Rearranging Array Alternation

```
Give DSA notes along with this code: /**
 * Rearranges the given array such that positive and negative numbers alternate.
 * Assumes the input array contains an equal number of positive and negative numbers.
 *
 * @param nums the input array of integers
 * @return a rearranged array with alternating positive and negative numbers
 *         starting with a positive number at index 0
 */
public int[] rearrangeArray(int[] nums) {
    // Initialize result array
    int[] rearrangedArray = new int[nums.length];
    int positiveIndex = 0, negativeIndex = 1;

    // Iterate through the input array
    for (int val : nums) {
        if (val > 0) {
            rearrangedArray[positiveIndex] = val;
            positiveIndex += 2;
        } else {
            rearrangedArray[negativeIndex] = val;
            negativeIndex += 2;
        }
    }

    return rearrangedArray;
}
```

DSA Notes on Rearranging Array with Alternating Positive and Negative Numbers

This code solves the problem of rearranging an array such that positive and negative numbers alternate, starting with a positive number at index 0. Below are the detailed notes on the logic and its DSA concepts:

Key Concepts Covered:

1. **Two-Pointer Technique**:
    - The code uses two pointers, `positiveIndex` and `negativeIndex`, which point to the next available position for positive and negative numbers respectively.
    - `positiveIndex` starts at 0 (even indices), and `negativeIndex` starts at 1 (odd indices).
2. **Iterative Traversal**:
    - Each element in the array is traversed exactly once in a `for` loop, ensuring an $O(n)$ time complexity.
3. **Array Index Management**:
    - The `positiveIndex` and `negativeIndex` pointers are incremented by 2 to ensure alternating placement of positive and negative numbers.
4. **Assumptions and Constraints**:
    - It is assumed that the input array contains an equal number of positive and negative integers.
    - If the input does not meet this condition, additional logic would be required to handle the imbalance.

Code Walkthrough:

1. **Initialization**:
    - A new array `rearrangedArray` is created to store the output.
    - Two pointers, `positiveIndex` and `negativeIndex`, are initialized at 0 and 1, respectively.
2. **Iteration through the Input Array**:
    - Each number in the input array is checked:
        - If the number is positive, it is placed at the current `positiveIndex`, and `positiveIndex` is incremented by 2.
        - If the number is negative, it is placed at the current `negativeIndex`, and `negativeIndex` is incremented by 2.

3. **Output**:
    - The rearranged array is returned.

Time and Space Complexity:

1. **Time Complexity**:
    - $O(n)$: The array is traversed once, where n is the size of the array.
2. **Space Complexity**:
    - $O(n)$: A new array of the same size as the input is created to store the rearranged result.

Example Input/Output:

Input:

`nums = [3, -1, 2, -2, -3, 4]`

Output:

`[3, -1, 2, -2, 4, -3]`

Explanation:
The array is rearranged such that positive and negative numbers alternate, starting with a positive number.

Edge Cases:

1. **Balanced Input**:
    - `[1, -1, 2, -2]`: Works as expected.
2. **Unbalanced Input**:
    - If there are unequal numbers of positives and negatives, the logic will fail unless modified to handle such cases.

3. **All Positive or All Negative**:
   - For input like `[1, 2, 3]` or `[-1, -2, -3]`, the code requires a pre-check to ensure the alternating condition is achievable.

Potential Enhancements:

1. **Handle Unequal Counts**:
   - Use a separate logic to append remaining positive or negative numbers at the end if they cannot alternate completely.
2. **In-place Rearrangement**:
   - Modify the input array directly instead of using additional space for a new array (if allowed).

By using the provided code and understanding its logic, you can efficiently solve problems involving alternating arrangements of array elements.

2/2