Common Approach to solve Uncommon problems

Two Pointer Approach

It's a technique where you use **two pointers (variables)** to **traverse a data structure**, often from both ends, toward the middle — or sometimes in the same direction — depending on the problem.
Two pointer technique ek algorithmic technique hai jisme do pointers (variables) ka use hota hai jo kisi array/list ke elements ko efficiently process karte hain.

> **Note**
>
> Ye technique **array ko ek hi pass mein (O(n) time)** process karne mein help karti hai — bina extra space ke.

Where is it used?

Scenario	Example Problems	
Reversing things	Reverse a string or array	
Searching in sorted arrays	Two-sum in sorted array, pair sum	
Comparing	Palindrome check	
Removing duplicates	From sorted array	
Partitioning or rearranging	Dutch National Flag problem	
Merging	Merge two sorted arrays	
Sliding window problems	Optimized window expansion/compression	

When to use Two Pointer Approach?

Use it when:

1. You're working with a linear data structure (array, string, linked list).
2. You need to compare elements from both ends, or keep track of two positions.
3. The array is sorted, and you want to find pairs, triplets, or boundaries.
4. You want to minimize time and avoid nested loops (brute force `O(n^2)` → `O(n)` or `O(n log n)`).
   Template for opposite-end traversal:

```
let left = 0;
let right = arr.length - 1;
```

```
while (left < right) {
  // do something with arr[left] and arr[right]
  if (/* some condition */) {
    left++;
  } else {
    right--;
  }
}
```

Basic Concept:

- `left` array ke start par hai
- `right` array ke end par hai
- Dono move karte hain based on logic
  Jaise dono taraf se ek rassi kheech rahe ho — center mein milne tak.

> *Jaise dono taraf se ek rassi kheech rahe ho — center mein milne tak.*

What is Sliding Window?

> ***Sliding Window** is a technique used to efficiently solve problems that deal with **contiguous sequences (subarrays/substrings)** in an array or string.*

Sliding Window Technique में हम एक fixed या variable size की "window" (subarray) को धीरे-धीरे पूरे array पर slide कराते हैं।

> ⓘ **इसका मतलब है कि हम array का एक हिस्सा लेते हैं (जैसे index `i` से `j` तक), उस पर कुछ computation करते हैं, फिर window को एक step आगे बढ़ा देते हैं और अगले हिस्से पर computation करते हैं।**

कैसे काम करता है?
मान लो हमारे पास एक array है:
`arr = [1, 3, 2, 6, -1, 4, 1, 8, 2]`
अब मान लो problem statement है:

> *"Find the maximum sum of any subarray of size `k = 3`"*
> *यानी तुमको array के अंदर जितने भी 3 elements की continuous subarrays बन सकती हैं, उनमें से उस subarray को ढूंढना है जिसका sum सबसे ज्यादा हो।*

तू एक window (खिड़की) ले रहा है, जैसे size `k = 3`, और उस window को array पर left to right slide करवा रहा है।
हर बार:

- पीछे का एक element निकाल देता है

- आगे का एक नया element जोड़ देता है
  यानि जितना आया, उतना ही गया — इसलिए efficient हो जाता है!
  हर बार पुराना हटाओ, नया जोड़ो

क्यों ज़रूरत है ऐसा करने की?

ऐसे problems real life में performance, optimization, या pattern detection के लिए होते हैं, जैसे:

- Sensor readings में 3 लगातार values का peak जानना
- Stock prices में 3 दिन की highest average value जानना
- किसी audio signal में सबसे तेज़ 3 सेकंड का हिस्सा पहचानना
  Brute Force से हर बार तीन elements का sum निकालना पड़ेगा — बहुत time लगेगा (O(n* k) time)

```
for (let i = 0; i <= arr.length - k; i++) {
  let sum = 0;
  for (let j = i; j < i + k; j++) {
    sum += arr[j];
  }
}
```

Sliding Window:

1. पहले `k` elements का sum निकालो → यह पहला window है।
2. अब अगला window लेने के लिए:
   - जो element पीछे छूट रहा है उसे minus कर दो,
   - और नया element जो window में आ रहा है उसे add कर दो।
3. इस तरह हर बार sum को update करते रहो → O(n) में solve हो जाएगा।

```
let maxSum = 0;
let windowSum = 0;
let k = 3;

for (let i = 0; i < k; i++) {
  windowSum += arr[i];  // पहले k elements का sum
}

maxSum = windowSum;

for (let i = k; i < arr.length; i++) {
  windowSum += arr[i] - arr[i - k]; // एक add, एक minus
  maxSum = Math.max(maxSum, windowSum);
}
```

`windowSum += arr[i]- arr[i -k]`

> यही reason है कि हम `arr[i - k]` minus करते हैं — क्योंकि वो पिछली window का सबसे पहला element था जो अब window से बाहर हो गया।
>
> "जो खिड़की से बाहर गिरा, उसे minus कर दो —
> जो नया अंदर आया, उसे जोड़ दो!"

