

# Two Pointers

A comprehensive visual guide to all variants, when to use each, and how to recognise them in interview problems.

## 💡 Which variant should I use?

Answer each question to narrow down the right approach.

**Is the input a sorted array or can you sort it?**

+ Looking for a pair / triplet with a target sum?

→ Opposite Ends

**Need to find a sub-array / substring of fixed or variable length?**

Contiguous elements, possibly with a constraint (max sum, unique chars...)

→ Sliding Window

**Input is a linked list – detect cycle, find middle, Nth from end?**

One pointer moves faster than the other.

→ Fast & Slow

**Two sorted arrays / lists that need to be merged or compared?**

Each pointer lives in a different array.

→ Parallel Pointers

**In-place removal of duplicates / zeros, or partitioning by condition?**

One read-pointer, one write-pointer on the same array.

→ Read / Write

**Partition array around a pivot (QuickSort-style) or Dutch flag?**

Three regions: left, mid, right.

→ Three Pointers

## All Variants in Detail

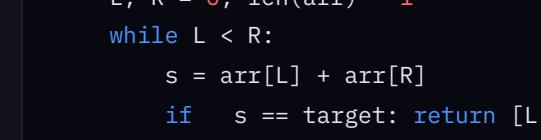
Click to expand – diagrams, use-cases, complexity, and code skeletons.

### ① Opposite Ends (Left ↔ Right)

Place one pointer at the start and one at the end. Move them toward each other based on a condition. Works best on sorted arrays.

Use when: Two Sum (sorted), 3Sum, Container With Most Water, Valid Palindrome, Trapping Rain Water (basic).

• L pointer • R pointer



If  $\text{arr}[L] + \text{arr}[R] < \text{target}$  → move L right | If  $> \text{target}$  → move R left

⌚ 0(n) ⚡ 0(1)

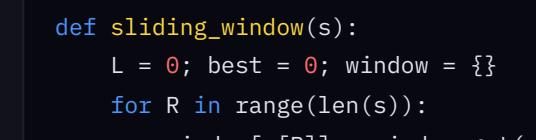
```
def two_sum_sorted(arr, target):
    L, R = 0, len(arr) - 1
    while L < R:
        s = arr[L] + arr[R]
        if s == target: return [L, R]
        elif s < target: L += 1
        else: R -= 1
    return []
```

### ② Sliding Window

Both pointers move in the same direction. A window [L...R] expands by moving R right, and shrinks by moving L right when a constraint is violated.

Use when: Longest substring without repeating chars, Max sum subarray of size K, Minimum window substring, Count subarrays with K distinct.

• Window region • L • R



Expand R → violate constraint → shrink L → track best answer

⌚ 0(n) ⚡ 0(k) for freq map

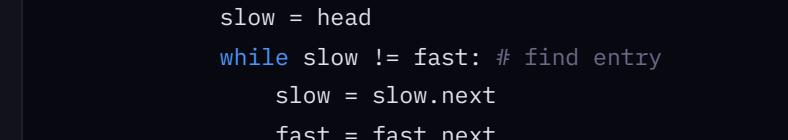
# Variable-size sliding window template

```
def sliding_window(s):
    L = 0; best = 0; window = {}
    for R in range(len(s)):
        window[s[R]] = window.get(s[R], 0) + 1
        while VIOLATED(window):
            window[s[L]] -= 1
            L += 1
        best = max(best, R - L + 1)
    return best
```

### ③ Fast & Slow (Floyd's Tortoise & Hare)

slow moves 1 step, fast moves 2 steps. If there's a cycle they'll meet. After meeting, reset one to head – they'll collide at cycle entry.

Use when: Detect cycle in linked list, Find cycle start, Find middle of list, Happy Number, Palindrome linked list.



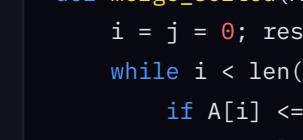
⌚ 0(n) ⚡ 0(1)

```
def detect_cycle(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast: # cycle found
            slow = head
            while slow != fast: # find entry
                slow = slow.next
                fast = fast.next
            return slow # cycle start
    return None
```

### ④ Parallel Pointers (Two Arrays)

One pointer in each of two sorted arrays/lists. Advance whichever is smaller (or equal). Used to merge, compare, or find intersection.

Use when: Merge Sorted Arrays, Intersection of Two Arrays, Compare Version Numbers, Shortest Word Distance.



$A[i] \leq B[j]$  → take  $A[i]$ ,  $i++$  | else → take  $B[j]$ ,  $j++$

⌚ 0(n+m) ⚡ 0(1) in-place

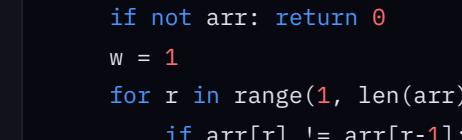
```
def merge_sorted(A, B):
    i = j = 0; result = []
    while i < len(A) and j < len(B):
        if A[i] <= B[j]:
            result.append(A[i]); i += 1
        else:
            result.append(B[j]); j += 1
    return result + A[i:] + B[j:]
```

### ⑤ Read / Write (In-Place Transform)

read scans the entire array; write only advances when valid data should be kept. Effectively compacts the array in-place.

Use when: Remove Duplicates (sorted), Remove Element, Move Zeroes to End, Compress String In-Place.

• write (w) • read (r) • duplicate/skip



⌚ 0(n) ⚡ 0(1)

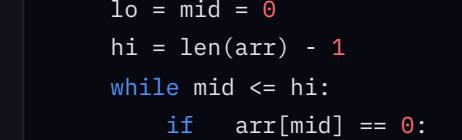
```
# Remove duplicates from sorted array
def remove_dups(arr):
    if not arr: return 0
    w = 1
    for r in range(1, len(arr)):
        if arr[r] != arr[w-1]:
            arr[w] = arr[r]
            w += 1
    return w # new length
```

### ⑥ Three Pointers (Dutch National Flag)

Three regions:  $[0..lo-1]$  = low,  $[lo..mid-1]$  = mid,  $[hi+1..n-1]$  = high. mid scans; swap with lo or hi based on value.

Use when: Sort Colors (0,1,2), Partition around pivot, Separate negative/zero/positive, QuickSort partition.

• lo (0s) • mid (1s) • hi (2s)



$arr[mid] = 0 \rightarrow \text{swap}(lo, mid)$ ,  $lo++, mid++$  |  $arr[mid] = 1 \rightarrow mid++$  |  $arr[mid] = 2 \rightarrow \text{swap}(mid, hi)$ ,  $hi--$

⌚ 0(n) ⚡ 0(1)

```
def sort_colors(arr):
    lo = mid = 0
    hi = len(arr) - 1
    while mid <= hi:
        if arr[mid] == 0:
            arr[lo], arr[mid] = arr[mid], arr[lo]
            lo += 1; mid += 1
        elif arr[mid] == 1: mid += 1
        else:
            arr[mid], arr[hi] = arr[hi], arr[mid]
            hi -= 1
```

## ⚡ Quick Reference Cheatsheet

Variant	Data Structure	Direction	Key Condition	Time
Opposite Ends	Sorted Array	$\leftarrow \rightarrow$	sum vs target	$O(n)$
Sliding Window	Array / String	$\rightarrow \rightarrow$	constraint on window	$O(n)$
Fast & Slow	Linked List	$\rightarrow \rightarrow \rightarrow$	cycle / middle	$O(n)$
Parallel	Two Sorted Arrays	$\rightarrow \rightarrow$	compare & advance smaller	$O(n+m)$
Read / Write	Array (in-place)	$\rightarrow \rightarrow$	filter / deduplicate	$O(n)$
Three Pointers	Array (in-place)	$lo \rightarrow mid \rightarrow hi$	partition by value	$O(n)$