Strings

1. Reverse a String

To reverse a string, we can iterate from the end of the string to the beginning and construct a new string.

```
public class ReverseString {
    public static String reverse(String str) {
        char[] chars = str.toCharArray();
        String reversed = "";
        for (int i = chars.length - 1; i >= 0; i--) {
            reversed += chars[i];
        }
        return reversed;
    }

    public static void main(String[] args) {
        String str = "hello";
        System.out.println(reverse(str)); // Output: "olleh"
    }
}
```

2. Check if a String is a Palindrome

A string is a palindrome if it reads the same backward as forward.

```
public class PalindromeCheck {
    public static boolean isPalindrome(String str) {
        int left = 0;
        int right = str.length() - 1;

        while (left < right) {
            if (str.charAt(left) != str.charAt(right)) {
                return false;
            }
            left++;
            right--;
        }
        return true;
    }

    public static void main(String[] args) {
        String str = "madam";
        System.out.println(isPalindrome(str)); // Output: true
    }
}
```

3. Determine if Two Strings are Anagrams

Two strings are anagrams if they contain the same characters in the same frequency.

```
public class AnagramCheck {
    public static boolean areAnagrams(String str1, String str2) {
        if (str1.length() != str2.length()) {
            return false;
        }

        int[] charCount = new int[256];
        for (int i = 0; i < str1.length(); i++) {
            charCount[str1.charAt(i)]++;
            charCount[str2.charAt(i)]--;
        }

        for (int count : charCount) {
            if (count != 0) {
                return false;
            }
        }
        return true;
    }

    public static void main(String[] args) {
        String str1 = "listen";
        String str2 = "silent";
        System.out.println(areAnagrams(str1, str2)); // Output: true
    }
}
```

4. Find the Longest Substring Without Repeating Characters

We can use a sliding window technique to find the longest substring without repeating characters.

```
public class LongestSubstring {
    public static int longestSubstringWithoutRepeating(String str) {
        int maxLength = 0;
        for (int start = 0; start < str.length(); start++) {
            boolean[] seen = new boolean[256];
            int length = 0;
            for (int end = start; end < str.length(); end++) {
                if (seen[str.charAt(end)]) {
                    break;
                }
                seen[str.charAt(end)] = true;
                length++;
            }
            if (length > maxLength) {
                maxLength = length;
            }
        }
        return maxLength;
    }

    public static void main(String[] args) {
        String str = "abcabcbb";
        System.out.println(longestSubstringWithoutRepeating(str)); // Output:
3
```

```
    }
}
```

5. Compress a String by Counting Consecutive Characters

To compress a string, we count the consecutive characters and form a new string.

```
public class StringCompression {
    public static String compress(String str) {
        String compressed = "";
        int count = 1;

        for (int i = 0; i < str.length(); i++) {
            if (i + 1 < str.length() && str.charAt(i) == str.charAt(i + 1)) {
                count++;
            } else {
                compressed += str.charAt(i);
                compressed += count;
                count = 1;
            }
        }
        return compressed.length() < str.length() ? compressed : str;
    }

    public static void main(String[] args) {
        String str = "aabcccccaaa";
        System.out.println(compress(str)); // Output: "a2b1c5a3"
    }
}
```

6. Check if One String is a Rotation of Another

To check if one string is a rotation of another, we can concatenate the first string with itself and see if the second string is a substring of this concatenated string.

```
public class StringRotation {
    public static boolean isRotation(String s1, String s2) {
        if (s1.length() != s2.length()) {
            return false;
        }
        String concatenated = s1 + s1;
        return concatenated.contains(s2);
    }

    public static void main(String[] args) {
        String s1 = "waterbottle";
        String s2 = "erbottlewat";
        System.out.println(isRotation(s1, s2)); // Output: true
    }
}
```

7. Tokenize a String into Words

To tokenize a string into words, we can iterate through the string and split it based on spaces.

```
public class TokenizeString {
    public static String[] tokenize(String str) {
        int wordCount = 0;
        for (char c : str.toCharArray()) {
            if (c == ' ') {
                wordCount++;
            }
        }
        String[] words = new String[wordCount + 1];
        String word = "";
        int index = 0;
        for (char c : str.toCharArray()) {
            if (c == ' ') {
                words[index++] = word;
                word = "";
            } else {
                word += c;
            }
        }
        words[index] = word;
        return words;
    }

    public static void main(String[] args) {
        String str = "Hello World Java Programming";
        String[] tokens = tokenize(str);
        for (String token : tokens) {
            System.out.println(token);
        }
        // Output:
        // Hello
        // World
        // Java
        // Programming
    }
}
```

