📘 DAY 3 — TWO POINTER PATTERN (NOTES)

PROBLEM 1 . SORT TWO COLORS – NOTES (C++ VERSION)

(Perfect for exams + DSA revision + interviews)

🔷 1. Problem Understanding

You are given an array nums that contains only **0s and 1s**.
Your job is to **sort it in-place** so that:

All 0s come first,
All 1s come after them.
Example:
Input: [0,1,1,0,1,1]
Output: [0,0,0,1,1,1]

🔷 2. Bruteforce Approach (2 Passes)

1st pass → count number of **0s** and number of **1s**
2nd pass → overwrite array

- first put all 0s
- then put all 1s

⏳ **Time = O(n)**
📦 **Space = O(1)**
But: Interviewer will ask → "Can you solve in **1 pass**?"

🔷 3. Optimal Approach – Two Pointer Technique (Single Pass)

This is the **Dutch National Flag (Simplified)** algorithm.
We maintain two pointers:

- **i** → points to place where next 0 should go (front)
- **j** → points to place where next 1 should go (back)

Process continues while i <= j.

✔ Rules:

- If nums[i] == 0 → correct place → **just move i forward**
- Else (nums[i] == 1) → we should put this 1 at the end
  - swap nums[i] with nums[j]
  - decrement j
  - **don't move i** (because swapped element may again be 1)

🔷 4. Intuition (Very Important)

At any moment:

- **All elements left of i are guaranteed 0s.**
- **All elements right of j are guaranteed 1s.**
- Middle part [i..j] is unprocessed.

🔷 5. Final C++ Code (Interview Ready)

```cpp
void sortTwoColors(vector<int>& nums) {
    int i = 0;
    int j = nums.size() - 1;

    while (i <= j) {
        if (nums[i] == 0) {
            i++;   // 0 is already in correct area
        }
        else {
            // nums[i] == 1 → place at the end
            swap(nums[i], nums[j]);
            j--;
        }
    }
}
```

🔷 6. Example Walkthrough

Array: 0 1 1 0 1 1
i = 0, j = 5

- nums[i] = 0 → i++
- nums[i] = 1 → swap with j
- keep reducing j
- only move i when 0 is found

Final result: 0 0 0 1 1 1

🔷 7. Time & Space Complexity

Complexity	Value
Time	O(n) (single pass)
Space	O(1)

PROBLEM 2. SORT THREE COLORS (0,1,2)

(Dutch National Flag – 3 Pointer Approach)

🔵 Problem

Array given with only **0, 1, 2**.
We must **sort them in-place** so that:

```scss
All 0s → left
All 1s → middle
All 2s → right
```

Allowed → Single pass, O(1) extra space.

🔵 Pointers Used

We use **3 pointers**:

Pointer	Meaning
i	Boundary for 0s (left region fully 0s)
k	Current element we are processing
j	Boundary for 2s (right region fully 2s)

Initial setup:

i = 0
k = 0
j = n - 1

⭐ Invariant at any moment

```css
0 ... i-1  → all 0s
i ... k-1  → all 1s (processed)
k ... j    → unknown (to process)
j+1 ... n  → all 2s
```

🔵 Loop Condition

```arduino
while (k <= j)
```

Because `k` is scanning the middle region until it crosses j.

🔵 Three Cases

✅ Case 1: nums[k] == 1

One is the middle color → correct zone →
Just move k ahead.

k++;

✅ Case 2: nums[k] == 2

Two must go to the end → swap with j:

```
swap(nums[k], nums[j]);
j--;
```

⚠ **Important:** DO NOT increment k here.
Because after swapping, nums[k] may again be 0 or 2 → needs processing.

✅ Case 3: nums[k] == 0

Zero must go to the front → swap with i:

```
swap(nums[k], nums[i]);
i++;
k++;
```

✔ Why increment both i and k?

- i points to the next blank spot for 0
- When nums[k] == 0, swapping with i ensures:
  - nums[i] ← 0
  - Element previously at nums[i] comes to k
- But because i < k, the element coming to k is always 1
  (since before i everything is 0, between i and k everything is 1)

Thus **k can safely move ahead**.

🔵 Final C++ Code (Perfect Version)

```cpp
class Solution {
public:
    void sortColors(vector<int>& nums) {
        int i = 0;                  // boundary for 0s
        int k = 0;                  // current pointer
        int j = nums.size() - 1;    // boundary for 2s

        while (k <= j) {

            if (nums[k] == 1) {
                k++;
            }
            else if (nums[k] == 2) {
                swap(nums[k], nums[j]);
                j--;
            }
            else { // nums[k] == 0
                swap(nums[k], nums[i]);
                i++;
                k++;
            }
        }
    }
```

Copy code

🔵 Dry Run Example

Input:

[2, 0, 2, 1, 1, 0]

1. k=0 → nums[k]=2 → swap with j
2. k=0 → nums[k]=0 → swap with i → i++, k++
3. k=1 → nums[k]=0 → swap with i → i++, k++
4. k=2 → nums[k]=2 → swap with j → j--
5. k=2 → nums[k]=1 → k++
6. k=3 → nums[k]=1 → k++

   Loop stops when k > j.

