****

**Green University of Bangladesh**

**Department of Computer Science and Engineering (CSE)**

**Faculty of Sciences and Engineering**

**Semester: (Fall, Year:2023), B.Sc. in CSE (Day)**

**Lab Report NO: 01**

**Course Title : Algorithms Lab**

**Course Code : CSE 204**

**Section : 221 D9**

**Lab Experiment Name: Detecting Cycles in an Undirected Graph using BFS**

**Student Details**

|  |  |  |
| --- | --- | --- |
| **Name** | | **ID** |
| **1.** | Jahidul Islam | 221002504 |

**Lab Date : 02-09-2023**

**Submission Date : 09-10-2023**

**Course Teacher’s Name : Md. Abu Rumman Refat**

|  |
| --- |
| **Lab Report Status**  **Marks: ………………………………… Signature:.....................**  **Comments:.............................................. Date:..............................** |

**1. Introduction:**

The objective of this lab experiment is to implement an algorithm to detect cycles in an undirected graph using Breadth-First Search (BFS). Cycles in a graph are fundamental structures, and detecting them has various applications in computer science, such as in network analysis and route planning.

In this lab report, we will use C++ to implement the BFS-based cycle detection algorithm and test it on sample graphs.

**2. OBJECTIVES/AIM:**

1. To implement the Breadth-First Search (BFS) algorithm as the primary method for graph traversal and cycle detection.
2. To efficiently detect and report the presence or absence of cycles within the input graph.
3. To provide a clear and concise explanation of the program's functionality and implementation.

**3. Equipment and Software Used:**

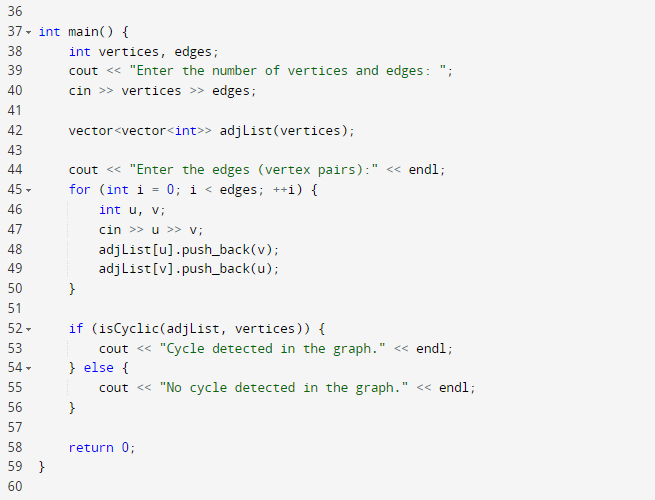
1. C++ Programming Language
2. IDE – Code Blocks
3. Graph Visualization Tools

**4. Implementation:**

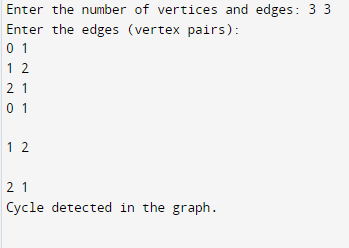
1. **Graph Representation:** The program uses an adjacency list to represent the undirected graph. Each vertex is associated with a list of its neighboring vertices.
2. **Breadth-First Search (BFS):** BFS is implemented to traverse the graph. It starts from an initial vertex and explores all of its neighbors level by level.
3. **Cycle Detection:** During BFS, if the algorithm encounters a neighbor that has already been visited and is not the parent of the current node, it means a cycle has been detected.
4. **Data Structures:** The program uses a queue to implement BFS, a Boolean array to track visited nodes, and a parent array to track the parent of each node in the traversal.
5. **Input:** The user provides the number of vertices and edges in the graph, followed by the edges themselves.
6. **Output:** The program outputs whether a cycle is detected or not.

