1. Merge Sort

Sort a given set of elements using **Merge Sort** and determine the time required to sort the elements. Repeat the experiment for different values of `n`, the number of elements in the list to be sorted, and **plot a graph** of the **time taken versus the number of elements**. The elements can be read from a file or generated using a random number generator.

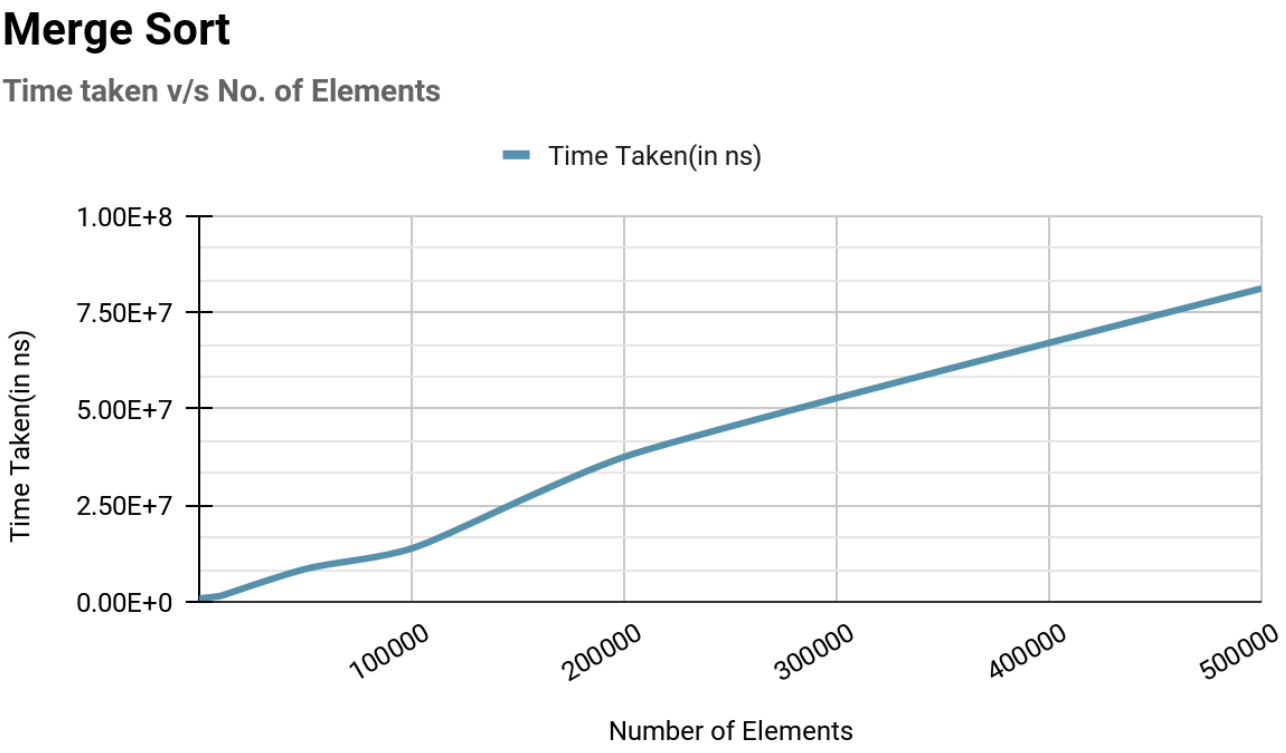
```
import java.util.Random;

public class MergeSort{
    // Merge Sort implementation
    public static void mergeSort(int[] array, int left, int right) {
        if (left < right) {
            int middle = (left + right) / 2;
            // Sort the first and second halves
            mergeSort(array, left, middle);
            mergeSort(array, middle + 1, right);
            // Merge the sorted halves
            merge(array, left, middle, right);
        }
    }
    public static void merge(int[] array, int left, int middle, int right) {
        int n1 = middle - left + 1;
        int n2 = right - middle;
        // Temporary arrays
        int[] L = new int[n1];
        int[] R = new int[n2];
        // Copy data to temp arrays
        System.arraycopy(array, left, L, 0, n1);
        System.arraycopy(array, middle + 1, R, 0, n2);
        // Merge the temp arrays
        int i = 0, j = 0;
        int k = left;
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                array[k] = L[i];
                i++;
            } else {

                array[k] = R[j];
                j++;
```

```
            }
            k++;
        }
        // Copy remaining elements of L[] if any
        while (i < n1) {
            array[k] = L[i];
            i++;
            k++;
        }
        // Copy remaining elements of R[] if any
        while (j < n2) {
            array[k] = R[j];
            j++;
            k++;
        }
    }
    // Generate an array with random integers
    public static int[] generateRandomArray(int size) {
        Random random = new Random();
        int[] array = new int[size];
        for (int i = 0; i < size; i++) {
            array[i] = random.nextInt(10000);
        }
        return array;
    }
    public static void main(String[] args) {
        // Define the sizes of arrays to test
        int[] sizes = {100, 1000, 5000, 10000, 50000, 100000, 200000, 500000};
        for (int size : sizes) {
            int[] array = generateRandomArray(size);
            // Record start time in nanoseconds
            long startTime = System.nanoTime();
            // Sort the array using merge sort
            mergeSort(array, 0, array.length - 1);
            // Record end time in nanoseconds
            long endTime = System.nanoTime();
            // Calculate and display the time taken in nanoseconds
            long timeTaken = endTime - startTime;
            System.out.println("Size: " + size + ", Time taken: "
                                                + timeTaken + " ns");
        }
    }
}
```

