**CODING PRACTICE PROBLEMS**

1. next permutation

class Solution {

public void nextPermutation(int[] nums) {

int ind1=-1;

int ind2=-1;

// step 1 find breaking point

for(int i=nums.length-2;i>=0;i--){

if(nums[i]<nums[i+1]){

ind1=i;

break;

}

}

// if there is no breaking point

if(ind1==-1){

reverse(nums,0);

}

else{

// step 2 find next greater element and swap with ind2

for(int i=nums.length-1;i>=0;i--){

if(nums[i]>nums[ind1]){

ind2=i;

break;

}

}

swap(nums,ind1,ind2);

// step 3 reverse the rest right half

reverse(nums,ind1+1);

}

}

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

int temp=nums[i];

nums[i]=nums[j];

nums[j]=temp;

}

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

int i=start;

int j=nums.length-1;

while(i<j){

swap(nums,i,j);

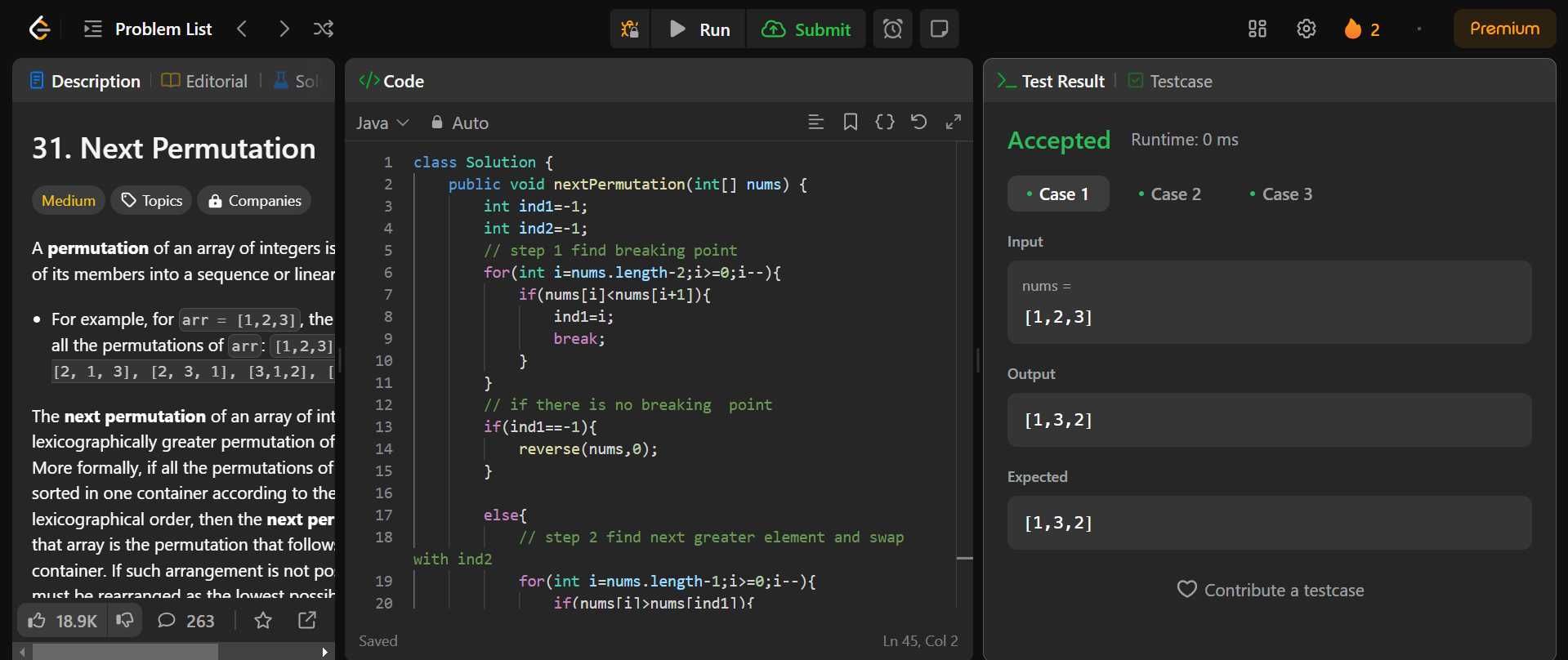
i++;

j--;

}

}

}



1. Spiral matrix

class Solution {

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

        int rows = matrix.length;

        int cols = matrix[0].length;

        int x = 0;

        int y = 0;

        int dx = 1;

        int dy = 0;

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

        for (int i = 0; i < rows \* cols; i++) {

            res.add(matrix[y][x]);

            matrix[y][x] = -101; // the range of numbers in matrix is from -100 to 100

            if (!(0 <= x + dx && x + dx < cols && 0 <= y + dy && y + dy < rows) || matrix[y+dy][x+dx] == -101) {

                int temp = dx;

                dx = -dy;

                dy = temp;

