**19-11-2024**

**CODING PRACTICE PROBLEMS**

**1.Next permutation**

import java.util.\*;

public class Problem1 {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter the number of elements: ");

int n = sc.nextInt();

int[] nums = new int[n];

System.out.println("Enter the elements:");

for (int i = 0; i < n; i++) {

nums[i] = sc.nextInt();

}

nextPermutation(nums);

System.out.println("Next permutation: " + Arrays.toString(nums));

}

public static void nextPermutation(int[] nums) {

int i = nums.length - 2;

while (i >= 0 && nums[i] >= nums[i + 1]) {

i--;

}

if (i >= 0) {

int j = nums.length - 1;

while (nums[j] <= nums[i]) {

j--;

}

swap(nums, i, j);

}

reverse(nums, i + 1);

}

private static void swap(int[] nums, int i, int j) {

int temp = nums[i];

nums[i] = nums[j];

nums[j] = temp;

}

private static void reverse(int[] nums, int start) {

int end = nums.length - 1;

while (start < end) {

swap(nums, start, end);

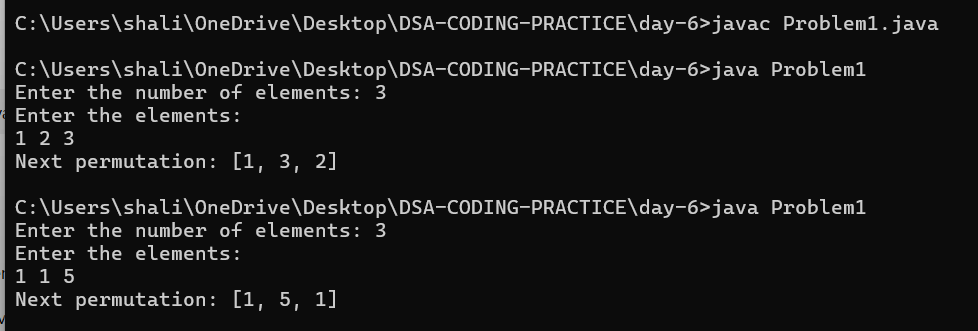
start++;

end--;

}

}

}



**2.Spiral matrix**

import java.util.\*;

public class Problem1 {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter the number of rows: ");

int m = sc.nextInt();

System.out.print("Enter the number of columns: ");

int n = sc.nextInt();

int[][] matrix = new int[m][n];

System.out.println("Enter the elements of the matrix:");

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

for (int j = 0; j < n; j++) {

matrix[i][j] = sc.nextInt();

}

}

List<Integer> result = spiralOrder(matrix);

System.out.println("Spiral order: " + result);

}

public static List<Integer> spiralOrder(int[][] matrix) {

List<Integer> result = new ArrayList<>();

if (matrix == null || matrix.length == 0) {

return result;

}

int top = 0, bottom = matrix.length - 1, left = 0, right = matrix[0].length - 1;

while (top <= bottom && left <= right) {

for (int i = left; i <= right; i++) {

result.add(matrix[top][i]);

}

top++;

for (int i = top; i <= bottom; i++) {

result.add(matrix[i][right]);

}

right--;

if (top <= bottom) {

for (int i = right; i >= left; i--) {

result.add(matrix[bottom][i]);

}

bottom--;

}

if (left <= right) {

for (int i = bottom; i >= top; i--) {

result.add(matrix[i][left]);

}

left++;

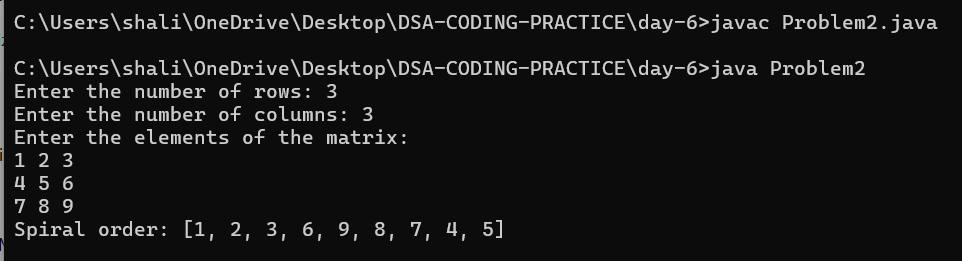
}

}

return result;

