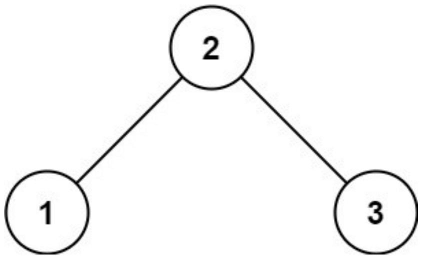
DSA PRACTICE -7

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 subtreeof 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.

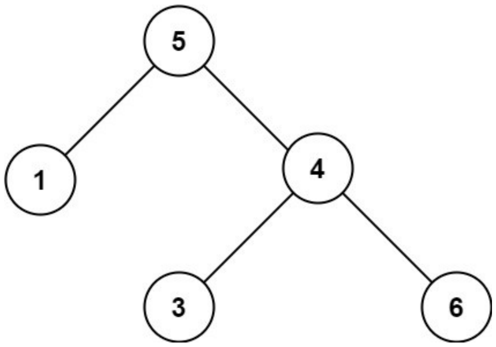
Example 1:



Input: root = [2,1,3]

Output: true

Example 2:



Input: root = [5,1,4,null,null,3,6]

Output: false

Explanation: The root node's value is 5 but its right child's value is 4.

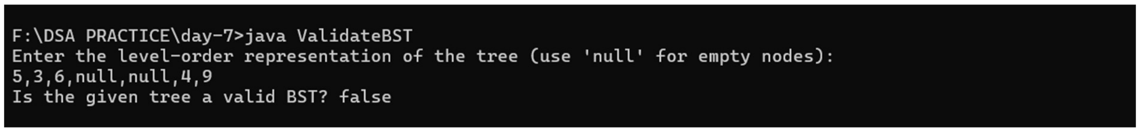
Constraints:

- The number of nodes in the tree is in the range [1, 10^4].
- $-2^{31} <= \text{Node.val} <= 2^{31} – 1$

Solution:

```
import java.util.*;

class TreeNode {

    int val;

    TreeNode left;

    TreeNode right;

    TreeNode(int x) { val = x; }

}

class ValidateBST {

    public boolean isValidBST(TreeNode root) {

        return BSTU(root, Long.MIN_VALUE, Long.MAX_VALUE);

    }

    private boolean BSTU(TreeNode node, long min, long max) {

        if (node == null) return true;

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

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

    }

    public TreeNode constructTree(Integer[] values) {

        if (values == null || values.length == 0 || values[0] == null) return null;

        TreeNode root = new TreeNode(values[0]);

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

        queue.add(root);

        int i = 1;

        while (!queue.isEmpty() && i < values.length) {

            TreeNode current = queue.poll();


            if (values[i] != null) {

                current.left = new TreeNode(values[i]);

                queue.add(current.left);

            }
            i++;

            if (i < values.length && values[i] != null) {

                current.right = new TreeNode(values[i]);

                queue.add(current.right);

            }

            i++;

        }

        return root;

    }

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.println("Enter the level-order representation of the tree (use 'null' for empty
nodes):");

        String input = scanner.nextLine();

        String[] parts = input.split(",");

        Integer[] values = new Integer[parts.length];

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

            values[i] = parts[i].trim().equalsIgnoreCase("null") ? null :
Integer.parseInt(parts[i].trim());

        }

        ValidateBST solution = new ValidateBST();

        TreeNode root = solution.constructTree(values);

        boolean isValid = solution.isValidBST(root);

        System.out.println("Is the given tree a valid BST? " + isValid);

    }

}
```

Output:



```
F:\DSA PRACTICE\day-7>java ValidateBST
Enter the level-order representation of the tree (use 'null' for empty nodes):
5,3,6,null,null,4,9
Is the given tree a valid BST? false
```

TimeComplexity:O(n)

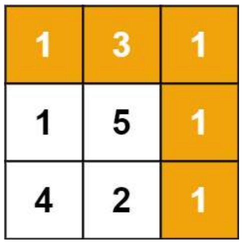
SpaceComplexity:O(n)

2. 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.

Example 1:



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

Output: 7

Explanation: Because the path 1 → 3 → 1 → 1 → 1 minimizes the sum.