            }

            x += dx;

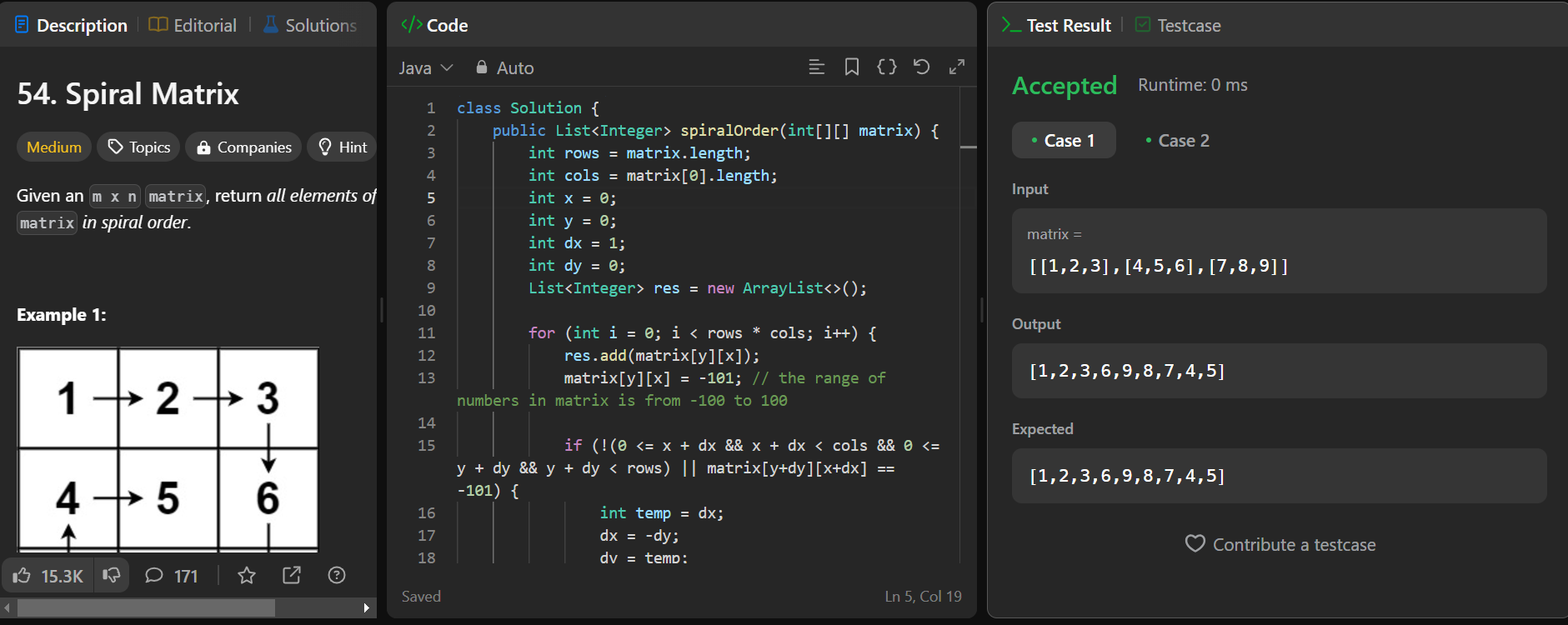
            y += dy;

        }

        return res;

    }

}



1. Longest substring without repeating characters

class Solution {

public int lengthOfLongestSubstring(String s) {

int n = s.length();

int maxLength = 0;

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

int left = 0;

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

if (!charSet.contains(s.charAt(right))) {

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

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

} else {

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

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

left++;

