**NARESH M\_AI&DS\_DSA\_DAY-7**

**1.Next permutation**

A **permutation** of an array of integers is an arrangement of its members into a sequence or linear order.

* For example, for arr = [1,2,3], the following are all the permutations of arr: [1,2,3], [1,3,2], [2, 1, 3], [2, 3, 1], [3,1,2], [3,2,1].

The **next permutation** of an array of integers is the next lexicographically greater permutation of its integer. More formally, if all the permutations of the array are sorted in one container according to their lexicographical order, then the **next permutation** of that array is the permutation that follows it in the sorted container. If such arrangement is not possible, the array must be rearranged as the lowest possible order (i.e., sorted in ascending order).

* For example, the next permutation of arr = [1,2,3] is [1,3,2].
* Similarly, the next permutation of arr = [2,3,1] is [3,1,2].
* While the next permutation of arr = [3,2,1] is [1,2,3] because [3,2,1] does not have a lexicographical larger rearrangement.

Given an array of integers nums, *find the next permutation of* nums.

The replacement must be in place and use only constant extra memory.

**Example 1:**

**Input:** nums = [1,2,3]

**Output:** [1,3,2]

**Example 2:**

**Input:** nums = [3,2,1]

**Output:** [1,2,3]

**Example 3:**

**Input:** nums = [1,1,5]

**Output:** [1,5,1]

**Constraints:**

* 1 <= nums.length <= 100
* 0 <= nums[i] <= 100

Code:

class Solution {

    public void nextPermutation(int[] nums) {

        int n = nums.length;

        int i = n - 2;

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

            i--;

        }

        if (i >= 0) {

            int j = n - 1;

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

                j--;

            }

            swap(nums, i, j);

        }

        reverse(nums, i + 1, n - 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) {

        while (start < end) {

            swap(nums, start, end);

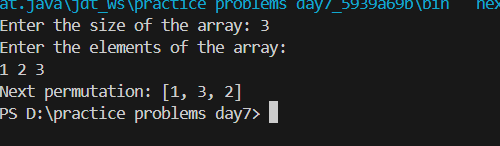
            start++;

            end--;

        }

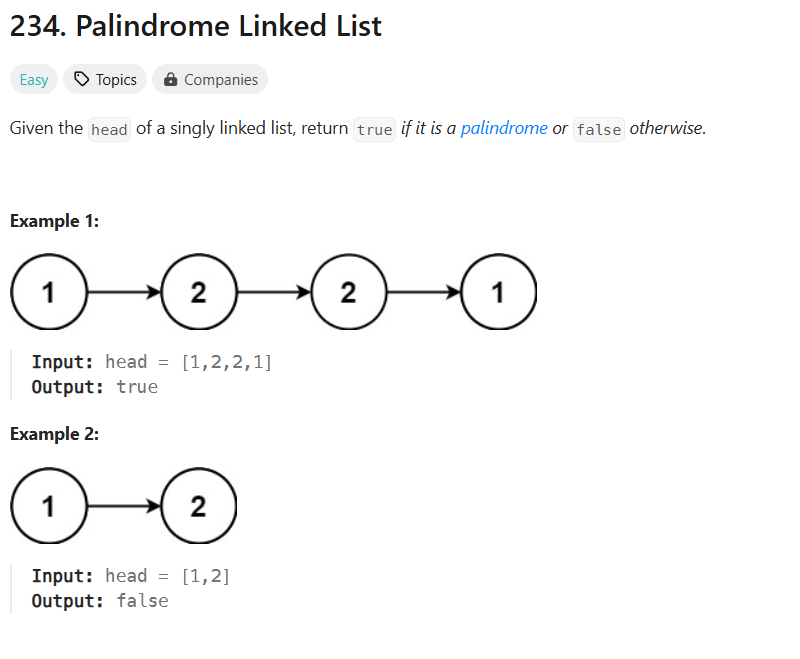
    }

}



**Time complexity: O(n)**

**2.**

****

**Code:**

import java.util.Scanner;

import java.util.Stack;

class ListNode {

    int val;

    ListNode next;

    ListNode(int val) {

        this.val = val;

        this.next = null;

    }

}

class Solution {

    public boolean isPalindrome(ListNode head) {

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

        ListNode curr = head;

        while (curr != null) {

            stack.push(curr.val);

            curr = curr.next;

        }

        curr = head;

        while (curr != null && curr.val == stack.pop()) {

            curr = curr.next;

        }

        return curr == null;

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

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

        int n = sc.nextInt();

        if (n <= 0) {

            System.out.println("The linked list is empty.");

            System.out.println("Output: true");

            return;

        }

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

        ListNode head = new ListNode(sc.nextInt());

        ListNode current = head;

        for (int i = 1; i < n; i++) {

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

            current = current.next;

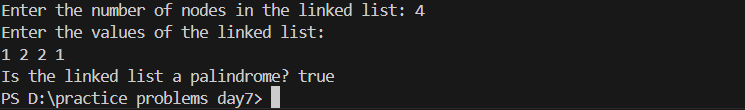
        }

        Solution solution = new Solution();

        System.out.println("Is the linked list a palindrome? " + result);

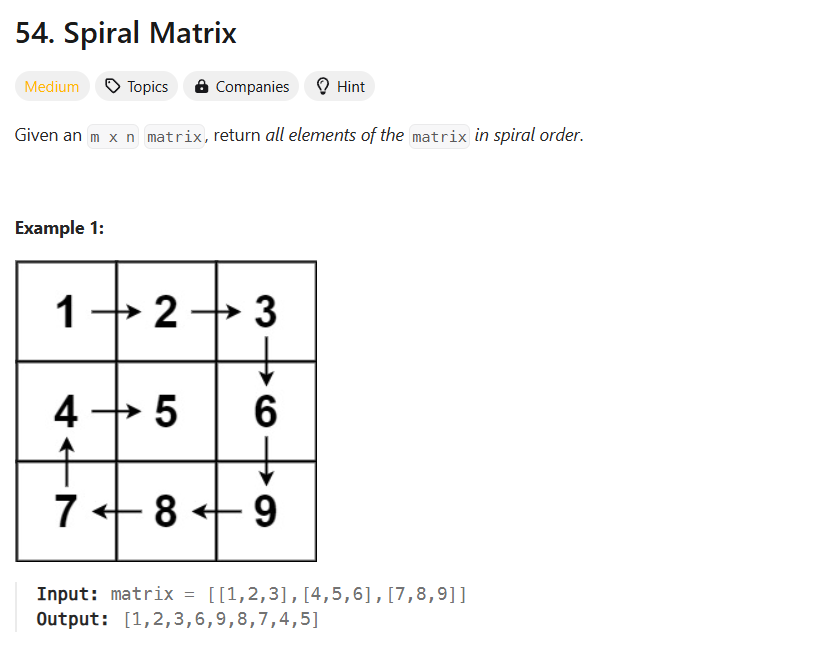
    }

}

****

**Time complexity: O(n)**

**3.**

****

**Code:**

import java.util.ArrayList;

import java.util.List;

import java.util.Scanner;

class spiralMatrix {

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

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

        int n = matrix.length;

        int m = matrix[0].length;

        int top = 0, left = 0, bottom = n - 1, right = m - 1;

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

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

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

            }

            top++;

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

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

            }

            right--;

            if (top <= bottom) {

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

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

                }

                bottom--;

            }

            if (left <= right) {

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

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

                }

                left++;

            }

        }

        return ans;

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

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

        int rows = sc.nextInt();

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

        int cols = sc.nextInt();

        int[][] matrix = new int[rows][cols];

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

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

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

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

            }

        }

        spiralMatrix solution = new spiralMatrix();

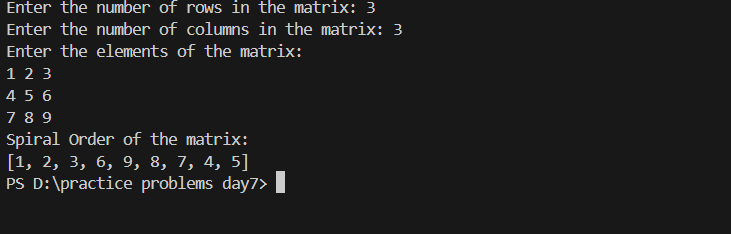
        List<Integer> result = solution.spiralOrder(matrix);

        System.out.println("Spiral Order of the matrix:");

        System.out.println(result);

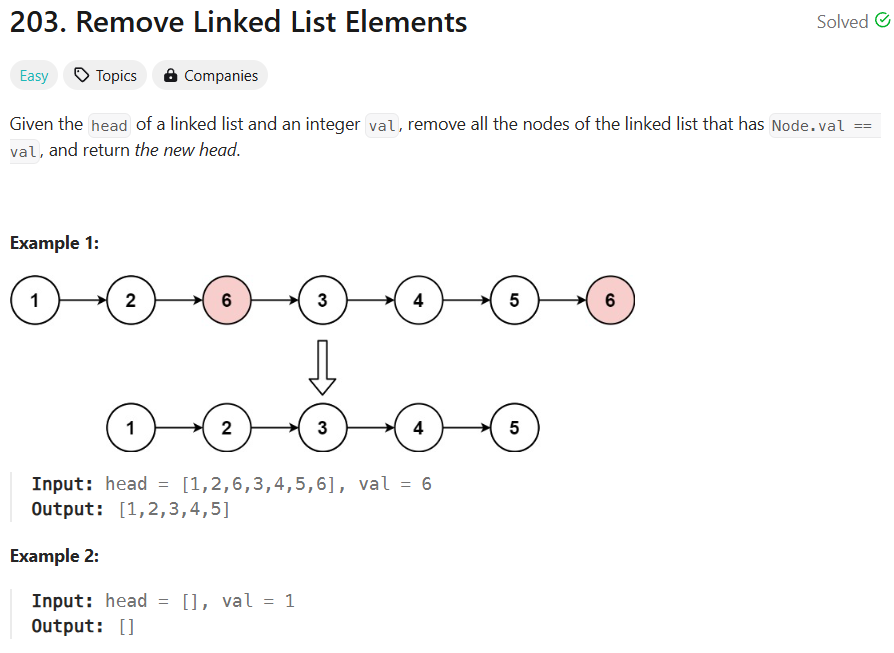
    }

}

****

**Time complexity : O(mxn)**

**4.**

****

**Code:**

class ListNode {

    int val;

    ListNode next;

    ListNode(int val) {

        this.val = val;

        this.next = null;

    }

    ListNode(int val, ListNode next) {

        this.val = val;

        this.next = next;

    }

}

class removeLinkedListElements {

    public ListNode removeElements(ListNode head, int val) {

        // Create a dummy node to handle edge cases

        ListNode dummy = new ListNode(0);

        dummy.next = head;

        ListNode current = dummy;

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

            // If the next node has the target value, remove it

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

                current.next = current.next.next;

            } else {

                // Otherwise, move to the next node

                current = current.next;

            }

        }

        return dummy.next; // Return the new head (dummy.next is the real head)

    }

    public static 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) {

        // Create linked list 1 -> 2 -> 6 -> 3 -> 4 -> 5 -> 6

        ListNode head = new ListNode(1);

        head.next = new ListNode(2);

        head.next.next = new ListNode(6);

        head.next.next.next = new ListNode(3);

        head.next.next.next.next = new ListNode(4);

        head.next.next.next.next.next = new ListNode(5);

        head.next.next.next.next.next.next = new ListNode(6);

        removeLinkedListElements solution = new removeLinkedListElements();

        ListNode newHead = solution.removeElements(head, 6);

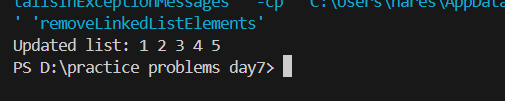
        System.out.print("Updated list: ");

        printList(newHead);

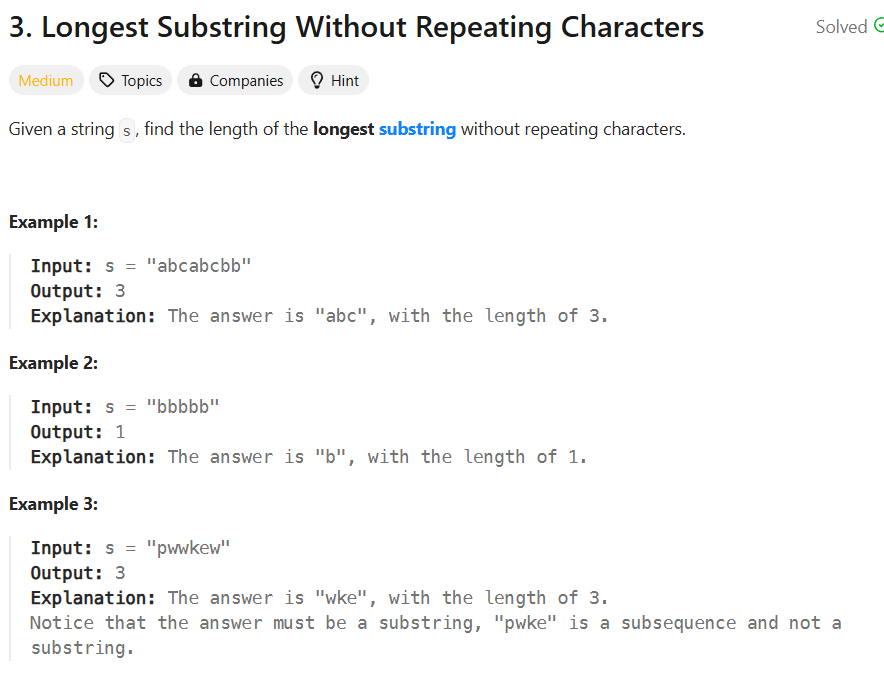
    }

}

**Time Complexity: O(n)**



**5.**

****

**Code:**

import java.util.HashSet;

class longestSubstringWithoutRepeating {

    public int lengthOfLongestSubstring(String s) {

        int n = s.length();

        int l = 0, r = 0;

        int max = 0;

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

        while (r < n) {

            if (!set.contains(s.charAt(r))) {

                set.add(s.charAt(r));

                r++;

                max = Math.max(max, r - l);

            } else {

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

                l++;

            }

        }

        return max;

    }

    public static void main(String[] args) {

        longestSubstringWithoutRepeating solution = new longestSubstringWithoutRepeating();

        java.util.Scanner scanner = new java.util.Scanner(System.in);

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

        String s = scanner.nextLine();

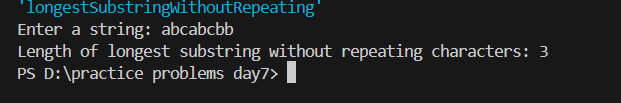
        int result = solution.lengthOfLongestSubstring(s);

