch (int Array [], unsigned int Size, int SearchKey);
   Precondition: None
   Post condition: return first index of search key in the array if found, or -1 otherwise
2. int fnBinarySearch (int Array [],unsigned int Size, int SearchKey);
   Precondition: Array is sorted in ascending order
   Post condition: return first index of search key in the array if found, or -1 otherwise
3. void fnBubbleSort (int Array [],unsigned int Size, int SortKey);
   Precondition: None
   Post condition: Array should be ordered according to sort key, in ascending order if SortKey=0 and in descending order otherwise
4. void fnSelectionSort (int Array [],unsigned int Size, int SortKey);
   Precondition: None
   Post condition: Array should be ordered according to sort key, in ascending order if SortKey=0 and in descending order otherwise

Task 01: Perform the complexity analysis for the following.

Write a driver program that populates an array of size **n** with random values from a specified range of values. Select a random SearchKey from the same set of values and fill the following table after the execution of fnLinearSearch.

Exp. No.	**Input Size**	**Input Rang**	**Search Key**	**Instruction Executed**
1				
2				
3				

Write a driver program that populate an array of size n with random values from a specified range of values and sort it in ascending order. Select a random search key from the same set of values and fill the following table after the execution of fnBinarySearch

Exp. No.	**Input Size**	**Input Rang**	**Search Key**	**Instruction Executed**
1				
2				
3				

Write a driver program that populates an array of size n with random values from a specified range of values. Sort the instance array using fnBubbleSort

Exp. No.	**Input Size**	**Input Rang**	**Instruction Executed**
1			
2			
3			

Write a driver program that populates an array of size n with random values from a specified range of values. Sort the instance array using fnSelectionSort

Exp. No.	Input Size	Input Rang	Instruction Executed
1			
2			
3			

Task 02: Calculate the running time complexity for the following.

Use the following driver program, get system time before the execution and after completion of above functions and write your calculated time for execution below in the table.

Driver Program:

```
#include <stdio.h>
#include <dos.h>
#include <constream.h>
int main(void) {
    struct time First, Second;
    clrscr();
    gettime(&First);
        //
        // Write your function call here
        //
    gettime(&Second);
        //
        // Compute the difference and display as following
        //
    printf("The current time is: %2d:%02d:%02d.%02d\n", First.ti_hour, First.ti_min,
    First.ti_sec, First.ti_hund);
    printf("The current time is: %2d:%02d:%02d.%02d\n", Second.ti_hour, Second.ti_min,
    Second.ti_sec, Second.ti_hund);
    return 0;
}
```

Function	Input Size	Input Rang	Search/Sort Key	Execution Time
fnLinearSearch				
fnBinarySearch				
fnBubbleSort				
fnSelectionSort				

Post-Lab Activities:

Why efficient solution?

We've been struggling to figure out how well our computer programs work, especially when they're doing complicated tasks. Now, you might wonder why it's important to make our programs work faster. In this lab, we'll show you why it matters and use pictures to help explain why some programs are much better than others when dealing with big jobs.

For a specific job, we'll start by doing it the easy way, the way that makes sense to most people. After that, we'll try doing the same job using a smart, faster method.

Let's say the job is finding all the prime numbers in a certain range of numbers. We'll compare the easy way with the smart way. The smart way uses something called the Sieve of Eratosthenes, which is a clever technique that's really good at finding prime numbers quickly.

The Naive Approach - Trial Division for Prime Numbers

To begin, let's examine a simple method for generating prime numbers. This approach, known as the trial division method, involves individually checking each number to see if it's a prime. The time complexity of the function that follows is approximately $O(n\sqrt{n})$.

```
void prime_numbers(int limit) {
    for (int num = 2; num <= limit; ++num) {
        bool is_prime = true;

        if (num <= 1)
            is_prime = false;

        for (int i = 2;  i *i  <  num; ++i) {
            if (num % i == 0) {
                is_prime = false;
                break;
            }
        }

        if (is_prime) {
            std::cout << num << " ";
        }
    }
}
```