}

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

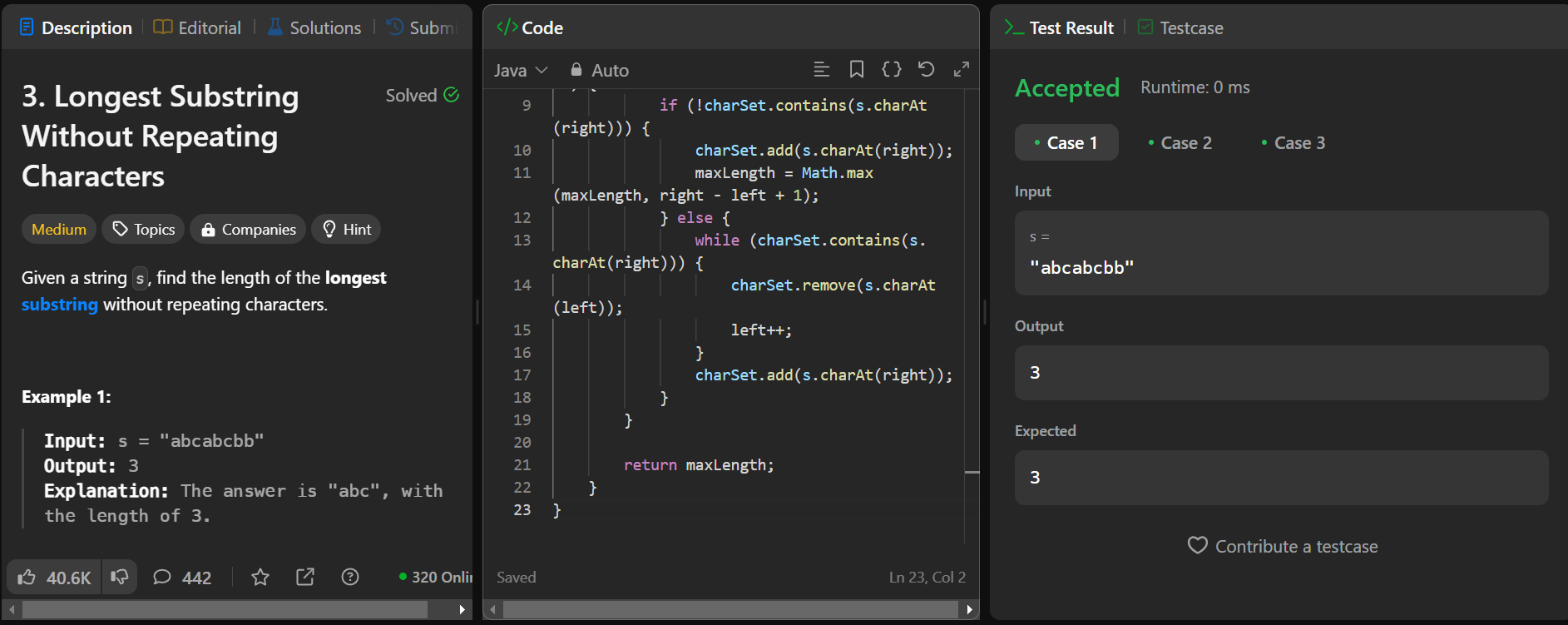
}

}

return maxLength;

}

}



1. Remove linked list elements

class Solution {

public ListNode removeElements(ListNode head, int val) {

ListNode ans = new ListNode(0, head);

ListNode dummy = ans;

while (dummy != null) {

while (dummy.next != null && dummy.next.val == val) {

dummy.next = dummy.next.next;

}

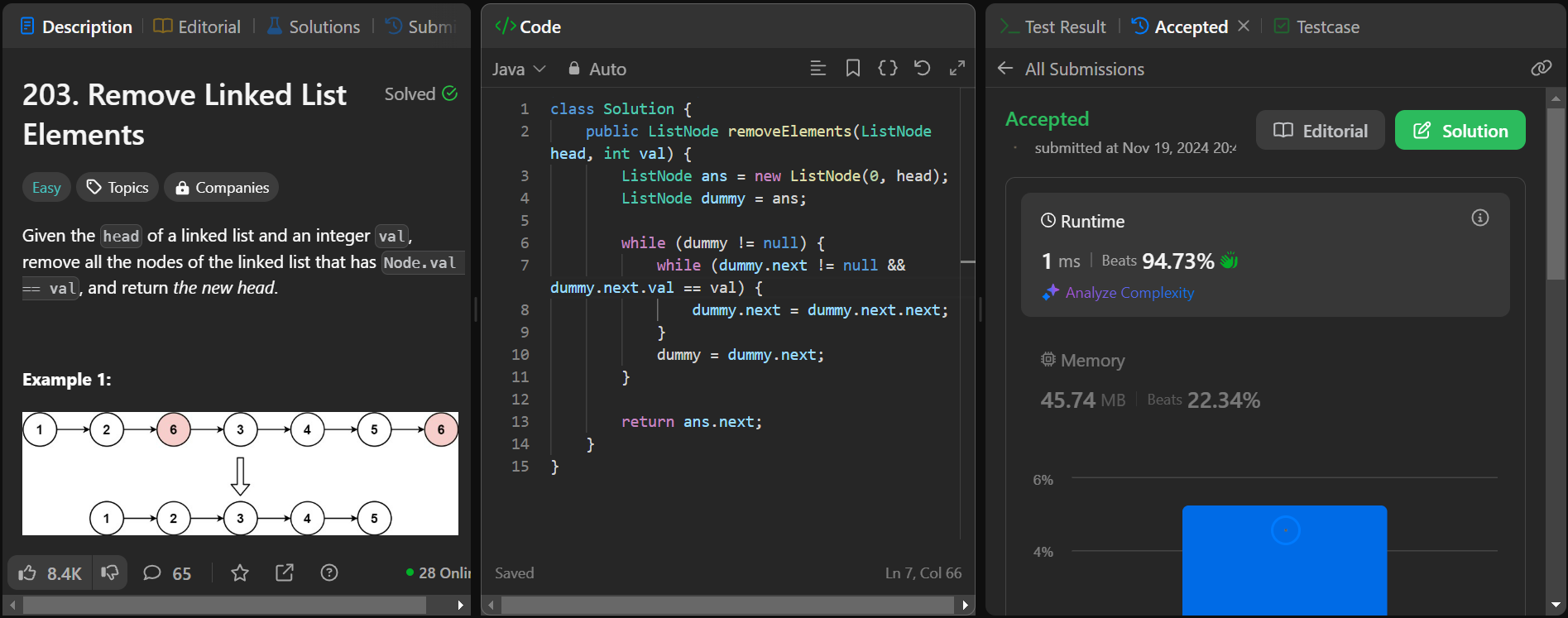
dummy = dummy.next;

}

return ans.next;

}

}



1. Palindrome linked list

class Solution {

public boolean isPalindrome(ListNode head) {

ListNode slow = head, fast = head, prev, temp;

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

slow = slow.next;

fast = fast.next.next;

}

prev = slow;

slow = slow.next;

prev.next = null;

while (slow != null) {

temp = slow.next;

slow.next = prev;

prev = slow;

slow = temp;

}

fast = head;

slow = prev;

while (slow != null) {

if (fast.val != slow.val) return false;

fast = fast.next;

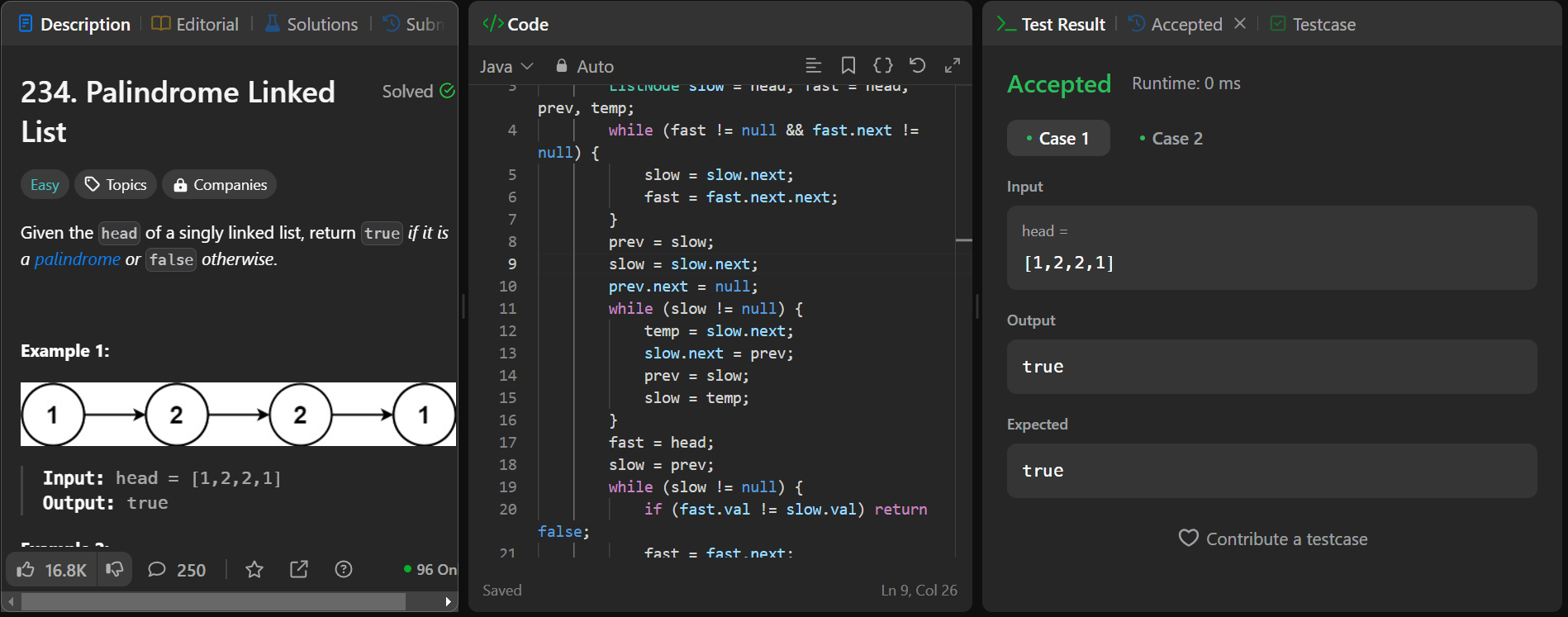
slow = slow.next;