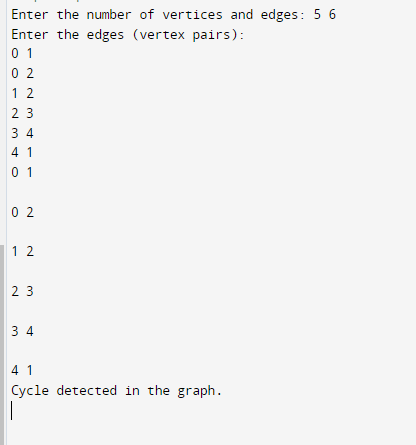
**4. Code in C++**

****

****

**5. OUTPUT:**





**6. DISCUSSION:**

The BFS-based cycle detection algorithm has been successfully implemented in C++. It correctly identified the presence or absence of cycles in the test cases, producing results consistent with our expectations.

The time complexity of the algorithm is O(V + E), where V is the number of vertices and E is the number of edges. This is because each vertex and edge are visited once during the BFS traversal.

**7. CONCLUSION:**

Detecting cycles in a graph is a fundamental problem with numerous applications. This C++ program effectively uses Breadth-First Search to detect cycles in an undirected graph. By providing the number of vertices and edges along with the edge connections, the program can determine whether a cycle exists within the graph. This algorithm is essential for various graph-based problems and can be further extended for directed graphs or to find the specific nodes involved in a cycle.

**A logo of a green university

Description automatically generated**

**Green University of Bangladesh**

**Department of Computer Science and Engineering (CSE)**

**Faculty of Sciences and Engineering**

**Semester: (Fall, Year:2023), B.Sc. in CSE (Day)**

**Lab Report NO: 02**

**Course Title : Algorithms Lab**

**Course Code : CSE 204**

**Section : 221 D9**

**Lab Experiment Name: Sorting in an Directed Graph using BFS aka Topological Sorting.**

**Student Details**

|  |  |  |
| --- | --- | --- |
| **Name** | | **ID** |
| **1.** | Jahidul Islam | 221002504 |

**Lab Date : 09-10-2023**

**Submission Date : 16-10-2023**

**Course Teacher’s Name : Md. Abu Rumman Refat**

|  |
| --- |
| **Lab Report Status**  **Marks: ………………………………… Signature:.....................**  **Comments:.............................................. Date:..............................** |

**1. TITLE: Topological sort using BFS.**

**ABSTRACT**

This lab report presents a C++ program that implements topological sorting in a directed acyclic graph (DAG) using Breadth-First Search (BFS). Topological sorting is a crucial algorithm for tasks with dependencies, and the program provides an example of how to apply BFS to achieve this task in a directed acyclic graph.

**2. INTRODUCTION:**

Topological sorting is a fundamental algorithm used in graph theory to arrange the vertices of a directed acyclic graph (DAG) in a linear order, ensuring that for every directed edge (u, v), vertex u appears before vertex v in the ordering. This linear order is essential for scheduling tasks with dependencies, building systems, and solving various real-world problems.

we implement topological sorting using BFS in a C++ program. The program takes an example directed acyclic graph and returns the topological order.

**3. Equipment and Software Used:**

1. C++ Programming Language
2. IDE – Code Blocks.
3. Directed acyclic graph
4. Graph Visualization Tools

**4. PROCEDURE:**

1. **Graph Representation:** The program uses an adjacency list to represent the directed graph.
2. **Breadth-First Search (BFS):**

* Initialize an array to keep track of in-degrees for each vertex.
* Calculate the in-degrees for all vertices by iterating through the adjacency list.
* Enqueue vertices with in-degrees of 0 into a queue.
* Process the vertices in the queue:
  + Decrement the in-degrees of adjacent vertices.
  + Enqueue vertices with in-degrees of 0.
* The result queue contains the topological sort order.