Example 2:

Input: grid = [[1,2,3],[4,5,6]]

Output: 12

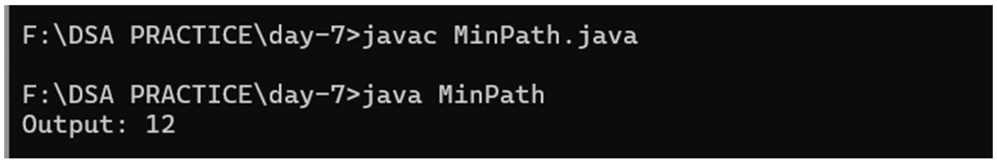
Constraints:

- m == grid.length
- n == grid[i].length
- 1 <= m, n <= 200
- 0 <= grid[i][j] <= 200

Solution:

```
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];
    }
    public static void main(String[] args) {
        Solution solution = new Solution();
        int[][] grid = {
            {1, 2, 3},
            {4, 5, 6}
        };
        int result = solution.minPathSum(grid);
        System.out.println("Output: " + result);
    }
}
```

Output:



```
F:\DSA PRACTICE\day-7>javac MinPath.java

F:\DSA PRACTICE\day-7>java MinPath
Output: 12
```

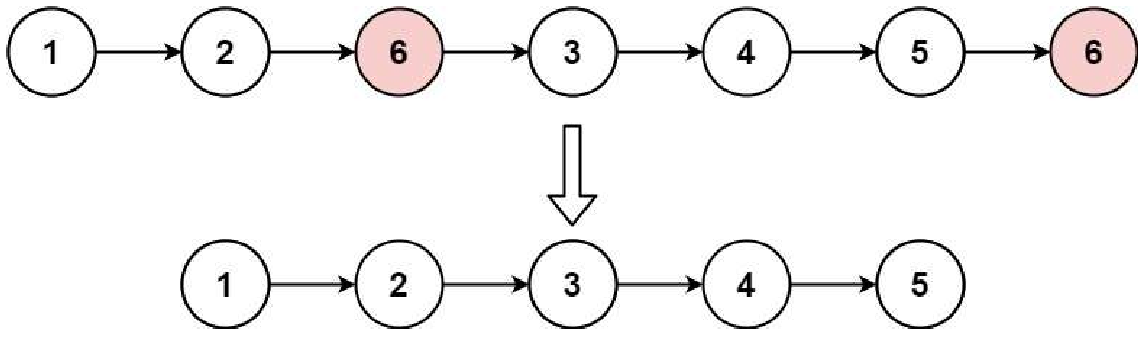
TimeComplexity:O(n*m)

SpaceComplexity:O(1)

3.Remove LinkedListNode:

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*.

Example 1:



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

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

Example 2:

Input: head = [], val = 1

Output: []

Example 3:

Input: head = [7,7,7,7], val = 7

Output: []

Constraints:

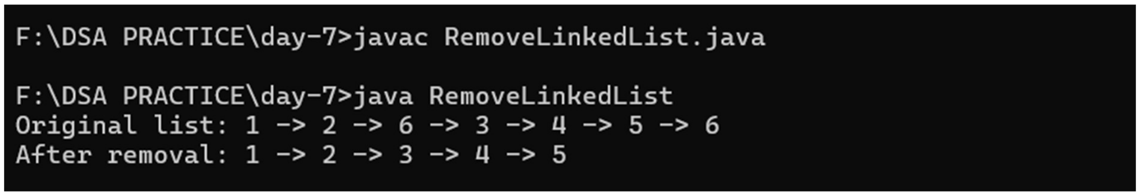
- The number of nodes in the list is in the range [0, 10^4].
- 1 <= Node.val <= 50
- 0 <= val <= 50

Solution:

```
class ListNode {
   int val;
   ListNode next;
   ListNode() {}
   ListNode(int val) { this.val = val; }
   ListNode(int val, ListNode next) { this.val = val; this.next = next; }
}
```

```
class Solution {
    public ListNode removeElements(ListNode head, int val) {
        ListNode res = new ListNode(0);
        ListNode dummy = res;

        while (head != null) {
            if (head.val != val) {
                dummy.next = head;
                dummy = dummy.next;
            }
            head = head.next;
        }
        dummy.next = null;
        return res.next;
    }
    public static ListNode createLinkedList(int[] values) {
        ListNode dummy = new ListNode(0);
        ListNode current = dummy;
        for (int value : values) {
            current.next = new ListNode(value);
            current = current.next;
        }
        return dummy.next;
    }
    public static void printLinkedList(ListNode head) {
        while (head != null) {
            System.out.print(head.val);
            if (head.next != null) System.out.print(" -> ");
            head = head.next;
        }
        System.out.println();
    }
    public static void main(String[] args) {
        int[] values = {1, 2, 6, 3, 4, 5, 6};
        int val = 6;
        ListNode head = createLinkedList(values);
        System.out.print("Original list: ");
        printLinkedList(head);
        Solution solution = new Solution();
        ListNode result = solution.removeElements(head, val);
        System.out.print("After removal: ");
        printLinkedList(result);
    }
}
```

