🚀 DSA Pattern Course 2025 — Day 4: Two Pointer Pattern Mastery (C++)

⭐ Today's Goal: Three Medium-Level Problems → One Pass Pattern

---

🟦 1. Sort Two Colors (Only 0s and 1s)

(Custom problem → Three Colors Problem ko samajhne ka base)

🧠 Problem Statement

Ek array `nums` diya hai jisme **sirf 0s aur 1s** hain.

List ko **in-place** sort karna hai:

```
All 0s → left All 1s → right
```

🔴 Approach 1: Brute Force (Two-Pass Counting)

✔️ Step 1: Count

- `C0` → number of 0s
- `C1` → number of 1s

✔️ Step 2: Overwrite

- First `C0` times write `0`
- Then `C1` times write `1`

🧮 Complexity

- **Time:** `O(N) + O(N) = O(N)`
- **Space:** `O(1)` (only counters)

🔸 Best brute force → but not one pass.

Interviewer: "One pass me solve karo!"

🟢 Approach 2: Optimal One Pass (Two Pointers)

Pointer Meaning:

Pointer	Start	Invariant
i (Left)	0	`i` se pehle sab **0s**
j (Right)	n−1	`j` ke baad sab **1s**

Only **i pointer** traversal karega (`while(i <= j)`)

📝 Logic (while i <= j)

Case 1: nums[i] == 0

✔️ Already correct

👉 Move `i++`

Case 2: nums[i] == 1

❌ Wrong place → should be on right

👉 Swap `nums[i]` with `nums[j]`

👉 Move `j--`

❗ Don't increment `i`

(kyunki j se aaya element ab i par check hona chahiye)

💻 Code (C++)

```
void sortTwoColors(vector<int>& nums) { int i = 0; // left pointer → 0s region
int j = nums.size() - 1; // right pointer → 1s region // Process until both poi
nters meet while (i <= j) { // If nums[i] is 0 → already at correct place if (n
ums[i] == 0) { i++; } // If nums[i] is 1 → send it to right side else { swap(nu
ms[i], nums[j]); j--; // decrease right boundary // i ko yahan increase nahi ka
rte // kyunki swapped value ko dobara check karna zaroori hai } } }
```

🟥2 . LeetCode 75 – Sort Colors (Brute Force: Counting Method)

Most clean & most accepted brute force.

🧠 Intuition (Hinglish + Simple)

Array me sirf **3 values** hoti hain → `0`, `1`, `2`

Toh brute force ka best tareeka hai:

1️⃣ **Count karo** kitne 0, kitne 1, kitne 2

2️⃣ **Rewrite** kar do array ko is order me:

```
0 → 1 → 2
```

Bas. Sorting bhi nahi chahiye.

Extra space bhi nahi lagta.

Logic bhi crystal clear.

🧩 Step-by-Step Breakdown

Step 1: Count Frequency

- Har element ek loop me traverse hoga
- Values:
  - Agar 0 → `zero++`
  - Agar 1 → `one++`
  - Agar 2 → `two++`

Step 2: Rewrite Array

- Index = 0 se start
- Pehle saare `zero` values (0)
- Fir saare `one` values (1)
- Fir saare `two` values (2)
- Every assignment uses

  `nums[index++] = x;`

🔥 index++ (Important for beginners)

`index++` ka matlab:

👉 Pehle **current index** use hota hai

👉 Fir index **1 se increase** ho jaata hai

Example:

- index = 0 → use hua → index = 1
- index = 1 → use hua → index = 2
- index = 2 → use hua → index = 3

Yeh ensure karta hai ki **har naya element next position par hi jaaye.**

💻 Brute Force Code (C++)

```
void sortColors(vector<int>& nums) { int zero = 0, one = 0, two = 0; // Step 1: Counting frequency for(int x : nums){ if(x == 0) zero++; else if(x == 1) one++; else two++; } // Step 2: Rewrite array int index = 0; while(zero--){ nums[index++] = 0; } while(one--){ nums[index++] = 1; } while(two--){ nums[index++] = 2; } }
```

📝 Dry Run (Clear & Simple)

Input:

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

Step 1 → Count

```
zero = 2 one = 2 two = 2
```

Step 2 → Rewrite

```
nums[0] = 0 nums[1] = 0 nums[2] = 1 nums[3] = 1 nums[4] = 2 nums[5] = 2
```

Final Output:

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

⏳ Time Complexity (Detailed + Interview Explanation)

✔️ Step 1: Counting

- Hum array ko **1 bar traverse** karte hain → length = n

  Cost: O(n)

✔️ Step 2: Rewrite

- Rewrite me total operations = **zero + one + two**
- Yeh values milke **n hi hoti hain**

  Cost: O(n)

🎯 Total Time Complexity

```
O(n) + O(n) = O(n)
```

Why it's efficient?

- No nested loops
- No sorting algorithm
- Two linear passes only

Interview me confidently bolo:

👉 **"Brute force but still linear-time."**

💾 Space Complexity (Deep Explanation)

Variables used:

```
zero → stores count of 0s one → stores count of 1s two → stores count of 2s ind
ex → pointing to current write position
```

Bas **4 integer variables**.

No extra array, no vector, nothing else.

🎯 Total Space: O(1)

Constant space

Input array ke size par depend nahi hota.

Why O(1)?

- Hum sirf counts store karte hain
- Rewrite bhi **usi same array me** hota hai (in-place)

✔️ Final Summary (Easy to Recall)

Step	Work
Count	Find 0s, 1s, 2s
Rewrite	Fill array: 0 → 1 → 2
TC	**O(n)**
SC	**O(1)**
Why good?	Fast, clean, interview-suitable brute force

🟦 2. LeetCode 75 – Sort Colors (Optimal: Two Pointer / Dutch National Flag Algorithm)

Most important + most asked + zero confusion version.

🧠 Intuition (Hinglish + Crystal Clear)

Array me sirf **0, 1, 2** hota hai.

Hume array ko *in-place* sort karna hai.

Goal:

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

Toh hum 3 pointer use karte:

1) low

→ Jaha next 0 rakhna hai

2) mid

→ Current element jise check karna hai

3) high

→ Jaha next 2 rakhna hai

🎯 Core Idea (Very Important)

Rules based on nums[mid]:

Case 1 → nums[mid] == 0

👉 Swap(nums[low], nums[mid])

👉 low++, mid++

Because 0 belongs to left.

Case 2 → nums[mid] == 1

👉 mid++

Because 1 already at correct middle region.

Case 3 → nums[mid] == 2

👉 Swap(nums[mid], nums[high])

👉 high--

BUT mid++ nahi,

kyunki swap ke baad jo element aya hai mid par, usko fir se check karna padta hai.

🧩 Why Two Pointer Works Perfectly?

- 0s ko left bhejo
- 2s ko right bhejo
- 1s automatically center me aa jaate

Ek hi pass me sorting complete.

💻 Optimal Code (C++)

```
void sortColors(vector<int>& nums) { int low = 0, mid = 0, high = nums.size() - 1; while(mid <= high) { if(nums[mid] == 0) { swap(nums[low], nums[mid]); low++; mid++; } else if(nums[mid] == 1) { mid++; } else { // nums[mid] == 2 swap(nums[mid], nums[high]); high--; } } }
```

📝 Dry Run (Step-by-Step Clear Walkthrough)

Input:

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

Start:

```
low=0, mid=0, high=5
```

Step 1:

mid=0 → 2

Swap(mid, high)

Array → `[0, 0, 2, 1, 1, 2]`

high=4

(mid same: 0)

Step 2:

mid=0 → 0

Swap(low, mid)

Array → `[0, 0, 2, 1, 1, 2]`

low=1, mid=1

Step 3:

mid=1 → 0

Swap(low, mid)

Array → `[0, 0, 2, 1, 1, 2]`

low=2, mid=2

Step 4:

mid=2 → 2

Swap(mid, high)

Array → `[0, 0, 1, 1, 2, 2]`

high=3

Step 5:

mid=2 → 1

mid = 3

Step 6:

mid=3 → 1

mid = 4 > high → STOP

Final Output:

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

⏳ Time Complexity (Detail + Interview-Perfect)

Single while loop

```
mid moves from 0 → n-1 (maximum once) low moves forward only high moves backwar
d only
```

👉 No element is processed more than **once**.

Final Time Complexity

```
O(n)
```

Better than brute force?

→ Same time complexity, but **1 pass** instead of 2.

💾 Space Complexity

Hum sirf use kar rahe:

```
low, mid, high → 3 variables
```

Koi extra array nahi.

Sorting **in-place** hoti hai.

Space Complexity

```
O(1)
```

⭐ Why TWO POINTER is Optimal?

- **One-pass algorithm**
- **In-place** (no extra space)
- **No frequency counting**
- **No sorting**
- **Fastest + cleanest**
- Interviewer always expects this solution

✔️ Final Summary (Easy to Revise)

Pointer	Meaning
low	yaha next 0 jaayega
mid	current element
high	yaha next 2 jaayega

Case	Action
nums[mid] == 0	swap(low, mid), low++, mid++
nums[mid] == 1	mid++
nums[mid] == 2	swap(mid, high), high--

Result	Value
TC	**O(n)**
SC	**O(1)**
Passes	**Single pass**

🟥 3. LeetCode #19 – Remove N-th Node From End of Linked List

⭐ Brute Force Solution (2-Pass Approach)

✔ Easiest

✔ High clarity

✔ No confusion

✔ Interview-approved brute force

🧠 Problem in Simple Hinglish

Ek linked list di hui hai.

Hume **end se N-th node** ko delete karna hai.

Example:

List = `1 → 2 → 3 → 4 → 5`, n = 2

→ End se 2nd = **4**, so result = `1 → 2 → 3 → 5`

🚀 Brute Force Intuition (Super Clear)

Linked list me **piche se count nahi kar sakte**,

isiliye hum kya karte?

🔥 Step 1:

Pehle **total length L** nikal lo.

🔥 Step 2:

Agar length L pata hai...

Toh **end se n-th** node ka matlab hota hai:

```
Start se (L - n + 1)-th node delete karna
```

But delete karne ke liye hume **previous node** chahiye.

Yani hume reach karna hoga:

```
(L - n)-th node
```

🔥 Step 3:

Simply uska next skip kar do:

```
prev->next = prev->next->next
```

🧩 Edge Case (Important)

Agar tumhe **head** remove karna hai, that means:

```
n == length
```

→ Directly return `head->next` .

💻 Brute Force Code (C++) – Cleanest

```
ListNode* removeNthFromEnd(ListNode* head, int n) { // Step 1: Find total lengt
h of list int length = 0; ListNode* temp = head; while(temp != nullptr) { lengt
h++; temp = temp->next; } // Edge case: remove head if(n == length) { return he
ad->next; } // Step 2: Reach (length - n)-th node (previous node) temp = head;
for(int i = 1; i < length - n; i++) { temp = temp->next; } // Step 3: Delete ne
xt node temp->next = temp->next->next; return head; }
```

📝 Dry Run (Ultra Clear)

Input:

```
head = 1 → 2 → 3 → 4 → 5 n = 2
```

✔ Step 1: Find Length

```
length = 5
```

✔ Step 2: Which node to delete?

Delete = `(5 - 2 + 1) = 4th node` → value = **4**

Previous node index = `5 - 2 = 3` → value = **3**

✔ Step 3: Delete

```
3 → next = 4 → next 3 → next = 5
```

Final list:

```
1 → 2 → 3 → 5
```

⏳ Time Complexity (Deep but Simple)

Pass 1 → Length count

O(n)

Pass 2 → Delete position find

O(n)

Total:

```
O(n) + O(n) = O(2n) = O(n)
```

Linear time brute force.

💾 Space Complexity

Sirf 2–3 pointers and integers:

```
temp, length
```

Total space:

```
O(1)
```

Constant space, no extra list.

⭐ Why This is Pure BRUTE FORCE?

✔ Linked list ko **do baar traverse** karna

✔ Simple logic: pehle length, phir delete

✔ Easy to implement

✔ Beginner-friendly

✔ Interview me jab woh bole:

“Brute force solution batao” → yahi bolna

🟢 Final Revision Table

Task	Meaning
Step 1	Length count
Step 2	(L - n)-th node tak jao
Step 3	next skip karke delete karo
TC	**O(n)**
SC	**O(1)**
Style	2-pass

🟦3. LeetCode #19 – Remove N-th Node From End of List

⭐ Optimal Solution — Two Pointer (One-Pass Method)

✔ Fastest

✔ Most elegant

✔ Most asked in FAANG rounds

✔ Single traversal only

🧠 Intuition (Super Clear Hinglish)

Hume end se N-th node delete karni hai.

But hum list ko **piche se** traverse nahi kar sakte.

Toh hum kya karte?

🔥 Two Pointer Trick:

- **fast pointer** ko **n steps aage** chala do
- **slow pointer** ko head par hi rakho
- Ab dono ko **same speed** se move karo
- Jab fast end (NULL) par pahunch jaaye →

  slow **exactly delete hone wali node ke pehle** hoga

Yahi magic hai two-pointer technique ka.

🎯 Why It Works? (Very Visual)

Suppose list ka end se N=2 delete karna hai.

fast pointer ko:

```
fast → n steps ahead
```

So gap = **n nodes** between fast & slow.

Then:

- Dono ko ek sath move karo
- Jab fast end tak pahunchta hai:
  - slow → (end se n)-th node ke just pehle hota hai

Yani perfect delete position mil jaati hai.

🟨 Edge Case: Delete head

Agar fast initially n step aage le jaane se fast NULL ho jaaye →

🔹 Means humko **head delete** karna hai.

Solution:

```
return head->next;
```

💻 Optimal Two-Pointer Code (C++)

```
ListNode* removeNthFromEnd(ListNode* head, int n) { ListNode* fast = head; List
Node* slow = head; // Step 1: Move fast pointer n steps ahead for(int i = 0; i
< n; i++) { fast = fast->next; } // If fast is NULL, delete head if(fast == NUL
L) { return head->next; } // Step 2: Move both pointers together while(fast->ne
xt != NULL) { fast = fast->next; slow = slow->next; } // Step 3: Delete slow->n
ext slow->next = slow->next->next; return head; }
```

📝 Dry Run (Crystal Clear & Intuitive)

List:

```
1 → 2 → 3 → 4 → 5 n = 2
```

Step 1 → Move fast n steps

fast moves 2 steps:

```
fast at 3 slow at 1
```

Step 2 → Move both

Move until fast->next becomes NULL:

Movement	fast	slow
1	4	2
2	5	3
3	NULL (stop)	3

Now slow is at **3** → this is **previous node** of the node to delete.

Step 3 → Delete

```
slow->next = 4 slow->next = slow->next->next
```

Final list:

```
1 → 2 → 3 → 5
```

Perfect.

⏳ Time Complexity (Best Possible)

Step 1: Move fast n steps → O(n)

Step 2: Move until end → O(n)

BUT both are **just consecutive steps in ONE traversal**, not two full traversals.

⭐ Final TC:

```
O(n)
```

Single pass algorithm.

💾 Space Complexity

Sirf do pointers:

```
fast, slow
```

⭐ Final SC:

```
O(1)
```

⭐ Why This is the Optimal Solution?

- Only **one traversal**
- Uses **two pointers**, perfect for linked list
- No need to count length
- Handles all cases cleanly
- Fastest and shortest code
- Interviewers LOVE this approach

✔️ Final Revision Sheet

Step	Explanation
1	fast ko n steps aage bhejo
2	fast-null → delete head
3	fast & slow ko ek sath chalao
4	slow bilkul delete ke pehle hoga