Output:

```
sunil@sunil:~daaLabManual/program01$ java MergeSort.java
Size: 100, Time taken: 90745 ns
Size: 1000, Time taken: 853677 ns
Size: 5000, Time taken: 1155572 ns
Size: 10000, Time taken: 1493659 ns
Size: 50000, Time taken: 8477575 ns
Size: 100000, Time taken: 13802170 ns
Size: 200000, Time taken: 37517808 ns
Size: 500000, Time taken: 81148003 ns
```

Graph:



Time complexity for **Merge Sort** is **O(n*logn).**

2. Quick Sort

Sort a given set of elements using **Quick sort** and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and **plot a graph** of the **time taken versus number of elements**. The elements can be read from a file or generated using a random number generator.

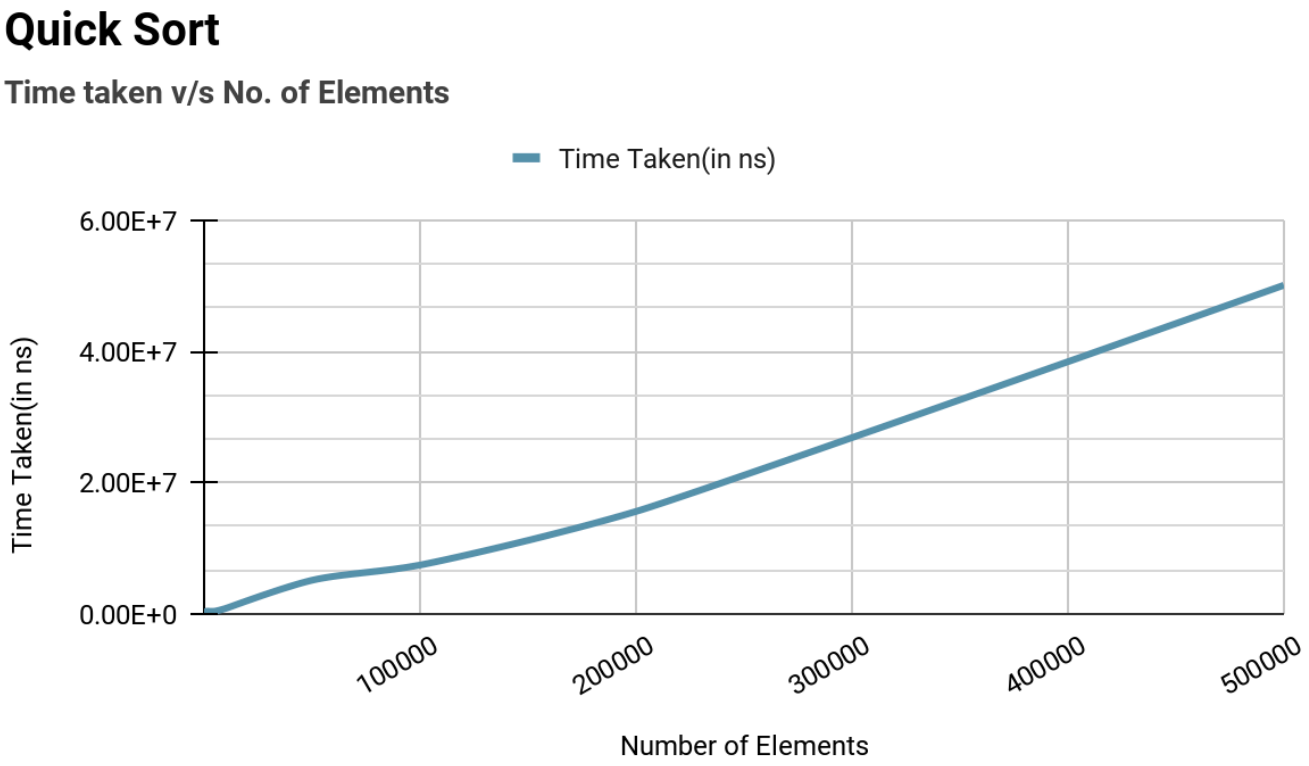
```
import java.util.Random;

public class QuickSort {
    // Quick Sort implementation
    public static void quickSort(int[] array, int low, int high) {
        if (low < high) {
            int pi = partition(array, low, high);
            // Recursively sort elements before and after partition
            quickSort(array, low, pi - 1);
            quickSort(array, pi + 1, high);
        }
    }

    public static int partition(int[] array, int low, int high) {
        int pivot = array[high];
        int i = (low - 1); // Index of smaller element
        for (int j = low; j < high; j++) {
            // If current element is smaller than or equal to pivot
            if (array[j] <= pivot) {
                i++;
                // Swap array[i] and array[j]
                int temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }
        // Swap array[i + 1] and array[high] (or pivot)
