**DSA Practice Problems**

**Set – 7**

1. **Next Permutaion:**

Given an array of integers **arr[]** representing a permutation, implement the **next permutation** that rearranges the numbers into the lexicographically next greater permutation. If no such permutation exists, rearrange the numbers into the lowest possible order (i.e., sorted in ascending order).

Note - A permutation of an array of integers refers to a specific arrangement of its elements in a sequence or linear order.

**Input:** arr = [2, 4, 1, 7, 5, 0]

**Output:** [2, 4, 5, 0, 1, 7]

**Explanation:** The next permutation of the given array is {2, 4, 5, 0, 1, 7}.

**Code:**

class Solution {

void nextPermutation(int[] arr) {

// code here

int pivot=-1;

int n=arr.length;

for(int i=n-2;i>=0;i--){

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

pivot = i;

break;

}

}

if(pivot==-1){

reverse(arr,0,n-1);

return;

}

for(int i=n-1;i>pivot;i--){

if (arr[i]>arr[pivot]){

swap(arr,i,pivot);

break;

}

}

reverse(arr,pivot+1,n-1);

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

}

}

public static void reverse(int[] arr, int start, int end) {

while (start < end) {

swap(arr, start++, end--);

}

}

public static void swap(int[] arr, int i, int j) {

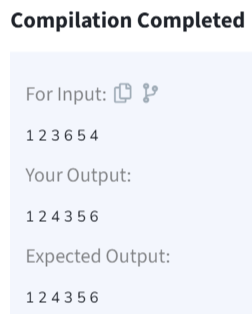
int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}



**Time Complexity: O(n)**

1. **Spiral Matrix:**

Given an m x n matrix, return *all elements of the* matrix *in spiral order*.

**Input:** matrix = [[1,2,3],[4,5,6],[7,8,9]]

**Output:** [1,2,3,6,9,8,7,4,5]

**Code:**

class Solution {

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

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

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

            return res;

        int startRow = 0;

        int endRow = matrix.length - 1;

        int startCol = 0;

        int endCol = matrix[0].length - 1;

        while (startRow <= endRow && startCol <= endCol) {

            // top

            for (int top = startCol; top <= endCol; top++) {

                res.add(matrix[startRow][top]);

            }

            // right

            for (int right = startRow + 1; right <= endRow; right++) {

                res.add(matrix[right][endCol]);

            }

            // bottom

            for (int bottom = endCol - 1; bottom >= startCol; bottom--) {

                if (startRow == endRow) {

                    break;

                }

                res.add(matrix[endRow][bottom]);

            }

            // left

            for (int left = endRow - 1; left >= startRow + 1; left--) {

                if (startCol == endCol) {

                    break;

                }

                res.add(matrix[left][startRow]);

            }

            startRow++;

            endRow--;

            startCol++;

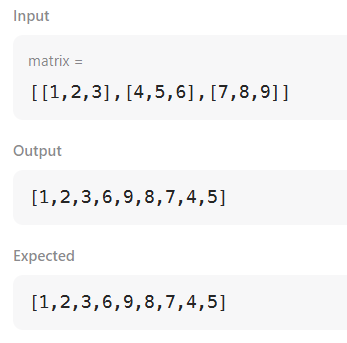
            endCol--;

        }

        return res;

    }

}



**Time Complexity: O(n\*m)**

1. **Longest Substring without repeated characters:**

Given a string **s**, find the length of the longest substring with all distinct characters.

**Input:** s = "geeksforgeeks"

**Output:** 7

**Explanation**: "eksforg" is the longest substring with all distinct characters.

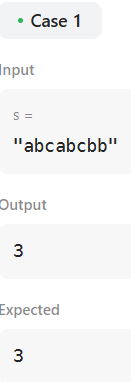
**Code:**

class Solution {

public int lengthOfLongestSubstring(String s) {

int maxLength = 0;

int left = 0;

 Map<Character, Integer> count = new HashMap<>();

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

char c = s.charAt(right);

count.put(c, count.getOrDefault(c, 0) + 1);

while (count.get(c) > 1) {

char leftChar = s.charAt(left);

count.put(leftChar, count.get(leftChar) - 1);

left++;

}

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

}

return maxLength;

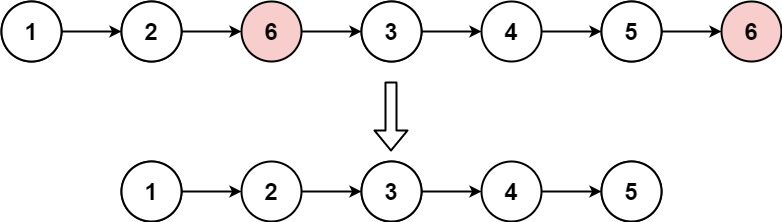
}

}

**Time Complexity: O(n)**

1. **Remove linked list Elements:**

Given the head of a linked list and an integer val, remove all the nodes of the linked list that has Node.val == val, and return *the new head*.



