Sorting 0s, 1s, and 2s

This section covers the problem of sorting an array containing only 0s, 1s, and 2s. We'll explore various solutions, from brute force to the most optimal approach using the Dutch National Flag algorithm.

Brute Force Solution

The initial approach involves using a standard sorting algorithm like **merge sort**.

- **Time Complexity:** $O(n \log n)$
- **Space Complexity:** $O(n)$ (due to the temporary array used in merge sort)

The interviewer will likely ask for a more optimized solution.

Better Solution: Counting

This solution involves counting the occurrences of 0, 1, and 2 in the array and then overwriting the array with the sorted values.

1. **Count Occurrences:** Iterate through the array once to count the number of 0s, 1s, and 2s.
2. **Overwrite Array:** Overwrite the array based on the counts. First, fill the array with the number of 0s, then the number of 1s, and finally the number of 2s.

```
count0 = number of 0s
count1 = number of 1s
count2 = number of 2s
```

Then overwrite:

```
array[0 ... count0-1] = 0
array[count0 ... count0 + count1 -1] = 1
array[count0 + count1 ... n-1] = 2
```

- **Time Complexity:** $O(n)$ (one loop to count, and another to overwrite) which is effectively $2n$ or $O(n)$
- **Space Complexity:** $O(1)$ (no extra space used, modifying the given array)

However, the interviewer might push for an even more optimal solution that requires only one iteration.

Most Optimal Solution: Dutch National Flag Algorithm

This algorithm uses three pointers (**low**, **mid**, **high**) and revolves around three rules:

- Everything from `0` to `low - 1` will be `0`.
- Everything from `low` to `mid - 1` will be `1`.
- Everything from `high + 1` to `n - 1` will be `2`.

Visualization

```
0 ... low-1   |   low ... mid-1   |   mid ... high   |   high+1 ... n-1
---------------------------------------------------------------------
      0s      |         1s        |   Unsorted 0s,1s,2s  |        2s
```

Initially:

- `mid` points to the start of the array.
- `high` points to the end of the array.
- `low` points to the start of the array.

The algorithm works based on the value of `array[mid]`:

1. **If `array[mid] == 0`:**
    - Swap `array[mid]` with `array[low]`.
    - Increment both `low` and `mid`.
2. **If `array[mid] == 1`:**
    - Increment `mid`.
3. **If `array[mid] == 2`:**
    - Swap `array[mid]` with `array[high]`.
    - Decrement `high`.
    - Note: `mid` is NOT incremented in this case.

Algorithm Rules

> **Rule 1:** Everything from `0` to `low - 1` is `0`.
>
> **Rule 2:** Everything from `low` to `mid - 1` is `1`.
>
> **Rule 3:** Everything from `high + 1` to `n - 1` is `2`.

Thought Process

If `array[mid]` is `0`, it should be placed at the beginning of the array. So, it's swapped with the element at the `low` index.

If `array[mid]` is `1`, it's already in the correct section, so `mid` is incremented.

If `array[mid]` is `2`, it should be placed at the end of the array, so it's swapped with the element at the `high` index.

Time and Space Complexity

- **Time Complexity:** $O(n)$ (single iteration)
- **Space Complexity:** $O(1)$ (constant extra space)

Dutch National Flag Algorithm

The Dutch National Flag algorithm sorts an array containing only 0s, 1s, and 2s in one pass. It maintains three sections within the array:

- 0 to `low - 1`: Contains all 0s
- `low` to `mid - 1`: Contains all 1s
- `mid` to `high`: Contains unsorted elements
- `high + 1` to end: Contains all 2s

The algorithm uses three pointers: `low`, `mid`, and `high`.

Algorithm Steps ⌧

The algorithm iterates as long as `mid <= high`. Here are the three possible scenarios for `arr[mid]`:

1. If `arr[mid]` is 0:
    - Swap `arr[low]` and `arr[mid]`.
    - Increment both `low` and `mid`.
    - By swapping the zero with the element at the `low` index, you're placing the zero in its correct sorted position at the beginning of the array. Since `mid` now points to an element just after a sorted zero, it's safe to increment `mid` to continue the sorting process.
2. If `arr[mid]` is 1:
    - Increment `mid`.
    - Since '1' is already in its correct order, no swapping is needed
3. If `arr[mid]` is 2:
    - Swap `arr[mid]` and `arr[high]`.
    - Decrement `high`.
    - By swapping `arr[mid]` with `arr[high]`, you're placing the '2' at the end of the array where it belongs. Unlike the '0' case, you don't increment `mid` because the element swapped into `arr[mid]` is new and still needs to be processed.

Visual Example

Step	Condition	Action
`arr[mid] == 0`	Zero found	Swap `arr[low]` and `arr[mid]`, then increment both `low` and `mid`.
`arr[mid] == 1`	One found	Increment `mid`.
`arr[mid] == 2`	Two found	Swap `arr[mid]` and `arr[high]`, then decrement `high`.

Code Implementation

```python
def dutch_national_flag(arr):
    low = 0
    mid = 0
    high = len(arr) - 1

    while mid <= high:
        if arr[mid] == 0:
            arr[low], arr[mid] = arr[mid], arr[low]
            low += 1
            mid += 1
        elif arr[mid] == 1:
            mid += 1
        else:  # arr[mid] == 2
            arr[mid], arr[high] = arr[high], arr[mid]
            high -= 1
```

Importance of Understanding

It's important to understand the underlying logic behind each step:

- Why move `low` and `mid` when encountering a 0?
- Why only move `mid` when encountering a 1?
- Why move `high` but not `mid` when encountering a 2?

Understanding these nuances helps in explaining the algorithm during interviews and enhances problem-solving skills.## Time and Space Complexity of the Dutch National Flag Algorithm

The Dutch National Flag algorithm sorts an array containing 0s, 1s, and 2s in place. This section will cover the time and space complexity of this sorting algorithm.

Time Complexity

The algorithm iterates through the array from `mid` to `high` to sort the unsorted section.

Initially, `mid` points to the start of the array, and `high` points to the end, covering the entire length `n` of the array.

At each step, one of the following operations is performed:

- If `array[mid]` is 0, swap `array[low]` and `array[mid]`, and increment both `low` and `mid`.
- If `array[mid]` is 1, increment `mid`.
- If `array[mid]` is 2, swap `array[mid]` and `array[high]`, and decrement `high`.

In each of these three steps, either `mid` is incremented, or `high` is decremented. This means that with each step, one element is sorted.

Since all `n` elements are initially unsorted, the algorithm takes `n` steps to sort all elements. Therefore, the **time complexity** is $O(n)$, representing a single iteration through the array.

To better understand the time complexity, you can manually track iterations using a sample array. Count each iteration as +1 to visualize how the algorithm processes the array and relate it to $O(n)$.

Space Complexity

The **space complexity** of the Dutch National Flag algorithm is $O(1)$ because it sorts the array in place without using any additional data structures that scale with the input size. The algorithm only uses a few extra variables (like `low`, `mid`, and `high`), so the space required does not depend on the size of the array.