}

}



**3.Longest substring without repeating characters**

import java.util.\*;

public class Problem3 {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter the string: ");

String s = sc.nextLine();

int result = lengthOfLongestSubstring(s);

System.out.println("Length of the longest substring without repeating characters: " + result);

}

public static int lengthOfLongestSubstring(String s) {

Set<Character> set = new HashSet<>();

int maxLength = 0, left = 0;

for (int right = 0; right < s.length(); right++) {

while (set.contains(s.charAt(right))) {

set.remove(s.charAt(left));

left++;

}

set.add(s.charAt(right));

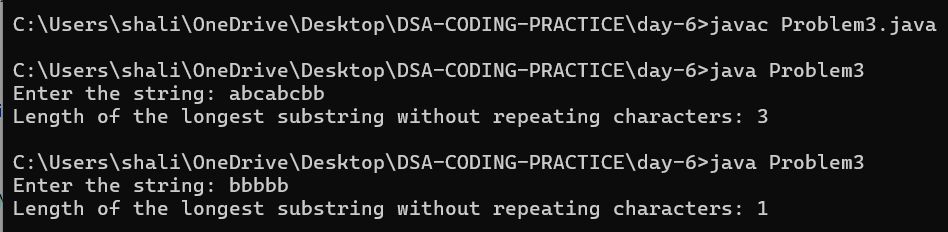
maxLength = Math.max(maxLength, right - left + 1);

}

return maxLength;

}

}



**4.Remove linked list elements**

import java.util.\*;

public class Problem4 {

static class ListNode {

int val;

ListNode next;

ListNode(int val) {

this.val = val;

}

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter the number of nodes: ");

int n = sc.nextInt();

System.out.println("Enter the values of the nodes:");

ListNode dummy = new ListNode(0), current = dummy;

for (int i = 0; i < n; i++) {

current.next = new ListNode(sc.nextInt());

current = current.next;

}

System.out.print("Enter the value to remove: ");

int val = sc.nextInt();

ListNode newHead = removeElements(dummy.next, val);

System.out.print("Modified list: ");

while (newHead != null) {

System.out.print(newHead.val + " ");

newHead = newHead.next;

}

}

public static ListNode removeElements(ListNode head, int val) {

ListNode dummy = new ListNode(0);

dummy.next = head;

ListNode current = dummy;

while (current.next != null) {

if (current.next.val == val) {

current.next = current.next.next;

} else {

current = current.next;

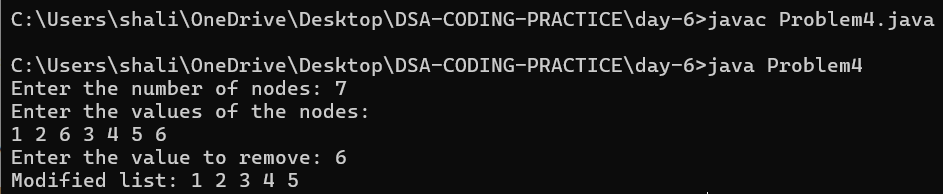
}

}

return dummy.next;

}

}



**5.Palindrome linked list**

import java.util.\*;

public class Problem5 {

static class ListNode {

int val;

ListNode next;

ListNode(int val) {

this.val = val;

}

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter the number of nodes: ");

int n = sc.nextInt();

System.out.println("Enter the values of the nodes:");

ListNode dummy = new ListNode(0), current = dummy;

for (int i = 0; i < n; i++) {

current.next = new ListNode(sc.nextInt());

current = current.next;

}

boolean result = isPalindrome(dummy.next);

System.out.println("Is palindrome: " + result);

}

public static boolean isPalindrome(ListNode head) {

if (head == null || head.next == null) {

return true;

}

ListNode slow = head, fast = head;

while (fast != null && fast.next != null) {

slow = slow.next;

fast = fast.next.next;

}

ListNode secondHalf = reverseList(slow);

ListNode firstHalf = head;

while (secondHalf != null) {

if (firstHalf.val != secondHalf.val) {

return false;

}

firstHalf = firstHalf.next;

secondHalf = secondHalf.next;

}

return true;