Output:



```
F:\DSA PRACTICE\day-7>javac RemoveLinkedList.java

F:\DSA PRACTICE\day-7>java RemoveLinkedList
Original list: 1 -> 2 -> 6 -> 3 -> 4 -> 5 -> 6
After removal: 1 -> 2 -> 3 -> 4 -> 5
```

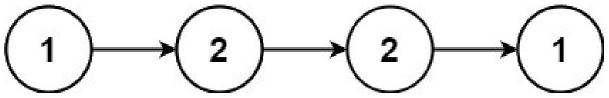
TimeComplexity:O(n)

SpaceComplexity:O(n)

4. Palindrome Linked List

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

Example 1:

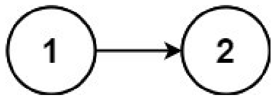


$$\begin{tikzcd} \boxed{1} \arrow[r] & \boxed{2} \arrow[r] & \boxed{2} \arrow[r] & \boxed{1} \end{tikzcd}$$

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

Output: true

Example 2:



Input: head = [1,2]

Output: false

Constraints:

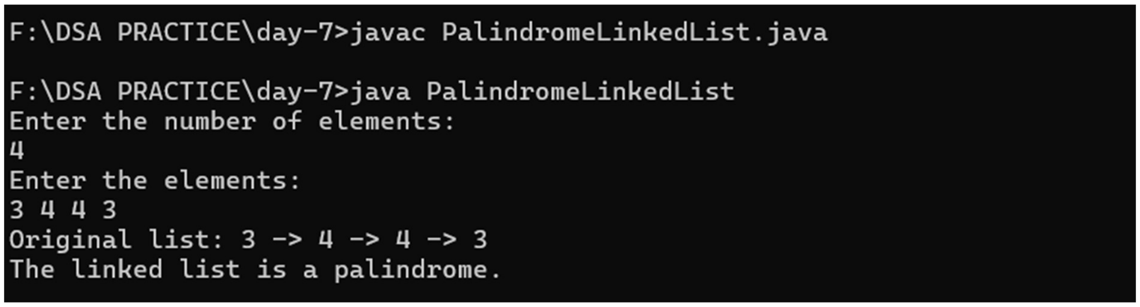
- The number of nodes in the list is in the range [1, 10^5].
- 0 <= Node.val <= 9

Solution:

```
import java.util.*;
class ListNode {
  int val;
  ListNode next;
  ListNode() {}
  ListNode(int val) {
    this.val = val;
  }
  ListNode(int val, ListNode next) {
    this.val = val;
    this.next = next;
  }
}
class PalindromeLinkedList {
  public boolean isPalindrome(ListNode head) {
    List<Integer> l = new ArrayList<>();
    while (head != null) {
      l.add(head.val);
      head = head.next;
    }
    int le = 0;
    int r = l.size() - 1;
    while (le < r && l.get(le).equals(l.get(r))) {
        le++;
        r--;
    }
    return le >= r;
}
public static ListNode createLinkedList(Scanner sc) {
    System.out.println("Enter the number of elements:");
    int n = sc.nextInt();
    if (n == 0) return null;
    System.out.println("Enter the elements:");
    ListNode dummy = new ListNode(0);
    ListNode current = dummy;
    for (int i = 0; i < n; i++) {
        int value = sc.nextInt();
        current.next = new ListNode(value);
        current = current.next;
    }
    return dummy.next;
}
public static void printLinkedList(ListNode head) {
    while (head != null) {
        System.out.print(head.val);
        if (head.next != null) System.out.print(" -> ");
        head = head.next;
    }
```

```
        System.out.println();
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        ListNode head = createLinkedList(sc);
        System.out.print("Original list: ");
        printLinkedList(head);
        PalindromeLinkedList solution = new PalindromeLinkedList();
        boolean isPalindrome = solution.isPalindrome(head);
        if (isPalindrome) {
            System.out.println("The linked list is a palindrome.");
        } else {
            System.out.println("The linked list is not a palindrome.");
        }
        sc.close();
    }
}
```

Output:



```
F:\DSA PRACTICE\day-7>javac PalindromeLinkedList.java

F:\DSA PRACTICE\day-7>java PalindromeLinkedList
Enter the number of elements:
4
Enter the elements:
3 4 4 3
Original list: 3 -> 4 -> 4 -> 3
The linked list is a palindrome.
```

TimeComplexity:0(n)

SpaceComplexity:0(n)

5. Longest Substring Without Repeating Characters

Given a string s, find the length of the longest   substring without repeating characters.

