1.Next Permutation
Last Updated : 13 Nov, 2024
Given an array arr[] of size n, the task is to print the lexicographically next greater permutation of the given array. If there does not exist any greater permutation, then find the lexicographically smallest permutation of the given array.
Let us understand the problem better by writing all permutations of [1, 2, 4] in lexicographical order
[1, 2, 4], [1, 4, 2], [2, 1, 4], [2, 4, 1], [4, 1, 2] and [4, 2, 1]
If we give any of the above (except the last) as input, we need to find the next one in sequence. If we give last as input, we need to return the first one.

Solution:

```
import java.util.Arrays;

class Per{
    static void nextPermutation(int[] arr) {
        int n = arr.length;
        int pivot = -1;
        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);
    }
    private static void reverse(int[] arr, int start, int end) {
        while (start < end) {
            swap(arr, start++, end--);
        }
    }

    private static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    public static void main(String[] args) {
        int[] arr = { 2, 4, 1, 7, 5, 0 };
        nextPermutation(arr);

        for(int i = 0; i < arr.length; i++)
        System.out.print(arr[i] + " ");
    }
}
```

Output:

```
2 4 5 0 1 7
```

Time Complexity:O(n)
Space Complexity:O(1)

2.Print a given matrix in spiral form

Given an m x n matrix, the task is to print all elements of the matrix in spiral form.

Examples:

```
 Input: matrix = {{1,    2,   3,   4},
                   {5,    6,   7,   8},
                  {9,   10,  11,  12},
                 {13,  14,  15,  16 }}
Output: 1 2 3 4 8 12 16 15 14 13 9 5 6 7 11 10
```

Explanation: The output is matrix in spiral format.

Solution:

```
import java.util.*;
public class SpiralPrintMatrix {
    public static void spiralPrint(int m, int n, int[][] a)
    {       int top = 0, bottom = m - 1, left = 0,
            right = n - 1;
        while (top <= bottom && left <= right) {
            for (int i = left; i <= right; ++i) {
                System.out.print(a[top][i] + " ");
            }
            top++;

            for (int i = top; i <= bottom; ++i) {
                System.out.print(a[i][right] + " ");
            }
            right--;
            if (top <= bottom) {
                for (int i = right; i >= left; --i) {
                    System.out.print(a[bottom][i] + " ");
                }
                bottom--;
            }

            if (left <= right) {
                for (int i = bottom; i >= top; --i) {
                    System.out.print(a[i][left] + " ");
                }
                left++;
            }
        }
    }

    public static void main(String[] args)
    {
        int[][] matrix = { { 1, 2, 3, 4 },
                           { 5, 6, 7, 8 },
                           { 9, 10, 11, 12 },
                           { 13, 14, 15, 16 } };

        spiralPrint(matrix.length, matrix[0].length,
                    matrix);
    }
}
```

Output:

```
1 2 3 4 8 12 16 15 14 13 9 5 6 7 11 10
```

Time Complexity:O(m*n)
Space Complexity:O(1)

3.Longest substring without repeating characters
Last Updated : 27 Aug, 2024
Given a string s, find the length of the longest substring without repeating characters.
Examples:
Input: “ABCBC”
Output: 3
Explanation: The longest substring without repeating characters is “ABC”

Solution:

```
class GfG {

    static int longestUniqueSubstr(String s) {
        int n = s.length();

        int res = 0;

        int[] lastIndex = new int[256];
        for (int i = 0; i < 256; i++) {
            lastIndex[i] = -1;
        }
        int start = 0;

        for (int end = 0; end < n; end++) {
             start = Math.max(start, lastIndex[s.charAt(end)] + 1);
            res = Math.max(res, end - start + 1);
            lastIndex[s.charAt(end)] = end;
        }

        return res;
    }

    public static void main(String[] args) {
        String s = "geeksforgeeks";
        System.out.println(longestUniqueSubstr(s));
    }
}
```

Output:

```
7
```

Time Complexity:O(n)
Space Complexity:O(1)

