Sort Colors Algorithm

Give DSA notes along with code public static void sortColors(int[] arr) {
    int low = 0, mid = 0, high = arr.length - 1;

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

      if (arr[mid] == 0) {
        arr[mid++] = arr[low];
        arr[low++] = 0;
      } else if (arr[mid] == 1) {
        ++mid;
      } else if (arr[mid] == 2) {
        arr[mid] = arr[high];
        arr[high] = 2;
        --high;
      }

    }
  }

DSA Notes: Dutch National Flag Problem

The given code is an implementation of the **Dutch National Flag Problem** or **Sort Colors Problem**. The goal is to sort an array of integers where the values are only 0, 1, and 2 in a single pass (O(n) time complexity) and with constant extra space (O(1)).

Problem Description

You are given an array containing only three distinct integers: 0, 1, and 2. Write a function to sort this array in ascending order.

Approach

This problem can be efficiently solved using the **Three Pointers Approach**:

1. **Pointers:**
   - `low` points to the next position where `0` should go.
   - `mid` scans through the array.
   - `high` points to the next position where `2` should go.
2. **Logic:**
   - Traverse the array with `mid`.
   - Swap elements to their correct positions:
     - If `arr[mid] == 0`, swap `arr[low]` with `arr[mid]`, then increment both `low` and `mid`.
     - If `arr[mid] == 1`, just increment `mid`.
     - If `arr[mid] == 2`, swap `arr[mid]` with `arr[high]` and decrement `high` (do not increment `mid` here to re-evaluate the swapped element).
3. **Invariant:**
   - At any point in the traversal:
     - All elements before `low` are `0`.
     - All elements after `high` are `2`.
     - Elements between `low` and `mid` are `1`.

Code Implementation (Java)

java

```java
public static void sortColors(int[] arr) {
    int low = 0, mid = 0, high = arr.length - 1;

    while (mid <= high) {
        if (arr[mid] == 0) {
            // Swap arr[mid] and arr[low], increment both pointers
            int temp = arr[low];
            arr[low] = arr[mid];
            arr[mid] = temp;
            low++;
```

```
            mid++;
        } else if (arr[mid] == 1) {
            // No swap needed, just increment mid
            mid++;
        } else if (arr[mid] == 2) {
            // Swap arr[mid] and arr[high], decrement high
            int temp = arr[mid];
            arr[mid] = arr[high];
            arr[high] = temp;
            high--;
        }
    }
}
```

Dry Run Example

Input: `arr = [2, 0, 2, 1, 1, 0]`

Process:

1. Initial state: `low = 0`, `mid = 0`, `high = 5`
2. `arr[mid] = 2`: Swap with `arr[high]`, decrement `high`.
   - Array: `[0, 0, 2, 1, 1, 2]`
   - `low = 0`, `mid = 0`, `high = 4`
3. `arr[mid] = 0`: Swap with `arr[low]`, increment `low` and `mid`.
   - Array: `[0, 0, 2, 1, 1, 2]`
   - `low = 1`, `mid = 1`, `high = 4`
4. `arr[mid] = 0`: Swap with `arr[low]`, increment `low` and `mid`.
   - Array: `[0, 0, 2, 1, 1, 2]`
   - `low = 2`, `mid = 2`, `high = 4`
5. `arr[mid] = 2`: Swap with `arr[high]`, decrement `high`.
   - Array: `[0, 0, 1, 1, 2, 2]`
   - `low = 2`, `mid = 2`, `high = 3`

6. `arr[mid] = 1`: Increment `mid`.
    - Array: `[0, 0, 1, 1, 2, 2]`
    - `low = 2`, `mid = 3`, `high = 3`
7. `arr[mid] = 1`: Increment `mid`.
    - Array: `[0, 0, 1, 1, 2, 2]`
    - `low = 2`, `mid = 4`, `high = 3`

Output: `[0, 0, 1, 1, 2, 2]`

Complexity Analysis

- **Time Complexity:** O(n) — Each element is processed at most once.
- **Space Complexity:** O(1) — No extra space is used.

Key Takeaways

1. This approach is optimal for sorting arrays with only three unique values.
2. It avoids using a comparison-based sorting algorithm (like QuickSort or MergeSort).
3. The code is in-place, saving memory.

Let me know if you need more examples or alternative approaches!

2/2