        int temp = array[i + 1];
        array[i + 1] = array[high];
        array[high] = temp;

        return i + 1;
    }
```

```
    // Generate an array with random integers
    public static int[] generateRandomArray(int size) {
        Random random = new Random();
        int[] array = new int[size];
        for (int i = 0; i < size; i++) {
            array[i] = random.nextInt(10000)
        }
        return array;
    }

    public static void main(String[] args) {
        // Define the sizes of arrays to test
        int[] sizes = {100, 1000, 5000, 10000, 50000, 100000, 200000, 500000};
        for (int size : sizes) {
            int[] array = generateRandomArray(size);
            // Record start time in nanoseconds
            long startTime = System.nanoTime();
            // Sort the array using quick sort
            quickSort(array, 0, array.length - 1);
            // Record end time in nanoseconds
            long endTime = System.nanoTime();
            // Calculate and display the time taken in nanoseconds
            long timeTaken = endTime - startTime;
            System.out.println("Size: " + size + ", Time taken: "
                                                + timeTaken + " ns");
        }
    }
}
```

Output:

```
sunil@sunil:~daaLabManual/program02$ java QuickSort.java
Size: 100, Time taken: 31928 ns
Size: 1000, Time taken: 390821 ns
Size: 5000, Time taken: 418341 ns
Size: 10000, Time taken: 836881 ns
Size: 50000, Time taken: 5127389 ns
Size: 100000, Time taken: 7445483 ns
Size: 200000, Time taken: 15604122 ns
Size: 500000, Time taken: 50061383 ns
```

Graph:



Time complexity for **Quick Sort** is **O(n*logn).**

3. Binary Search Tree

Write a program to perform **insert** and **delete** operations in **Binary Search Tree**.

```
class BinarySearchTree {
    // Node class representing a node in the BST
    class Node {
        int key;
        Node left, right;
        public Node(int item) {
            key = item;
            left = right = null;
        }
    }
    // Root of the BST
    Node root;
    // Constructor
    BinarySearchTree() {
        root = null;
    }
    // Insert a new key in the BST
    void insert(int key) {
        root = insertRec(root, key);
    }
    // A recursive function to insert a new key in the BST
    Node insertRec(Node root, int key) {
        // If the tree is empty, return a new node
        if (root == null) {
            root = new Node(key);
            return root;
        }
        // Otherwise, recur down the tree
        if (key < root.key) {
            root.left = insertRec(root.left, key);
        } else if (key > root.key) {
            root.right = insertRec(root.right, key);
        }
        // Return the unchanged node pointer
        return root;
    }

    // Delete a key from the BST
```

```
    void deleteKey(int key) {
        root = deleteRec(root, key);
    }
    // A recursive function to delete a key in the BST
    Node deleteRec(Node root, int key) {
        // Base case: If the tree is empty
        if (root == null) {
            return root;
        }
        // Otherwise, recur down the tree
        if (key < root.key) {
            root.left = deleteRec(root.left, key);
        } else if (key > root.key) {
            root.right = deleteRec(root.right, key);
        } else {
            // Node with only one child or no child
            if (root.left == null) {
                return root.right;
            } else if (root.right == null) {
                return root.left;
            }
            // Node with two children:
           // Get the inorder successor (smallest in the right subtree)
            root.key = minValue(root.right);
            // Delete the inorder successor
            root.right = deleteRec(root.right, root.key);
        }
        return root;
    }
    // A utility function to find the minimum value node in a given BST
    int minValue(Node root) {
        int minValue = root.key;
        while (root.left != null) {
            minValue = root.left.key;
            root = root.left;
        }
        return minValue;
    }
    // This method calls the in-order traversal of the BST
    void inorder() {
        inorderRec(root);
    }
    // Function to do in-order traversal of the BST
```

```
    void inorderRec(Node root) {
        if (root != null) {
            inorderRec(root.left);
            System.out.print(root.key + " ");
            inorderRec(root.right);
        }
    }
    public static void main(String[] args) {
        BinarySearchTree bst = new BinarySearchTree();
        int[] nodes = {50, 30, 20, 40, 70, 60, 80};
        // Insert nodes
        for(int node : nodes){
            bst.insert(node);
        }
        // Print the in-order traversal of the BST
        System.out.println("In-order traversal of the given tree:");
        bst.inorder();
        System.out.println();
        // Delete nodes
        int[] deleteNodes = {20, 30, 50};
        for(int node : deleteNodes){
            bst.deleteKey(node);
            System.out.println("In-order traversal after deleting "
                                                    + node + ": ");
            bst.inorder();
            System.out.println();
        }
    }
}
```

Output:

```
sunil@sunil:~daaLabManual/program3$ java BinarySearchTree.java
In-order traversal of the given tree:
20 30 40 50 60 70 80
In-order traversal after deleting 20:
30 40 50 60 70 80
In-order traversal after deleting 30:
40 50 60 70 80
In-order traversal after deleting 50:
40 60 70 80
```

4. Breadth-First Search

Print all the nodes reachable from a given starting node in a **Di-graph** using the **Breadth-First Search** method.

```
import java.util.*;

public class BFSDigraph {
    // Graph class to represent a directed graph using adjacency list
    static class Graph {
        private int vertices; // Number of vertices
        private LinkedList<Integer>[] adjList; // Adjacency list
        // Constructor
        Graph(int vertices) {
            this.vertices = vertices;
            adjList = new LinkedList[vertices];
            for (int i = 0; i < vertices; i++) {
                adjList[i] = new LinkedList<>();
            }
        }
        // Function to add an edge to the graph
        void addEdge(int v, int w) {
            adjList[v].add(w); // Add w to v's list
        }
        // Function to perform BFS from a given starting node
        void BFS(int start) {
            // Mark all the vertices as not visited
            boolean[] visited = new boolean[vertices];
            // Create a queue for BFS
            LinkedList<Integer> queue = new LinkedList<>();
            // Mark the current node as visited and enqueue it
            visited[start] = true;
            queue.add(start);
            while (queue.size() != 0) {
                // Dequeue a vertex from the queue and print it
                start = queue.poll();
                System.out.print(start + " ");
                // Get all adjacent vertices of the dequeued vertex
                // If an adjacent vertex has not been visited, mark it as
                //visited and enqueue it
```

```
                for (int n : adjList[start]) {
                    if (!visited[n]) {
                        visited[n] = true;
                        queue.add(n);
                    }
                }
            }
        }
    }
    public static void main(String[] args) {
        // Create a graph with 6 vertices
        Graph graph = new Graph(6);
        // Add edges to the digraph
        graph.addEdge(0, 1);
        graph.addEdge(0, 2);
        graph.addEdge(1, 2);
        graph.addEdge(2, 0);
        graph.addEdge(2, 3);
        graph.addEdge(3, 3);
        graph.addEdge(4, 5);
        // Print all nodes reachable from node 2
        System.out.println("Following are the nodes reachable from node 2:");
        graph.BFS(2);
        System.out.println();
    }
}
```