}

return true;

}

}



1. Minimum path sum

class Solution {

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

int m = grid.length;

int 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]);

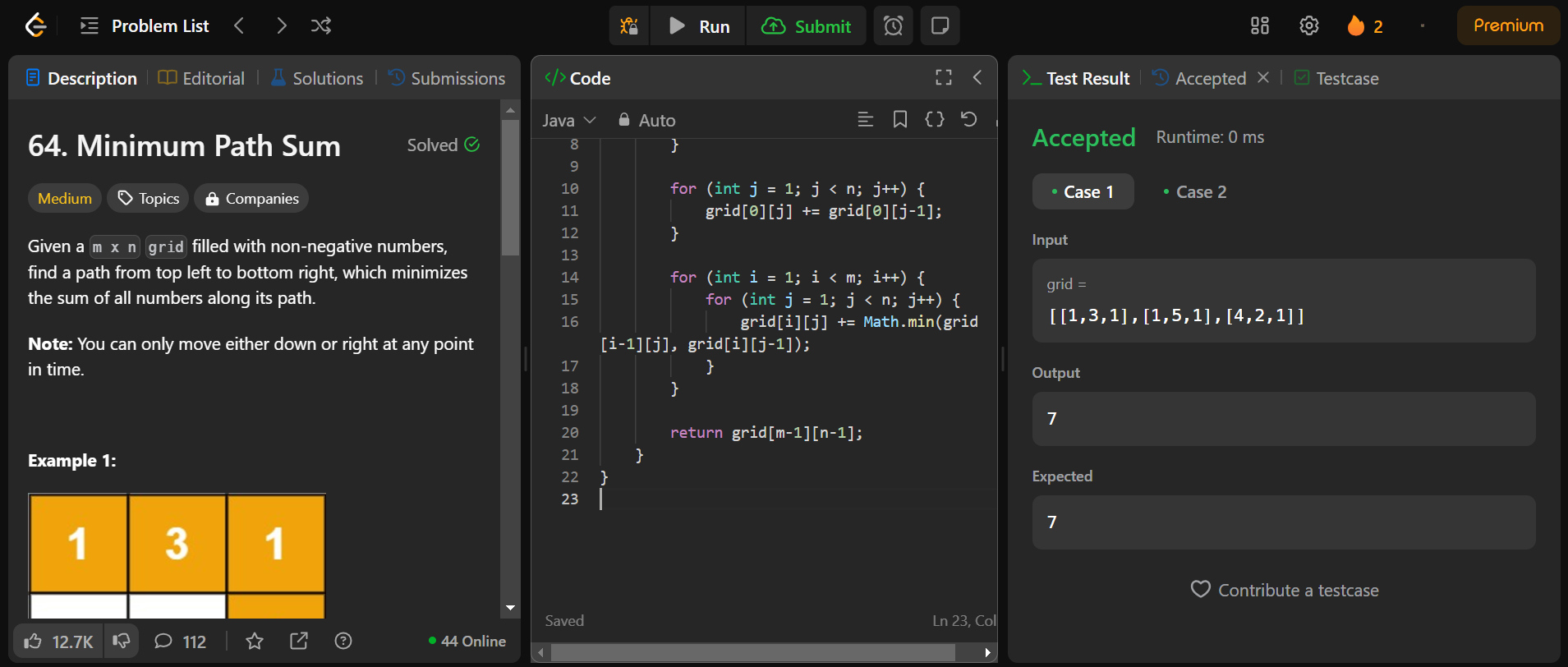
}

}

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

}

}



1. Validate binary search tree

public class Solution {

public boolean isValidBST(TreeNode root) {

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

}

public boolean isValidBST(TreeNode root, long minVal, long maxVal) {

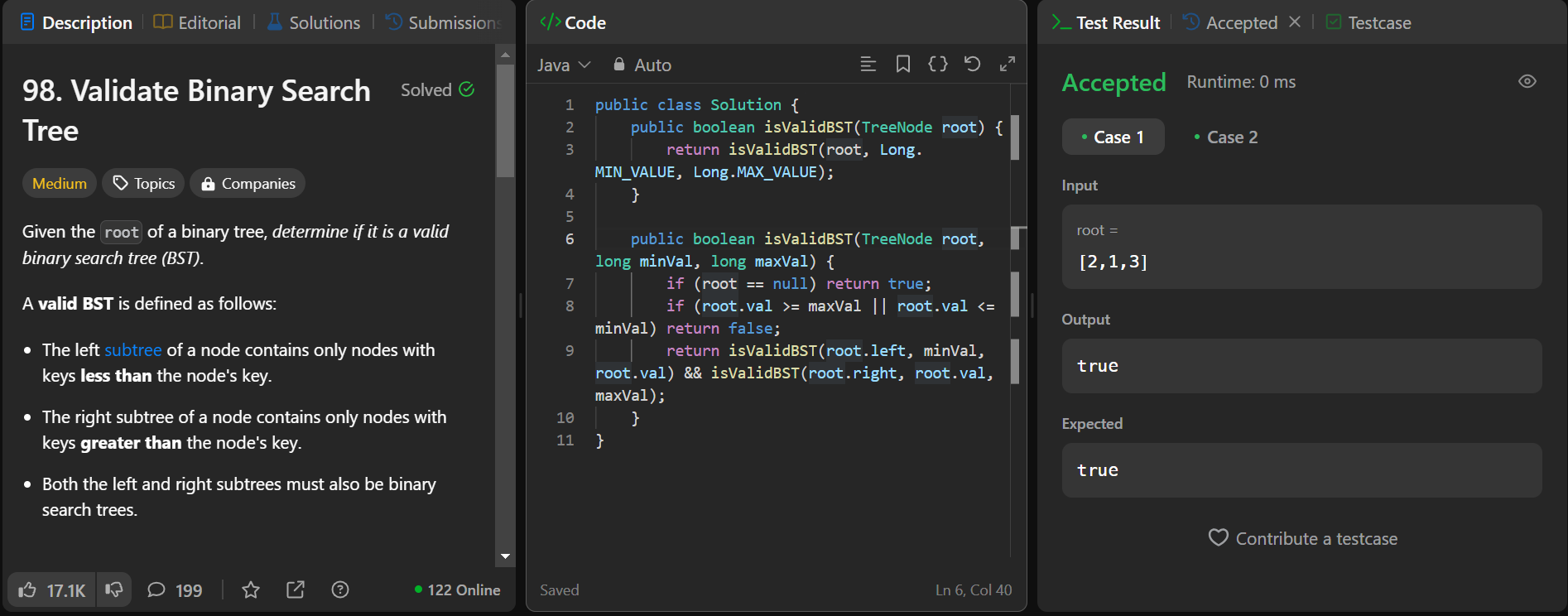
if (root == null) return true;

if (root.val >= maxVal || root.val <= minVal) return false;

return isValidBST(root.left, minVal, root.val) && isValidBST(root.right, root.val, maxVal);

}

}



8 . Word ladder

class Solution {

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

// Create a set of all words in the word list for quick lookup.

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

// Flag to check if the endWord is present in the word list.

Boolean isPresent = false;

// Add all words from the word list to the set.

wordSet.addAll(wordList);

// Check if the endWord is in the word list.

for (String currWord : wordList) {

if (endWord.equals(currWord)) {

isPresent = true;

break; // If found, break out of the loop.

}

}

// If endWord is not in the list, return 0 as there's no valid transformation.

if (!isPresent) return 0;

// Use a queue to perform BFS (Breadth-First Search).

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

// Start BFS with the beginWord.

wordQueue.add(beginWord);

// Distance from the beginWord (initially 0).

int distance = 0;

// BFS loop: continue until the queue is empty.

while (!wordQueue.isEmpty()) {

int size = wordQueue.size();

distance++; // Increment distance at each level of BFS.

// Process each word in the current level.

while (size-- != 0) {

String currWord = wordQueue.poll(); // Get the front word from the queue.

// Try changing each character in the current word.

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

char[] temp = currWord.toCharArray(); // Convert the word to a character array.

// Replace the character at index i with every letter from 'a' to 'z'.

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

temp[i] = j;

String newWord = new String(temp); // Form a new word.

// If the new word matches the endWord, return the distance + 1.

if (newWord.equals(endWord)) return distance + 1;

// If the new word is in the word set, it's a valid transformation.

if (wordSet.contains(newWord)) {

wordQueue.add(newWord); // Add it to the queue for further exploration.

wordSet.remove(newWord); // Remove it from the set to avoid revisiting.

// Print the new word being added to the queue (for debugging).