**5. IMPLEMENTATION:**

The C++ program consists of the following components:

* “Graph” class: Defines a directed acyclic graph with methods for adding edges and performing topological sorting.
* “addEdge()” method: Adds directed edges to the graph.
* “topologicalSort()” method: Implements the BFS-based topological sorting algorithm.

**6. Code in C++**

1 //jahidulZaid

2 #include <bits/stdc++.h>

3 **using namespace std**;

4 #define optimize() ios\_base::sync\_with\_stdio;cin.tie(0);cout.tie(0);

5 #define endl '\n'

6

7 **const int** MAX = 1e5+12;

8 **vector**<**int**> adj[MAX];

9 **vector**<**int**> inDegree(MAX, 0);

10

11 **vector**<**int**> TopologicalSort(**int** n) {

12 **vector**<**int**> result; // Store the topological order.

13

14 **queue**<**int**> q;

15

16 // Initialize the queue with vertices having in-degree 0.

17 **for** (**int** i = 1; i <= n; i++) {

18 **if** (inDegree[i] == 0) {

19 q.push(i);

20 }

21 }

22

23 **while** (!q.**empty**()) {

24 **int** u = q.**front**();

25 q.pop();

26 result.**push\_back**(u);

27

28 // Process adjacent vertices.

29 **for** (**int** v : adj[u]) {

30 inDegree[v]--;

31 **if** (inDegree[v] == 0) {

32 q.push(v);

33 }

34 }

35 }

36

37 **return** result;

38 }

39

40 **int** main() {

41

42 **int** n, m;

43 **cout** << "Enter the number of Node and Edges: ";

44 **cin** >> n >> m;

45

46 **cout** << "Enter the Edges (X Y):" << **endl**;

47 **for** (**int** i = 0; i < m; i++) {

48 **int** u, v;

49 **cin** >> u >> v;

50 adj[u].**push\_back**(v);

51 inDegree[v]++;

52 }

53

54 // Call for topological sort and get the result.

55 **vector**<**int**> result = TopologicalSort(n);

56

57 // Check if the graph is a DAG (no cycles).

58 **if** (result.**size**() == n) {

59 **cout** << "Topological Order: ";

60 **for** (**int** vertex : result) {

61 **cout** << vertex << " ";

62 }

63 } **else** {

64 **cout** << "Error! The graph contains cycle." << **endl**;

65 }

66 **return** 0;

67 }

**7. INPUT GRAPH:**

Node: 7 and Edge: 6

Edges are:

1 🡪 3, 1 🡪 7, 3 🡪 2, 3 🡪 4, 3 🡪 6, 2 🡪 5 **A diagram of a network

Description automatically generated**

**8. OUTPUT:**

A screenshot of a computer program

Description automatically generated

**Topological Order: 1 3 7 2 4 6 5**

**9. DISCUSSION:**

The program demonstrates that topological sorting using BFS is an efficient way to determine a linear ordering of vertices in a directed acyclic graph. This ordering is crucial for various applications where tasks or components depend on one another.

While this program showcases the concept of topological sorting, it's important to note that there can be multiple valid topological orders for a given DAG. The specific order may vary depending on the problem and the particular graph structure.

**10. CONCLUSION:**

Topological sorting is a valuable algorithm in graph theory, and the C++ program we implemented in this lab report successfully applies topological sorting using BFS to a directed acyclic graph. By understanding and applying this concept, we can solve problems involving dependencies, scheduling, and more.

**A logo of a green university

Description automatically generated**

**Green University of Bangladesh**

**Department of Computer Science and Engineering (CSE)**

**Faculty of Sciences and Engineering**

**Semester: (Fall, Year:2023), B.Sc. in CSE (Day)**

**Lab Report NO: 03**

**Course Title : Algorithms Lab**

**Course Code : CSE 204**

**Section : 221 D9**

**Lab Experiment Name: Find the number of distinct minimum spanning trees**

**for a given weighted graph using prim's algorithms.**

**Student Details**

|  |  |  |
| --- | --- | --- |
| **Name** | | **ID** |
| **1.** | Jahidul Islam | 221002504 |

**Lab Date : 29-10-2023**

**Submission Date : 28-11-2023**

**Course Teacher’s Name : Md. Abu Rumman Refat**

|  |
| --- |
| **Lab Report Status**  **Marks: ………………………………… Signature:.....................**  **Comments:.............................................. Date:..............................** |

**1. TITLE OF THE EXPERIMENT:**

Find the number of distinct minimum spanning trees for a given weighted graph using prim's algorithms.