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

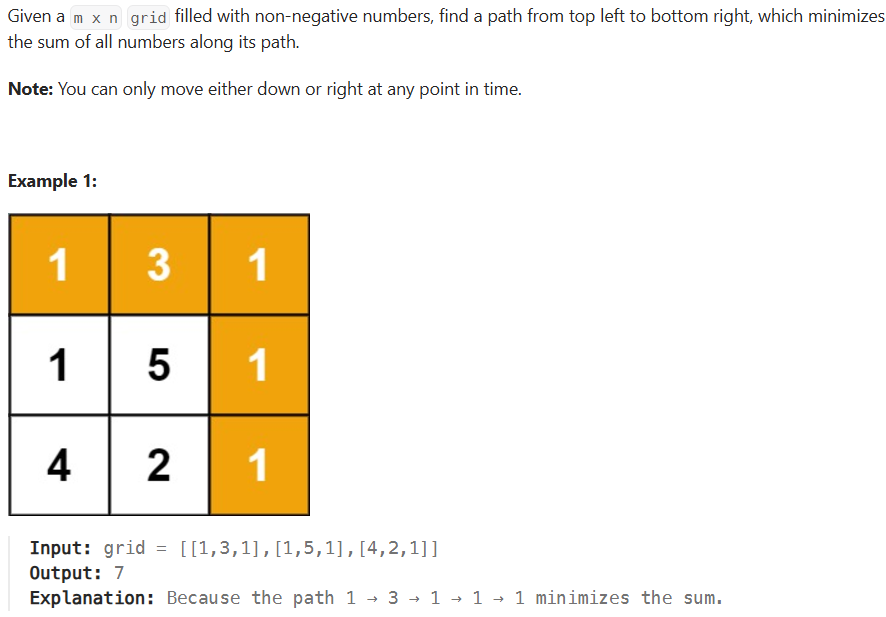
        scanner.close();

    }

}

**Time complexity: O(n)**

**6.Minimum path sum**



Code:

import java.util.Scanner;

class minPathSum{

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

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

        int n = sc.nextInt();

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

        int m = sc.nextInt();

        int[][] grid = new int[n][m];

        System.out.println("Enter the grid values row by row:");

        for (int i = 0; i < n; i++) {

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

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

            }

        }

        minPathSum obj = new minPathSum();

        int result = obj.minPathSum(grid);

        System.out.println("Minimum Path Sum: " + result);

        sc.close();

    }

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

        int n = grid.length;

        int m = grid[0].length;

        for (int i = 1; i < n; i++) {

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

        }

        for (int j = 1; j < m; j++) {

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

        }

        for (int i = 1; i < n; i++) {

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

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

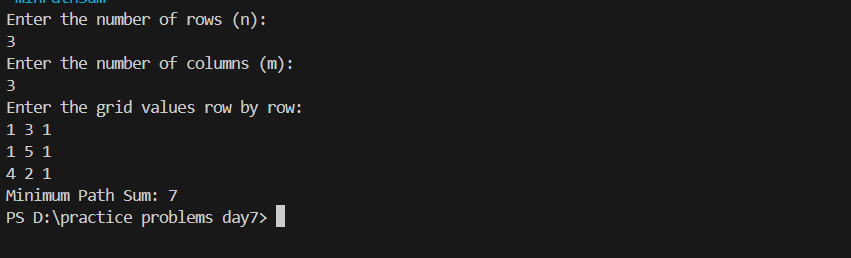
            }

        }

        return grid[n - 1][m - 1];

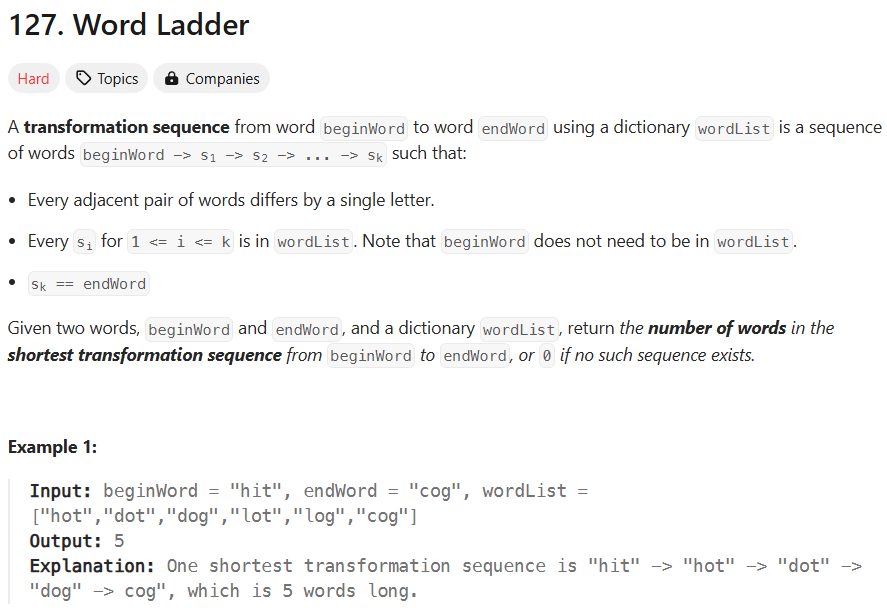
    }

}



Time complexity : O(nxm)

**7. Word ladder:**

****

**Code:**

import java.util.\*;

class Solution {

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

        // Add all words to a set for quick lookup

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

        if (!wordSet.contains(endWord)) {

            return 0; // If endWord is not in the list, no transformation is possible

        }

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

        queue.offer(new Pair(beginWord, 1)); // Start with beginWord and step count of 1

        while (!queue.isEmpty()) {

            Pair current = queue.poll();

            String word = current.word;

            int steps = current.steps;

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

                char[] charArray = word.toCharArray();

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

                    charArray[i] = c;

                    String nextWord = new String(charArray);

                    // If the transformed word is endWord, return the steps

                    if (nextWord.equals(endWord)) {

                        return steps + 1;

                    }

                    // If the transformed word is in the wordSet, add it to the queue

                    if (wordSet.contains(nextWord)) {

                        queue.offer(new Pair(nextWord, steps + 1));

                        wordSet.remove(nextWord); // Remove from set to mark as visited

                    }

                }

            }

        }

        return 0;

    }

}

class Pair {

    String word;

    int steps;

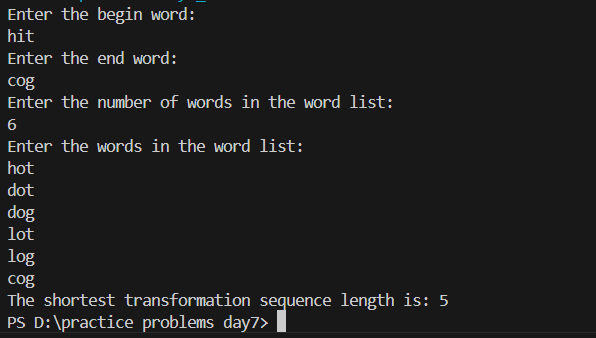
    Pair(String word, int steps) {

        this.word = word;

        this.steps = steps;

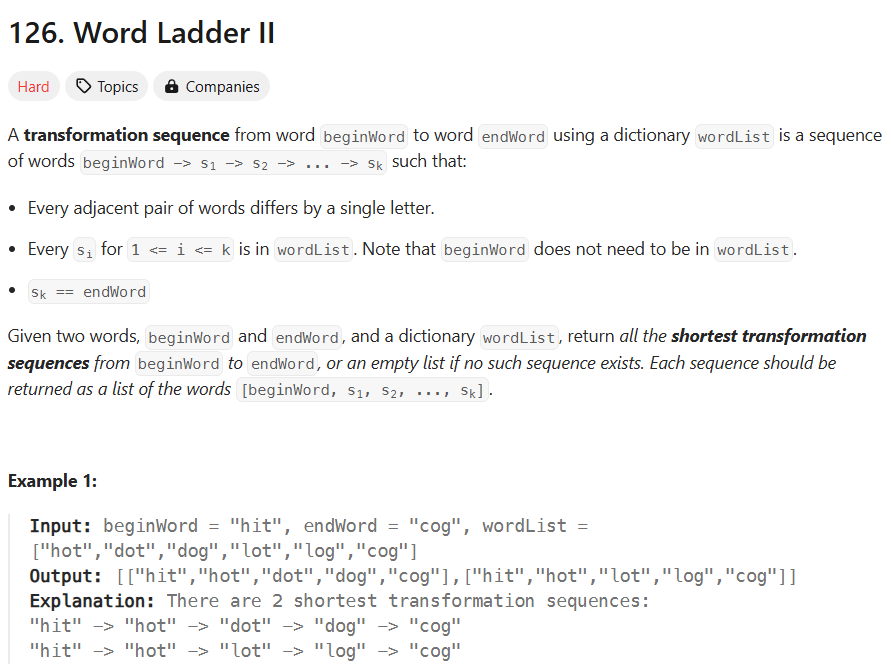
    }

}



Time Complexity: O(NxL)

8.Word Ladder II



Code:

import java.util.\*;

class wordLadderII {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

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

        String beginWord = sc.nextLine();

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

        String endWord = sc.nextLine();

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

        int n = sc.nextInt();

        sc.nextLine();

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

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

        for (int i = 0; i < n; i++) {

            wordList.add(sc.nextLine());

        }

        wordLadderII obj = new wordLadderII();

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

        if (result.isEmpty()) {

            System.out.println("No transformation sequence exists.");

        } else {

            System.out.println("Shortest transformation sequences:");

            for (List<String> seq : result) {

                System.out.println(seq);

            }

        }

        sc.close();

    }

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

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

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

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

        q.add(beginWord);

        hm.put(beginWord, 1);

        HashSet<String> hs = new HashSet<>(wordList);

        hs.remove(beginWord);

        while (!q.isEmpty()) {

            String word = q.poll();

            if (word.equals(endWord)) {

                break;

            }

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

                int level = hm.get(word);

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

                    char[] replaceChars = word.toCharArray();

                    replaceChars[i] = ch;

                    String replaceString = new String(replaceChars);

                    if (hs.contains(replaceString)) {

                        q.add(replaceString);

                        hm.put(replaceString, level + 1);

                        hs.remove(replaceString);

                    }

                }

            }

        }

        if (hm.containsKey(endWord)) {

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

            seq.add(endWord);

            dfs(endWord, seq, res, beginWord, hm);

        }

        return res;

    }

    public void dfs(String word, List<String> seq, List<List<String>> res, String beginWord, Map<String, Integer> hm) {

        if (word.equals(beginWord)) {

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

            Collections.reverse(ref);

            res.add(ref);

            return;

        }

        int level = hm.get(word);

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

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

                char[] replaceChars = word.toCharArray();

                replaceChars[i] = ch;

                String replaceStr = new String(replaceChars);

                if (hm.containsKey(replaceStr) && hm.get(replaceStr) == level - 1) {

                    seq.add(replaceStr);

                    dfs(replaceStr, seq, res, beginWord, hm);