4.
Delete all occurrences of a given key in a linked list
Last Updated : 04 Sep, 2024
Given a singly linked list, the task is to delete all occurrences of a given key in it.
Examples:
Input: head: 2 -> 2 -> 1 -> 8 -> 2 -> NULL, key = 2
Output: 1 -> 8 -> NULL

Solution:

```
class Node {
    int data;
    Node next;

    Node(int new_data) {
        data = new_data;
        next = null;
    }
}

public class GfG {

    static Node deleteOccurrences(Node head, int key) {
        Node curr = head, prev = null;

        while (curr != null) {
            if (curr.data == key) {

                if (prev == null) {
                    head = curr.next;
                }

                else {
                    prev.next = curr.next;
                }

                curr = curr.next;

            }

             else {

                prev = curr;
                curr = curr.next;
            }
        }

        return head;
    }
    static void printList(Node curr) {
        while (curr != null) {
            System.out.print(" " + curr.data);
            curr = curr.next;
        }
    }

    public static void main(String[] args) {

        // Create a hard-coded linked list:
        // 2 -> 2 -> 1 -> 8 -> 2 -> NULL
        Node head = new Node(2);
        head.next = new Node(2);
        head.next.next = new Node(1);
        head.next.next.next = new Node(8);
        head.next.next.next.next = new Node(2);

        int key = 2;

        head = deleteOccurrences(head, key);
        printList(head);
    }
}
```

Output:

```
 1 8
```

Time Complexity:O(n)
Space Complexity:O(1)

5.Palindrome Linked List
Last Updated : 14 Sep, 2024
Given a singly linked list. The task is to check if the given linked list is palindrome or not.
Examples:
Input: head: 1->2->1->1->2->1
Output: true
Explanation: The given linked list is 1->2->1->1->2->1 , which is a palindrome and Hence, the output is true.

Solution:

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

class GfG {
```

```
// Function to reverse a linked list
static Node reverseList(Node head) {
    Node prev = null;
    Node curr = head;
    Node next;

    while (curr != null) {
        next = curr.next;
        curr.next = prev;
        prev = curr;
        curr = next;
    }
    return prev;
}

// Function to check if two lists are identical
static boolean isIdentical(Node n1, Node n2) {
    while (n1 != null && n2 != null) {
        if (n1.data != n2.data)
            return false;
        n1 = n1.next;
        n2 = n2.next;
    }
    return true;
}

// Function to check whether the list is palindrome
static boolean isPalindrome(Node head) {
    if (head == null || head.next == null)
        return true;

    Node slow = head, fast = head;

    while (fast.next != null
          && fast.next.next != null) {
        slow = slow.next;
        fast = fast.next.next;
    }

    Node head2 = reverseList(slow.next);
    slow.next = null;

    boolean ret = isIdentical(head, head2);
```

```
        head2 = reverseList(head2);
        slow.next = head2;

        return ret;
    }

    public static void main(String[] args) {

      // Linked list : 1->2->3->2->1
        Node head = new Node(1);
        head.next = new Node(2);
        head.next.next = new Node(3);
        head.next.next.next = new Node(2);
        head.next.next.next.next = new Node(1);

        boolean result = isPalindrome(head);

        if (result)
            System.out.println("true");
        else
            System.out.println("false");
    }
}
```
Output:

```
true
```

Time Complexity:O(n)
Space Complexity:O(1)

6.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.

Solution:

```
public 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];
    }

    public static void main(String[] args) {
        Solution solution = new Solution();

        int[][] grid = {
            {1, 3, 1},
            {1, 5, 1},
            {4, 2, 1}
        };

        int result = solution.minPathSum(grid);
        System.out.println(result);
    }
}
```

Output:

```
7
```

Time Complexity:O(m*n)
Space Complexity:O(1)

7.Check if a Binary Tree is BST or not
Last Updated : 25 Sep, 2024
Given the root of a binary tree. Check whether it is a Binary Search Tree or not. A Binary Search Tree (BST) is a node-based binary tree data structure with the following properties.
All keys in the left subtree are smaller than the root and all keys in the right subtree are greater.
Both the left and right subtrees must also be binary search trees.
Each key must be distinct.

Solution:

```
class Solution {
    public boolean isValidBST(TreeNode root) {
        return validate(root, null, null);}
```

```
  private boolean validate(TreeNode node, Integer lower, Integer upper) {
     if (node == null) return true;

     if ((lower != null && node.val <= lower) ||
        (upper != null && node.val >= upper)) {
        return false;
     }

     return validate(node.left, lower, node.val) &&
           validate(node.right, node.val, upper);
  }
  public static void main(String[] args) {
     Solution solution = new Solution();

     TreeNode validRoot = new TreeNode(5);
     validRoot.left = new TreeNode(3);
     validRoot.right = new TreeNode(7);
     validRoot.left.left = new TreeNode(1);
     validRoot.left.right = new TreeNode(4);
     validRoot.right.left = new TreeNode(6);
     validRoot.right.right = new TreeNode(8);

     TreeNode invalidRoot = new TreeNode(5);
     invalidRoot.left = new TreeNode(3);
     invalidRoot.right = new TreeNode(7);
     invalidRoot.left.left = new TreeNode(1);
     invalidRoot.left.right = new TreeNode(6);

     System.out.println(solution.isValidBST(validRoot));
     System.out.println(solution.isValidBST(invalidRoot));
  }
}
class TreeNode {
  int val;
  TreeNode left;
  TreeNode right;
  TreeNode(int x) { val = x; }
}
```

Output:

```
true
```

Time Complexity:O(n)

Space Complexity:O(h)

8.Word Ladder (Length of shortest chain to reach a target word)
Last Updated : 10 Jul, 2024
Given a dictionary, and two words 'start' and 'target' (both of same length). Find length of the smallest chain from 'start' to 'target' if it exists, such that adjacent words in the chain only differ by one character and each word in the chain is a valid word i.e., it exists in the dictionary. It may be assumed that the 'target' word exists in dictionary and length of all dictionary words is same.
Example:
Input: Dictionary = {POON, PLEE, SAME, POIE, PLEA, PLIE, POIN}, start = TOON, target = PLEA
Output: 7
Explanation: TOON – POON – POIN – POIE – PLIE – PLEE – PLEA

Solution:
```
import java.util.*;

class GFG
{

static int shortestChainLen(String start,
                            String target,
                            Set<String> D)
{

     if(start == target)
      return 0;
    if (!D.contains(target))
        return 0;

    int level = 0, wordlength = start.length();

    Queue<String> Q = new LinkedList<>();
    Q.add(start);

    // While the queue is non-empty
    while (!Q.isEmpty())
    {

        // Increment the chain length
        ++level;

        // Current size of the queue
        int sizeofQ = Q.size();

        // Since the queue is being updated while
        // it is being traversed so only the
        for (int i = 0; i < sizeofQ; ++i)
        {

            // Remove the first word from the queue
            char []word = Q.peek().toCharArray();
            Q.remove();

            // For every character of the word
            for (int pos = 0; pos < wordlength; ++pos)
            {

                // Retain the original character
                // at the current position
                char orig_char = word[pos];

                // Replace the current character with
                // every possible lowercase alphabet
                for (char c = 'a'; c <= 'z'; ++c)
                {
                    word[pos] = c;

                    // If the new word is equal
                    // to the target word
                    if (String.valueOf(word).equals(target))
                        return level + 1;

                    // Remove the word from the set
                    // if it is found in it
                    if (!D.contains(String.valueOf(word)))
                        continue;
                    D.remove(String.valueOf(word));

                    // And push the newly generated word
                    // which will be a part of the chain
                    Q.add(String.valueOf(word));
                }

                // Restore the original character
                // at the current position
                word[pos] = orig_char;
            }
        }
    }
return 0;}
```

```
public static void main(String[] args)
{
    // make dictionary
    Set<String> D = new HashSet<String>();
    D.add("poon");
    D.add("plee");
    D.add("same");
    D.add("poie");
    D.add("plie");
    D.add("poin");
    D.add("plea");
    String start = "toon";
    String target = "plea";
    System.out.print("Length of shortest chain is: "
        + shortestChainLen(start, target, D));
}
}
```

Output:

