1D Array Practice

1. Find the Largest Element in an Array

Problem: Write a Java program to find the largest element in a 1D array.

Approach:

1. Initialize a variable `max` to store the largest value, setting it to the first element of the array.
2. Traverse the array starting from the second element.
3. For each element, compare it with `max`. If the current element is greater, update `max`.
4. By the end of the traversal, `max` will contain the largest element.

Time Complexity: O(n)O(n)O(n)

Solution:

```
public class LargestElement {
    public static void main(String[] args) {
        int[] arr = {1, 2, 9, 4, 5};
        int max = arr[0];
        for (int i = 1; i < arr.length; i++) {
            if (arr[i] > max) {
                max = arr[i];
            }
        }
        System.out.println("Largest element: " + max);
    }
}
```

2. Reverse an Array

Problem: Write a Java program to reverse the elements of a 1D array.

Approach:

1. Use two pointers: one starting at the beginning (`i`) and one at the end (`n - i - 1`).
2. Swap the elements at these pointers.
3. Move the pointers towards the center until they meet or cross each other.
4. This results in the array being reversed.

Time Complexity: O(n)O(n)O(n)

Solution:

```
public class ReverseArray {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int n = arr.length;
        for (int i = 0; i < n / 2; i++) {
            int temp = arr[i];
            arr[i] = arr[n - i - 1];
            arr[n - i - 1] = temp;
        }
        for (int i : arr) {
            System.out.print(i + " ");
        }
    }
}
```

---

3. Sum of Elements in an Array

Problem: Write a Java program to calculate the sum of all elements in a 1D array.

Approach:

1. Initialize a variable `sum` to 0.
2. Iterate over the array and add each element to `sum`.
3. After the loop, `sum` will contain the total sum of all elements in the array.

Time Complexity: O(n)O(n)O(n)

Solution:

```
public class SumOfArray {
    public static void main(String[] args) {
        int[] arr = {5, 8, 3, 10};
        int sum = 0;
        for (int num : arr) {
            sum += num;
        }
        System.out.println("Sum of elements: " + sum);
    }
}
```

4. Count Occurrences of an Element

Problem: Write a Java program to count the occurrences of a given element in a 1D array.

Approach:

1. Initialize a counter variable `count` to 0.
2. Traverse the array and for each element, check if it matches the target element.
3. If there is a match, increment `count`.
4. By the end of the loop, `count` will have the total occurrences of the target element.

Time Complexity: O(n)O(n)O(n)

Solution:

```
public class CountOccurrences {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 2, 4, 2, 5};
        int target = 2;
        int count = 0;
        for (int num : arr) {
            if (num == target) {
                count++;
            }
        }
        System.out.println("Occurrences of " + target + ": "
+ count);
    }
}
```

---

5. Find the Second Largest Element

Problem: Write a Java program to find the second largest element in a 1D array.

Approach:

1. Initialize two variables, `largest` and `secondLargest`, with very low values (like `Integer.MIN_VALUE`).
2. Traverse the array and compare each element:
   - If the element is greater than `largest`, update `secondLargest` to be `largest` and `largest` to the current element.
   - If the element is not equal to `largest` but greater than `secondLargest`, update `secondLargest`.
3. This ensures that at the end, `secondLargest` will hold the second largest value.

Time Complexity: O(n)O(n)O(n)

Solution:

```
public class SecondLargest {
    public static void main(String[] args) {
        int[] arr = {3, 5, 7, 2, 8};
        int largest = Integer.MIN_VALUE, secondLargest = Inte
ger.MIN_VALUE;
        for (int num : arr) {
            if (num > largest) {
                secondLargest = largest;
                largest = num;
            } else if (num > secondLargest && num != largest)
{
                secondLargest = num;
            }
        }
        System.out.println("Second largest element: " + secon
dLargest);
    }
}
```

6. Check if Array is Sorted

Problem: Write a Java program to check if a 1D array is sorted in ascending order.

Approach:

1. Assume the array is sorted (`isSorted = true`).
2. Iterate through the array and compare adjacent elements.
3. If any element is greater than the next one, set `isSorted` to `false` and break the loop.
4. If no such case is found, the array is confirmed to be sorted.

Time Complexity: O(n)O(n)O(n)

Solution:

```
public class CheckSorted {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        boolean isSorted = true;
        for (int i = 0; i < arr.length - 1; i++) {
            if (arr[i] > arr[i + 1]) {
                isSorted = false;
                break;
            }
        }
        System.out.println("Array is sorted: " + isSorted);
    }
}
```

---

7. Move Zeros to End

Problem: Write a Java program to move all zeros to the end of the array while maintaining the order of non-zero elements.

Approach:

1. Use a variable `index` to track the position where the next non-zero element should be placed.
2. Traverse the array and whenever a non-zero element is found, place it at `index` and increment `index`.
3. After the traversal, fill the remaining positions from `index` to the end of the array with zeros.

Time Complexity: O(n)O(n)O(n)

Solution:

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

8. Remove Duplicates from a Sorted Array

Problem: Write a Java program to remove duplicates from a sorted 1D array.

Approach:

1. Use a variable `j` to track the position of the next unique element.
2. Iterate through the array and compare each element with the next one.
3. If the elements are not equal, copy the element at `i` to `j` and increment `j`.
4. After the loop, `j` will mark the length of the array with unique elements.

Time Complexity: O(n)O(n)O(n)

Solution:

```
public class RemoveDuplicates {
    public static void main(String[] args) {
        int[] arr = {1, 1, 2, 2, 3, 4, 4};
        int j = 0;
        for (int i = 0; i < arr.length - 1; i++) {
            if (arr[i] != arr[i + 1]) {
                arr[j++] = arr[i];
            }
        }
        arr[j++] = arr[arr.length - 1];
        for (int i = 0; i < j; i++) {
            System.out.print(arr[i] + " ");
        }
    }
}
```

9. Find Missing Number

Problem: Write a Java program to find the missing number in an array containing numbers from 1 to N.

Approach:

1. Use the formula for the sum of the first `N` natural numbers: `totalSum = n * (n + 1) / 2`, where `n` is the length of the array plus one.
2. Calculate the actual sum of the array elements (`arraySum`).
3. The missing number will be the difference between `totalSum` and `arraySum`.

Time Complexity: O(n)O(n)O(n)

Solution:

```
public class FindMissingNumber {
```

```
    public static void main(String[] args) {
        int[] arr = {1, 2, 4, 6, 3, 7, 8};
        int n = arr.length + 1;
        int totalSum = n * (n + 1) / 2;
        int arraySum = 0;
        for (int num : arr) {
            arraySum += num;
        }
        System.out.println("Missing number: " + (totalSum - a
rraySum));
    }
}
```

---

10. Rotate Array by K Positions

Problem: Write a Java program to rotate a 1D array by `k` positions to the right.

Approach:

1. Use a three-step reversal technique:
    - Reverse the entire array.
    - Reverse the first `k` elements.
    - Reverse the remaining `n - k` elements.
2. This approach allows the array to be rotated in-place, avoiding the need for additional arrays.

Time Complexity: O(n)O(n)O(n)

Solution:

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

```
        int n = arr.length;
        k = k % n;
        reverse(arr, 0, n - 1);
        reverse(arr, 0, k - 1);
        reverse(arr, k, n - 1);
        for (int num : arr) {
            System.out.print(num + " ");
        }
    }

    private static void reverse(int[] arr, int start, int en
d) {
        while (start < end) {
            int temp = arr[start];
            arr[start] = arr[end];
            arr[end] = temp;
            start++;
            end--;
        }
    }
}
```