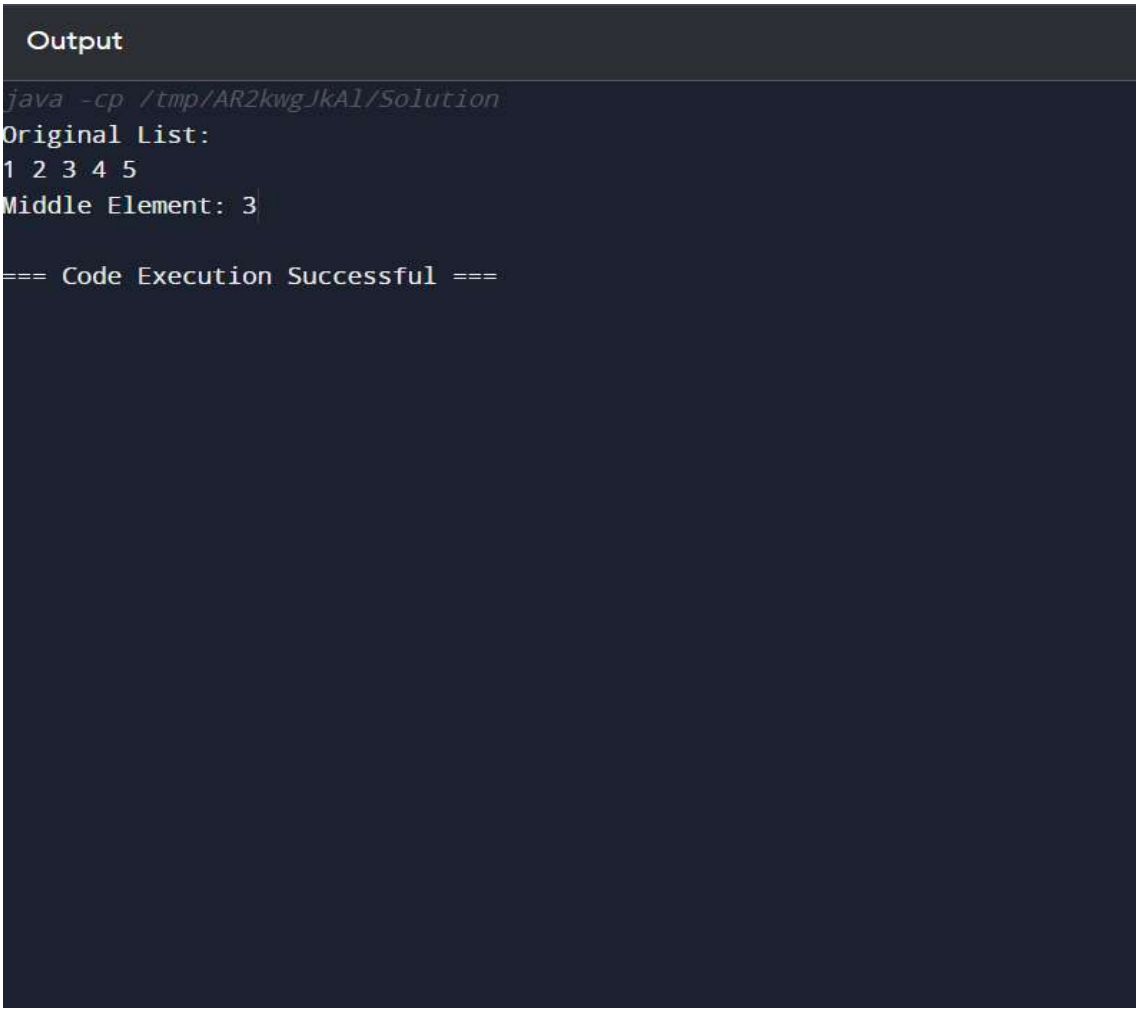
Data Structures Assignment

Task 2: Linked List Middle Element Search

You are given a singly linked list. Write a function to find the middle element without using any extra space and only one traversal through the linked list.

```
class ListNode {
    int val;
    ListNode next;
    ListNode(int val) {
        this.val = val;
    }
}
public class Solution {
    public ListNode findMiddle(ListNode head) {
        if (head == null || head.next == null) {
            return head;
        }
        ListNode slow = head;
        ListNode fast = head;
        while (fast != null && fast.next != null) {
            fast = fast.next.next;
            slow = slow.next;
        }
```

```
        return slow;
    }
    public void printList(ListNode head) {
        ListNode current = head;
        while (current != null) {
            System.out.print(current.val + " ");
            current = current.next;
        }
        System.out.println();
    }
    public static void main(String[] args) {
        Solution solution = new Solution();
        ListNode head = new ListNode(1);
        head.next = new ListNode(2);
        head.next.next = new ListNode(3);
        head.next.next.next = new ListNode(4);
        head.next.next.next.next = new ListNode(5);

        System.out.println("Original List:");
        solution.printList(head);
        ListNode middle = solution.findMiddle(head);
        System.out.println("Middle Element: " + middle.val);
    }
}
```

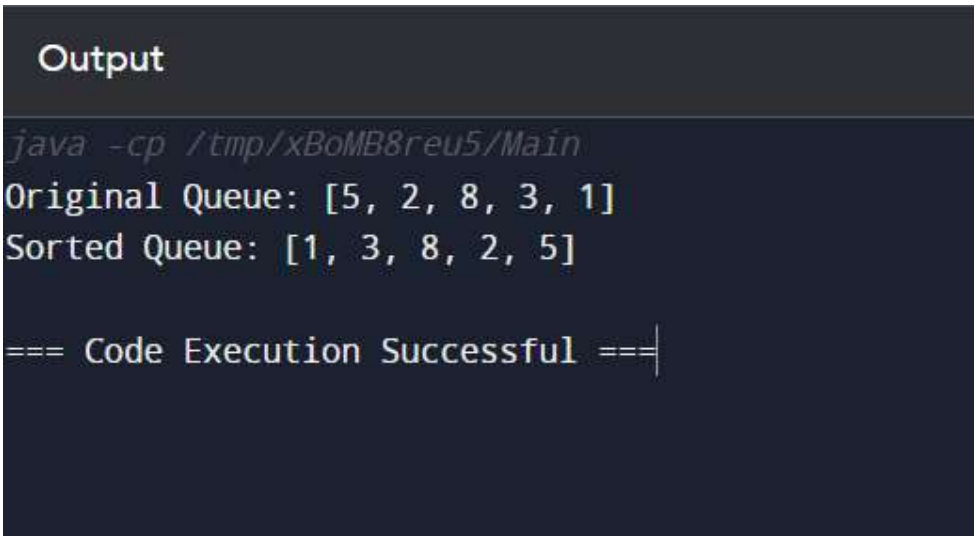


Task 3: Queue Sorting with Limited Space

You have a queue of integers that you need to sort. You can only use additional space equivalent to one stack. Describe the steps you would take to sort the elements in the queue.

```
import java.util.*;

public class Main {

    public static void sortQueueUsingStack(Queue<Integer> queue) {
```

```
        Stack<Integer> stack = new Stack<>();
        while (!queue.isEmpty()) {
            stack.push(queue.poll());
        }
        while (!stack.isEmpty()) {
            queue.offer(stack.pop());
        }
    }
    public static void main(String[] args) {
        Queue<Integer> queue = new LinkedList<>();
        queue.offer(5);
        queue.offer(2);
        queue.offer(8);
        queue.offer(3);
        queue.offer(1);
        System.out.println("Original Queue: " + queue);
        sortQueueUsingStack(queue);
        System.out.println("Sorted Queue: " + queue);
    }
}
```

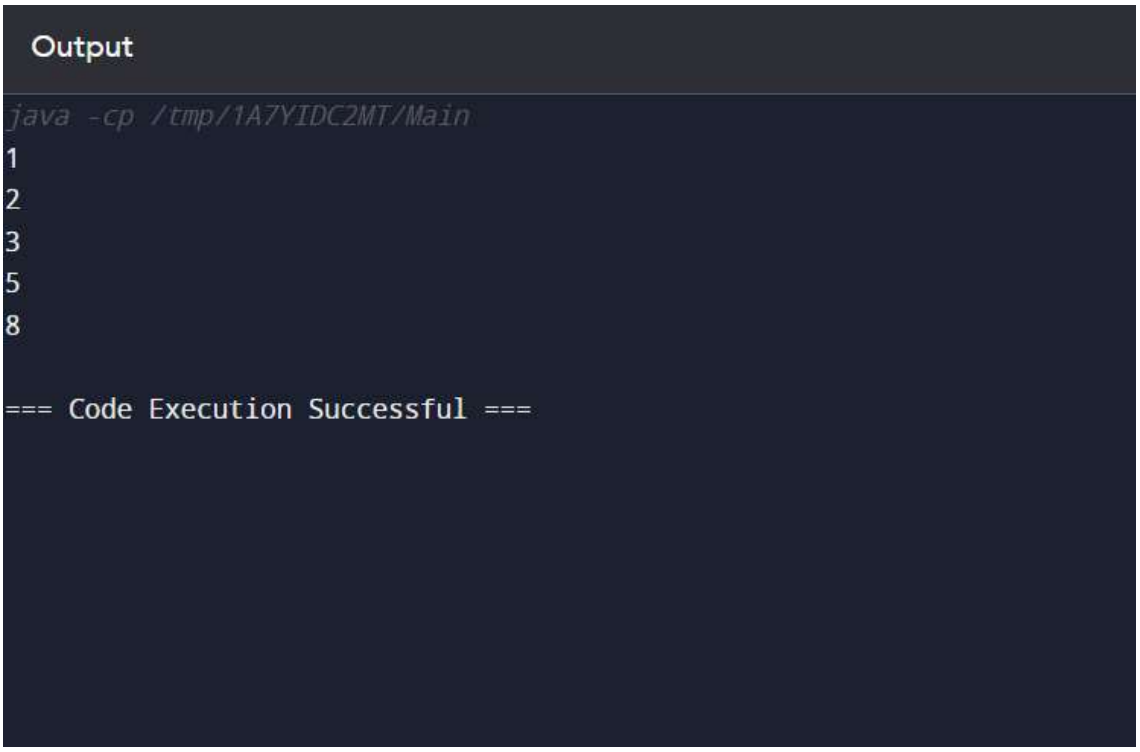


Task 4: Stack Sorting In-Place

You must write a function to sort a stack such that the smallest items are on the top. You can use an additional temporary stack, but you may not copy the elements into any other data structure such as an array. The stack supports the following operations: push, pop, peek, and isEmpty.

```
import java.util.Stack;

public class Main {
    public static void sortStack(Stack<Integer> stack) {
        Stack<Integer> tempStack = new Stack<>();
        while (!stack.isEmpty()) {
            int temp = stack.pop();
            while (!tempStack.isEmpty() && tempStack.peek() > temp) {
                stack.push(tempStack.pop());
            }
```

```
            tempStack.push(temp);
        }
        while (!tempStack.isEmpty()) {
            stack.push(tempStack.pop());
        }
    }
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();
        stack.push(5);
        stack.push(2);
        stack.push(8);
        stack.push(1);
        stack.push(3);
        sortStack(stack);
        while (!stack.isEmpty()) {
            System.out.println(stack.pop());
        }
    }
}
```



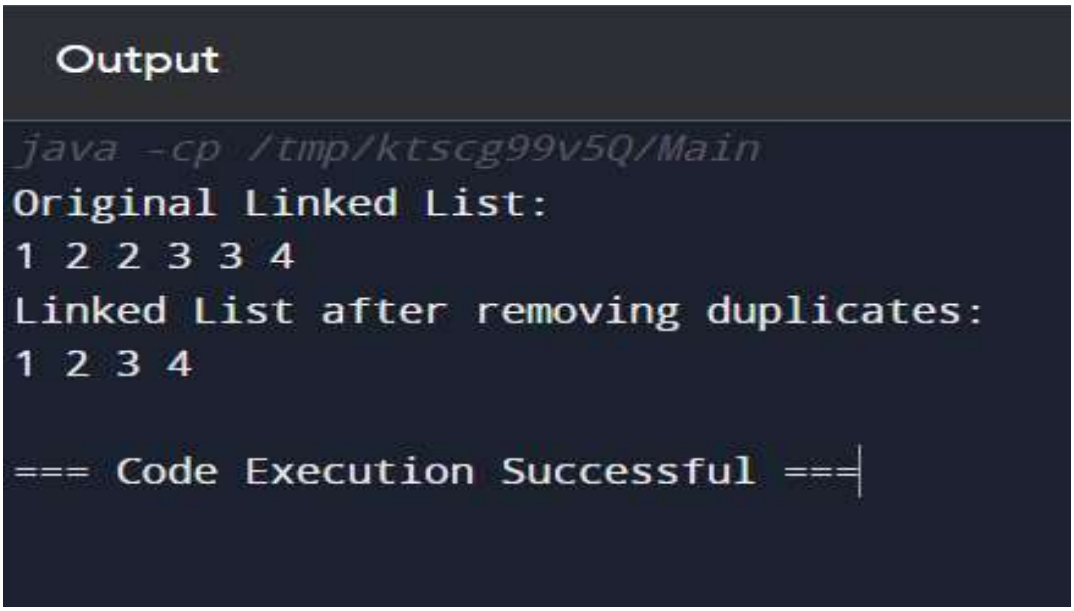
Task 5: Removing Duplicates from a Sorted Linked List

A sorted linked list has been constructed with repeated elements. Describe an algorithm to remove all duplicates from the linked list efficiently.

```
class Node {
   int data;
   Node next;
   Node(int data) {
      this.data = data;
      this.next = null;
   }
}
```

```
class LinkedList {
    Node head;



    void removeDuplicates() {
        Node current = head;
        while (current != null && current.next != null) {
            if (current.data == current.next.data) {
                current.next = current.next.next;
            } else {
                current = current.next;
            }
        }
    }
    void printList() {
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
        System.out.println();
    }
}
public class Main {
```

```
    public static void main(String[] args) {
        LinkedList list = new LinkedList();
        list.head = new Node(1);
        list.head.next = new Node(2);
        list.head.next.next = new Node(2);
        list.head.next.next.next = new Node(3);
        list.head.next.next.next.next = new Node(3);
        list.head.next.next.next.next.next = new Node(4);
        System.out.println("Original Linked List: ");
        list.printList();
        list.removeDuplicates();
        System.out.println("Linked List after removing duplicates: ");
        list.printList();
    }
}
```

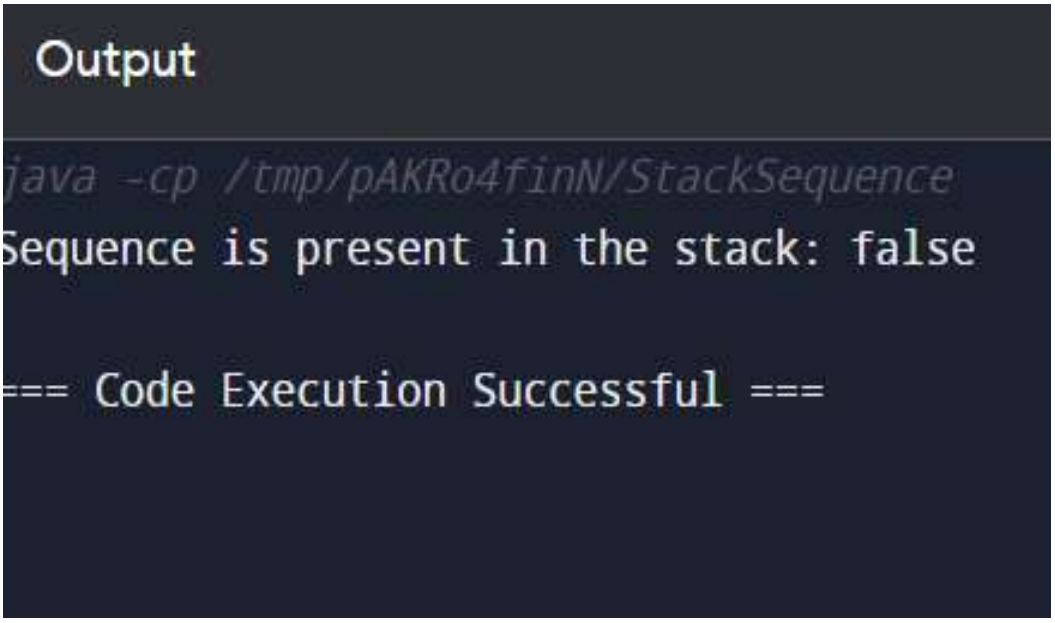


Task 6: Searching for a Sequence in a Stack

Given a stack and a smaller array representing a sequence, write a function that determines if the sequence is present in the stack. Consider the sequence present if, upon popping the elements, all elements of the array appear consecutively in the stack.

```
import java.util.Stack;

public class StackSequence {

   public static boolean isSequencePresent(Stack<Integer> stack, int[] sequence) {

      int sequenceIndex = sequence.length - 1;

      Stack<Integer> tempStack = new Stack<>();



      while (!stack.isEmpty() && sequenceIndex >= 0) {

         int currentElement = stack.pop();

         if (currentElement == sequence[sequenceIndex]) {

            sequenceIndex--;

         } else {

              tempStack.push(currentElement);

         }

      }

     while (!tempStack.isEmpty()) {

         stack.push(tempStack.pop());

      }
```

```
        return sequenceIndex == -1;
    }
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();
        stack.push(5);
        stack.push(1);
        stack.push(4);
        stack.push(3);
        stack.push(2);
        int[] sequence = {4, 5, 3, 1, 2};
        boolean isPresent = isSequencePresent(stack, sequence);
        System.out.println("Sequence is present in the stack: " + isPresent);
    }
}
```



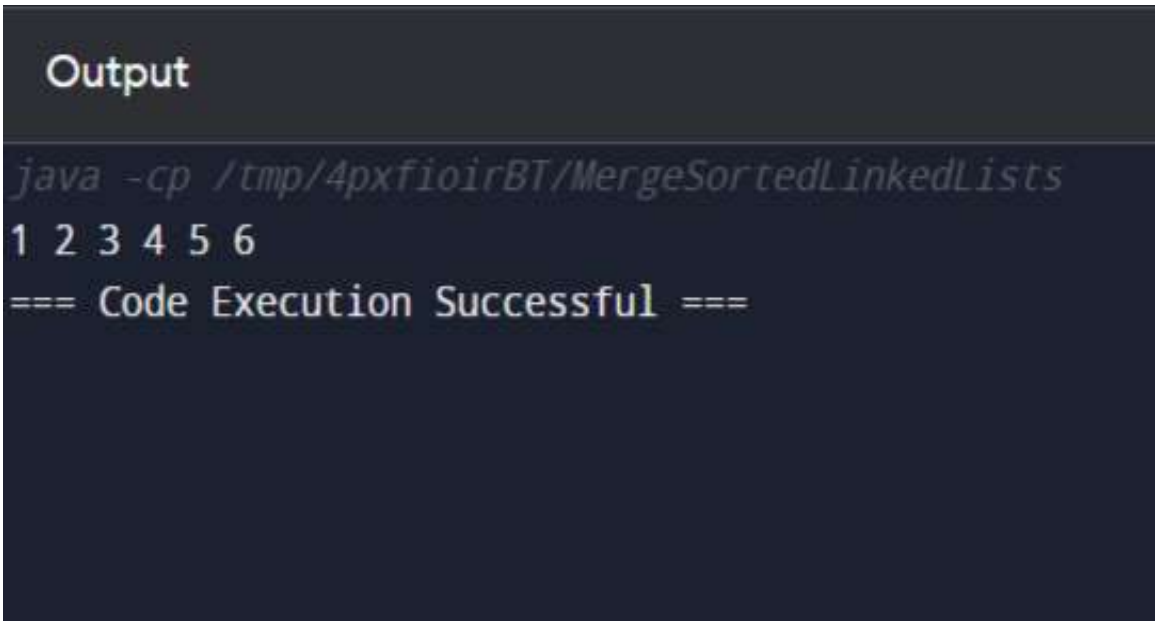
Task 7: Merging Two Sorted Linked Lists

You are provided with the heads of two sorted linked lists. The lists are sorted in ascending order. Create a merged linked list in ascending order from the two input lists without using any extra space (i.e., do not create any new nodes).

```
class ListNode {
    int val;
    ListNode next;
    ListNode(int val) {
        this.val = val;
    }
}
public class MergeSortedLinkedLists {
    public static ListNode mergeLists(ListNode l1, ListNode l2) {
        if (l1 == null) {
            return l2;
        }
        if (l2 == null) {
            return l1;
        }

        ListNode head = (l1.val < l2.val) ? l1 : l2;
        ListNode current1 = head == l1 ? l1 : l2;
        ListNode current2 = head == l1 ? l2 : l1;
```

```
    ListNode prev = null;
    ListNode temp = null;
     while (current1 != null && current2 != null) {
        if (current1.val <= current2.val) {
            prev = current1;
            current1 = current1.next;
        } else {
            temp = current2.next;
            prev.next = current2;
            current2.next = current1;
            prev = current2;
            current2 = temp;
        }
    }

    if (current1 == null) {
        prev.next = current2;
    }

    return head;
}

public static void main(String[] args) {
```

```
        ListNode l1 = new ListNode(1);
        l1.next = new ListNode(3);
        l1.next.next = new ListNode(5);

        ListNode l2 = new ListNode(2);
        l2.next = new ListNode(4);
        l2.next.next = new ListNode(6);

        ListNode mergedList = mergeLists(l1, l2);
         while (mergedList != null) {
            System.out.print(mergedList.val + " ");
            mergedList = mergedList.next;
        }
    }
}
```

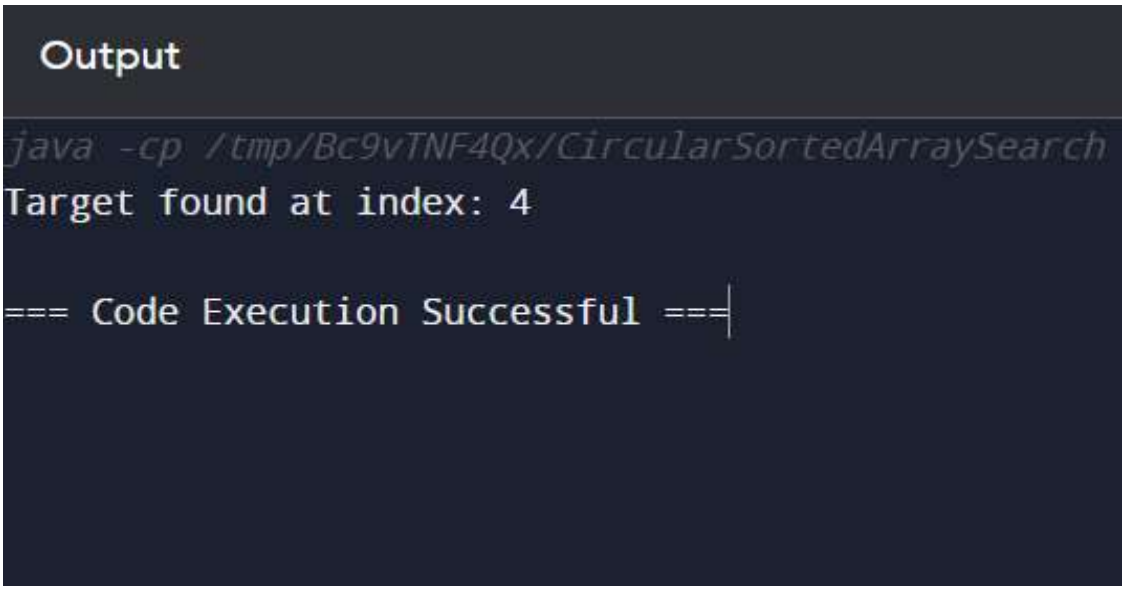


Task 8: Circular Queue Binary Search

Consider a circular queue (implemented using a fixed-size array) where the elements are sorted but have been rotated at an unknown index. Describe an approach to perform a binary search for a given element within this circular queue.

```
public class CircularSortedArraySearch {
    public static int search(int[] arr, int target) {
        int left = 0;
        int right = arr.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (arr[mid] == target) {
                return mid;
            }
            if (arr[left] <= arr[mid]) {
                if (target >= arr[left] && target < arr[mid]) {
                    right = mid - 1;
                } else {
                    left = mid + 1;
                }
            }
            else {
```

```
            if (target > arr[mid] && target <= arr[right]) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
    }
    return -1;
  }
  public static void main(String[] args) {
    int[] arr = {4, 5, 6, 7, 0, 1, 2};
    int target = 0;
    int index = search(arr, target);
    System.out.println("Target found at index: " + index);
  }
}
```