Figure. 25 (Prime Number Naive Approach Code)

Sieve of Eratosthenes Algorithms for Prime Number:

Next, we'll delve into the Sieve of Eratosthenes, which is a much more efficient method for generating prime numbers. This algorithm methodically removes numbers that are not prime, making it considerably faster. The running time of this algorithm is $O(n * \log(\log n))$, which is much quicker than $O(n\sqrt{n})$. Here's an explanation of how it operates:

```cpp
void SieveOfEratosthenes(int n)
{
    // Create a boolean array "prime[0..n]" and initialize
    // all entries it as true. A value in prime[i] will
    // finally be false if i is Not a prime, else true.
    bool prime[n + 1];
    memset(prime, true, sizeof(prime));

    for (int p = 2; p * p <= n; p++) {
        // If prime[p] is not changed, then it is a prime
        if (prime[p] == true) {
            // Update all multiples of p greater than or
            // equal to the square of it numbers which are
            // multiple of p and are less than p^2 are
            // already been marked.
            for (int i = p * p; i <= n; i += p)
                prime[i] = false;
        }
    }

    // Print all prime numbers
    for (int p = 2; p <= n; p++)
        if (prime[p])
            cout << p << " ";
}
```

Figure. 26 (Sieve of Eratosthenes Algorithms for Prime Number Code)

Comparison of both algorithms:

We will use a graphing tool to visually evaluate the performance of both of these algorithms. Specifically, we are making use of the graphing tool accessible on desmos.com.

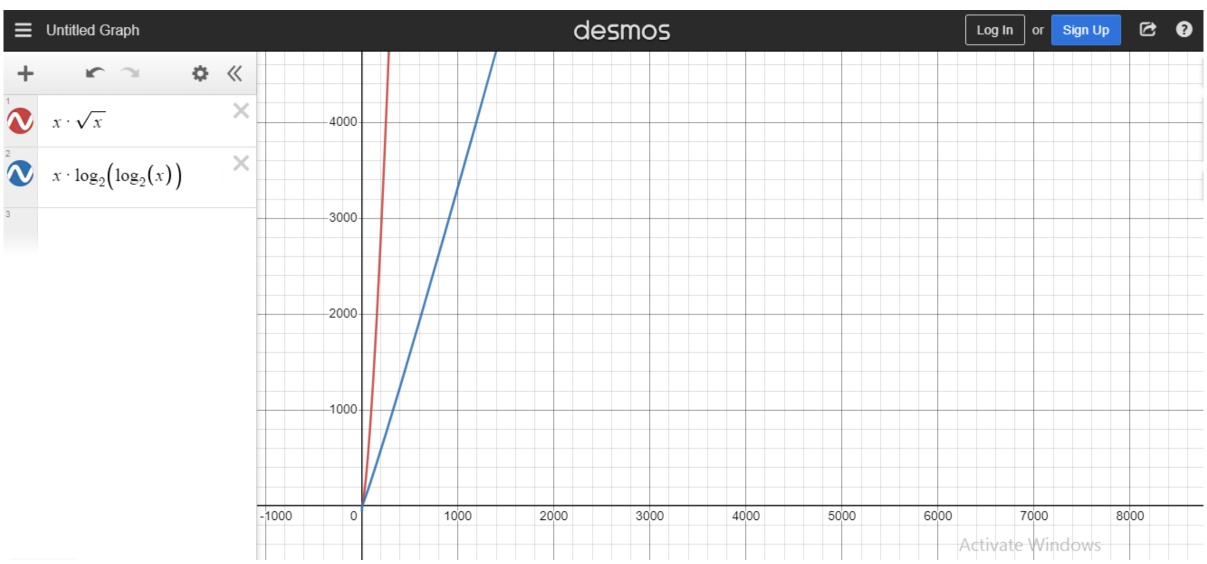

Figure. 27 (Comparison of Both Algorithms)

For the large input, say one million, simple algorithm will take almost 25 times more time than efficient one!

