Two Pointers Pattern

The **Two Pointers** technique involves using two indices (or pointers) to scan through an array or string, often from opposite ends or sometimes in the same direction. It's used to solve problems efficiently, usually in linear time, by reducing unnecessary comparisons and iterations.

Classic Use Cases:

- Reverse an array or string in-place
- Remove duplicates from a sorted array
- Find pairs in an array that sum to a target value (in sorted arrays)
- Partition arrays
- Move zeroes to the end

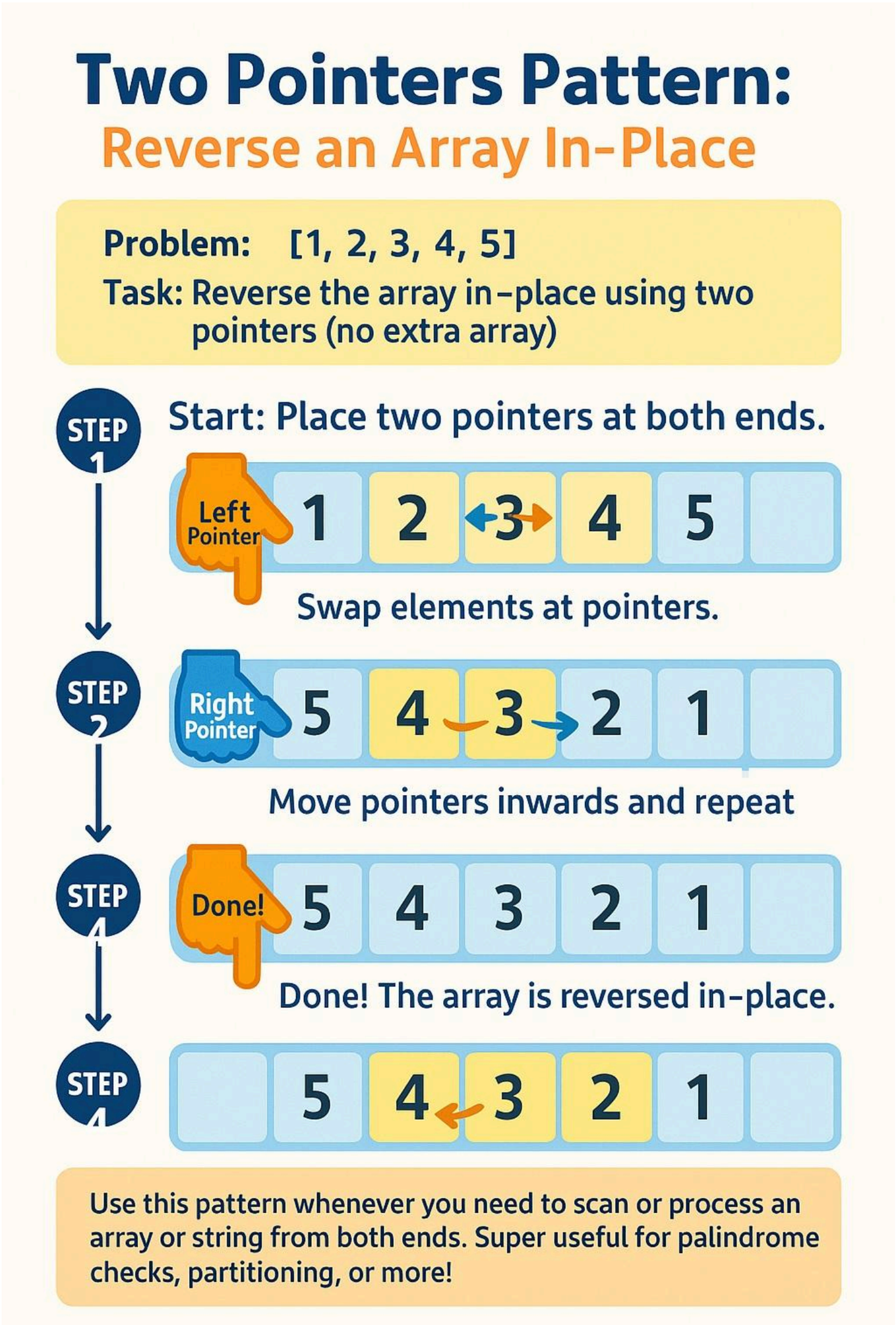
Most Basic Example: Reverse an Array

Problem:

Reverse an array in-place (no extra array allowed).