}

private static ListNode reverseList(ListNode head) {

ListNode prev = null;

while (head != null) {

ListNode nextNode = head.next;

head.next = prev;

prev = head;

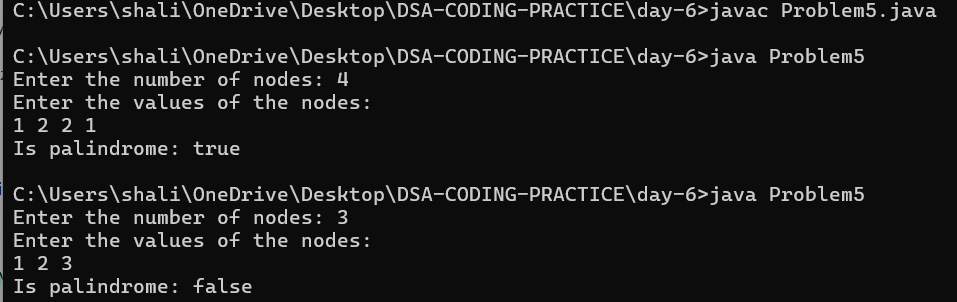
head = nextNode;

}

return prev;

}

}



**6.Minimum path sum**

import java.util.\*;

public class Problem6 {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter the number of rows: ");

int m = sc.nextInt();

System.out.print("Enter the number of columns: ");

int n = sc.nextInt();

int[][] grid = new int[m][n];

System.out.println("Enter the elements of the grid:");

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

for (int j = 0; j < n; j++) {

grid[i][j] = sc.nextInt();

}

}

int result = minPathSum(grid);

System.out.println("Minimum path sum: " + result);

}

public static int minPathSum(int[][] grid) {

int m = grid.length, n = grid[0].length;

for (int i = 1; i < m; i++) {

grid[i][0] += grid[i - 1][0];

}

for (int j = 1; j < n; j++) {

grid[0][j] += grid[0][j - 1];

}

for (int i = 1; i < m; i++) {

for (int j = 1; j < n; j++) {

grid[i][j] += Math.min(grid[i - 1][j], grid[i][j - 1]);

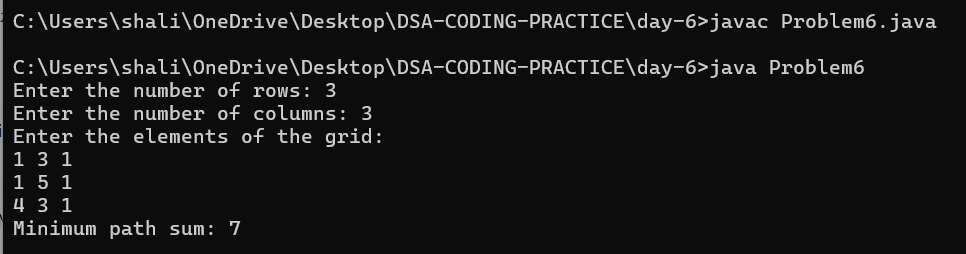
}

}

return grid[m - 1][n - 1];

}

}



**7.Validate binary search tree**

import java.util.\*;

public class Problem7 {

static class TreeNode {

int val;

TreeNode left, right;

TreeNode(int val) {

this.val = val;

}

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.println("Enter the number of nodes:");

int n = sc.nextInt();

if (n == 0) {

System.out.println("Tree is empty.");

return;

}

System.out.println("Enter the values of the nodes in level-order (use -1 for null):");

TreeNode root = buildTree(sc, n);

boolean result = isValidBST(root);

System.out.println("Is valid BST: " + result);

}

public static TreeNode buildTree(Scanner sc, int n) {

if (n == 0) return null;

TreeNode root = new TreeNode(sc.nextInt());

Queue<TreeNode> queue = new LinkedList<>();

queue.add(root);

for (int i = 1; i < n; ) {

TreeNode current = queue.poll();

int leftVal = sc.nextInt();

if (leftVal != -1) {

current.left = new TreeNode(leftVal);

queue.add(current.left);

}

i++;

if (i < n) {

int rightVal = sc.nextInt();

if (rightVal != -1) {

current.right = new TreeNode(rightVal);

queue.add(current.right);

}

i++;

}

}

return root;