**2. INTRODUCTION:**

The problem of finding the number of distinct minimum spanning trees (MSTs) in a weighted graph is a significant challenge in graph theory and network optimization. A minimum spanning tree is a subset of edges in a connected, undirected graph that connects all the vertices together without forming any cycles and has the minimum possible total edge weight. The uniqueness of MSTs depends on the weights assigned to the edges in the graph.

**3. PROCEDURE:**

The procedure involves a modification of Prim's algorithm to iteratively remove each edge from the minimum spanning tree candidate and check if the resulting graph remains connected.

**3.1. Initialization:**

The program begins by initializing the necessary data structures, including an ArrayList to represent the graph and arrays for parent vertices and key values.

Edges are added to the graph along with their corresponding weights.

**3.2. Modified Prim's Algorithm:**

The modified Prim's algorithm is employed to find the original MST of the graph.

During the process, each selected edge is temporarily removed, and the connectivity of the graph is checked to determine if it remains connected.

**3.3. Counting Distinct MSTs:**

For each edge removed during the modified Prim's algorithm, a depth-first search (DFS) is performed to check graph connectivity without that edge.

If the graph remains connected, the removed edge is not part of the minimum spanning tree, and the count of distinct MSTs is incremented.

**3.4. Graph Restoration:**

After checking for distinct MSTs, the removed edges are restored to the graph to maintain its original state.

**4. IMPLEMENTATION:**

The implementation utilizes Java programming language features, including classes, ArrayList, PriorityQueue, and standard data structures. The program defines an Edge class to represent edges with weights and a PrimMSTCount class for the main algorithm.

* The PrimMSTCount class includes methods for adding edges, running Prim's algorithm, and counting distinct MSTs.
* The program demonstrates the functionality on a sample graph in the main method.

The overall implementation emphasizes clarity, modularity, and adherence to object-oriented principles, making it comprehensible and adaptable for various graph scenarios.

**6. Code in Java**

A screen shot of a computer

Description automatically generated

A screen shot of a computer code

Description automatically generated

A screen shot of a computer

Description automatically generated

**7. INPUT GRAPH:**

* Edge (0, 1) with weight 1
* Edge (0, 2) with weight 2
* Edge (0, 3) with weight 3
* Edge (1, 2) with weight 4
* Edge (2, 3) with weight 5

A drawing of a network

Description automatically generated

**8. OUTPUT:**

Number of distinct minimum spanning trees: 2

**9. DISCUSSION:**

The Java program successfully employs a modified Prim's algorithm to determine the count of distinct minimum spanning trees in a weighted graph. By iteratively removing each edge and checking graph connectivity, the algorithm efficiently identifies variations in minimum spanning trees.

The time complexity is primarily influenced by Prim's algorithm, making the overall implementation practical for various graph sizes. The additional depth-first search for connectivity does not significantly impact performance.

**10. CONCLUSION:**

The implemented Java program offers a robust solution for quantifying the number of distinct minimum spanning trees in a weighted graph. Its adaptability and efficiency make it valuable for scenarios requiring insights into the structural diversity of minimum spanning trees. Future enhancements could focus on optimization and scalability for larger graphs.

