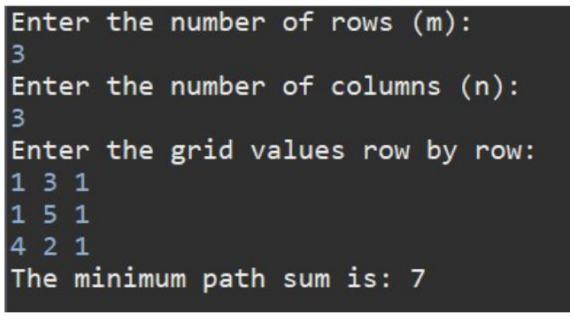
DSA Problem:

1. Minimum Path Sum:

```
import java.util.Arrays;

public class MinimumPathSum {
  public static int minPathSum(int[][] grid) {
    int rows = grid.length;
    int cols = grid[0].length;

    // Create a DP array to store the minimum path sums
    int[][] dp = new int[rows][cols];

    // Initialize the starting point
    dp[0][0] = grid[0][0];

    // Fill the first row (can only come from the left)
    for (int col = 1; col < cols; col++) {
      dp[0][col] = dp[0][col - 1] + grid[0][col];
    }

    // Fill the first column (can only come from above)
    for (int row = 1; row < rows; row++) {
      dp[row][0] = dp[row - 1][0] + grid[row][0];
    }

    // Fill the rest of the grid
    for (int row = 1; row < rows; row++) {
      for (int col = 1; col < cols; col++) {
        dp[row][col] = Math.min(dp[row - 1][col], dp[row][col - 1]) +
grid[row][col];
      }
    }

    // The bottom-right cell contains the minimum path sum
    return dp[rows - 1][cols - 1];
  }
```

```
    public static void main(String[] args) {
        int[][] grid = {
            {1, 3, 1},
            {1, 5, 1},
            {4, 2, 1}
        };

        System.out.println("Minimum Path Sum: " + minPathSum(grid));
    }
}
```

Output:



2. palindrome linked list:

```
class ListNode {
    int val;
    ListNode next;

    ListNode(int val) {
        this.val = val;
        this.next = null;
```

```
    }
}

public class PalindromeLinkedList {
    public static boolean isPalindrome(ListNode head) {
        if (head == null || head.next == null) {
            return true; // A single node or empty list is a palindrome
        }

        // Step 1: Find the middle of the linked list
        ListNode slow = head;
        ListNode fast = head;

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

        // Step 2: Reverse the second half of the list
        ListNode secondHalf = reverseList(slow);

        // Step 3: Compare the two halves
        ListNode firstHalf = head;
        ListNode tempSecondHalf = secondHalf; // Save this to
restore the list later
        boolean isPalindrome = true;

        while (tempSecondHalf != null) {
            if (firstHalf.val != tempSecondHalf.val) {
                isPalindrome = false;
                break;
            }
            firstHalf = firstHalf.next;
```

```
            tempSecondHalf = tempSecondHalf.next;
        }

        // Step 4: Restore the original structure of the list (optional)
        reverseList(secondHalf);

        return isPalindrome;
    }

    // Helper function to reverse a linked list
    private static ListNode reverseList(ListNode head) {
        ListNode prev = null;
        while (head != null) {
            ListNode next = head.next;
            head.next = prev;
            prev = head;
            head = next;
        }
        return prev;
    }

    // Helper function to create and print a linked list
    public static void printList(ListNode head) {
        while (head != null) {
            System.out.print(head.val + " -> ");
            head = head.next;
        }
        System.out.println("null");
    }

    public static void main(String[] args) {
        // Create a test linked list: 1 -> 2 -> 2 -> 1
        ListNode head = new ListNode(1);
```

```
        head.next = new ListNode(2);
        head.next.next = new ListNode(2);
        head.next.next.next = new ListNode(1);

        System.out.print("Original List: ");
        printList(head);

        boolean result = isPalindrome(head);

        System.out.println("Is Palindrome: " + result);
        System.out.print("Restored List: ");
        printList(head); // To verify the original structure is retained
    }
}
```