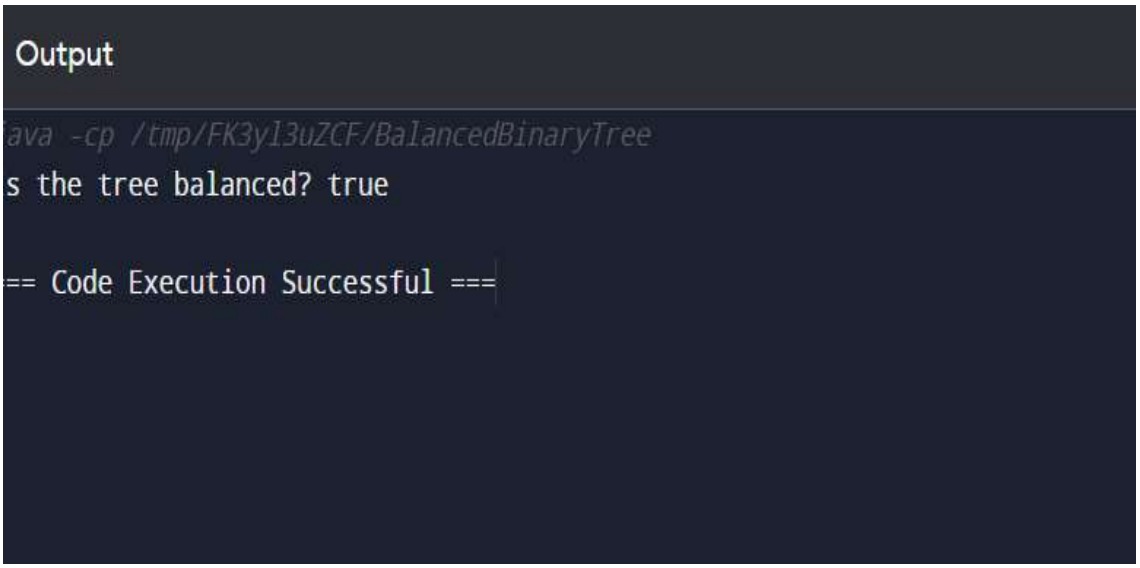
Day 7 and 8:

Task 1: Balanced Binary Tree Check

Write a function to check if a given binary tree is balanced. A balanced tree is one where the height of two subtrees of any node never differs by more than one.

```
class TreeNode {
   int val;
   TreeNode left;
   TreeNode right;
   TreeNode(int x) { val = x; }
}


public class BalancedBinaryTree {
   public static void main(String[] args) {
      TreeNode root = new TreeNode(1);
      root.left = new TreeNode(2);
      root.right = new TreeNode(3);
      root.left.left = new TreeNode(4);
      root.left.right = new TreeNode(5);
      BalancedBinaryTree solution = new BalancedBinaryTree();
      System.out.println("Is the tree balanced? " + solution.isBalanced(root));
   }
   public boolean isBalanced(TreeNode root) {
```

```
        return getHeight(root) != -1;
    }
    private int getHeight(TreeNode node) {
        if (node == null) return 0;

        int leftHeight = getHeight(node.left);
        if (leftHeight == -1) return -1;

        int rightHeight = getHeight(node.right);
        if (rightHeight == -1) return -1;

    if (Math.abs(leftHeight - rightHeight) > 1) return -1;

        return Math.max(leftHeight, rightHeight) + 1;
    }
}
```



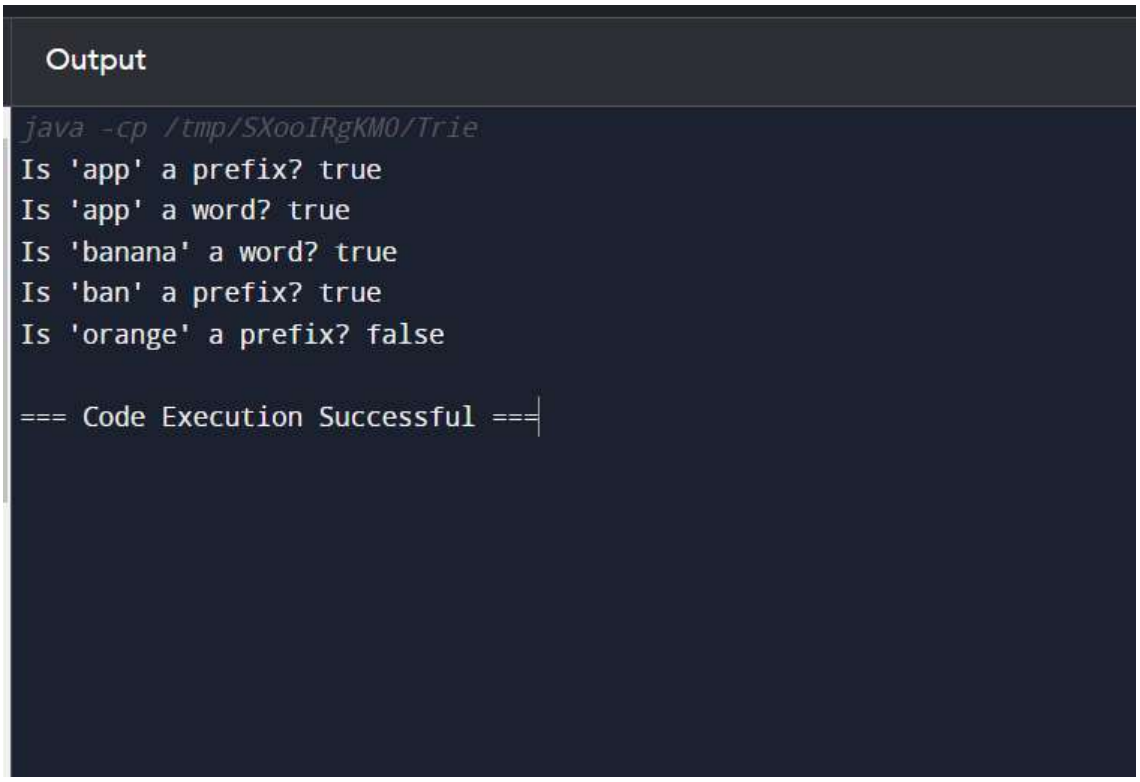
Task 2: Trie for Prefix Checking

Implement a trie data structure in C# that supports insertion of strings and provides a method to check if a given string is a prefix of any word in the trie.

```
class TrieNode {
    TrieNode[] children;
    boolean isEndOfWord;
    public TrieNode() {
        children = new TrieNode[26];
        isEndOfWord = false;
    }
}
public class Trie {
    private TrieNode root;

    public Trie() {
        root = new TrieNode();
    }
    public void insert(String word) {
        TrieNode current = root;
        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (current.children[index] == null) {
                current.children[index] = new TrieNode();
```

```
            }
            current = current.children[index];
        }
        current.isEndOfWord = true;
    }

    public boolean search(String word) {
        TrieNode current = root;
        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (current.children[index] == null) {
                return false;
            }
            current = current.children[index];
        }
        return current != null && current.isEndOfWord;
    }

    public boolean startsWith(String prefix) {
        TrieNode current = root;
        for (char c : prefix.toCharArray()) {
            int index = c - 'a';
            if (current.children[index] == null) {
                return false;
```

```
            }
            current = current.children[index];
        }
        return true;
    }

    public static void main(String[] args) {
        Trie trie = new Trie();

        trie.insert("apple");
        trie.insert("app");
        trie.insert("banana");

        System.out.println("Is 'app' a prefix? " + trie.startsWith("app"));
        System.out.println("Is 'app' a word? " + trie.search("app"));
        System.out.println("Is 'banana' a word? " + trie.search("banana"));
        System.out.println("Is 'ban' a prefix? " + trie.startsWith("ban"));
        System.out.println("Is 'orange' a prefix? " + trie.startsWith("orange"));
    }
}
```



Task 3: Implementing Heap Operations

Code a min-heap in C# with methods for insertion, deletion, and fetching the minimum element. Ensure that the heap property is maintained after each operation."