System.out.println(newWord);

}

}

}

}

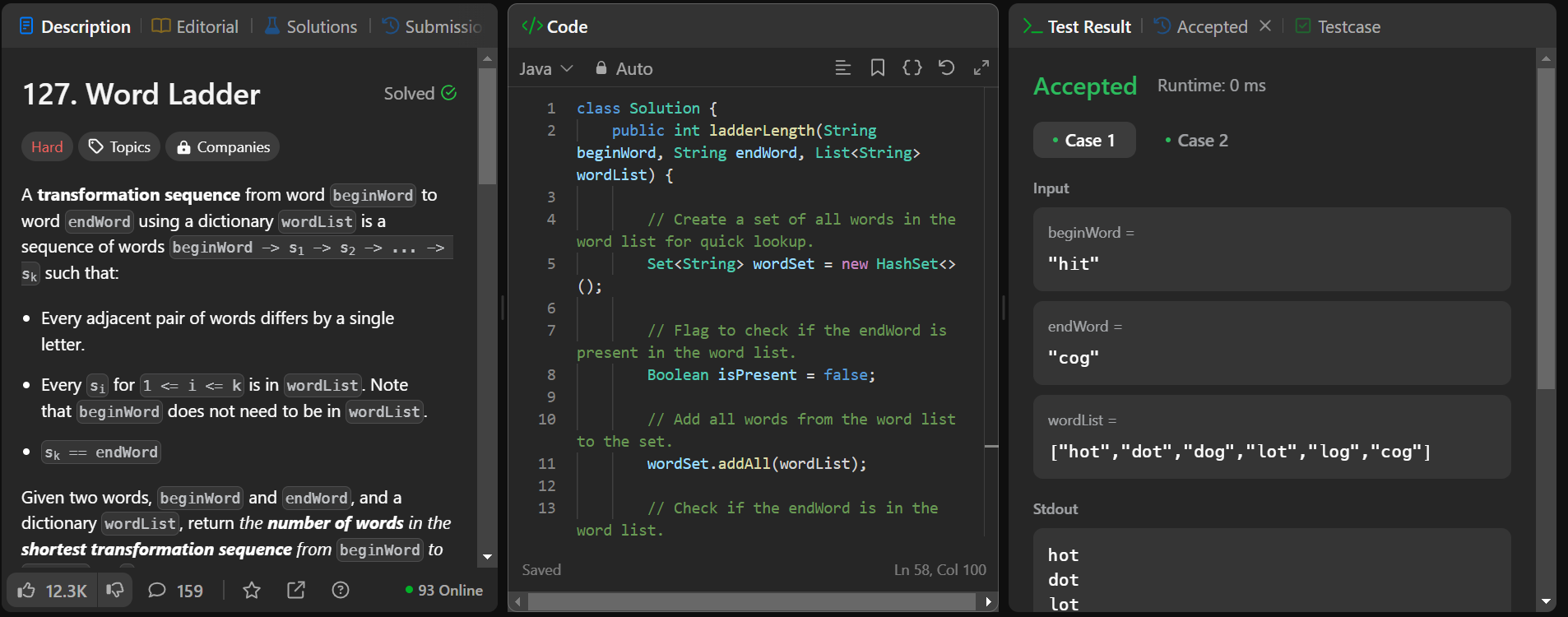
}

// If no transformation sequence leads to the endWord, return 0.

return 0;

}

}



9 . Word ladder -II

class Solution {

String b;

// Create a hashmap of type word->level to get the idea

// on which level the word comes after the transformations.

HashMap < String, Integer > mpp;

// A list for storing the final answer.

List < List < String >> ans;

private void dfs(String word, List < String > seq) {

// Function for implementing backtracking using the created map

// in reverse order to find the transformation sequence in less time.

// Base condition :

// If word equals beginWord, we’ve found one of the sequences

// simply reverse the sequence and return.

if (word.equals(b)) {

// Since java works with reference, create

// a duplicate and store the reverse of it

List < String > dup = new ArrayList < > (seq);

Collections.reverse(dup);

ans.add(dup);

return;

}

int steps = mpp.get(word);

int sz = word.length();

// Replace each character of the word with letters from a-z

// and check whether the transformed word is present in the map

// and at the previous level or not.

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

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

char replacedCharArray[] = word.toCharArray();

replacedCharArray[i] = ch;

String replacedWord = new String(replacedCharArray);

if (mpp.containsKey(replacedWord) &&

mpp.get(replacedWord) + 1 == steps) {

seq.add(replacedWord);

dfs(replacedWord, seq);

// pop the current word from the back of the queue

// to traverse other possibilities.

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

}

}

}

}

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

// Push all values of wordList into a set

// to make deletion from it easier and in less time complexity.

Set < String > st = new HashSet < String > ();

int len = wordList.size();

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

st.add(wordList.get(i));

}

// Perform BFS traversal and push the string in the queue

// as soon as they’re found in the wordList.