}

public static boolean isValidBST(TreeNode root) {

return validate(root, Long.MIN\_VALUE, Long.MAX\_VALUE);

}

private static boolean validate(TreeNode node, long min, long max) {

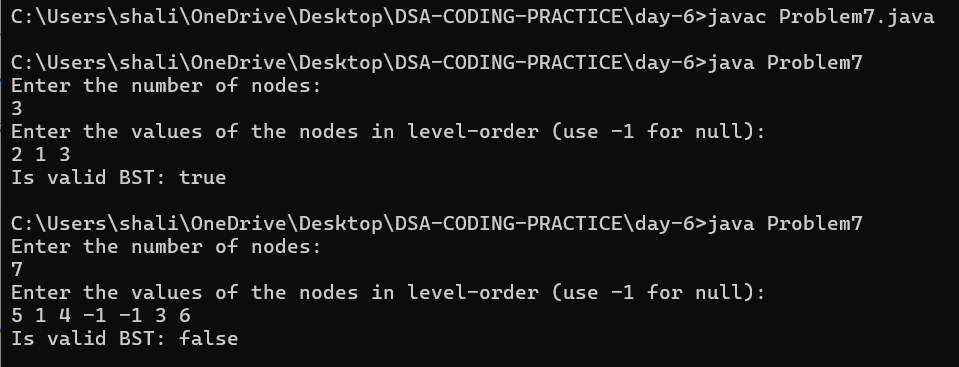
if (node == null) return true;

if (node.val <= min || node.val >= max) return false;

return validate(node.left, min, node.val) && validate(node.right, node.val, max);

}

}



**8.Word ladder**

import java.util.\*;

public class Problem8 {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter the beginWord: ");

String beginWord = sc.next();

System.out.print("Enter the endWord: ");

String endWord = sc.next();

System.out.print("Enter the number of words in the wordList: ");

int n = sc.nextInt();

List<String> wordList = new ArrayList<>();

System.out.println("Enter the words in the wordList:");

for (int i = 0; i < n; i++) {

wordList.add(sc.next());

}

int result = ladderLength(beginWord, endWord, wordList);

System.out.println("Shortest transformation sequence length: " + result);

}

public static int ladderLength(String beginWord, String endWord, List<String> wordList) {

Set<String> wordSet = new HashSet<>(wordList);

if (!wordSet.contains(endWord)) {

return 0;

}

Queue<String> queue = new LinkedList<>();

queue.add(beginWord);

int steps = 1;

while (!queue.isEmpty()) {

int size = queue.size();

for (int i = 0; i < size; i++) {

String currentWord = queue.poll();

if (currentWord.equals(endWord)) {

return steps;

}

for (String neighbor : getNeighbors(currentWord, wordSet)) {

queue.add(neighbor);

}

}

steps++;

}

return 0;

}

private static List<String> getNeighbors(String word, Set<String> wordSet) {

List<String> neighbors = new ArrayList<>();

char[] charArray = word.toCharArray();

for (int i = 0; i < charArray.length; i++) {

char originalChar = charArray[i];

for (char c = 'a'; c <= 'z'; c++) {

if (c == originalChar) continue;

charArray[i] = c;

String newWord = new String(charArray);

if (wordSet.contains(newWord)) {

neighbors.add(newWord);

wordSet.remove(newWord);

}

}

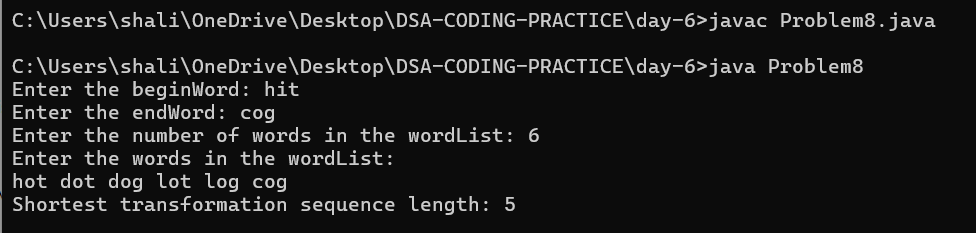
charArray[i] = originalChar;

}

return neighbors;

}

}



**9.Word ladder -II**

import java.util.\*;

public class Problem9 {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter the beginWord: ");

String beginWord = sc.next();