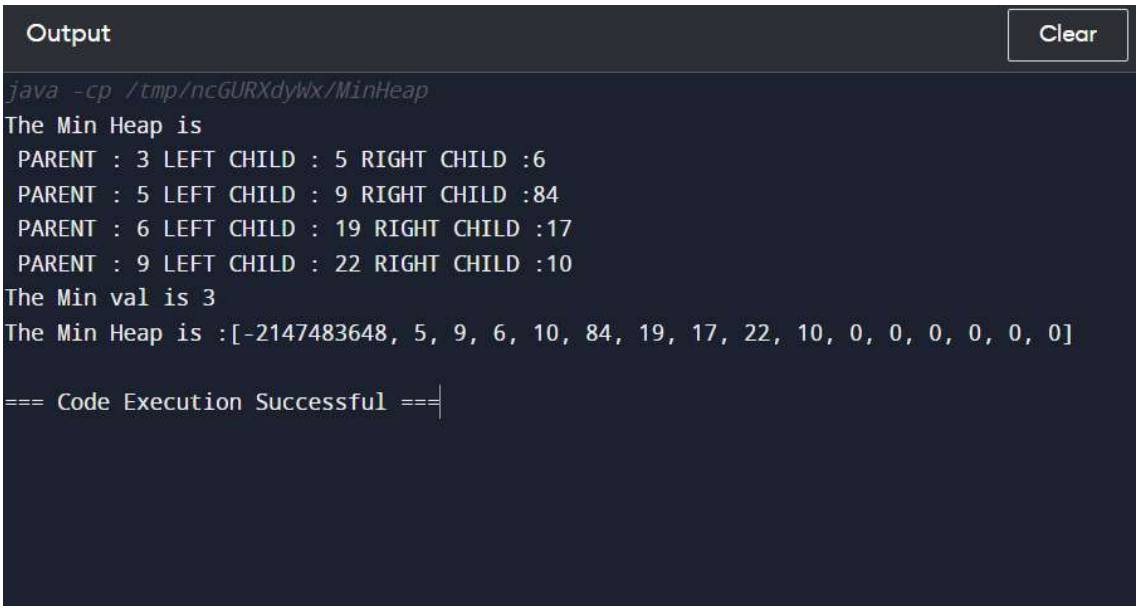
```
import java.util.Arrays;

public class MinHeap {

   private int[] Heap;

   private int size;

   private int maxsize;

   private static final int FRONT = 1;

   public MinHeap(int maxsize) {

      this.maxsize = maxsize;

      this.size = 0;
```

```
    Heap = new int[this.maxsize + 1];
    Heap[0] = Integer.MIN_VALUE;
  }
 private int parent(int pos) {
    return pos / 2;
  }

  private int leftChild(int pos) {
    return (2 * pos);
  }

  private int rightChild(int pos) {
    return (2 * pos) + 1;
  }

  private boolean isLeaf(int pos) {
    return pos >= (size / 2) && pos <= size;
  }

  private void swap(int fpos, int spos) {
    int tmp;
    tmp = Heap[fpos];
    Heap[fpos] = Heap[spos];
    Heap[spos] = tmp;
```

```
    }

    private void minHeapify(int pos) {
        if (!isLeaf(pos)) {
            if (Heap[pos] > Heap[leftChild(pos)] || Heap[pos] > Heap[rightChild(pos)]) {
                if (Heap[leftChild(pos)] < Heap[rightChild(pos)]) {
                    swap(pos, leftChild(pos));
                    minHeapify(leftChild(pos));
                } else {
                    swap(pos, rightChild(pos));
                    minHeapify(rightChild(pos));
                }
            }
        }
    }

    public void insert(int element) {
        Heap[++size] = element;
        int current = size;

        while (Heap[current] < Heap[parent(current)]) {
            swap(current, parent(current));
            current = parent(current);
```

```
    }
  }

  public void print() {
    for (int i = 1; i <= size / 2; i++) {
      System.out.print(" PARENT : " + Heap[i] + " LEFT CHILD : " +
Heap[2 * i] + " RIGHT CHILD :" + Heap[2 * i + 1]);
      System.out.println();
    }
  }
  public int remove() {
    int popped = Heap[FRONT];
    Heap[FRONT] = Heap[size--];
    minHeapify(FRONT);
    return popped;
  }
  public static void main(String[] args) {
    MinHeap minHeap = new MinHeap(15);
    minHeap.insert(5);
    minHeap.insert(3);
    minHeap.insert(17);
    minHeap.insert(10);
    minHeap.insert(84);
    minHeap.insert(19);
```

```
        minHeap.insert(6);
        minHeap.insert(22);
        minHeap.insert(9);
        System.out.println("The Min Heap is ");
        minHeap.print();
        System.out.println("The Min val is " + minHeap.remove());
        System.out.println("The Min Heap is :" + Arrays.toString(minHeap.Heap));
    }
}
```



Task 4: Graph Edge Addition Validation

Given a directed graph, write a function that adds an edge between two nodes and then checks if the graph still has no cycles. If a cycle is created, the edge should not be added.

```
import java.util.ArrayList;
```

```
import java.util.HashSet;
import java.util.List;
import java.util.Set;
public class Graph {
    private int V;
    private List<List<Integer>> adj;
    public Graph(int V) {
        this.V = V;
        adj = new ArrayList<>(V);
        for (int i = 0; i < V; i++) {
            adj.add(new ArrayList<>());
        }
    }
    public void addEdge(int u, int v) {
        adj.get(u).add(v);
    }
    private boolean isCyclicUtil(int v, boolean[] visited, boolean[] recursionStack) {
        if (recursionStack[v]) return true;
        if (visited[v]) return false;
        visited[v] = true;
        recursionStack[v] = true;
        for (int neighbor : adj.get(v)) {
            if (isCyclicUtil(neighbor, visited, recursionStack)) return true;
```

```
    }
    recursionStack[v] = false;
    return false;
}
public boolean isCyclic() {
    boolean[] visited = new boolean[V];
    boolean[] recursionStack = new boolean[V];
    for (int i = 0; i < V; i++) {
        if (isCyclicUtil(i, visited, recursionStack)) return true;
    }
    return false;
}
public boolean addEdgeWithoutCycle(int u, int v) {
    addEdge(u, v);
    if (isCyclic()) {
        adj.get(u).remove(adj.get(u).size() - 1);
        System.out.println("Edge not added: It creates a cycle");
        return false;
    } else {
        System.out.println("Edge added successfully");
        return true;
    }
}
```

```
    public static void main(String[] args) {
        Graph g = new Graph(4);
        g.addEdge(0, 1);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        System.out.println("Graph has cycle: " + g.isCyclic());
        g.addEdgeWithoutCycle(2, 3);
        System.out.println("Graph after adding edge: " + g.adj);
    }
}
```

Output

```
java -cp /tmp/FqJvhWBeMv/Graph
Graph has cycle: true
Edge not added: It creates a cycle
Graph after adding edge: [[1], [2], [0], []]

=== Code Execution Successful ===
```

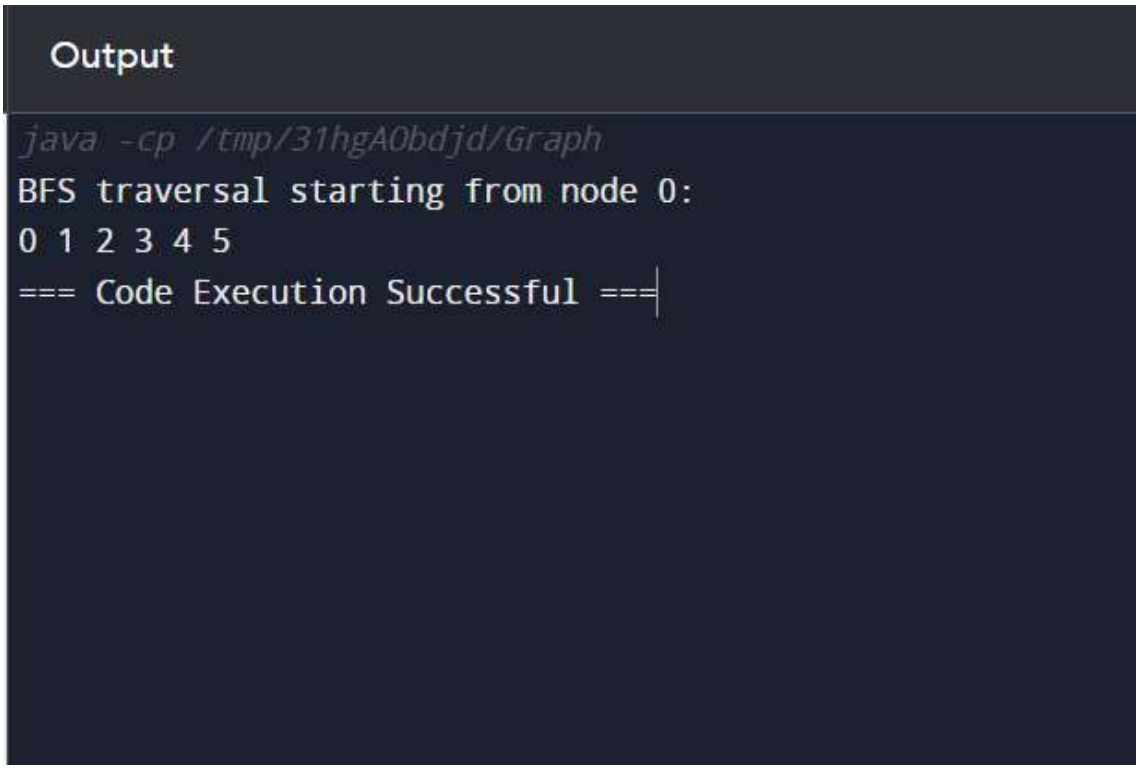
Task 5: Breadth-First Search (BFS) Implementation

For a given undirected graph, implement BFS to traverse the graph starting from a given node and print each node in the order it is visited.

```
import java.util.*;
public class Graph {
   private int V;
   private List<List<Integer>> adj;
   public Graph(int V) {
      this.V = V;
      adj = new ArrayList<>(V);
      for (int i = 0; i < V; i++) {
         adj.add(new ArrayList<>());
      }
   }
   public void addEdge(int u, int v) {
      adj.get(u).add(v);
      adj.get(v).add(u);
   }
   public void BFS(int start) {
      boolean[] visited = new boolean[V];
      Queue<Integer> queue = new LinkedList<>();
      visited[start] = true;
      queue.offer(start);
```

```
    while (!queue.isEmpty()) {
      int node = queue.poll();
      System.out.print(node + " ");
       for (int neighbor : adj.get(node)) {
         if (!visited[neighbor]) {
            visited[neighbor] = true;
            queue.offer(neighbor);
         }
      }
   }
}


public static void main(String[] args) {
   Graph g = new Graph(6);
   g.addEdge(0, 1);
   g.addEdge(0, 2);
   g.addEdge(1, 3);
   g.addEdge(1, 4);
   g.addEdge(2, 4);
   g.addEdge(3, 5);
   g.addEdge(4, 5);
  System.out.println("BFS traversal starting from node 0:");
   g.BFS(0);
```

```
    }
}
```



Task 6: Depth-First Search (DFS) Recursive

Write a recursive DFS function for a given undirected graph. The function should visit every node and print it out.

```
import java.util.*;

public class Graph {
    private int V;
    private List<List<Integer>> adj;
    public Graph(int V) {
        this.V = V;
```

```
    adj = new ArrayList<>(V);
    for (int i = 0; i < V; i++) {
        adj.add(new ArrayList<>());
    }
}
public void addEdge(int u, int v) {
    adj.get(u).add(v);
    adj.get(v).add(u);
}
private void DFSUtil(int v, boolean[] visited) {
    visited[v] = true;
    System.out.print(v + " ");

    for (int neighbor : adj.get(v)) {
        if (!visited[neighbor]) {
            DFSUtil(neighbor, visited);
        }
    }
}
public void DFS(int start) {
    boolean[] visited = new boolean[V];
    DFSUtil(start, visited);
}
```

```
    public static void main(String[] args) {
        Graph g = new Graph(6);
        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 3);
        g.addEdge(1, 4);
        g.addEdge(2, 4);
        g.addEdge(3, 5);
        g.addEdge(4, 5);

        System.out.println("DFS traversal starting from node 0:");
        g.DFS(0);
    }
}
```

Output

```
java -cp /tmp/CH0kQr3SqW/Graph
DFS traversal starting from node 0:
0 1 3 5 4 2
=== Code Execution Successful ===
```

Day 9 and 10:

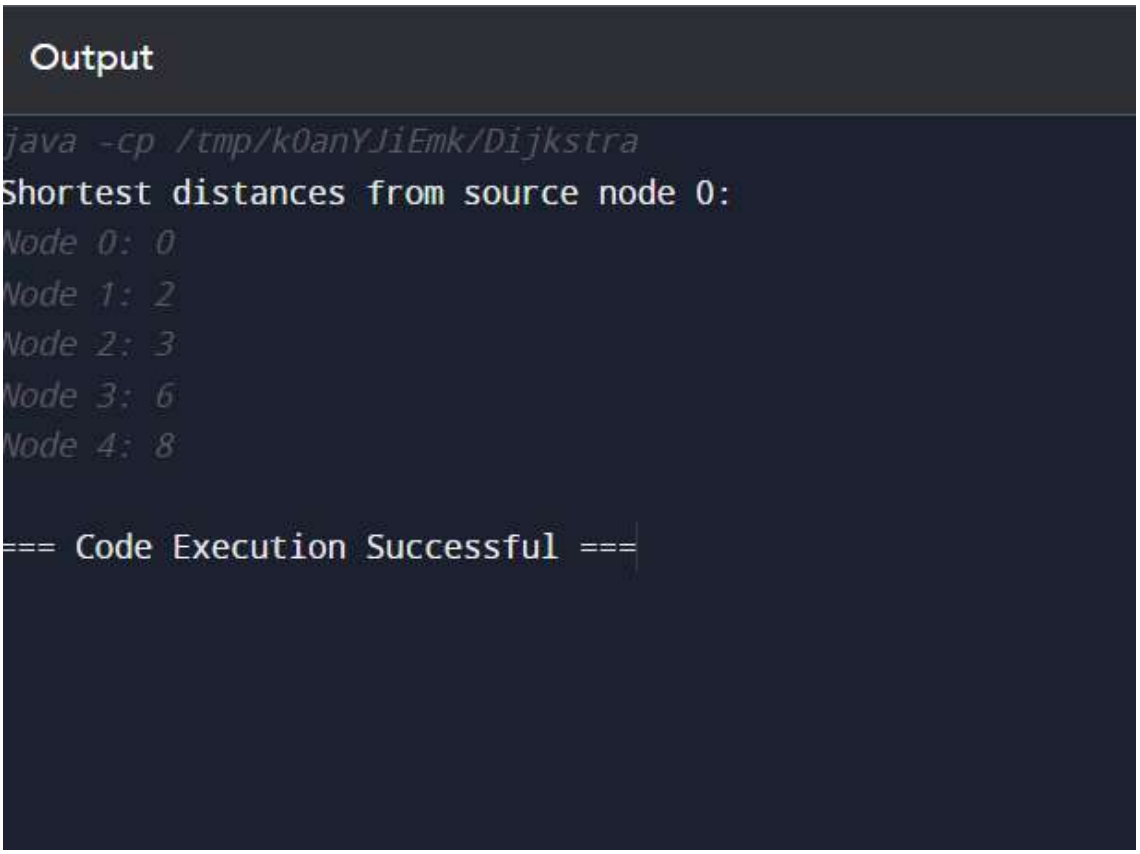
Task 1: Dijkstra's Shortest Path Finder

Code Dijkstra's algorithm to find the shortest path from a start node to every other node in a weighted graph with positive weights.

```
import java.util.*;
public class Dijkstra {
private int V;
   private List<List<Node>> adj;
   static class Node implements Comparable<Node> {
      int vertex;
      int weight;
      Node(int vertex, int weight) {
         this.vertex = vertex;
         this.weight = weight;
      }
      public int compareTo(Node other) {
         return Integer.compare(weight, other.weight);
      }
   }
   public Dijkstra(int V) {
      this.V = V;
      adj = new ArrayList<>(V);
      for (int i = 0; i < V; i++) {
```

```
        adj.add(new ArrayList<>());
      }
   }
   public void addEdge(int u, int v, int weight) {
      adj.get(u).add(new Node(v, weight));
      adj.get(v).add(new Node(u, weight));
   }



   public void dijkstra(int source) {
      int[] dist = new int[V];
      Arrays.fill(dist, Integer.MAX_VALUE);  infinity
      dist[source] = 0;
      PriorityQueue<Node> pq = new PriorityQueue<>();
      pq.offer(new Node(source, 0));
      while (!pq.isEmpty()) {
         Node node = pq.poll();
         int u = node.vertex;
         for (Node neighbor : adj.get(u)) {
            int v = neighbor.vertex;
            int weight = neighbor.weight;
            if (dist[u] != Integer.MAX_VALUE && dist[u] + weight <
dist[v]) {
               dist[v] = dist[u] + weight;
```

```
                pq.offer(new Node(v, dist[v]));
            }
        }
    }
    System.out.println("Shortest distances from source node " + source + ":");
    for (int i = 0; i < V; i++) {
        System.out.println("Node " + i + ": " + dist[i]);
    }
  }
  public static void main(String[] args) {
    int V = 5;
    Dijkstra graph = new Dijkstra(V);
    graph.addEdge(0, 1, 2);
    graph.addEdge(0, 2, 4);
    graph.addEdge(1, 2, 1);
    graph.addEdge(1, 3, 7);
    graph.addEdge(2, 3, 3);
    graph.addEdge(2, 4, 5);
    graph.addEdge(3, 4, 2);
    graph.dijkstra(0);
  }
}
```



Task 2: Kruskal's Algorithm for MST

Implement Kruskal's algorithm to find the minimum spanning tree of a given connected, undirected graph with non-negative edge weights.