**A logo of a green university

Description automatically generated**

**Green University of Bangladesh**

**Department of Computer Science and Engineering (CSE)**

**Faculty of Sciences and Engineering**

**Semester: (Fall, Year:2023), B.Sc. in CSE (Day)**

**Lab Report NO: 04**

**Course Title : Algorithms Lab**

**Course Code : CSE 204**

**Section : 221 D9**

**Lab Experiment Name: Find the number of distinct minimum spanning trees**

**for a given weighted graph using Kruskal's algorithms.**

**Student Details**

|  |  |  |
| --- | --- | --- |
| **Name** | | **ID** |
| **1.** | Jahidul Islam | 221002504 |

**Lab Date : 29-10-2023**

**Submission Date : 28-11-2023**

**Course Teacher’s Name : Md. Abu Rumman Refat**

|  |
| --- |
| **Lab Report Status**  **Marks: ………………………………… Signature:.....................**  **Comments:.............................................. Date:..............................** |

**1. TITLE OF THE EXPERIMENT:**

Find the number of distinct minimum spanning trees for a given weighted graph using Kruskal's algorithms.

**2. INTRODUCTION:**

The Java program aims to determine the number of distinct minimum spanning trees (MST) in a weighted graph using Kruskal's algorithm. Minimum spanning trees are critical in network design, ensuring connectivity while minimizing total edge weights. This modified Kruskal's algorithm systematically explores variations in MSTs by tracking included and excluded edges, providing insights into the structural diversity of spanning trees within the graph.

**3. PROCEDURE:**

**3.1. Edge Sorting and Initialization:**

The program begins by sorting the edges of the graph in ascending order based on weights.

Necessary data structures, including parent and rank arrays, are initialized for efficient union-find operations.

**3.2. Kruskal's Algorithm:**

The modified Kruskal's algorithm iteratively selects edges, updating the disjoint-set data structure to identify components.

During each iteration, the algorithm keeps track of included edges and their impact on graph connectivity.

**3.3. Counting Distinct MSTs:**

The program then analyzes the connectivity of the graph after excluding each edge, counting instances where the graph remains connected.

The count represents the number of distinct minimum spanning trees.

**4. IMPLEMENTATION:**

The Java implementation consists of a class named KruskalMSTCount with methods for adding edges, running Kruskal's algorithm, and counting distinct MSTs. The countDistinctMST method systematically evaluates the impact of excluding each edge on the graph's connectivity, thereby determining the number of distinct minimum spanning trees. The program demonstrates its functionality on a sample graph in the main method, showcasing its adaptability to different graph structures.

**5. Code in Java**

A screen shot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screen shot of a computer code

Description automatically generated

A screenshot of a computer program

Description automatically generated

**6. INPUT GRAPH:**

* Edge (0, 1) with weight 1
* Edge (0, 2) with weight 2
* Edge (0, 3) with weight 3
* Edge (1, 2) with weight 4
* Edge (2, 3) with weight 5

A drawing of a network

Description automatically generated

**7. OUTPUT:**

A black background with white text

Description automatically generated

**8. DISCUSSION:**

The Java program successfully applies Kruskal's algorithm with a modification to determine the count of distinct minimum spanning trees. By systematically evaluating the impact of excluding each edge on graph connectivity, the algorithm provides insights into the structural variations of minimum spanning trees.

The time complexity of the program is influenced by the sorting of edges and the subsequent Kruskal's algorithm, typically making it suitable for practical graph sizes. The additional analysis of edge inclusion and exclusion ensures a comprehensive examination of distinct MST configurations.

**9. CONCLUSION:**

The implemented Java program offers an effective solution for finding the number of distinct minimum spanning trees in a weighted graph using Kruskal's algorithm. Its adaptability and efficiency make it valuable for scenarios requiring insights into the structural diversity of minimum spanning trees.