Final:

[0,0,1,1,2,2]

🔵 Time & Space Complexity

Complexity	Value
Time	O(n)
Space	O(1)

Single pass → best possible.

🔥 In short – One rule for remembering

- **0 → swap with i, i++, k++**
- **1 → k++**
- **2 → swap with j, j--**

This is the Dutch National Flag algorithm.

PROBLEM 3. REMOVE Nth NODE FROM END OF LIST – COMPLETE NOTES (C++ Version)

(LeetCode 19 – Medium – Very Important Interview Question)

🔷 1. Problem Statement

You are given the **head of a singly linked list**.
You must **remove the N-th node from the end** of the list and return the modified head.
Example:

List: 1 → 2 → 3 → 4 → 5
n = 2

2nd from end = node 4
After deletion:
1 → 2 → 3 → 5

🔷 2. Key Concepts Needed

✔ (A) How to delete a node in singly linked list?

To delete a node curr, you MUST know the node **before it** → prev.
Then:

prev->next = curr->next
✔ Node gets removed automatically.
This is why we track → **previous** + **current**

✔ (B) The HEAD deletion problem

If the head itself needs to be removed, normal prev logic fails (because head has no previous).

Solution → Use a dummy node

dummy → head → node2 → node3 ...
This makes deletion uniform:

- delete head
- delete middle
- delete last
  → everything is handled the same way.

🔷 3. Approach 1 (Two Pass / Length Based)

Simple logic:

1. First pass → find length L of the list
2. Node to delete from front =

   d = L - n + 1
3.
4. Move (d-1) steps from dummy
5. Delete next node
6. Return dummy->next

Example

List: 1 → 2 → 3 → 4 → 5, n=2
Length = 5
Delete node = 5 - 2 + 1 = **4th** node
Move 3 steps from dummy → we reach node 3 → delete next (4).

🔷 Approach 1 – C++ Code

```cpp
ListNode* removeNthFromEnd(ListNode* head, int n) {

    ListNode* dummy = new ListNode(0);
    dummy->next = head;

    // Step 1: find length
    int len = 0;
    ListNode* t = head;
    while (t) {
        len++;
        t = t->next;
    }

    // Step 2: find (len-n+1)-th node from front
    int d = len - n + 1;

    // Step 3: move to (d-1)th node
    ListNode* prev = dummy;
    for (int i = 1; i < d; i++) {
        prev = prev->next;
    }

    // Step 4: delete
    prev->next = prev->next->next;

    return dummy->next;
}
```

✔ **Time: O(n)**
✔ **Space: O(1)**

🔷 4. Approach 2 (One Pass – Two Pointer / Optimized)

This is the **famous two-pointer trick**.

Idea:

We use two pointers:

fast
slow

Step-by-step:

1. Move fast pointer **n steps forward**
2. Now move **fast + slow together** until fast reaches NULL
3. At this moment:

slow is exactly at the (d-1)th node
i.e., **just before the node that must be deleted**

1. So delete:

slow->next = slow->next->next

Why does this work?

Think of it like:

- fast runs ahead by n nodes
- slow then starts from dummy
- When fast reaches the end,
- slow will be at the correct deletion point
  (because fast was exactly n nodes ahead).

🔷 Approach 2 – C++ Code (BEST SOLUTION)

```cpp
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {

        ListNode* dummy = new ListNode(0);
        dummy->next = head;

        ListNode* fast = dummy;
        ListNode* slow = dummy;

        // Step 1: Move fast n+1 steps (to get slow at node-before deletion)
        for (int i = 0; i <= n; i++)
            fast = fast->next;

        // Step 2: Move both while fast != NULL
        while (fast) {
            fast = fast->next;
            slow = slow->next;
        }

        // Step 3: Delete
        slow->next = slow->next->next;

        return dummy->next;
    }
};
```

✔ Single pass O(n)
✔ Constant space
✔ Handles head deletion
✔ Very popular interview pattern

🔷 5. Quick Dry Run

List: 1 → 2 → 3 → 4 → 5, n=2

1. Move fast 3 steps (n+1)
   → fast at node 3
   → slow at dummy
2. Move both until fast hits NULL
   fast: 3→4→5→NULL
   slow: dummy→1→2→3
3. slow points to node 3 → delete next (node 4)

Result:

1 → 2 → 3 → 5

🔥 Summary Table

Approach	Passes	Time	Space	Notes
Length-based	Two-pass	O(n)	O(1)	Easy, clear
Two-pointer	One-pass	O(n)	O(1)	BEST, asked in interviews