# 75. Sort Colors

Medium Array Two Pointers Sorting

## Problem Description

In this problem, you are given an array `nums` which contains `n` elements. Each element represents a color coded as an integer: `0` for red, `1` for white, and `2` for blue. Your task is to sort this array in a way that the colors are grouped together and in the order of red, white, and blue. The sorting has to be done in-place, without using any extra space for another array, and you cannot use the library's sort function.

## Intuition

To solve this problem, the solution approach uses a variant of the famous Dutch National Flag algorithm proposed by Edsger Dijkstra. The crux of this algorithm is a three-way partitioning technique that segments an array into three parts to sort the elements of three different types.

In this case, we will maintain three pointers:

1. `i` - All elements before index `i` are `0`s (reds).
2. `j` - All elements from index `j` onwards are `2`s (blues).
3. `k` - Current element that is being inspected.

Initially, `i` is set to `-1`, indicating there are no `0`s in the beginning, and `j` is set to the length of `nums`, indicating there are no `2`s at the end. The `k` pointer will start from `0` and move towards `j`.

We iterate through the array with `k`, and when we find a `0`, we increment `i` and swap the values at `i` and `k`. If we find a `2`, we decrement `j` and swap the values at `k` and `j` but we don't move the pointer `k` because the new element we swapped from position `j` might be `0`, so it needs to be rechecked. If the element is `1`, it's already in its correct place since we are ensuring all `0`s and `2`s are moved to their correct places. So, for `1`, we just move `k` forward.

We continue this process until `k` meets `j`, at which point all elements to the left of `i` are `0`s, elements between `i` and `j` are `1`s, and all elements from `j` onwards are `2`s, resulting in a sorted array.

## Solution Approach

The solution implements the Dutch National Flag algorithm, which is a partitioning strategy.

Here's a step-by-step explanation using the Reference Solution Approach:

1. Initialize three pointers (`i`, `j`, and `k`):
   - `i` starts just before the array at `-1`. This will eventually track the position up to which `0`s have been sorted.
   - `j` starts after the end of the array at `len(nums)`. This will eventually track the position from which `2`s have been sorted.
   - `k` starts at `0` and is used to iterate through the array.
2. Perform iterations while `k < j`:
   - If `nums[k] == 0`, this element needs to be moved to the front.
     - Increment `i` to move it to the next unsorted position.
     - Swap the elements at `i` and `k` (`nums[i], nums[k] = nums[k], nums[i]`), effectively moving the `0` to its correct place.
     - Increment `k` to move on to the next element.
   - Else, if `nums[k] == 2`, this element needs to be moved to the end.
     - Decrement `j` to move it towards the first unsorted position from the end.
     - Swap the elements at `k` and `j` (`nums[j], nums[k] = nums[k], nums[j]`), moving the `2` closer to its correct place. Here we don't increment `k` because the newly swapped element could be `0` or `1` and it has not been evaluated yet.
   - If `nums[k] == 1`, no action is needed as `1`s are automatically sorted when `0`s and `2`s are moved to their correct places.
     - Simply increment `k` to continue to the next element.

By following this approach, we continue to partition the array into three parts: `0`s before `i`, `1`s between `i` and `j`, and `2`s after `j`. The loop continues until `k` becomes equal to `j`, meaning all elements have been examined and placed in their correct position. Therefore, the array is now sorted in-place with red (`0`), white (`1`), and blue (`2`) colors in the correct order without using any additional space or the library sort function.

### Example Walkthrough

Let's say we have an array `nums` as `[2, 0, 1, 2, 1, 0]`. We need to sort this array using the Dutch National Flag algorithm so that all `0`s (reds) come first, followed by `1`s (whites), and then `2`s (blues).

Here's a step-by-step process of how the algorithm would sort this array:

1. Initialize the pointers `i`, `j`, and `k`:
   - `i` is set to `-1`
   - `j` is set to `6` (since the array length is `6`)
   - `k` is set to `0`
2. Start iterating with `k` while `k < j` (while `k` is less than `6`):
   - Iteration 1:
     - `nums[k]` is `2`. Since `k==0`, we need to move this `2` to the end.
     - We decrement `j` to `5`.
     - We swap `nums[k]` with `nums[j]`. So the array becomes `[0, 0, 1, 2, 1, 2]`.
     - We don't increment `k` as we need to evaluate the swapped element.
   - Iteration 2:
     - Now `nums[k]` is `0`. This needs to go at the beginning.
     - We increment `i` to `0`.
     - We swap `nums[i]` with `nums[k]`. The array is still `[0, 0, 1, 2, 1, 2]` since both `nums[i]` and `nums[k]` are `0`.
     - We increment `k` to `1`.
   - Iteration 3:
     - `nums[k]` is another `0`.
     - We increment `i` to `1`.
     - We swap `nums[i]` with `nums[k]`, but the array remains unchanged `[0, 0, 1, 2, 1, 2]` as they are the same value.
     - Increment `k` to `2`.
   - Iteration 4:
     - `nums[k]` is `1`. This is already in the correct position.
     - We simply increment `k` to `3`.
   - Iteration 5:
     - `nums[k]` is `2`, needs to move to the end.
     - We decrement `j` to `4`.
     - We swap `nums[k]` with `nums[j]`. Now the array looks like `[0, 0, 1, 1, 2, 2]`.
     - Do not increment `k` as we need to evaluate the swapped element.
   - Iteration 6:
     - `nums[k]` is now `1`. It should stay in place.
     - Increment `k` to `4`. Now `k == j`, so we stop.