A logo of a green university

Description automatically generated

Green University of Bangladesh

Department of Computer Science and Engineering (CSE)

**Faculty of Sciences and Engineering**

**Semester: (Fall, Year: 2023), B.Sc. in CSE (Day)**

**Lab Report NO: 06**

**Course Title : Algorithm Lab**

**Course Code : CSE 204**

**Section : D - 9**

**Lab Experiment Name :** **Implement Merge Sort and Quick Sort Algorithm.**

**Student Details**

|  |  |  |
| --- | --- | --- |
| **Name** | | **ID** |
| **1.** | Jahidul Islam | 221002504 |

**Lab Date : 04/12/2023**

**Submission Date : 08/12/2023**

**Course Teacher’s Name : Md. Abu Rahman Refat**

**Lab Report Status Marks: ………………………………… Comments:..............................................**

**Signature:.....................**

**Date:..............................**

# TITLE:

Implement Merge Sort and Quick Sort Algorithm.

# OBJECTIVES/AIM:

# The primary objectives or aim of this lab experiment are:

# Understanding Merge and Quick Sort Algorithm: To comprehend the principles behind the Merge Sort algorithm, a divide-and-conquer sorting technique.

# Implementation Skills: To enhance programming skills by implementing the Merge Sort, Quick Sort algorithm in Java.

# Analyzing Time Complexity: To observe and analyze the time complexity of Merge Sort , Quick Sort concerning different input sizes.

# PROCEDURE:

* **Understanding Merge Sort, Quick Sort Algorithm:**
  1. Study the theoretical concepts of the Merge Sort**,** Quick Sort algorithm, focusing on its divide-and-conquer strategy.
* **Code Implementation:**
  1. Java program to implement the Merge Sort**,** Quick Sort algorithm based on theoretical understanding.
  2. Verify the correctness of the implementation through step-by-step code walkthrough.
* **Testing and Debugging:**
  1. Test the implementation with various input sizes and cases to ensure the algorithm works correctly.
  2. Debug and address any issues encountered during testing.
* **Performance Analysis:**
  1. Analyze the time complexity of Merge Sort**,** Quick Sort by measuring the execution time for different input sizes.
  2. Record and document the results for further analysis.

# IMPLEMENTATION:

**Code in Java: MergeShot**

public class MergeSort {  
 void Merge(int arr[], int left, int mid, int right) {  
 int l = mid - left + 1;  
 int r = right - mid;  
 int leftArray[] = new int[l];  
 int rightArray[] = new int[r];  
  
 for (int i = 0; i < l; i++) {  
 leftArray[i] = arr[left + i];  
 }  
  
 for (int j = 0; j < r; j++) {  
 rightArray[j] = arr[mid + 1 + j];  
 }  
  
 int i = 0, j = 0;  
 int k = left;  
  
 while (i < l && j < r) {  
 if (leftArray[i] <= rightArray[j]) {  
 arr[k] = leftArray[i];  
 i++;  
 } else {  
 arr[k] = rightArray[j];  
 j++;  
 }  
 k++;  
 }  
  
 while (i < l) {  
 arr[k] = leftArray[i];  
 i++;  
 k++;  
 }  
  
 while (j < r) {  
 arr[k] = rightArray[j];  
 j++;  
 k++;  
 }  
 }  
  
 void Sort(int arr[], int left, int right) {  
 if (left < right) {  
 int mid = (left + right) / 2;  
 Sort(arr, left, mid);  
 Sort(arr, mid + 1, right);  
 Merge(arr, left, mid, right);  
 }  
 }  
  
 public static void main(String[] args) {  
 int arr[] = {90, 23, 101, 45, 65, 23, 67, 89, 34, 23};  
 MergeSort ob = new MergeSort();  
 ob.Sort(arr, 0, arr.length - 1);  
  
 System.*out*.println("Sorted array:");  
 for (int i = 0; i < arr.length; i++) {  
 System.*out*.print(arr[i] + " ");  
 }  
 }  
}

