What is the Problem?

Imagine you have a line of toys on a shelf. Some of the toys are cool (non-zero), and some are boring (zero). You want to rearrange the toys so that all the cool toys are on the left side of the shelf, and all the boring toys are on the right side. But you can't throw away any toys — you just need to move them around.

This is exactly what the "Move Zeroes" problem is about! We have a list of numbers (like `[0, 1, 0, 3, 12]`), and we need to move all the zeroes to the end while keeping the non-zero numbers in the same order.

Naive Approach - Using an Additional Array

Step-by-Step Explanation

1. **Create a New Shelf (List):**
   Imagine you have a second shelf where you put all the cool toys (non-zero numbers). This new shelf is called `result`.
2. **Pick Cool Toys:**
   Go through each toy (number) in the original shelf (`nums`). If the toy is cool (not zero), put it on the new shelf (`result`).
3. **Count Boring Toys:**
   After picking all the cool toys, count how many boring toys (zeroes) were left on the original shelf. For example, if there were 5 toys total and 3 cool toys, then there must be `5 - 3 = 2` boring toys.
4. **Add Boring Toys to the End:**
   Now, take all the boring toys (zeroes) and add them to the end of the new shelf (`result`).
5. **Replace the Original Shelf:**
   Finally, copy everything from the new shelf (`result`) back to the original shelf (`nums`).

Code Example

Here’s how the code works:

python Copy

```python
def moveZeroes(nums):
    # Step 1: Create a new shelf for cool toys
    result = []

    # Step 2: Pick cool toys (non-zero numbers)
    for num in nums:
        if num != 0:
            result.append(num)

    # Step 3: Count boring toys (zeroes)
    zero_count = len(nums) - len(result)

    # Step 4: Add boring toys to the end
    result.extend([0] * zero_count)

    # Step 5: Replace the original shelf with the new one
    for i in range(len(nums)):
        nums[i] = result[i]
```

Why Is This Not Perfect?

This approach works, but it uses an extra shelf (`result`), which takes up more space. If the original shelf is very big, this might not be the best way because we’re wasting space on the second shelf.

Optimal Approach - Two Pointer Technique

Step-by-Step Explanation

Now, let's try to solve the problem without using a second shelf. Instead, we'll rearrange the toys directly on the original shelf.

1. **Point at the First Spot:**

   Imagine you have a helper who points at the first spot on the shelf where a cool toy should go. This pointer is called `last_non_zero_found_at` , and it starts at position `0` .

2. **Walk Through the Shelf:**

   You walk through the shelf one toy at a time. For each toy:

   - If it's cool (non-zero), swap it with the toy at the spot your helper is pointing to.
   - Then, tell your helper to move one spot to the right.

3. **Repeat Until Done:**

   Keep walking and swapping until you've checked all the toys. At the end, all the cool toys will be on the left, and the boring toys will naturally stay on the right.

Code Example

Here's how the code works:

python

```python
def moveZeroes(nums):
    # Step 1: Helper points at the first spot
    last_non_zero_found_at = 0

    # Step 2: Walk through the shelf
    for i in range(len(nums)):
        if nums[i] != 0:
            # Step 3: Swap cool toy with the spot helper is pointing to
            nums[last_non_zero_found_at], nums[i] = nums[i], nums[last_non_zero_found_at]

            # Step 4: Move helper to the next spot
            last_non_zero_found_at += 1
```

Why Is This Better?

This approach doesn't use a second shelf, so it saves space. It also only walks through the shelf once, making it faster.

Example Walkthrough

Let's see how the two-pointer technique works with an example:

Original shelf: `[0, 1, 0, 3, 12]`

1. Start with `last_non_zero_found_at = 0` .
2. Look at the first toy (`0`). It's boring, so do nothing.
3. Look at the second toy (`1`). It's cool! Swap it with the toy at position `0` . Now the shelf looks like `[1, 0, 0, 3, 12]` . Move the helper to position `1` .
4. Look at the third toy (`0`). It's boring, so do nothing.
5. Look at the fourth toy (`3`). It's cool! Swap it with the toy at position `1` . Now the shelf looks like `[1, 3, 0, 0, 12]` . Move the helper to position `2` .
6. Look at the fifth toy (`12`). It's cool! Swap it with the toy at position `2` . Now the shelf looks like `[1, 3, 12, 0, 0]` . Move the helper to position `3` .

Final shelf: `[1, 3, 12, 0, 0]`

Complexity Analysis

Naive Approach:

- **Time Complexity:** O(n) — You check every toy twice (once to pick cool toys, once to copy back).
- **Space Complexity:** O(n) — You use a second shelf to store cool toys.

Optimal Approach:

- **Time Complexity:** O(n) — You only check every toy once.
- **Space Complexity:** O(1) — You don't use a second shelf; you rearrange toys directly on the original shelf.

Which One Should You Use?

The two-pointer technique is better because it's faster and doesn't waste space. It's like being able to rearrange the toys perfectly without needing a second shelf!

Final Answer

The optimal solution is the **Two Pointer Technique** , and the final code is:

python Copy

```python
def moveZeroes(nums):
    last_non_zero_found_at = 0
    for i in range(len(nums)):
        if nums[i] != 0:
            nums[last_non_zero_found_at], nums[i] = nums[i], nums[last_non_zero_found_at]
            last_non_zero_found_at += 1
```

Output for `[0, 1, 0, 3, 12]` **:**

$$\boxed{[1, 3, 12, 0, 0]}$$

Ask Explain