Output:

```
sunil@sunil:~daaLabManual/program4$ java BFSDigraph.java

Following are the nodes reachable from node 2:
2 0 3 1
```

5. Topological Sort

Obtain the **Topological ordering** of vertices in a given digraph.

```
import java.util.*;

public class TopologicalSort {
    private int V; // Number of vertices
    private List<Integer>[] adjList; // Adjacency list
    // Constructor
    @SuppressWarnings("unchecked")
    public TopologicalSort(int v) {
        V = v;
        adjList = new LinkedList[v];
        for (int i = 0; i < v; ++i)
            adjList[i] = new LinkedList<>();
    }
    // Function to add an edge to the graph
    private void addEdge(int v, int w) {
        adjList[v].add(w);
    }
    // A recursive function used by topologicalSort
    private void topologicalSortUtil(int v, boolean[] visited, Stack<Integer>
stack) {
        // Mark the current node as visited
        visited[v] = true;
        // Recur for all the vertices adjacent to this vertex
        for (Integer neighbor : adjList[v]) {
            if (!visited[neighbor])
                topologicalSortUtil(neighbor, visited, stack);
        }
        // Push the current vertex to the stack which stores the result
        stack.push(v);
    }
    // The function to do Topological Sort
    private void topologicalSort() {
        Stack<Integer> stack = new Stack<>();
        // Mark all the vertices as not visited
        boolean[] visited = new boolean[V];
        Arrays.fill(visited, false);
        // Call the recursive helper function to store Topological Sort starting
        // from all vertices one by one
```

```
        for (int i = 0; i < V; i++) {
            if (!visited[i])
                topologicalSortUtil(i, visited, stack);
        }
        // Print the contents of the stack
        System.out.println("Topological Sort:");
        while (!stack.isEmpty())
            System.out.print(stack.pop() + " ");
    }
    // Main method to test the Topological Sort
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the number of vertices: ");
        int V = scanner.nextInt();
        TopologicalSort g = new TopologicalSort(V);
        System.out.println("Enter the adjacency matrix:");
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (scanner.nextInt() == 1) {
                    g.addEdge(i, j);
                }
            }
        }
        g.topologicalSort();
        scanner.close();
        System.out.println();
    }
}
```

Output:

```
sunil@sunil:~daaLabManual/program5$ java TopologicalSort.java
Enter the number of vertices: 5
Enter the adjacency matrix:
0 5 0 0 0
0 0 7 10 0
0 0 0 3 0
0 0 0 0 6
0 2 0 0 0
Topological Sort:
4 3 2 1 0
```

6. Warshall's Algorithm

Compute the transitive closure of a given directed graph using **Warshall's Algorithm**.

```
import java.util.Scanner;

public class WarshallsAlgorithm {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the number of vertices: ");
        int vertices = scanner.nextInt();
        // Initialize the adjacency matrix
        int[][] graph = new int[vertices][vertices];
        // Get the adjacency matrix from the user
        System.out.println("Enter the adjacency matrix (0 for no edge, 1 for
                           edge):");
        for (int i = 0; i < vertices; i++) {
            for (int j = 0; j < vertices; j++) {
                graph[i][j] = scanner.nextInt();
            }
        }
        // Find the transitive closure using Warshall's Algorithm
        for (int k = 0; k < vertices; k++) {
            for (int i = 0; i < vertices; i++) {
                for (int j = 0; j < vertices; j++) {
                    graph[i][j] = graph[i][j] | (graph[i][k] & graph[k][j]);
                }
            }
        }
        // Display the transitive closure matrix
        System.out.println("Transitive Closure:");
        for (int i = 0; i < vertices; i++) {
            for (int j = 0; j < vertices; j++) {
                System.out.print(graph[i][j] + " ");
            }
            System.out.println();
        }
        scanner.close();
    }
}
```

Output:

```
sunil@sunil:~daaLabManual/program6$ java WarshallsAlgorithm.java
Enter the number of vertices: 5
Enter the adjacency matrix (0 for no edge, 1 for edge):
0 1 0 0 0
0 0 1 1 0
0 0 0 1 0
0 0 0 0 1
0 1 0 0 0
Transitive Closure:
0 1 1 1 1
0 1 1 1 1
0 1 1 1 1
0 1 1 1 1
0 1 1 1 1
```