Task 01: Efficient Approach

1.1 Maximum Subarray Sum

The task at hand is to find the maximum sum of a contiguous subarray within a given array of integers. In other words, we need to identify a subarray with consecutive elements such that the sum of its elements is as large as possible.

To tackle this problem, we'll begin by implementing a straightforward, brute force algorithm, which has a time complexity of O(n^3). Subsequently, we'll introduce Kadane's algorithm for a more efficient solution. We will then analyze the performance of both algorithms using a graph. Your objective is to provide an efficient solution to this problem.

Example:
Input: [-2, 1, -3, 4, -1, 2, 1, -5, 4]
Output: The maximum subarray sum is 6.
In this example, the subarray [4, -1, 2, 1] has the largest sum of 6.

1.2 Anagram Detection

An anagram detection problem involves determining whether two given strings are anagrams of each other. Anagrams are words or phrases that have the same characters but in a different order. For instance, "listen" and "silent" are anagrams.

Write a program to solve the anagram detection problem using the approach of character count arrays. You are given two input strings, and your task is to determine if they are anagrams.

Example:
String 1: "gram"
String 2: "mgra"
Output: The given strings are anagrams.

In this example, the strings "gram" and "mgra" have the same characters with the same frequencies, but in different orders, making them anagrams. Your program should be able to handle strings of varying lengths and should perform a case- insensitive comparison (Gram and mgra are also anagrams).

1.3 Merge Sorted Arrays

You are given two sorted arrays of integers, and your task is to merge them into a single sorted array. Write a program to efficiently merge two sorted arrays into one sorted array.

Example:
Input:
Array 1: [2, 5, 8, 12, 16]
Array 2: [4, 7, 9, 10, 14, 18]
Output: Merged Sorted Array: [2, 4, 5, 7, 8, 9, 10, 12, 14, 16, 18]

In this example, the two sorted arrays are merged into a single sorted array containing all the elements in ascending order.

Submissions:

- For In-Lab Activity:
  - Save the files on your PC.
  - TA's will evaluate the tasks offline.
- For Pre-Lab & Post-Lab Activity:
  - Submit the .c file on Google Classroom and name it to your roll no.

Evaluations Metric:

- All the lab tasks will be evaluated offline by TA's
- **Division of Pre-Lab marks:** **[40 marks]**
  - Task 01: Step Count and Time Equation [20 marks]
  - Task 02: Space Complexity Analysis [20 marks]
- **Division of In-Lab marks:** **[70 marks]**
  - Task 01: Perform the Complexity Analysis [35 marks]
  - Task 02: Calculate the Running Time Complexity [35 marks]
- **Division of Post-Lab marks:** **[40 marks]**
  - Task 01(1.1): Maximum Sub-Array Sum [10 marks]
  - Task 01(1.2): Anagram Detection [10 marks]
  - Task 01(1.3): Merge Sorted Arrays [10 marks]

References and Additional Material:

- Time Complexity and Space Complexity
  https://www.geeksforgeeks.org/time-complexity-and-space-complexity/

- Asymptotic Notations and How to find them
  https://www.geeksforgeeks.org/asymptotic-notations-and-how-to-calculate-them/?ref=ml_lbp

Lab Time Activity Simulation Log:

- Slot – 01 – 00:00 – 00:15: Class Settlement
- Slot – 02 – 00:15 – 00:30: In-Lab Task 01
- Slot – 03 – 00:30 – 00:45: In-Lab Task 01
- Slot – 04 – 00:45 – 01:00: In-Lab Task 01
- Slot – 05 – 01:00 – 01:15: In-Lab Task 01
- Slot – 06 – 01:15 – 01:30: In-Lab Task 01
- Slot – 07 – 01:30 – 01:45: In-Lab Task 02
- Slot – 08 – 01:45 – 02:00: In-Lab Task 02
- Slot – 09 – 02:00 – 02:15: In-Lab Task 02
- Slot – 10 – 02:15 – 02:30: In-Lab Task 02
- Slot – 11 – 02:30 – 02:45: In-Lab Task 02
- Slot – 12 – 02:45 – 03:00: Discussion on Post-Lab Task