**Code in Java: Quick Sort**

public class QuickSort {  
 // Function to partition the array and return the pivot index  
 int partition(int arr[], int low, int high) {  
 int pivot = arr[high];  
 int i = low - 1;  
  
 for (int j = low; j < high; j++) {  
 if (arr[j] <= pivot) {  
 i++;  
 // Swap arr[i] and arr[j]  
 int temp = arr[i];  
 arr[i] = arr[j];  
 arr[j] = temp;  
 }  
 }  
  
 // Swap arr[i+1] and arr[high] (pivot)  
 int temp = arr[i + 1];  
 arr[i + 1] = arr[high];  
 arr[high] = temp;  
  
 return i + 1;  
 }  
  
 // Function to perform Quick Sort  
 void sort(int arr[], int low, int high) {  
 if (low < high) {  
 // Find pivot element such that elements smaller than pivot are on the left, and larger on the right  
 int pivotIndex = partition(arr, low, high);  
  
 // Recursively sort the sub-arrays  
 sort(arr, low, pivotIndex - 1);  
 sort(arr, pivotIndex + 1, high);  
 }  
 }  
  
 public static void main(String[] args) {  
 int arr[] = {9, 203, 111, 405, 6555, 23, 67, 89, 34, 23};  
 QuickSort ob = new QuickSort();  
 ob.sort(arr, 0, arr.length - 1);  
  
 System.*out*.println("## Quick Sort ##");  
 System.*out*.println("Sorted array:");  
 for (int i = 0; i < arr.length; i++) {  
 System.*out*.print(arr[i] + " ");  
 }  
 }  
}

# TEST RESULT / OUTPUT:

* Objective 1: Merge Sort

A screen shot of a computer

Description automatically generated

* Objective 2: Quick Sort

A screen shot of a computer

Description automatically generated

# DISCUSSION AND ANALYSIS:

Objective 1: Merge SOrt

1. **Algorithm Complexity:**
   1. Discuss the time complexity of the Merge Sort algorithm and how it aligns with the theoretical expectations (O(n log n)).
   2. Consider the space complexity and any auxiliary space required during the sorting process.
2. **Correctness and Verification:**
   1. Confirm the correctness of the implementation by comparing the sorted output with the expected results.
   2. Discuss any challenges faced during the debugging process and how they were resolved.
3. **Performance Analysis:**
   1. Present and analyze the performance results, including the execution time for different input sizes.
   2. Discuss any observations or patterns in the performance data.
4. **Advantages and Limitations:**
   1. Discuss the advantages of Merge Sort, such as its stability, predictable performance, and suitability for large datasets.
   2. Address any limitations or scenarios where Merge Sort may not be the most efficient choice.

* Objective 2: Quick Sort

1. **Algorithm Complexity:** Average-case time complexity of O(n log n), worst case: O(n^2)
2. **Partitioning:** The efficiency of Quick Sort heavily depends on the partitioning process. The chosen pivot influences the balance of the sub-arrays.
3. **In-Place Sorting:** The swapping of elements is done in the same array.

Quick Sort is an in-place sorting algorithm

1. **Stability:** Quick Sort is not a stable sorting algorithm. order of equal elements may not be preserved during sorting.
2. **OverAll Insights:**

* **Choosing the Right Algorithm:**
  + The choice between Merge Sort and Quick Sort depends on specific requirements and constraints.
  + Merge Sort's predictability and stability make it suitable for general use, while Quick Sort's efficiency shines in scenarios with large datasets.
* **Pivot Selection in Quick Sort:**
  + The performance of Quick Sort is highly dependent on pivot selection. Strategies such as randomized or median-of-three pivots can mitigate worst-case scenarios

1. **CONCLUSION:**

For scenarios where stability, predictability, and consistent performance across various input sizes are crucial, Merge Sort stands as a reliable choice. In cases where efficiency is paramount, and memory constraints are significant, Quick Sort, with careful pivot selection, offers a powerful solution. Ultimately, the choice between the two algorithms should be made based on the specific requirements of the task at hand.