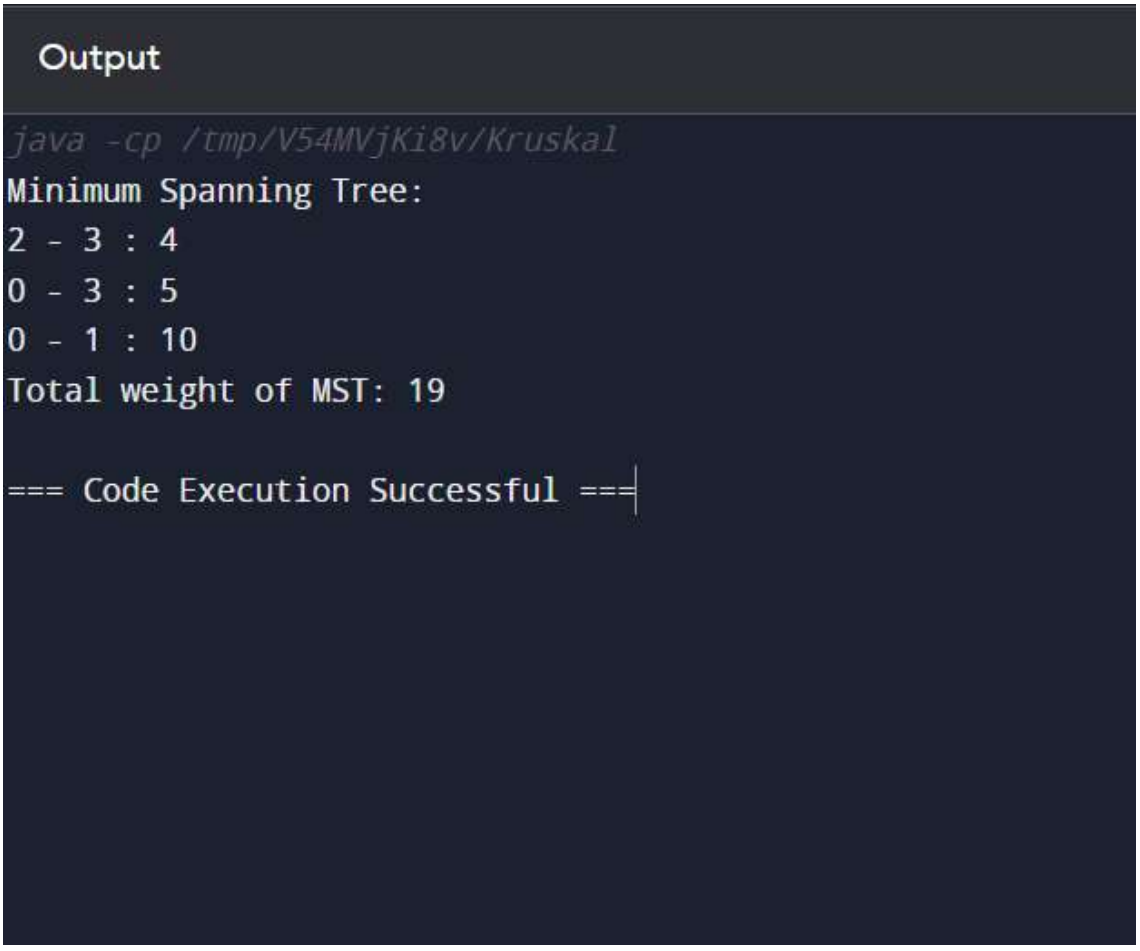
```
import java.util.*;

public class Kruskal {
    private int V;
    private List<Edge> edges;
    static class Edge implements Comparable<Edge> {
        int source;
        int destination;
        int weight;
```

```
    Edge(int source, int destination, int weight) {
        this.source = source;
        this.destination = destination;
        this.weight = weight;
    }
    public int compareTo(Edge other) {
        return Integer.compare(weight, other.weight);
    }
}
public Kruskal(int V) {
    this.V = V;
    edges = new ArrayList<>();
}
public void addEdge(int source, int destination, int weight) {
    edges.add(new Edge(source, destination, weight));
}
private int findParent(int[] parent, int vertex) {
    if (parent[vertex] != vertex) {
        parent[vertex] = findParent(parent, parent[vertex]);
    }
    return parent[vertex];
}
private void union(int[] parent, int[] rank, int x, int y) {
    int xRoot = findParent(parent, x);
```

```
    int yRoot = findParent(parent, y);

    if (rank[xRoot] < rank[yRoot]) {
        parent[xRoot] = yRoot;
    } else if (rank[yRoot] < rank[xRoot]) {
        parent[yRoot] = xRoot;
    } else {
        parent[yRoot] = xRoot;
        rank[xRoot]++;
    }
}


public void kruskalMST() {
    Collections.sort(edges);
     int[] parent = new int[V];
    int[] rank = new int[V];
    for (int i = 0; i < V; i++) {
        parent[i] = i;
        rank[i] = 0;
    }

    List<Edge> mst = new ArrayList<>();
    int mstWeight = 0;
```

```
        for (Edge edge : edges) {
            int sourceParent = findParent(parent, edge.source);
            int destParent = findParent(parent, edge.destination);

            if (sourceParent != destParent) {
                mst.add(edge);
                mstWeight += edge.weight;
                union(parent, rank, sourceParent, destParent);
            }
        }
        System.out.println("Minimum Spanning Tree:");
        for (Edge edge : mst) {
            System.out.println(edge.source + " - " + edge.destination + " : "
+ edge.weight);
        }
        System.out.println("Total weight of MST: " + mstWeight);
    }


    public static void main(String[] args) {
        int V = 4;
        Kruskal graph = new Kruskal(V);
        graph.addEdge(0, 1, 10);
```

```
        graph.addEdge(0, 2, 6);
        graph.addEdge(0, 3, 5);
        graph.addEdge(1, 3, 15);
        graph.addEdge(2, 3, 4);
        graph.kruskalMST();
    }
}
```



Task 3: Union-Find for Cycle Detection

Write a Union-Find data structure with path compression. Use this data structure to detect a cycle in an undirected graph.

```
import java.util.*;
public class UnionFind {
    private int[] parent;
    private int[] rank;
    public UnionFind(int n) {
        parent = new int[n];
        rank = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            rank[i] = 0;
        }
    }
    public int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }
    public void union(int x, int y) {
        int xRoot = find(x);
        int yRoot = find(y);
        if (xRoot == yRoot) return;
        if (rank[xRoot] < rank[yRoot]) {
            parent[xRoot] = yRoot;
```

```
    } else if (rank[yRoot] < rank[xRoot]) {
        parent[yRoot] = xRoot;
    } else {
        parent[yRoot] = xRoot;
        rank[xRoot]++;
    }
}


public boolean hasCycle(int[][] edges) {
    for (int[] edge : edges) {
        int x = edge[0];
        int y = edge[1];
        int xRoot = find(x);
        int yRoot = find(y);
        if (xRoot == yRoot) {
            return true;
        }
        union(x, y);
    }
    return false;
}
public static void main(String[] args) {
    int[][] edges = {{0, 1}, {1, 2}, {2, 3}, {3, 0}};
```

```
        int n = 4;
        UnionFind uf = new UnionFind(n);

        if (uf.hasCycle(edges)) {
            System.out.println("Cycle detected in the graph.");
        } else {
            System.out.println("No cycle detected in the graph.");
        }
    }
}
```

Output

```
java -cp /tmp/jzoP1LWeTn/UnionFind
Cycle detected in the graph.

=== Code Execution Successful ===
```