Output:

```
It is a palindrome
```

3. longest substring without reapeating string:

```
import java.util.HashSet;

public class LongestSubstringWithoutRepeating {
    public static int lengthOfLongestSubstring(String s) {
        if (s == null || s.isEmpty()) {
            return 0;
```

```
    }

    HashSet<Character> set = new HashSet<>();
    int maxLength = 0;
    int start = 0;

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

      // If character is already in the set, remove characters
from the start
      while (set.contains(currentChar)) {
        set.remove(s.charAt(start));
        start++;
      }

      // Add the current character to the set and update max
length
      set.add(currentChar);
      maxLength = Math.max(maxLength, end - start + 1);
    }

    return maxLength;
  }

  public static void main(String[] args) {
    String input = "abcabcbb";
    System.out.println("Input: " + input);
    int result = lengthOfLongestSubstring(input);
    System.out.println("Length of Longest Substring Without
Repeating Characters: " + result);
  }
```

```
}
```

Output:

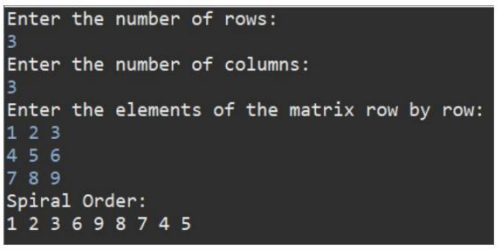
4. Spiral Matrix:

```
import java.util.ArrayList;
import java.util.List;

public class SpiralMatrix {
  public static List<Integer> spiralOrder(int[][] matrix) {
    List<Integer> result = new ArrayList<>();
    if (matrix == null || matrix.length == 0) {
      return result;
    }

    int top = 0;
    int bottom = matrix.length - 1;
    int left = 0;
    int right = matrix[0].length - 1;

    while (top <= bottom && left <= right) {
      // Traverse top row
      for (int i = left; i <= right; i++) {
        result.add(matrix[top][i]);
      }
      top++;

      // Traverse right column
```

```
        for (int i = top; i <= bottom; i++) {
            result.add(matrix[i][right]);
        }
        right--;

        // Traverse bottom row (if not already traversed)
        if (top <= bottom) {
            for (int i = right; i >= left; i--) {
                result.add(matrix[bottom][i]);
            }
            bottom--;
        }

        // Traverse left column (if not already traversed)
        if (left <= right) {
            for (int i = bottom; i >= top; i--) {
                result.add(matrix[i][left]);
            }
            left++;
        }
    }

    return result;
}

public static void main(String[] args) {
    int[][] matrix = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };
```

```
        System.out.println("Spiral Order: " +
spiralOrder(matrix));
    }
}
```

Output:



5. Next permutation:

```
import java.util.Arrays;

public class NextPermutation {
    public static void nextPermutation(int[] nums) {
        int n = nums.length;
        int i = n - 2;

        // Step 1: Find the first decreasing element
        while (i >= 0 && nums[i] >= nums[i + 1]) {
            i--;
        }

        if (i >= 0) {
            // Step 2: Find the next larger element to swap with
nums[i]
            int j = n - 1;
```

```
        while (nums[j] <= nums[i]) {
          j--;
        }
        // Swap nums[i] and nums[j]
        swap(nums, i, j);
      }

      // Step 3: Reverse the suffix starting from i + 1
      reverse(nums, i + 1, n - 1);
    }

    // Helper function to swap two elements in the array
    private static void swap(int[] nums, int i, int j) {
      int temp = nums[i];
      nums[i] = nums[j];
      nums[j] = temp;
    }

    // Helper function to reverse a subarray
    private static void reverse(int[] nums, int start, int end) {
      while (start < end) {
        swap(nums, start, end);
        start++;
        end--;
      }
    }

    public static void main(String[] args) {
      int[] nums = {1, 2, 3};
      System.out.println("Original Array: " +
  Arrays.toString(nums));
      nextPermutation(nums);
```

```
        System.out.println("Next Permutation: " +
Arrays.toString(nums));
    }
}
```

Output: