A) String Compression

```
public class StringCompression {

  public static String compressString(String input) {
    StringBuilder comp = new StringBuilder();
    int ct = 0;

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

      if (i + 1 >= input.length() || input.charAt(i) != input.charAt(i + 1)) {
        comp.append(input.charAt(i));
        comp.append(count);
        ct = 0;
      }
    }

    return comp.length() < input.length() ? comp.toString() : input;
  }

  public static void main(String[] args) {
    String input = "aabccccccaaa";
    String comp = compressString(input);
    System.out.println(comp);
  }
}
```

B) Linked list

```
class ListNode {
  int value;
  ListNode next;

  ListNode(int value) {
    this.value = value;
  }
}

public class KthToLastElement {

  public static int findKthToLast(ListNode head, int k) {
```

```
    ListNode slow = head;
    ListNode fast = head;

    // Move the fast pointer k nodes ahead
    for (int i = 0; i < k; i++) {
      if (fast == null) {
        return -1; // Handle the case where k is greater than the length of the list
      }
      fast = fast.next;
    }

    // Move both pointers until the fast one reaches the end
    while (fast != null && fast.next != null) {
      slow = slow.next;
      fast = fast.next;
    }

    if (slow == null) {
return -1; // Handle the case where k is greater than the length of the list
      }
      fast = fast.next;
    }

    // Move both pointers until the fast one reaches the end
    while (fast != null && fast.next != null) {
      slow = slow.next;
      fast = fast.next;
    }

    if (slow == null) {
      return -1;
    }

    return slow.value;
  }

  public static void main(String[] args) {
    ListNode node5 = new ListNode(5);
    ListNode node4 = new ListNode(4);
    ListNode node3 = new ListNode(3);
    ListNode node2 = new ListNode(2);
    ListNode node1 = new ListNode(1);
```

```
        node1.next = node2;
        node2.next = node3;
        node3.next = node4;
        node4.next = node5;


        int result = findKthToLast(node1, 2);
        System.out.println(result);
    }
}
```

C) Stack Minimum

```
    import java.util.Stack;

public class MinStack {
    private Stack<Integer> stack;
    private Stack<Integer> minStack;

    public MinStack() {
        stack = new Stack<>();
        minStack = new Stack<>();
    }

    public void push(int x) {
        stack.push(x);
        if (minStack.isEmpty() || x <= minStack.peek()) {
            minStack.push(x);
        }
    }

    public void pop() {
        if (!stack.isEmpty()) {
            if (stack.peek().equals(minStack.peek())) {
                minStack.pop();
            }
            stack.pop();
        }
    }

    public int top() {
        return stack.isEmpty() ? -1 : stack.peek();
    }
```

```
    public int getMin() {
        return minStack.isEmpty() ? -1 : minStack.peek();
    }
public static void main(String[] args) {
        MinStack minStack = new MinStack();
        minStack.push(3);
        minStack.push(5);
        System.out.println(minStack.getMin()); // Output: 3
        minStack.push(2);
        System.out.println(minStack.getMin()); // Output: 2
        minStack.pop();
        System.out.println(minStack.getMin()); // Output: 3
    }
}
```

D) Array Integers

```
public class TrappingRainWater {
    public static int trap(int[] height) {
        int n = height.length;
        if (n <= 2) {
            return 0; // No trapping possible with less than 3 positions
        }

        int left = 0, right = n - 1;
        int leftMax = 0, rightMax = 0;
        int trappedWater = 0;

        while (left < right) {
            leftMax = Math.max(leftMax, height[left]);
            rightMax = Math.max(rightMax, height[right]);

            if (height[left] < height[right]) {
                trappedWater += Math.max(0, leftMax - height[left]);
                left++;
            } else {
                trappedWater += Math.max(0, rightMax - height[right]);
                right--;
            }
        }

        return trappedWater;
    }
```

```
    public static void main(String[] args) {
        int[] elevation = {2, 1, 3, 0, 1, 2, 3};
        System.out.println("Amount of water trapped: " + trap(elevation)); // Output: 7
    }
}
```

E) Coin denominations

```
import java.util.*;

public class OptimalChange {
    public static List<Integer> findOptimalChange(int[] coins, int amount) {
        int[] dp = new int[amount + 1];
        int[] lastCoin = new int[amount + 1];
        Arrays.fill(dp, Integer.MAX_VALUE - 1);
        dp[0] = 0;

        for (int coin : coins) {
            for (int i = coin; i <= amount; i++) {
                if (dp[i - coin] + 1 < dp[i]) {
                    dp[i] = dp[i - coin] + 1;
                    lastCoin[i] = coin;
                }
            }
        }

        List<Integer> result = new ArrayList<>();
        int remaining = amount;
        while (remaining > 0) {
            int coin = lastCoin[remaining];
            result.add(coin);
            remaining -= coin;
        }

        return result;
    }

    public static void main(String[] args) {
        int[] availableCoins = {1, 2, 5, 8, 10};
        int change = 7;
        List<Integer> optimalChange = findOptimalChange(availableCoins, change);

        System.out.println("Optimal Change for " + change + ": " + optimalChange);
    }
}
```

F) Dot Product

The dot product and cross product are mathematical operations involving vectors.

Dot Product:
- Definition: The dot product of two vectors is the sum of the product of their corresponding components.
- Formula: $$\mathbf{A} \cdot \mathbf{B} = A_x \cdot B_x + A_y \cdot B_y + A_z \cdot B_z$$
- Use Cases in Graphics:
  - Illumination calculations, such as in shading models.
  - Determining the angle between two vectors.
  - Texture mapping.

Cross Product: *
- Definition: The cross product of two vectors results in a vector that is perpendicular to both input vectors.
- Formula: $$\mathbf{A} \times \mathbf{B} = \begin{bmatrix} A_y \cdot B_z - A_z \cdot B_y \\ A_z \cdot B_x - A_x \cdot B_z \\ A_x \cdot B_y - A_y \cdot B_x \end{bmatrix}$$
- Use Cases in Graphics:
  - Calculating surface normals.
  - Generating a third vector perpendicular to a plane defined by two vectors.
  - Implementing rotations and transformations.

For in-depth study, you can refer to resources like [Khan Academy](https://www.khanacademy.org/math/linear-algebra/vectors-and-spaces-dot-cross-products) and [Wolfram MathWorld](https://mathworld.wolfram.com/DotProduct.html) for dot product, and [Khan Academy](https://www.khanacademy.org/math/linear-algebra/vectors-and-spaces/cross-products/v/vector-cross-product-introduction) and [Wolfram MathWorld](https://mathworld.wolfram.com/CrossProduct.html) for cross product.

Bonus - Ray Intersection:
- For ray-plane intersection, you can refer to [Scratchapixel](https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-plane-intersection) or [Real-Time Rendering](http://www.realtimerendering.com/intersections.html).
- For ray-sphere intersection, resources like [Real-Time Rendering](http://www.realtimerendering.com/intersections.html) provide valuable information.
- Ray-triangle intersection details can be found in resources like [Tomas Akenine-Möller's Real-Time Rendering](http://www.realtimerendering.com/intersections.html) or [Scratchapixel](https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing/ray-triangle-intersection-geometric-solution).

G) The implementation of string is well known topic I known. The class building process and constructor implementation is one of my faviourate topics that I learned from coding ninjas and qspiders . Also with the help of oop concept it makes me easier to implement and understand the string concept very well. String topic is very unique as compared to other topics like array and the topic of string finds me very interesting .

H) Challenges

1. *Memory Issues:*
   - *Suspects:*
     - Uninitialized pointers.
     - Memory leaks.
     - Invalid memory access (buffer overflows/underflows).
   - *Isolation Steps:*
     - Run static code analysis tools to identify potential memory issues.
     - Use tools like Valgrind or AddressSanitizer to detect memory-related problems.
     - Check for proper memory deallocation.

2. *Exception Handling:*
   - *Suspects:*
     - Unhandled exceptions.
     - Improper exception handling.
   - *Isolation Steps:*
     - Ensure all potential exceptions are caught and handled appropriately.
     - Review exception logs.

3. *External Dependencies:*
   - *Suspects:*
     - Incorrect usage of external libraries.
     - Version mismatches.
   - *Isolation Steps:*
     - Check compatibility with external libraries.
     - Verify proper initialization and usage.

4. *Hardware/Environment Issues:*
   - *Suspects:*
     - Hardware failures.
     - Environmental factors (e.g., inconsistent network connectivity).
   - *Isolation Steps:*
     - Test on different hardware.
     - Monitor environmental conditions.

5. *Compiler and Optimization Settings:*
   - *Suspects:*
     - Compiler optimizations causing unexpected behavior.

- *Isolation Steps:*
  - Disable certain optimizations and retest.

9. *Operating System Compatibility:*
   - *Suspects:*
     - Code not compatible with the OS.
   - *Isolation Steps:*
     - Verify compatibility with the target operating system.
     - Check for OS-specific issues.