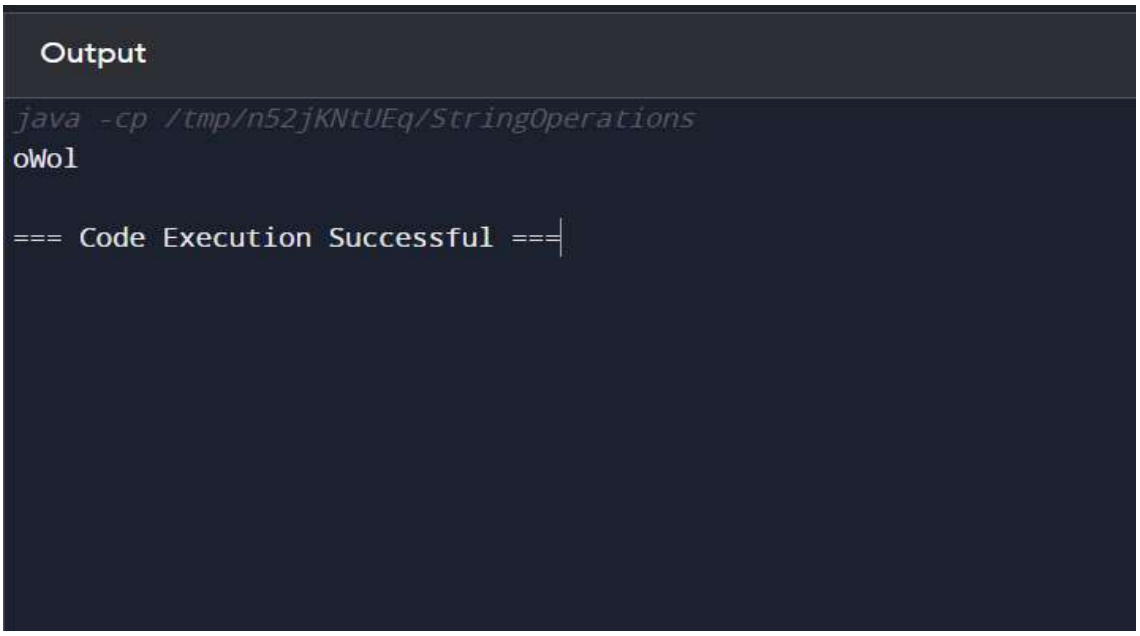
Day 11:

Task 1: String Operations

Write a method that takes two strings, concatenates them, reverses the result, and then extracts the middle substring of the given length. Ensure your method handles edge cases, such as an empty string or a substring length larger than the concatenated string.

```
public class StringOperations {
    public static String extractMiddleSubstring(String str1, String str2, int length) {
        String concatenated = str1.concat(str2);
        StringBuilder reversed = new StringBuilder();
        reversed.append(concatenated);
        reversed.reverse();
        int startIndex = (reversed.length() - length) / 2;
        if (startIndex < 0) {
            return "Invalid input";
        }
        return reversed.substring(startIndex, startIndex + length);
    }
    public static void main(String[] args) {
        String str1 = "Hello";
        String str2 = "World";
        int length = 4;
        System.out.println(extractMiddleSubstring(str1, str2, length));
```

```
    }
}
```

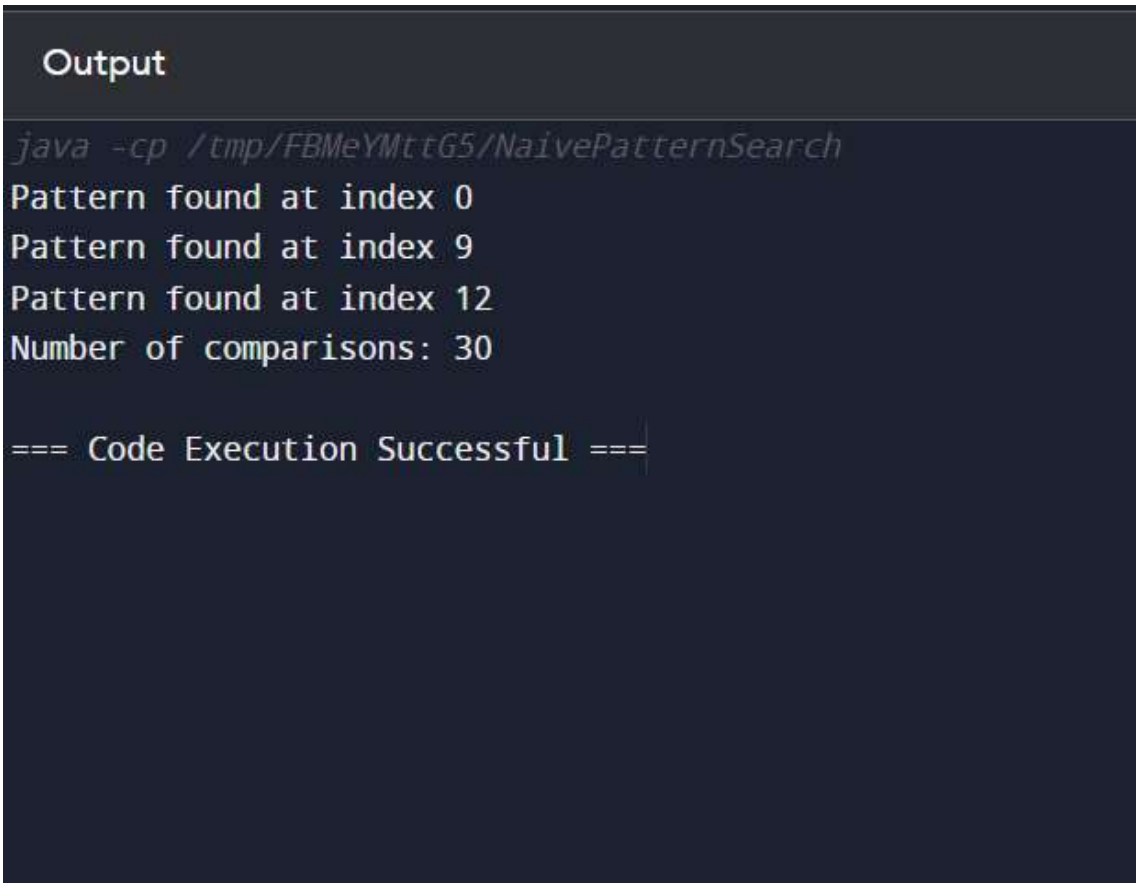


Task 2: Naive Pattern Search

Implement the naive pattern searching algorithm to find all occurrences of a pattern within a given text string. Count the number of comparisons made during the search to evaluate the efficiency of the algorithm.

```
public class NaivePatternSearch {
  public static int searchPattern(String text, String pattern) {
      int n = text.length();
      int m = pattern.length();
      int comparisons = 0;
      for (int i = 0; i <= n - m; i++) {
         int j;
```

```
            for (j = 0; j < m; j++) {
                comparisons++;
                if (text.charAt(i + j) != pattern.charAt(j)) {
                    break;
                }
            }
            if (j == m) {
                System.out.println("Pattern found at index " + i);
            }
        }
        return comparisons;
    }
    public static void main(String[] args) {
        String text = "AABAACAADAABAABA";
        String pattern = "AABA";
        int comparisons = searchPattern(text, pattern);
        System.out.println("Number of comparisons: " + comparisons);
    }
}
```



Task 3: Implementing the KMP Algorithm

Code the Knuth-Morris-Pratt (KMP) algorithm in C# for pattern searching which pre-processes the pattern to reduce the number of comparisons. Explain how this pre-processing improves the search time compared to the naive approach.

```
public class KMPAlgorithm {
  public static int[] computeLPSArray(String pattern) {
      int m = pattern.length();
      int[] lps = new int[m];
      int len = 0;
      int i = 1;
```

```
    while (i < m) {
        if (pattern.charAt(i) == pattern.charAt(len)) {
            len++;
            lps[i] = len;
            i++;
        } else {
            if (len != 0) {
                len = lps[len - 1];
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
    return lps;
}
public static void searchPattern(String text, String pattern) {
    int n = text.length();
    int m = pattern.length();
    int[] lps = computeLPSArray(pattern);
    int i = 0;
    int j = 0;
```

```
        while (i < n) {
            if (pattern.charAt(j) == text.charAt(i)) {
                i++;
                j++;
            }
            if (j == m) {
                System.out.println("Pattern found at index " + (i - j));
                j = lps[j - 1];
            } else if (i < n && pattern.charAt(j) != text.charAt(i)) {
                if (j != 0) {
                    j = lps[j - 1];
                } else {
                    i++;
                }
            }
        }
    }
    public static void main(String[] args) {
        String text = "ABABDABACDABABCABAB";
        String pattern = "ABABCABAB";
        System.out.println("Pattern found at indices:");
        searchPattern(text, pattern);
    }
}
```

Output

```
java -cp /tmp/Nfsevn4pJh/KMPAlgorithm
Pattern found at indices:
Pattern found at index 10

=== Code Execution Successful ===
```