Solution: Sort Colors

Let's solve the Sort Colors problem using the Two Pointers pattern.

We'll cover the following

Statement

Given an array, `colors`, which contains a combination of the following three elements:

- 0 (representing red)
- 1 (representing white)
- 2 (representing blue)

Sort the array in place so that the elements of the same color are adjacent, with the colors in the order of red, white, and blue. To improve your problem-solving skills, do not utilize the built-in sort function.

Constraints:

- $1 \leq$ `colors.length` ≤ 300
- `colors[i]` can only contain 0s, 1s, or 2s.

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive approach

The naive approach would be to sort the array. This would arrange the elements in the desired positions, i.e., 0s then 1s, and lastly, 2s. The time complexity of this approach would be $O(nlog(n))$, which is the time required to sort the array. The space complexity of this approach would be $O(1)$ since no extra space is being used.

Optimized approach using two pointers

The essence of the algorithm lies in efficiently partitioning the array into three sections corresponding to the colors red, white, and blue. We keep track of the boundaries of the red and blue sections while iterating through the array. Reds are kept on the left side of the array, while blues remain on the right side, with whites in between. When encountering a red (0), it's placed at the end of the red section, and when finding a blue (2), it's moved to the beginning of the blue section. Elements of value 1 (white) remain in place. This process continues until all the colors are separated, ensuring proper grouping of colors.

To implement it, we use two pointers (`start` and `end`) to traverse the array from either end. They keep track of the red and blue elements, respectively. In addition, we maintain another pointer (`current`) to keep track of the white elements. These pointers are used to traverse the array in one pass. They are initialized as follows:

- `start`: This pointer will initially point to the 0^{th} index of the array.
- `current`: This pointer will initially point to the 0^{th} index of the array.
- `end`: This pointer will initially point to the last index of the array.

Here's how the algorithm works:

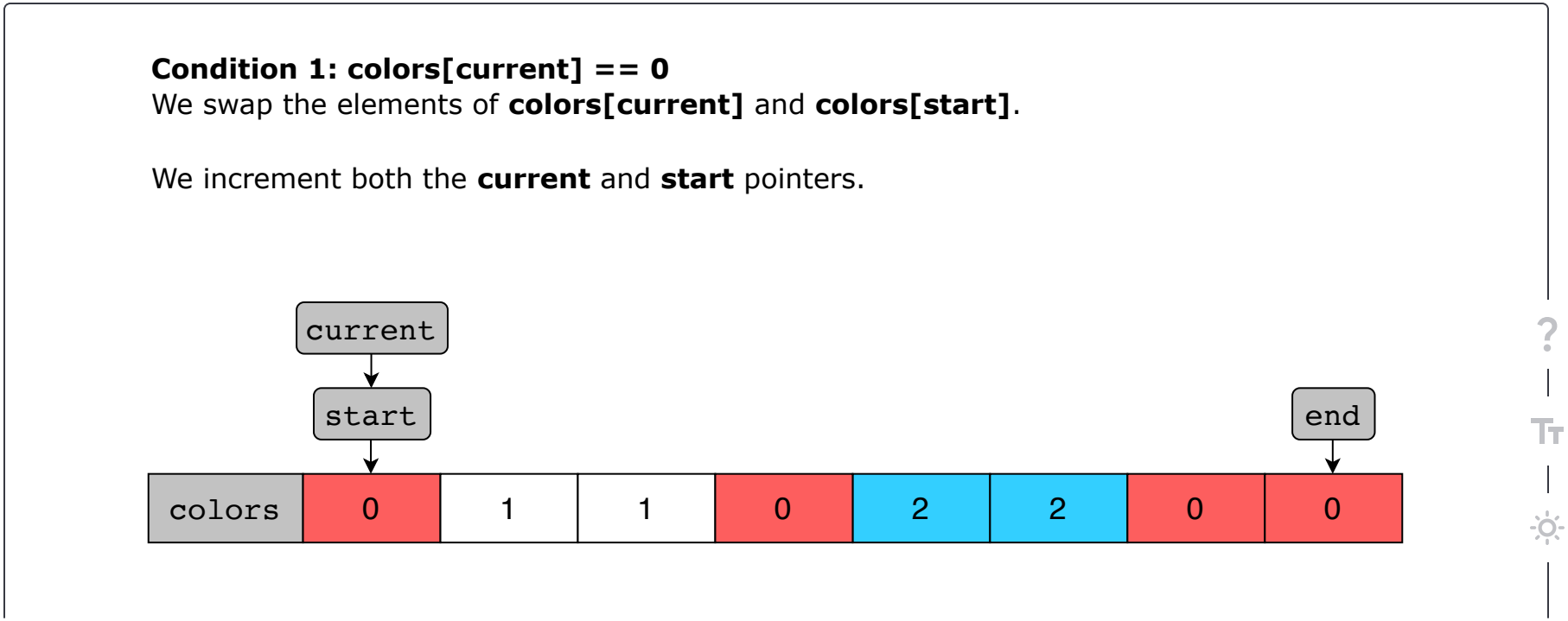
- **Condition 1:** If `colors[current]` is `0`, the `current` pointer points to red. Swap the elements of `colors[current]` and `colors[start]`. Next, move both the `start` and `current` pointers one position forward.
- **Condition 2:** Otherwise, if `colors[current]` is `1`, the `current` pointer points to white. Increment the `current` pointer by `1` to analyze the next element.

- **Condition 3:** Otherwise, `colors[current]` will be 2, i.e., the `current` pointer points to blue. Swap the elements of `colors[current]` and `colors[end]`. Next, move the `end` pointer one position backward.

> **Note:** When we decrement the `end` pointer, the `current` pointer remains unchanged since it has to analyze the swapped element to determine if further swapping is required with the `start` pointer.

- The three steps above are repeated until the `end` pointer becomes less than the `current` pointer, i.e., no elements are left to swap.

Let's look at the following illustration to get a better understanding of the solution:



Let's have a look at the code for the algorithm we just discussed:

Solution | Commented Solution

```
import java.util.*;

class Solution {
    public static int[] sortColors(int[] colors) {

        int start = 0;
        int current = 0;
        int end = colors.length - 1;

        while (current <= end) {

            if (colors[current] == 0) {
                int temp = colors[start];
                colors[start] = colors[current];
                colors[current] = temp;

                current++;
                start++;
            }

            else if (colors[current] == 1) {
                current++;
            }

            else {
```

```
                int temp = colors[current];
                colors[current] = colors[end];
```

Run | Ask AI Mentor | Save | Reset

Sort Colors

Solution summary

To summarize, solving this problem involves the following steps:

- Traverse the array and swap elements, as required, to organize them correctly.
- If the encountered color is red, swap its value with that of the `start` pointer. If the color is blue, swap its value with that of the `end` pointer.
- White elements are skipped, and pointers are adjusted accordingly.
- The array is sorted when the `end` pointer moves to the left of the `current` pointer.

Time complexity

The time complexity of this solution is $O(n)$ since we're only traversing the array once.

Space complexity

The space complexity of this solution is $O(1)$ since no extra space is used.

Back lesson

Mark As Completed

Next

Sort Colors

Reverse Words in a String