Problem 1.1: Find the Largest Number in an Array

Overview: The goal is to find the largest number in an unsorted array.

Pseudocode Explanation:

1. **Initialize**: Start with the first element of the array as the maximum (`max`).
2. **Iterate**: Loop through each element of the array (`A`).
3. **Compare and Update**: If the current element is greater than `max`, update `max` with this element.
4. **Result**: After the loop, `max` will hold the largest number in the array.

```
Algorithm FindLargest(A):
    max = A[0]
    for each element in A:
        if element > max:
            max = element
    return max
```

Problem 1.2: Find the Second Largest Number in an Array

Overview: The aim is to find the second largest number in an unsorted array.

Pseudocode Explanation:

1. **Initialize**: Start with two variables, `max` and `secondMax`, both set to very small values (like negative infinity).
2. **Iterate**: Loop through each element in the array.
3. **Update Max**: If the current element is greater than `max`, update `secondMax` to `max`'s value and then update `max` with the current element.
4. **Update Second Max**: If the current element is less than `max` but greater than `secondMax`, update `secondMax`.
5. **Result**: After the loop, `secondMax` will hold the second largest number.

```
Algorithm FindSecondLargest(A):
    max = -Infinity
    secondMax = -Infinity
    for each element in A:
        if element > max:
            secondMax = max
            max = element
        else if element > secondMax and element != max:
            secondMax = element
    return secondMax
```

Problem 2: Find the Missing Number in an Array

Overview: Find the missing number in an array containing unique integers between 0 and N.

Pseudocode Explanation:

1. **Calculate Expected Sum**: Compute the sum of the first N natural numbers using the formula `N*(N+1)/2`.
2. **Sum Array Elements**: Calculate the sum of all elements in the array.
3. **Find Missing Number**: Subtract the sum of array elements from the expected sum to get the missing number.

```
Algorithm FindMissingNumber(A, N):
    expectedSum = N * (N + 1) / 2
    actualSum = 0
    for each element in A:
        actualSum += element
    return expectedSum - actualSum
```

Problem 3: Check if Two Sequences are Permutations of the Same Set

Overview: Determine whether two sequences are permutations of each other.

Pseudocode Explanation:

1. **Size Check**: If the sizes of the two sequences are different, they cannot be permutations of each other.
2. **Create Sets**: Convert each sequence into a set to remove duplicates.
3. **Size Comparison**: If the sizes of the sets are different, return NO.
4. **Element Comparison**: Check if every element of one set is present in the other. If any element is not found, return NO.
5. **Result**: If all elements match, the sequences are permutations of each other.

```
Algorithm ArePermutations(Seq1, Seq2):
    if size(Seq1) != size(Seq2):
        return NO
    Set1 = new Set()
    Set2 = new Set()
    for each element in Seq1:
        Set1.add(element)
    for each element in Seq2:
        Set2.add(element)
    if Set1.size() != Set2.size():
        return NO
    for each element in Set1:
        if not Set2.contains(element):
            return NO
    return YES
```

Problem 4: Sum of a Range in an Array

Overview: Efficiently calculate the sum of elements in a given range of an array.

Pseudocode Explanation:

1. **Preprocessing - InitializePrefixSum**:
   - Create an array `PrefixSum` to store the cumulative sum up to each index.
   - Iterate through the array, updating `PrefixSum` such that each element at index `i` is the sum of all elements from 0 to `i` in the original array.
2. **RangeSum Calculation**:
   - To get the sum of a subarray from `L` to `R`, subtract the cumulative sum up to `L - 1` from the cumulative sum up to `R`.
   - This is efficient because it uses precomputed sums and requires only a constant time operation for each range sum query.

```
Algorithm InitializePrefixSum(A):
    N = size(A)
    PrefixSum[0..N] = new Array
    PrefixSum[0] = 0
    for i from 1 to N:
        PrefixSum[i] = PrefixSum[i - 1] + A[i - 1]
    return PrefixSum

Algorithm RangeSum(PrefixSum, L, R):
    return PrefixSum[R + 1] - PrefixSum[L]
```