**Input:** head = [1,2,6,3,4,5,6], val = 6

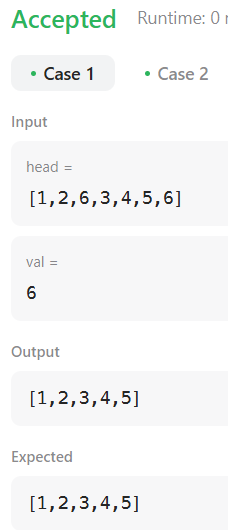
**Output:** [1,2,3,4,5]

**Code:**

class Solution {

    public ListNode removeElements(ListNode head, int val) {

        ListNode temp = new ListNode(0) , curr = temp;

        temp.next = head;

        while(curr.next != null ){

            if(curr.next.val == val) curr.next = curr.next.next;

            else curr = curr.next;

        }

        return temp.next;

    }

}

**Time Complexity: O(n)**

1. **Palindrome Linked List:**

Given the head of a singly linked list, return true*if it is a palindrome or*false*otherwise*.



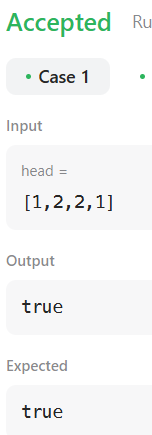
**Input:** head = [1,2,2,1]

**Output:** true

**Code:**

class Solution {

    public boolean isPalindrome(ListNode head) {

        List<Integer> list = new ArrayList();

        while(head != null) {

            list.add(head.val);

            head = head.next;

        }

        int left = 0;

        int right = list.size()-1;

        while(left < right && list.get(left) == list.get(right)) {

            left++;

            right--;

        }

        return left >= right;

    }

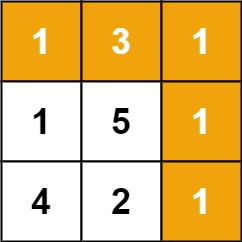
}

**Time Complexity: O(n)**

1. **Minimum path sum:**

Given a m x n grid filled with non-negative numbers, find a path from top left to bottom right, which minimizes the sum of all numbers along its path.

**Note:** You can only move either down or right at any point in time.



**Input:** grid = [[1,3,1],[1,5,1],[4,2,1]]

**Output:** 7

**Code:**

class Solution {

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

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

        for (int j = 1; j < n; j++) {

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

        }

        for (int i = 1; i < m; i++) {

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

        }

        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]);

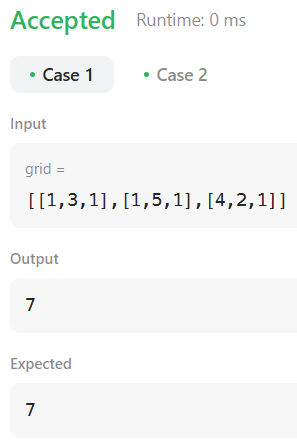
            }

        }

        return grid[m - 1][n - 1];

    }

}



**Time Complexity:o(m\*n)**

1. **Word ladder:**

Given two words, beginWord and endWord, and a dictionary wordList, return *the****number of words****in the****shortest transformation sequence****from* beginWord *to* endWord*, or*0*if no such sequence exists.*

**Input:** beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]

**Output:** 5

**Explanation:** One shortest transformation sequence is "hit" -> "hot" -> "dot" -> "dog" -> cog", which is 5 words long.

**Code:**

class Solution {

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

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

        Queue<String> queue = new LinkedList();

        Set<String> visited = new HashSet();

        queue.add(beginWord);

        visited.add(beginWord);

        int level = 0;

        while(!queue.isEmpty()) {

            for(int size = queue.size(); size > 0; size--) {

                String word = queue.poll();

                if(word.equals(endWord))

                    return level + 1;

                char[] ch = word.toCharArray();

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

                    char backup = ch[i];

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

                        ch[i] = c;

                        String nextWord = String.valueOf(ch);

                        if(!visited.contains(nextWord) && wordSet.contains(nextWord)) {

                            queue.add(nextWord);

                            visited.add(nextWord);

                        }

                    }

                    ch[i] = backup;

                }

            }

            level++;

        }

        return 0;

    }

}



**Time Complexity:O(m^2\*n)**

1. **Word ladder II:**

There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [ai, bi] indicates that you **must** take course bi first if you want to take course ai.

For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.

Return true if you can finish all courses. Otherwise, return false.

**Input:** numCourses = 2, prerequisites = [[1,0]]

**Output:** true

**Explanation:** There are a total of 2 courses to take.

To take course 1 you should have finished course 0. So it is possible.

**Code:**

class Solution {

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<>();

for(String w : wordList) hs.add(w);

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) == true){

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(N\*L + P\*L)**

1. **Course Schedule:**

There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [ai, bi] indicates that you **must** take course bi first if you want to take course ai.

For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.

Return true if you can finish all courses. Otherwise, return false.

**Example 1:**

**Input:** numCourses = 2, prerequisites = [[1,0]]

**Output:** true

**Explanation:** There are a total of 2 courses to take.

To take course 1 you should have finished course 0. So it is possible.

**Code:**

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);

}

}

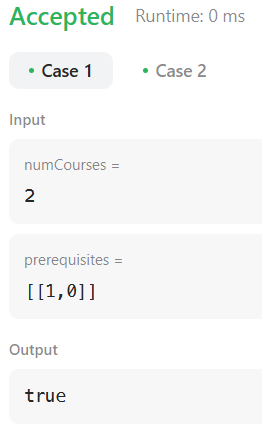
}

}

return ans.size() == n;

}

}



**Time Complexity:O(V+E)**

1. **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 subtree  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 Solution {

public boolean isValidBST(TreeNode root) {

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

}

private boolean valid(TreeNode node, long minimum, long maximum) {

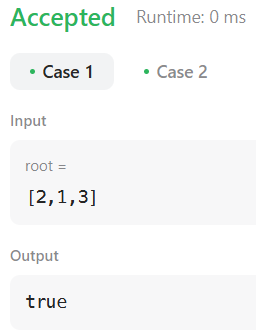
if (node == null) return true;

if (!(node.val > minimum && node.val < maximum)) return false;

return valid(node.left, minimum, node.val) && valid(node.right, node.val, maximum);

}

}



**Time Complexity:o(n)**

1. **Design tic tac toe:**

**Code:**

**import** java.util.Scanner;

**class** tictactoe {

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

**char**[][] board = **new** **char**[3][3];

**for** (**int** row = 0; row < board.length; row++) {

**for** (**int** col = 0; col < board[row].length; col++) {

board[row][col] = ' ';

}

}

**char** player = 'X';

**boolean** gameOver = **false**;

Scanner scanner = **new** Scanner(System.***in***);

**while** (!gameOver) {

*printBoard*(board);

System.***out***.print("Player " + player + " enter: ");

**int** row = scanner.nextInt();

**int** col = scanner.nextInt();

System.***out***.println();

**if** (board[row][col] == ' ') {

board[row][col] = player; // place the element

gameOver = *haveWon*(board, player);

**if** (gameOver) {

System.***out***.println("Player " + player + " has won: ");

} **else** {

player = (player == 'X') ? 'O' : 'X';

}

} **else** {

System.***out***.println("Invalid move. Try again!");

}

}

*printBoard*(board);

}

**public** **static** **boolean** haveWon(**char**[][] board, **char** player) {

// check the rows

**for** (**int** row = 0; row < board.length; row++) {

**if** (board[row][0] == player && board[row][1] == player && board[row][2] == player) {

**return** **true**;

}

}

**for** (**int** col = 0; col < board[0].length; col++) {

**if** (board[0][col] == player && board[1][col] == player && board[2][col] == player) {

**return** **true**;

}

}

// diagonal

**if** (board[0][0] == player && board[1][1] == player && board[2][2] == player) {

**return** **true**;

}

**if** (board[0][2] == player && board[1][1] == player && board[2][0] == player) {

**return** **true**;

}

**return** **false**;

}

**public** **static** **void** printBoard(**char**[][] board) {

**for** (**int** row = 0; row < board.length; row++) {

**for** (**int** col = 0; col < board[row].length; col++) {

System.***out***.print(board[row][col] + " | ");

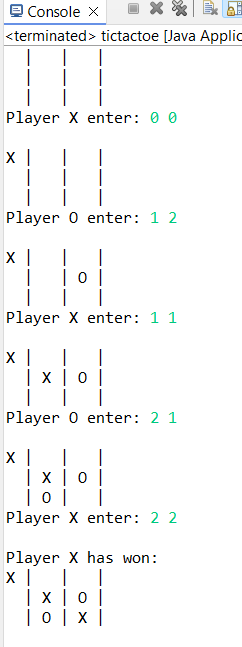
}

System.***out***.println();

}

}

}



**Time Complexity: O(1)**