Solution:

1. Set one pointer at the **start** (left = 0).
2. Set another at the **end** (right = n-1).
3. Swap elements at left and right.
4. Move left pointer forward, right pointer backward.
5. Repeat until left >= right.



C Code Example 1: Reverse an Array

```
#include <stdio.h>

// Function to reverse an array using two pointers
void reverseArray(int arr[], int n) {
```

```
    int left = 0;
    int right = n - 1;

    while (left < right) {
        // Swap arr[left] and arr[right]
        int temp = arr[left];
        arr[left] = arr[right];
        arr[right] = temp;

        left++;    // Move left pointer forward
        right--;   // Move right pointer backward
    }
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    reverseArray(arr, n);

    printf("Reversed array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Explanation:

- We point at the first and last element.
- Swap them (so 1 and 5 switch).
- Then move both pointers closer (now on 2 and 4), swap again.
- Continue until both pointers meet or cross.
- You get the reversed array.

Example 2: Remove Duplicates from a Sorted Array

Problem:

Given a **sorted** array, remove duplicates in-place so that each element appears only once, and return the new length.

(Classic LeetCode-style)

Step-by-step Solution:

1. Use `i` as the **slow** pointer (to mark where the next unique element should go).
2. Use `j` as the **fast** pointer to scan for unique elements.
3. If `arr[j]` is not equal to `arr[i]`, increment `i` and copy `arr[j]` to `arr[i]`.

C Code Example 2: Remove Duplicates

```
#include <stdio.h>

int removeDuplicates(int arr[], int n) {
    if (n == 0) return 0;
    int i = 0; // slow pointer

    for (int j = 1; j < n; j++) { // fast pointer
        if (arr[j] != arr[i]) {
            i++;
            arr[i] = arr[j];
        }
```

```
    }
    return i + 1; // Length of array without duplicates
}

int main() {
    int arr[] = {1, 1, 2, 2, 3, 4, 4};
    int n = sizeof(arr) / sizeof(arr[0]);

    int newLen = removeDuplicates(arr, n);

    printf("Array after removing duplicates: ");
    for (int i = 0; i < newLen; i++) {
        printf("%d ", arr[i]);
    }
    printf("\nNew length: %d\n", newLen);

    return 0;
}
```

Explanation:

- Because the array is sorted, duplicates are always next to each other.
- Whenever we see a new number, we copy it forward.
- At the end, the front part of the array contains only unique elements.

Example 3: Two Sum (Sorted Array)

Problem:
Given a sorted array, find two numbers that add up to a target sum.

Approach:

1. Point `left` at the start, `right` at the end.
2. If their sum is too small, move `left` forward.
3. If their sum is too large, move `right` backward.
4. If their sum is correct, done!

C Code Example 3: Two Sum Sorted

```
#include <stdio.h>

void twoSumSorted(int arr[], int n, int target) {
    int left = 0, right = n - 1;

    while (left < right) {
        int sum = arr[left] + arr[right];
        if (sum == target) {
            printf("Pair found: %d + %d = %d\n", arr[left], arr[right], target);
            return;
        } else if (sum < target) {
            left++;
        } else {
            right--;
        }
    }
    printf("No pair found with the given sum.\n");
}

int main() {
    int arr[] = {1, 2, 3, 4, 6, 8, 9};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 10;

    twoSumSorted(arr, n, target);
```

```
    return 0;
}
```

Explanation:

- Start from both ends.
- Add the two numbers.
- If sum is less, move left up (need bigger number).
- If sum is more, move right down (need smaller number).
- Repeat until you find the pair or pointers meet.

When To Use This Pattern

- When problem asks about pairs/triplets/subarrays/sections in arrays or strings.
- Often when array is **sorted** or you need to process both ends.

Key Takeaways

- Use two pointers to reduce time and extra space.
- Great for problems involving comparisons or scanning from both sides.
- Super useful for in-place modifications.

Practice Challenge

Try using two pointers to **move all zeros to the end** of an array while keeping the order of other numbers.