Example 1:

Input: s = "abcabcbb"

Output: 3

Explanation: The answer is "abc", with the length of 3.

Example 2:

Input: s = "bbbbb"

Output: 1

Explanation: The answer is "b", with the length of 1.

Example 3:

Input: s = "pwwkew"

Output: 3

Explanation: The answer is "wke", with the length of 3.

Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

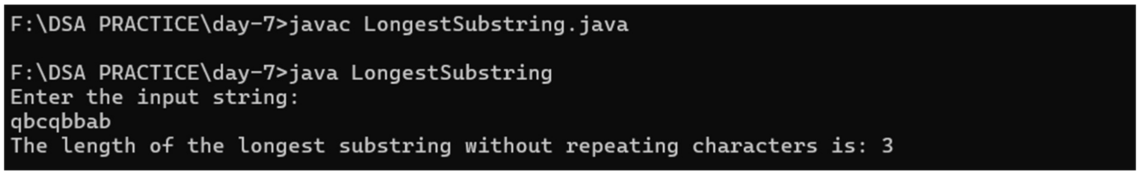
Constraints:

- 0 <= s.length <= 5 * 10^4
- s consists of English letters, digits, symbols and spaces.

Solution:

```
import java.util.*
class LongestSubstring {
    public int lengthOfLongestSubstring(String s) {
        Map<Character, Integer> charIndexMap = new HashMap<>();
        int maxLength = 0;
        int start = 0;

        for (int end = 0; end < s.length(); end++) {
            char current = s.charAt(end);

            if (charIndexMap.containsKey(current) && charIndexMap.get(current) >= start) {
                start = charIndexMap.get(current) + 1;
            }

            charIndexMap.put(current, end);

            maxLength = Math.max(maxLength, end - start + 1);
        }

        return maxLength;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the input string:");
        String input = sc.nextLine();

        LongestSubstring solution = new LongestSubstring();
        int result = solution.lengthOfLongestSubstring(input);

        System.out.println("The length of the longest substring without repeating characters is:
" + result);
        sc.close();
    }
}
```

Output:



```
F:\DSA PRACTICE\day-7>javac LongestSubstring.java

F:\DSA PRACTICE\day-7>java LongestSubstring
Enter the input string:
qbcqbbab
The length of the longest substring without repeating characters is: 3
```

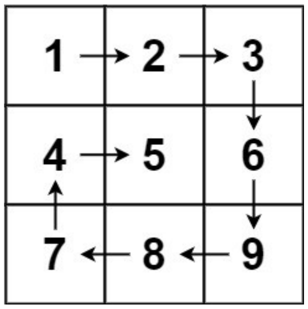
Time Complexity: O(n)

Space Complexity: O(n)

6. Spiral Matrix

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

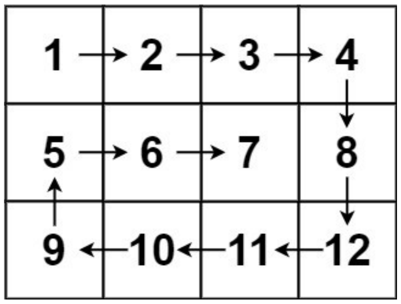
Example 1:



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

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

Example 2:



Input: matrix = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]

Output: [1,2,3,4,8,12,11,10,9,5,6,7]

Constraints:

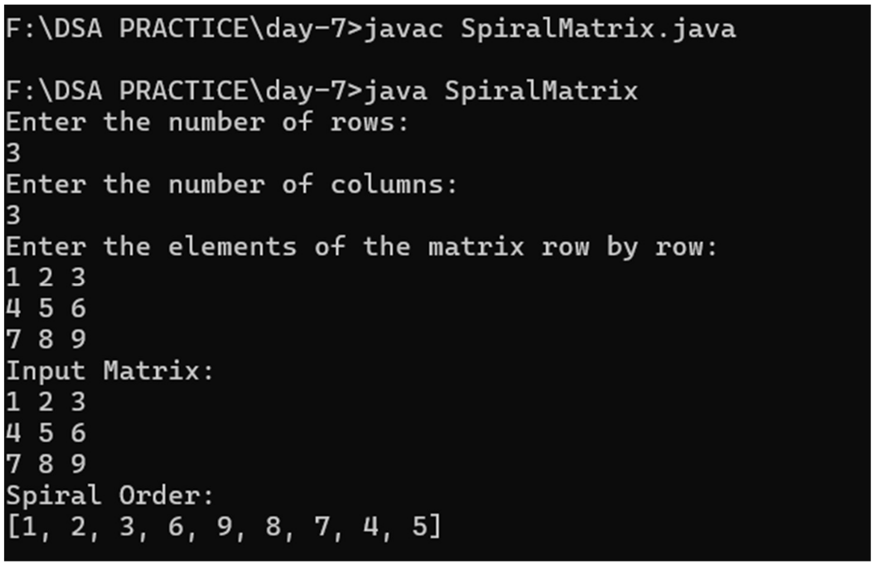
- m == matrix.length
- n == matrix[i].length
- 1 <= m, n <= 10
- -100 <= matrix[i][j] <= 100