The final sorted array is `[0, 0, 1, 1, 2, 2]`, with all the colors grouped together in the correct order without using any extra space or sorting functions.

## Python Solution

```python
class Solution:
    def sortColors(self, nums: List[int]) -> None:
        # Initialize pointers for the next position of 0, the next position of 2, and the current element
        next_zero_index, next_two_index, current_index = -1, len(nums), 0

        # Process elements until the current_index reaches the next_two_index
        while current_index < next_two_index:
            if nums[current_index] == 0:
                # Move the 0 to the next position for 0
                next_zero_index += 1
                nums[next_zero_index], nums[current_index] = nums[current_index], nums[next_zero_index]
                # Move to the next element
                current_index += 1
            elif nums[current_index] == 2:
                # Move the 2 to the next position for 2
                next_two_index -= 1
                nums[next_two_index], nums[current_index] = nums[current_index], nums[next_two_index]
                # Do not increment current_index because we need to check the newly swapped element
            else:
                # If the current element is a 1, just move to the next element
                current_index += 1
        # The function modifies the list in place, so there is no return value
```

## Java Solution

```java
class Solution {

    // Method to sort the array containing 0s, 1s, and 2s
    public void sortColors(int[] nums) {
        // Initialize pointers for the current element (currIndex),
        // the last position of 0 (lastZeroIndex) and the first position of 2 (firstTwoIndex)
        int lastZeroIndex = -1;
        int firstTwoIndex = nums.length;
        int currIndex = 0;

        // Process elements until currIndex reaches firstTwoIndex
        while (currIndex < firstTwoIndex) {
            if (nums[currIndex] == 0) {
                // If the current element is 0, swap it to the position after the last 0 we found
                swap(nums, ++lastZeroIndex, currIndex++);
            } else if (nums[currIndex] == 2) {
                // If the current element is 2, swap it with the element at the position
                // just before the first 2 we found
                swap(nums, --firstTwoIndex, currIndex);
            } else {
                // If the current element is 1, just move to the next element
                ++currIndex;
            }
        }
    }

    // Helper method to swap two elements in an array
    private void swap(int[] nums, int i, int j) {
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }
}
```

## C++ Solution

```cpp
#include <vector>
using namespace std;

class Solution {
public:
    // This function is used to sort the colors, represented by numbers 0, 1, and 2.
    // It uses the Dutch National Flag algorithm to sort in place with O(n) complexity.
    void sortColors(vector<int>& nums) {
        // Initialize pointers:
        // 'left' is the position where the next 0 should go,
        // 'right' is the position one more than where the next 2 should go,
        // 'current' is the current index being considered.
        int left = -1, right = nums.size(), current = 0;

        while (current < right) { // Process elements until 'current' reaches 'right'
            if (nums[current] == 0) {
                // When a 0 is found, swap it with the element at 'left' position,
                // then move both 'left' and 'current' one step right.
                swap(nums[++left], nums[current++]);
            } else if (nums[current] == 2) {
                // When a 2 is found, swap it with the element just before 'right' position,
                // then decrement 'right' to move it leftward.
                // Note 'current' is not incremented because the swapped element needs to be checked.
                swap(nums[--right], nums[current]);
            } else {
                // If the element is 1, just move 'current' one step to the right.
                ++current;
            }
        }
    }
};
```

## Typescript Solution

```typescript
/**
 * Sorts an array of numbers in-place, so that all 0s come first,
 * followed by all 1s, and then all 2s. This pattern is known as the Dutch national flag problem.
 *
 * @param {number[]} nums - The input array containing 0s, 1s, and 2s.
 */
function sortColors(nums: number[]): void {
    let zeroIndex = -1; // Initialize the index where 0s will be placed.
    let twoIndex = nums.length; // Initialize the index where 2s will be placed.
    let currentIndex = 0; // The current index we're scanning from the array.

    while (currentIndex < twoIndex) {
        if (nums[currentIndex] === 0) {
            // When the current element is 0, swap it with the element at zeroIndex,
            // then increment zeroIndex and currentIndex.
            zeroIndex++;
            [nums[zeroIndex], nums[currentIndex]] = [nums[currentIndex], nums[zeroIndex]];
            currentIndex++;
        } else if (nums[currentIndex] === 2) {
            // When the current element is 2, decrement twoIndex and swap the current element
            // with the element at twoIndex.
            twoIndex--;
            [nums[twoIndex], nums[currentIndex]] = [nums[currentIndex], nums[twoIndex]];
            // Do not increment currentIndex here because the element swapped from twoIndex
            // may be 0, which will need to be moved to zeroIndex in the next iteration.
        } else {
            // If the element is 1, just move on to the next element.
            currentIndex++;
        }
    }
}
```

## Time and Space Complexity

The time complexity of the code is `O(n)`, where `n` is the length of the input list `nums`. This is because the while loop iterates through each element of the list at most once. The variables `i`, `j`, and `k` are used to traverse the array without the need to revisit elements. The increment and decrement operations on `i`, `j`, and `k`, as well as the swaps, all occur in constant time, and the loop runs until `k` is no longer less than `j`.

The space complexity of the code is `O(1)` because the sorting is done in place. No additional storage is needed that scales with the input size `n`. The only extra space used is for the three pointers `i`, `j`, and `k`, which use a constant amount of space regardless of the size of the input list.