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

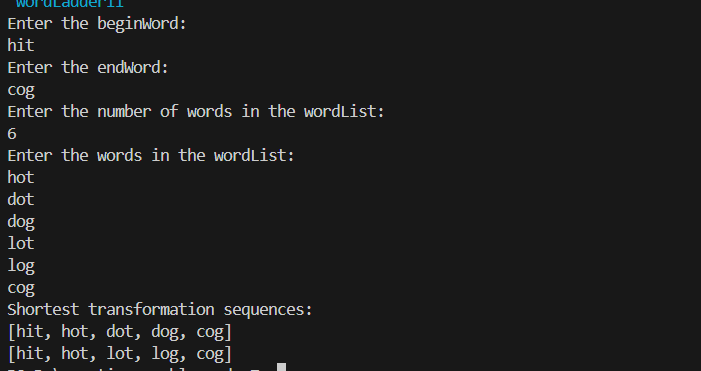
                }

            }

        }

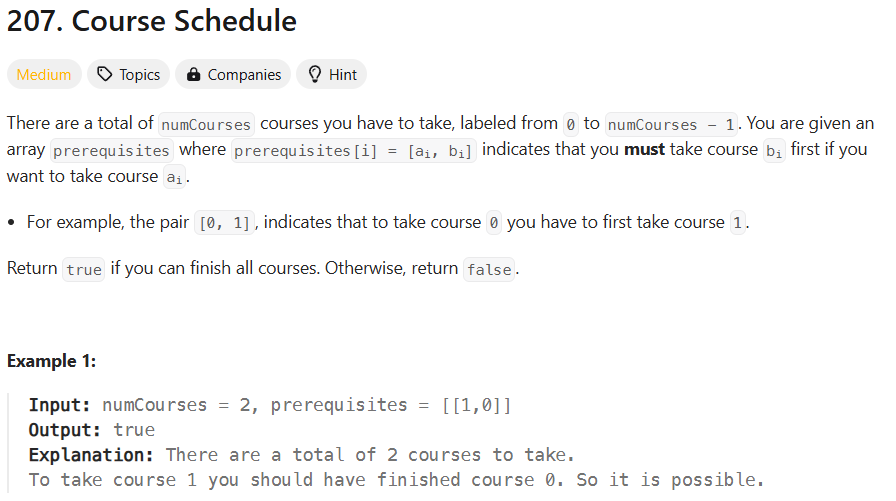
    }

}



Time complexity: O(NxL)

9.Course schedule:



**Code:**

class Solution {

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

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

        int[] inDegree = new int[numCourses];

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

            graph.add(new ArrayList<>());

        }

        for(int[] prereq: prerequisites){

            int course = prereq[0];

            int pre = prereq[1];

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

            inDegree[course]++;

        }

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

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

            if(inDegree[i] == 0){

                queue.offer(i);

            }

        }

        int processedCourses = 0;

        while(!queue.isEmpty()){

            int current = queue.poll();

            processedCourses++;

            for(int neighbor: graph.get(current)) {

                inDegree[neighbor]--;

                if(inDegree[neighbor] == 0){

                    queue.offer(neighbor);

                }

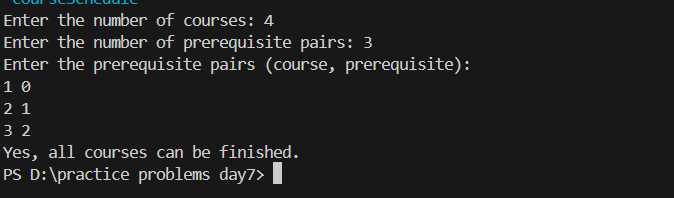
            }

        }

        return processedCourses == numCourses;

    }

}



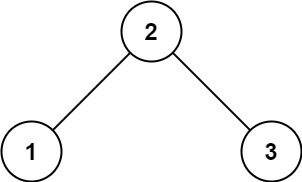
Time complexity: O(V+E)

**10. Validate Binary Search tree**

Given the root of a binary tree, determine if it is a valid binary search tree (BST).

A valid BST is defined as follows

* the left sub tree of a node contains only nodes with keys less than the node's key.
* The right subtree of a node contains only nodes with keys greater than the node's key.
* Both the left and right subtrees must also be binary search trees.



Input: root = [2,1,3]

Output: true

**Code:**

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode() {}

TreeNode(int val) { this.val = val; }

TreeNode(int val, TreeNode left, TreeNode right) {

this.val = val;

this.left = left;

this.right = right;

}

}

public class Main {

public static void main(String[] args) {

Solution solution = new Solution();

// Example 1: A valid BST

TreeNode validBST = new TreeNode(2, new TreeNode(1), new TreeNode(3));

System.out.println("Is the tree a valid BST? " + solution.isValidBST(validBST));

// Example 2: An invalid BST

TreeNode invalidBST = new TreeNode(5, new TreeNode(1), new TreeNode(4, new TreeNode(3), new TreeNode(6)));

System.out.println("Is the tree a valid BST? " + solution.isValidBST(invalidBST));

}

}

class Solution {

public boolean isValidBST(TreeNode root) {

if (root == null) return true;

return dfs(root, null, null);

}

private boolean dfs(TreeNode root, Integer min, Integer max) {

if (root == null) return true;

// Validate the current node

if ((min != null && root.val <= min) || (max != null && root.val >= max)) {

return false;

}

// Recursively validate the left and right subtrees

return dfs(root.left, min, root.val) && dfs(root.right, root.val, max);

}

}