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

b = beginWord;

q.add(beginWord);

mpp = new HashMap < > ();

// beginWord initialised with level 1.

mpp.put(beginWord, 1);

int sizee = beginWord.length();

st.remove(beginWord);

while (!q.isEmpty()) {

String word = q.peek();

int steps = mpp.get(word);

q.remove();

// Break out if the word matches the endWord.

if (word.equals(endWord)) break;

// Replace each character of the word with letters from a-z

// and check whether the transformed word is present in the

// wordList or not, if yes then push to queue

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

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

char replacedCharArray[] = word.toCharArray();

replacedCharArray[i] = ch;

String replacedWord = new String(replacedCharArray);

if (st.contains(replacedWord) == true) {

q.add(replacedWord);

st.remove(replacedWord);

// push the word along with its level

// in the map data structure.

mpp.put(replacedWord, steps + 1);

}

}

}

}

ans = new ArrayList < > ();

// If we reach the endWord, we stop and move to step-2

// that is to perform reverse dfs traversal.

if (mpp.containsKey(endWord) == true) {

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

seq.add(endWord);

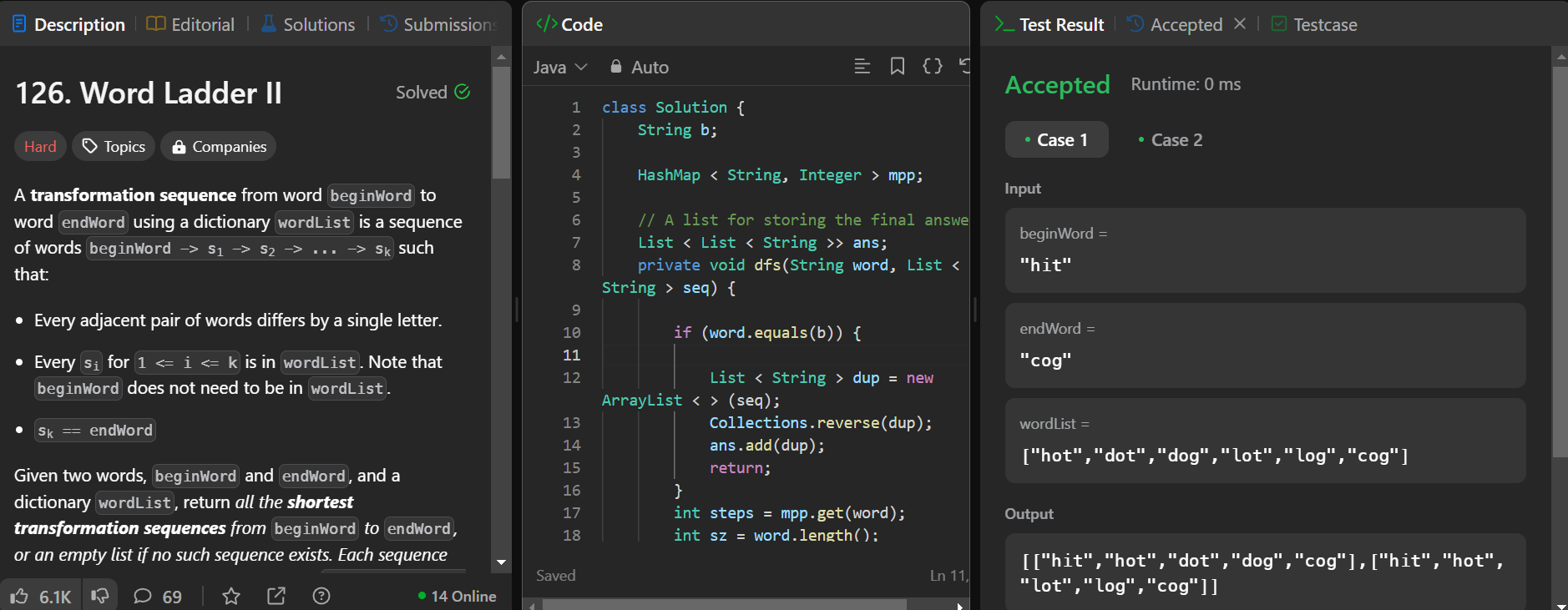
dfs(endWord, seq);

}

return ans;

}

}



10 . course schedule

class Solution {

public boolean canFinish(int n, int[][] prerequisites) {

List<Integer>[] adj = new List[n];

int[] indegree = new int[n];

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

for (int[] pair : prerequisites) {

int course = pair[0];

int prerequisite = pair[1];

if (adj[prerequisite] == null) {

adj[prerequisite] = new ArrayList<>();

}

adj[prerequisite].add(course);

indegree[course]++;

}

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

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

if (indegree[i] == 0) {

queue.offer(i);

}

}

while (!queue.isEmpty()) {

int current = queue.poll();

ans.add(current);

if (adj[current] != null) {

for (int next : adj[current]) {

indegree[next]--;

if (indegree[next] == 0) {

queue.offer(next);

}

}

}

}

return ans.size() == n;

}

}

