Remove Duplicates Algorithm

Give DSA notes along with this code.

```
public int removeDuplicates(int[] arr) {
    // Index to track the position of the last unique element
    int uniqueIndex = 0;

    // Iterate through the array starting from the second element
    for (int i = 1; i < arr.length; i++) {
        // Check if the current element is different from the last unique element
        if (arr[i] != arr[uniqueIndex]) {
            // Increment the unique index and update it with the current unique element
            arr[++uniqueIndex] = arr[i];
        }
    }

    // Return the count of unique elements in the array
    return uniqueIndex + 1;
}
```

DSA Notes: Removing Duplicates from a Sorted Array

Problem Overview:

The problem involves removing duplicate elements from a sorted array in-place, such that each element appears only once. After removing duplicates, the array will contain only the unique elements, and we need to return the count of unique elements.

Approach:

1. **Input Constraints**:
    - The input is a sorted array of integers, which makes it easier to identify duplicates since any repeated elements will be adjacent.
2. **Two Pointers Approach**:
    - **Pointer 1** (`uniqueIndex`): This pointer tracks the position of the last unique element in the array. Initially, it points to the first element.

   - **Pointer 2** (`i`): This pointer scans through the entire array from the second element to the end.

3. **Logic**:
   - We loop through the array starting from the second element (`i = 1`), comparing each element with the last unique element (pointed to by `uniqueIndex`).
   - If the current element is different from the last unique element, it is added as a new unique element.
   - We increment the `uniqueIndex` and move the new unique element to its correct position.
   - At the end of the iteration, `uniqueIndex + 1` will give the number of unique elements.

4. **Edge Case**:
   - If the array is empty, the number of unique elements is `0` .

Time Complexity:

- **Time Complexity**: O(n), where `n` is the length of the array. We only loop through the array once.
- **Space Complexity**: O(1), as we are modifying the array in-place without using extra space.

Code:

```java
public int removeDuplicates(int[] arr) {
    // Index to track the position of the last unique element
    int uniqueIndex = 0;

    // Iterate through the array starting from the second element
    for (int i = 1; i < arr.length; i++) {
        // Check if the current element is different from the last unique element
        if (arr[i] != arr[uniqueIndex]) {
            // Increment the unique index and update it with the current unique element
            arr[++uniqueIndex] = arr[i];
        }
    }
    // Return the count of unique elements in the array
    return uniqueIndex + 1;
}
```

Explanation of the Code:

1. **Initialization**:
   - `uniqueIndex = 0`: This points to the first element in the array, which is always unique initially.
2. **Loop**:
   - The loop starts from `i = 1`, because we assume the first element is unique.
   - For each element at index `i`, we compare it with `arr[uniqueIndex]`.
   - If they are different, we increment `uniqueIndex` and assign `arr[i]` to `arr[uniqueIndex]`.
3. **Return Value**:
   - After the loop, `uniqueIndex + 1` gives the count of unique elements, as `uniqueIndex` represents the index of the last unique element.

Example Walkthrough:

Consider the input array: `[1, 1, 2, 3, 3, 4, 5]`

- Initially, `uniqueIndex = 0`, so the first element `1` is considered unique.
- The second element `1` is the same as the first, so we skip it.
- The third element `2` is different from `1`, so `uniqueIndex` is incremented to `1`, and `arr[1]` is set to `2`.
- The fourth element `3` is different from `2`, so `uniqueIndex` is incremented to `2`, and `arr[2]` is set to `3`.
- The fifth element `3` is the same as the previous `3`, so we skip it.
- The sixth element `4` is different from `3`, so `uniqueIndex` is incremented to `3`, and `arr[3]` is set to `4`.
- The seventh element `5` is different from `4`, so `uniqueIndex` is incremented to `4`, and `arr[4]` is set to `5`.

Final array after duplicates are removed: `[1, 2, 3, 4, 5, _, _]`

- The number of unique elements is `5`.

Conclusion:

This approach ensures that duplicates are removed efficiently with minimal space usage, and the final count of unique elements is obtained.