Solution:

```
import java.util.*;
class SpiralMatrix {
  public List<Integer> spiralOrder(int[][] matrix) {
    int rows = matrix.length;
    int cols = matrix[0].length;
    int row = 0;
    int col = -1;
    int direction = 1;
    List<Integer> result = new ArrayList<>();
    while (rows > 0 && cols > 0) {
      for (int i = 0; i < cols; i++) {
        col += direction;
        result.add(matrix[row][col]);
      }
      rows--;
      for (int i = 0; i < rows; i++) {
        row += direction;
        result.add(matrix[row][col]);
      }
      cols--;
      direction *= -1;
    }
    return result;
  }
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter the number of rows:");
    int rows = sc.nextInt();
    System.out.println("Enter the number of columns:");
```

```
        int cols = sc.nextInt();

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

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

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

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

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

            }

        }

        System.out.println("Input Matrix:");

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

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

                System.out.print(matrix[i][j] + " ");

            }

            System.out.println();

        }

        SpiralMatrix solution = new SpiralMatrix();

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

        System.out.println("Spiral Order:");

        System.out.println(result);

        sc.close();

    }

}
```

Output:



```
F:\DSA PRACTICE\day-7>javac SpiralMatrix.java

F:\DSA PRACTICE\day-7>java SpiralMatrix
Enter the number of rows:
3
Enter the number of columns:
3
Enter the elements of the matrix row by row:
1 2 3
4 5 6
7 8 9
Input Matrix:
1 2 3
4 5 6
7 8 9
Spiral Order:
[1, 2, 3, 6, 9, 8, 7, 4, 5]
```

Timecomplexity:O(m*n)

Spacecomplexity:O(m*n)

7.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]

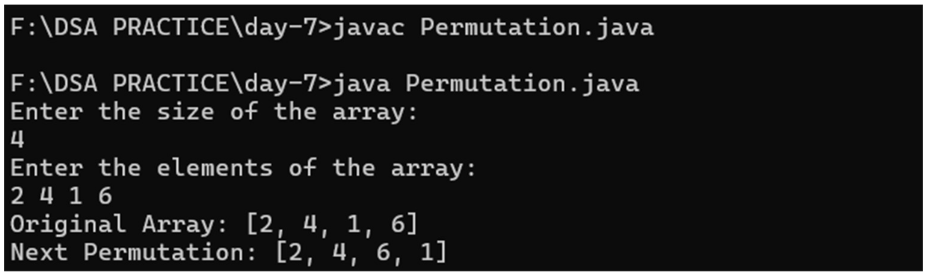
Constraints:

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

Solution:

```
import java.util.*;


class Permutation {
    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 void swap(int[] nums, int x, int y) {
        int temp = nums[x];
        nums[x] = nums[y];
        nums[y] = temp;
    }

    private void reverse(int[] nums, int start, int end) {
        while (start < end) {
            swap(nums, start, end);
            start++;
            end--;
        }
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the size of the array:");
        int n = sc.nextInt();
        int[] nums = new int[n];
        System.out.println("Enter the elements of the array:");
        for (int i = 0; i < n; i++) {
            nums[i] = sc.nextInt();
        }
        System.out.println("Original Array: " + Arrays.toString(nums));
        Permutation solution = new Permutation();
        solution.nextPermutation(nums);
        System.out.println("Next Permutation: " + Arrays.toString(nums));
        sc.close();
    }

}
```

Output:



```
F:\DSA PRACTICE\day-7>javac Permutation.java

F:\DSA PRACTICE\day-7>java Permutation.java
Enter the size of the array:
4
Enter the elements of the array:
2 4 1 6
Original Array: [2, 4, 1, 6]
Next Permutation: [2, 4, 6, 1]
```

Timecomplexity:O(n)

SpaceComplexity:O(1)