```
Length of shortest chain is: 7
```

Time Complexity:O(n*m)
Space Complexity:O(m+n)

9.Word Ladder II
Difficulty: HardAccuracy: 50.0%Submissions: 33K+Points: 8
Given two distinct words startWord and targetWord, and a list denoting wordList of unique words of equal lengths. Find all shortest transformation sequence(s) from startWord to targetWord. You can return them in any order possible.
Keep the following conditions in mind:
A word can only consist of lowercase characters.
Only one letter can be changed in each transformation.
Each transformed word must exist in the wordList including the targetWord.
startWord may or may not be part of the wordList.
Return an empty list if there is no such transformation sequence.
The first part of this problem can be found here.
Example 1:
Input:
startWord = "der", targetWord = "dfs",
wordList = {"des","der","dfr","dgt","dfs"}
Output:

```
der dfr dfs
der des dfs
```

Solution:
```
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);
                }
            }
        }

public class Main {
    public static void main(String[] args) {
        Solution solution = new Solution();

        String beginWord = "hit";
        String endWord = "cog";
        List<String> wordList = Arrays.asList("hot", "dot", "dog", "lot", "log", "cog");

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

        for (List<String> sequence : result) {
            System.out.println(sequence);}}}
    }
}
```
Output:

```
[hit, hot, dot, dog, cog]
[hit, hot, lot, log, cog]
```

Time Complexity:O(n*L)
Space Complexity:O(n+L)

10.Course Schedule
Difficulty: MediumAccuracy: 51.77%Submissions: 74K+Points: 4
There are a total of n tasks you have to pick, labelled from 0 to n-1. Some tasks may have prerequisite tasks, for example to pick task 0 you have to first finish tasks 1, which is expressed as a pair: [0, 1]
Given the total number of n tasks and a list of prerequisite pairs of size m. Find a ordering of tasks you should pick to finish all tasks.
Note: There may be multiple correct orders, you just need to return any one of them. If it is impossible to finish all tasks, return an empty array. Driver code will print "No Ordering Possible", on returning an empty array. Returning any correct order will give the output as 1, whereas any invalid order will give the output 0.
Example 1:
Input:
n = 2, m = 1
prerequisites = {{1, 0}}
Output:
1
Explanation:
The output 1 denotes that the order is valid. So, if you have, implemented your function correctly, then output would be 1 for all test cases. One possible order is [0, 1].

Solution:
```java
import java.util.*;

public class Solution {
    public boolean canFinish(int numCourses, int[][] prerequisites) {
        int counter = 0;
        if (numCourses <= 0) {
            return true;
        }

        // Initialize inDegree array and adjacency list
        int[] inDegree = new int[numCourses];
        List<List<Integer>> graph = new ArrayList<>();
        for (int i = 0; i < numCourses; i++) {
            graph.add(new ArrayList<>());
        }

        // Build the graph and update inDegree for each node
        for (int[] edge : prerequisites) {
            int parent = edge[1];
            int child = edge[0];
            graph.get(parent).add(child);
            inDegree[child]++;
        }
        // Initialize the queue with courses having no prerequisites (inDegree = 0)
        Queue<Integer> sources = new LinkedList<>();
        for (int i = 0; i < numCourses; i++) {
            if (inDegree[i] == 0) {
                sources.offer(i);
            }
        }

        // Process nodes with no prerequisites
        while (!sources.isEmpty()) {
            int course = sources.poll(); // dequeue
            counter++;

            // Process all the children of the current course
            for (int child : graph.get(course)) {
                inDegree[child]--;
                if (inDegree[child] == 0) {
                    sources.offer(child); // enqueue child if inDegree becomes 0
                }
            }
        }

        // If we processed all courses, return true
        return counter == numCourses;
    }

    public static void main(String[] args) {
        Solution solution = new Solution();

        // Example 1
        int numCourses1 = 2;
        int[][] prerequisites1 = {{1, 0}};
        System.out.println("Can finish all courses (Example 1): " + solution.canFinish(numCourses1,
prerequisites1));