8. Generate All Permutations of a String

To generate all permutations of a string, we can use recursion.

```
public class StringPermutations {
    public static void permute(String str, String prefix) {
        if (str.length() == 0) {
            System.out.println(prefix);
        } else {
            for (int i = 0; i < str.length(); i++) {
                String rem = str.substring(0, i) + str.substring(i + 1);
                permute(rem, prefix + str.charAt(i));
            }
        }
    }
    public static void main(String[] args) {
        String str = "abc";
        permute(str, "");
        // Output:
        // abc
        // acb
        // bac
        // bca
        // cab
        // cba
    }
}
```

9. Count Vowels and Consonants in a String

To count vowels and consonants in a string, we can iterate through the string and check each character.

```
public class VowelConsonantCount {
    public static void countVowelsAndConsonants(String str) {
        int vowels = 0, consonants = 0;
        str = str.toLowerCase();
        for (char c : str.toCharArray()) {
            if (c >= 'a' && c <= 'z') {
                if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u')
{
                    vowels++;
                } else {
                    consonants++;
                }
            }
        }
        System.out.println("Vowels: " + vowels);
        System.out.println("Consonants: " + consonants);
    }

    public static void main(String[] args) {
        String str = "Hello World";
        countVowelsAndConsonants(str);
        // Output:
        // Vowels: 3
        // Consonants: 7
    }
}
```

10. Check if a String is a Pangram

A pangram is a string that contains every letter of the alphabet at least once.

```
public class PangramCheck {
    public static boolean isPangram(String str) {
        boolean[] seen = new boolean[26];
        str = str.toLowerCase();
        for (char c : str.toCharArray()) {
            if (c >= 'a' && c <= 'z') {
                seen[c - 'a'] = true;
            }
        }
        for (boolean b : seen) {
            if (!b) {
                return false;
            }
        }
        return true;
    }

    public static void main(String[] args) {
        String str = "The quick brown fox jumps over the lazy dog";
        System.out.println(isPangram(str)); // Output: true
    }
}
```

Let's move on to the array problems:

Arrays

1. Find Pairs of Integers in an Array that Sum up to a Target

To find pairs of integers that sum up to a target, we can use a nested loop.

```
public class PairSum {
    public static void findPairs(int[] arr, int target) {
        for (int i = 0; i < arr.length; i++) {
            for (int j = i + 1; j < arr.length; j++) {
                if (arr[i] + arr[j] == target) {
                    System.out.println("(" + arr[i] + ", " + arr[j] + ")");
                }
            }
        }
    }

    public static void main(String[] args) {
        int[] arr = {2, 7, 11, 15, -2};
        int target = 9;
        findPairs(arr, target);
        // Output:
        // (2, 7)
        // (11, -2)
    }
}
```

2. Remove Duplicates from a Sorted Array

To remove duplicates from a sorted array, we can use two pointers.

```
public class RemoveDuplicates {
    public static int removeDuplicates(int[] arr) {
        if (arr.length == 0) return 0;
        int uniqueIndex = 1;
        for (int i = 1; i < arr.length; i++) {
            if (arr[i] != arr[i - 1]) {
                arr[uniqueIndex++] = arr[i];
            }
        }
        return uniqueIndex;
    }

    public static void main(String[] args) {
        int[] arr = {1, 1, 2, 2, 3, 4, 4, 5};
        int newLength = removeDuplicates(arr);
        for (int i = 0; i < newLength; i++) {
            System.out.print(arr[i] + " ");
        }
        // Output: 1 2 3 4 5
    }
}
```

3. Rotate an Array to the Right by a Given Number of Steps

To rotate an array to the right by a given number of steps, we can use a reverse approach.

```
public class RotateArray {
    public static void reverse(int[] arr, int start, int end) {
        while (start < end) {
            int temp = arr[start];
            arr[start] = arr[end];
            arr[end] = temp;
            start++;
            end--;
        }
    }

    public static void rotate(int[] arr, int k) {
        int n = arr.length;
        k = k % n; // In case k is greater than n
        reverse(arr, 0, n - 1);
        reverse(arr, 0, k - 1);
        reverse(arr, k, n - 1);
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5, 6, 7};
        int k = 3;
        rotate(arr, k);
        for (int num : arr) {
            System.out.print(num + " ");
        }
        // Output: 5 6 7 1 2 3 4
    }
}
```

4. Merge Two Sorted Arrays into a Single Sorted Array

To merge two sorted arrays, we can use a two-pointer technique.

```
public class MergeSortedArrays {
    public static int[] merge(int[] arr1, int[] arr2) {
        int[] result = new int[arr1.length + arr2.length];
        int i = 0, j = 0, k = 0;

        while (i < arr1.length && j < arr2.length) {
            if (arr1[i] <= arr2[j]) {
                result[k++] = arr1[i++];
            } else {
                result[k++] = arr2[j++];
            }
        }

        while (i < arr1.length) {
            result[k++] = arr1[i++];
        }

        while (j < arr2.length) {
            result[k++] = arr2[j++];
        }

        return result;
    }

    public static void main(String[] args) {
        int[] arr1 = {1, 3, 5, 7};
        int[] arr2 = {2, 4, 6, 8};
        int[] result = merge(arr1, arr2);
        for (int num : result) {
            System.out.print(num + " ");
        }
        // Output: 1 2 3 4 5 6 7 8
    }
}
```

5. Find the Missing Number in an Array from 1 to N

To find the missing number in an array, we can use the sum formula for the first N natural numbers.

```
public class MissingNumber {
    public static int findMissingNumber(int[] arr, int n) {
        int expectedSum = n * (n + 1) / 2;
        int actualSum = 0;
        for (int num : arr) {
            actualSum += num;
        }
        return expectedSum - actualSum;
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 4, 5, 6};
        int n = 6;
        System.out.println(findMissingNumber(arr, n)); // Output: 3
    }
}
```

6. Find the Contiguous Subarray with the Largest Sum

To find the contiguous subarray with the largest sum, we can use Kadane's algorithm.

```
public class LargestSumSubarray {
    public static int maxSubArraySum(int[] arr) {
        int maxSoFar = arr[0];
        int maxEndingHere = arr[0];

        for (int i = 1; i < arr.length; i++) {
            maxEndingHere = Math.max(arr[i], maxEndingHere + arr[i]);
            maxSoFar = Math.max(maxSoFar, maxEndingHere);
        }

        return maxSoFar;
    }

    public static void main(String[] args) {
        int[] arr = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
        System.out.println(maxSubArraySum(arr)); // Output: 6
    }
}
```

7. Check if an Array Contains Duplicates

To check if an array contains duplicates, we can use a nested loop.

```
public class ContainsDuplicates {
    public static boolean containsDuplicates(int[] arr) {
        for (int i = 0; i < arr.length; i++) {
            for (int j = i + 1; j < arr.length; j++) {
                if (arr[i] == arr[j]) {
                    return true;
                }
            }
        }
        return false;
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5, 3};
        System.out.println(containsDuplicates(arr)); // Output: true
    }
}
```

8. Calculate the Product of Array Except Self

To calculate the product of the array except self, we can use two passes.

```
public class ProductExceptSelf {
    public static int[] productExceptSelf(int[] arr) {
        int n = arr.length;
        int[] leftProducts = new int[n];
        int[] rightProducts = new int[n];
        int[] result = new int[n];

        leftProducts[0] = 1;
        for (int i = 1; i < n; i++) {
            leftProducts[i] = leftProducts[i - 1] * arr[i - 1];
        }

        rightProducts[n - 1] = 1;
        for (int i = n - 2; i >= 0; i--) {
            rightProducts[i] = rightProducts[i + 1] * arr[i + 1];
        }

        for (int i = 0; i < n; i++) {
            result[i] = leftProducts[i] * rightProducts[i];
        }

        return result;
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4};
        int[] result = productExceptSelf(arr);
        for (int num : result) {
            System.out.print(num + " ");
        }
        // Output: 24 12 8 6
    }
}
```

9. Move Zeroes to the End of an Array

To move zeroes to the end of an array, we can use the two-pointer technique.

```
public class MoveZeroes {
    public static void moveZeroes(int[] arr) {
        int index = 0;
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] != 0) {
                arr[index++] = arr[i];
            }
        }
        while (index < arr.length) {
            arr[index++] = 0;
        }
    }

    public static void main(String[] args) {
        int[] arr = {0, 1, 0, 3, 12};
        moveZeroes(arr);
        for (int num : arr) {
            System.out.print(num + " ");
        }
        // Output: 1 3 12 0 0
    }
}
```

10. Find a Peak Element in an Array

A peak element is an element that is greater than its neighbors.

```
public class PeakElement {
    public static int findPeakElement(int[] arr) {
        for (int i = 0; i < arr.length; i++) {
            if ((i == 0 || arr[i] > arr[i - 1]) && (i == arr.length - 1 ||
arr[i] > arr[i + 1])) {
                return arr[i];
            }
        }
        return -1;
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 1};
        System.out.println(findPeakElement(arr)); // Output: 3
    }
}
```