System.out.print("Enter the endWord: ");

String endWord = sc.next();

System.out.print("Enter the number of words in the wordList: ");

int n = sc.nextInt();

List<String> wordList = new ArrayList<>();

System.out.println("Enter the words in the wordList:");

for (int i = 0; i < n; i++) {

wordList.add(sc.next());

}

List<List<String>> result = findLadders(beginWord, endWord, wordList);

System.out.println("All shortest transformation sequences:");

for (List<String> sequence : result) {

System.out.println(sequence);

}

}

public static List<List<String>> findLadders(String beginWord, String endWord, List<String> wordList) {

List<List<String>> results = new ArrayList<>();

Set<String> wordSet = new HashSet<>(wordList);

if (!wordSet.contains(endWord)) {

return results;

}

Map<String, List<String>> neighborsMap = new HashMap<>();

Map<String, Integer> distance = new HashMap<>();

bfs(beginWord, endWord, wordSet, neighborsMap, distance);

List<String> path = new ArrayList<>();

path.add(beginWord);

dfs(beginWord, endWord, neighborsMap, distance, path, results);

return results;

}

private static void bfs(String beginWord, String endWord, Set<String> wordSet, Map<String, List<String>> neighborsMap, Map<String, Integer> distance) {

Queue<String> queue = new LinkedList<>();

queue.add(beginWord);

distance.put(beginWord, 0);

for (String word : wordSet) {

neighborsMap.put(word, new ArrayList<>());

}

while (!queue.isEmpty()) {

String currentWord = queue.poll();

int currentDistance = distance.get(currentWord);

for (String neighbor : getNeighbors(currentWord, wordSet)) {

neighborsMap.get(currentWord).add(neighbor);

if (!distance.containsKey(neighbor)) {

distance.put(neighbor, currentDistance + 1);

if (neighbor.equals(endWord)) continue;

queue.add(neighbor);

}

}

}

}

private static List<String> getNeighbors(String word, Set<String> wordSet) {

List<String> neighbors = new ArrayList<>();

char[] charArray = word.toCharArray();

for (int i = 0; i < charArray.length; i++) {

char originalChar = charArray[i];

for (char c = 'a'; c <= 'z'; c++) {

if (c == originalChar) continue;

charArray[i] = c;

String newWord = new String(charArray);

if (wordSet.contains(newWord)) {

neighbors.add(newWord);

}

}

charArray[i] = originalChar;

}

return neighbors;

}

private static void dfs(String currentWord, String endWord, Map<String, List<String>> neighborsMap, Map<String, Integer> distance, List<String> path, List<List<String>> results) {

if (currentWord.equals(endWord)) {

results.add(new ArrayList<>(path));

return;

}

for (String neighbor : neighborsMap.getOrDefault(currentWord, new ArrayList<>())) {

if (distance.get(neighbor) == distance.get(currentWord) + 1) {

path.add(neighbor);

dfs(neighbor, endWord, neighborsMap, distance, path, results);

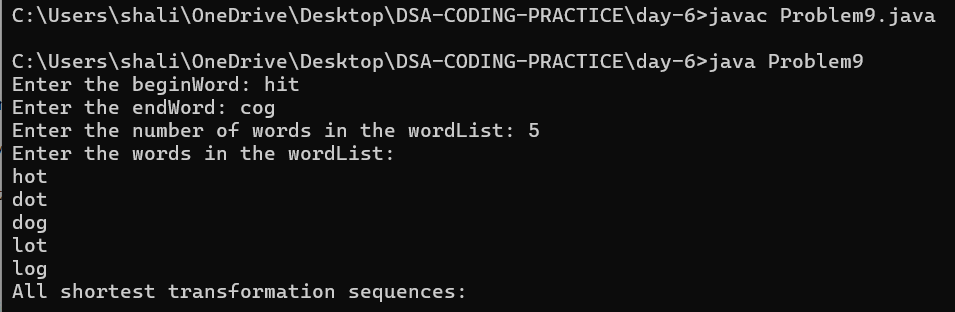
path.remove(path.size() - 1);

}

}

}

}



**10.Course schedule**

import java.util.\*;

public class Problem10 {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter the number of courses: ");

int numCourses = sc.nextInt();

System.out.print("Enter the number of prerequisites: ");

int n = sc.nextInt();

List<int[]> prerequisites = new ArrayList<>();

System.out.println("Enter the prerequisites pairs (course, prerequisite): ");

for (int i = 0; i < n; i++) {

int a = sc.nextInt();

int b = sc.nextInt();

prerequisites.add(new int[]{a, b});

}

boolean result = canFinish(numCourses, prerequisites);

System.out.println("Can finish all courses: " + result);

}

public static boolean canFinish(int numCourses, List<int[]> prerequisites) {

Map<Integer, List<Integer>> graph = new HashMap<>();

int[] indegree = new int[numCourses];

// Build the graph and indegree array

for (int[] prerequisite : prerequisites) {

int course = prerequisite[0];

int pre = prerequisite[1];

graph.putIfAbsent(pre, new ArrayList<>());

graph.get(pre).add(course);

indegree[course]++;

}

// Topological Sort using BFS (Kahn's Algorithm)

Queue<Integer> queue = new LinkedList<>();

for (int i = 0; i < numCourses; i++) {

if (indegree[i] == 0) {

queue.add(i);

}

}

int coursesTaken = 0;

while (!queue.isEmpty()) {

int currentCourse = queue.poll();

coursesTaken++;

if (graph.containsKey(currentCourse)) {

for (int nextCourse : graph.get(currentCourse)) {

indegree[nextCourse]--;

if (indegree[nextCourse] == 0) {

queue.add(nextCourse);

}

}

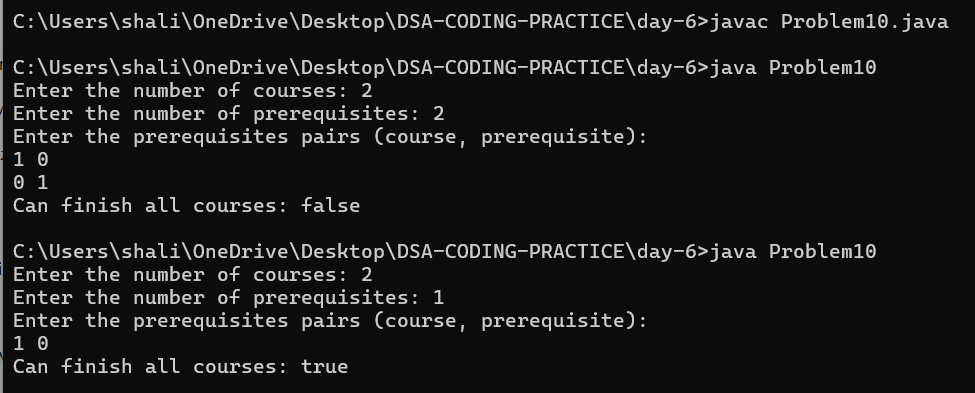
}

}

return coursesTaken == numCourses;

}

}



**11. tic tac toe**

import java.util.Scanner;

class Problem11 {

public boolean validTicTacToe(String[] board) {

int xCount = 0, oCount = 0;

for (String row : board) {

for (char c : row.toCharArray()) {

if (c == 'X') xCount++;

else if (c == 'O') oCount++;

}

}

if (oCount > xCount || xCount > oCount + 1) return false;

if (isWinner(board, 'X') && xCount != oCount + 1) return false;

if (isWinner(board, 'O') && xCount != oCount) return false;

return true;

}

private boolean isWinner(String[] board, char player) {

for (int i = 0; i < 3; i++) {

if (board[i].charAt(0) == player && board[i].charAt(1) == player && board[i].charAt(2) == player) return true;

if (board[0].charAt(i) == player && board[1].charAt(i) == player && board[2].charAt(i) == player) return true;

}

if (board[0].charAt(0) == player && board[1].charAt(1) == player && board[2].charAt(2) == player) return true;

if (board[0].charAt(2) == player && board[1].charAt(1) == player && board[2].charAt(0) == player) return true;

return false;

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

Problem11 problem = new Problem11();

String[] board = new String[3];

System.out.println("Enter the Tic-Tac-Toe board:");

for (int i = 0; i < 3; i++) {

board[i] = scanner.nextLine();

}

boolean result = problem.validTicTacToe(board);

System.out.println(result);

scanner.close();

}

}