        // Example 2
        int numCourses2 = 2;
        int[][] prerequisites2 = {{1, 0}, {0, 1}};
        System.out.println("Can finish all courses (Example 2): " + solution.canFinish(numCourses2,
prerequisites2));

        // Example 3
        int numCourses3 = 4;
        int[][] prerequisites3 = {{1, 0}, {2, 1}, {3, 2}};
        System.out.println("Can finish all courses (Example 3): " + solution.canFinish(numCourses3,
prerequisites3));
    }
}
```

Output:

```
Can finish all courses (Example 1): true
Can finish all courses (Example 2): false
Can finish all courses (Example 3): true
```

Time Complexity:O(v+e)
Space Complexity:O(v+e)

11.Low Level Design of Tic Tac Toe | System Design
Last Updated : 26 Aug, 2024
In classic games, few are as universally recognized and enjoyed as Tic Tac Toe. Despite its apparent simplicity, this game engages players of all ages in a battle of wits and strategy. In this article, we delve into the intricacies of designing a low-level structure for Tic Tac Toe that captures the essence of the game and highlights the underlying complexities.

Solution:

```
import java.util.Scanner;
public class TicTacToe {
    private char[][] board; // 3x3 game board
    private int
        currentPlayer; // 1 for Player 1, 2 for Player 2
    public TicTacToe()
    {
        board = new char[3][3];
        for (int i = 0; i < 3; ++i) {
            for (int j = 0; j < 3; ++j) {
                board[i][j] = ' ';}}
        currentPlayer = 1;
    }
    public void printBoard()
    {
        for (int i = 0; i < 3; ++i) {
            for (int j = 0; j < 3; ++j) {
                System.out.print(board[i][j]);
                if (j < 2) {
                    System.out.print(" | ");}}
            System.out.println();
            if (i < 2) {     System.out.println("---------");}}}
```

```
public boolean isBoardFull()
 {
     for (int i = 0; i < 3; ++i) {
         for (int j = 0; j < 3; ++j) {
             if (board[i][j] == ' ') {
                 return false;
             }
         }
     }
     return true;
 }
 public boolean makeMove(int row, int column)
 {
     if (row < 0 || row >= 3 || column < 0 || column >= 3
         || board[row][column] != ' ') {
         return false; // Invalid move
     }
     board[row][column]
         = (currentPlayer == 1) ? 'X' : 'O';
     currentPlayer
         = 3 - currentPlayer; // Switch player (1 to 2 or
                              // 2 to 1)
     return true;
 }
 public boolean checkWinner()
 {
     // Check rows, columns, and diagonals for a win
     for (int i = 0; i < 3; ++i) {
         if (board[i][0] != ' '
             && board[i][0] == board[i][1]
             && board[i][1] == board[i][2]) {
             return true; // Row win
         }
         if (board[0][i] != ' '
             && board[0][i] == board[1][i]
             && board[1][i] == board[2][i]) {
             return true; // Column win
         }
     }
     if (board[0][0] != ' ' && board[0][0] == board[1][1]
         && board[1][1] == board[2][2]) {
         return true; // Diagonal win
     }
     if (board[0][2] != ' ' && board[0][2] == board[1][1]
            && board[1][1] == board[2][0]) {
            return true; // Diagonal win
        }
        return false;
    }
    public static void main(String[] args)
    {
        TicTacToe game = new TicTacToe();
        Scanner scanner = new Scanner(System.in);
        int row, column;

        while (!game.isBoardFull() && !game.checkWinner()) {
            game.printBoard();

            System.out.print(
                "Player " + game.currentPlayer
                + ", enter your move (row and column): ");
            row = scanner.nextInt();
            column = scanner.nextInt();

            if (game.makeMove(row, column)) {
                System.out.println("Move successful!");
            }
            else {
                System.out.println(
                    "Invalid move. Try again.");
            }
        }
        game.printBoard();

        if (game.checkWinner()) {
            System.out.println("Player "
                          + (3 - game.currentPlayer)
                          + " wins!");
        }
        else {
            System.out.println("It's a draw!");
        }

        scanner.close();
    }
}
```
Time Complexity:O(n)
Space Complexity:O(1)