Instead of checking all possible combinations (brute force), it keeps a "window" (subsection) and slides it over the data to find a result in linear time (`O(n)`).

Why Use Sliding Window?

Because brute-force solutions (using nested loops) are too slow — often `O(n²)` or worse. Sliding Window helps you:

- Avoid unnecessary recalculations
- Reduce time complexity to `O(n)`
- Handle large input efficiently

How It Works (General Strategy)

1. Initialize window (start and end pointers)
2. Expand the window (move right)
3. Shrink the window (move left) if a condition breaks
4. Update result (`max` , `min` , etc.)
5. Repeat until end of input

दो तरह के Sliding Window:

Type	Use When Problem Mentions
Fixed-size window	Subarray/substring of size `k`
Variable-size window	Largest/smallest window satisfying some condition

1. Fixed Size Window (easy type)

> *Examples:*
>
> - *Max sum of subarray of size* `k`
> - *Count of subarrays with average > X of size* `k`

Variable Size Window (dynamic)

Generic Pattern (Variable Size):

```
let start = 0;
let answer = 0;
for (let end = 0; end < arr.length; end++) {
   // 1. Expand the window → include arr[end]
   // 2. While window is invalid → shrink from left
   while (/* some condition fails */) {
       // remove arr[start] from window
       start++;
   }
   // 3. Update answer using window size → (end - start + 1)
   answer = Math.max(answer, end - start + 1);
}
```

Core Concept — दोनों में दो pointer होते हैं

Pointer	काम क्या है?
`start`	window का left/starting edge
`end`	window का right/ending edge (expand करने वाला)

1. Fixed Size Sliding Window (Size = k)
   कब use होता है?
   जब question में बोले:
   `"Find something for all subarrays/substrings of fixed size k"`
   मतलब: Window size fix है — उसे ना छोटा करेंगे ना बड़ा।
   बस खिसकाएंगे
   Example:

```
arr = [1, 2, 3, 4, 5], k = 3
let windowSum = 0;
let start = 0;
for (let end = 0; end < arr.length; end++) {
    windowSum += arr[end];
    if (end >= k - 1) {
        // process result here
        maxSum = Math.max(maxSum, windowSum);
        // shrink window
        windowSum -= arr[start];
        start++;
    }
}
Steps:
पहले window: [1, 2, 3] → sum = 6
अगला window: subtract 1, add 4
→ [2, 3, 4] → sum = 9
```

```
फिर: subtract 2, add 5
→ [3, 4, 5] → sum = 12
```

Pointers Movement:

- `end` हर बार बढ़ेगा
- `start` तब ही बढ़ेगा जब window size `k` हो गया

Variable Size Sliding Window (Condition-based)

जब बोले:

> *"Find **longest/shortest** subarray/substring that satisfies some **condition**"*
> *Window size fix नहीं है — condition के हिसाब से बढ़ेगा या घटेगा*

Code Pattern:

```
let start = 0;
let set = new Set();
for (let end = 0; end < s.length; end++) {
    while (set.has(s[end])) {
        set.delete(s[start]);
        start++;
    }
    set.add(s[end]);
    maxLength = Math.max(maxLength, end - start + 1);
}
```

Pointers Movement:

- `end` हमेशा बढ़ेगा (expand)
- `start` तब बढ़ेगा जब condition fail हो जाए (shrink)

कैसे याद रखें?
Fixed size:
"माल अंदर करो, जब bag full हो जाए तो oldest निकालो"
Variable size:
"जितना condition माने, उतना फैलाओ — वरना काटो"

```
arr = [1, 2, 3, 4, 5, 6]
       ↑     ↑
     start  end   → (window size = 3)

window = [1, 2, 3]

अब move करो:
```

```
          ↑     ↑
        start  end
window = [2, 3, 4]
```

तो हाँ — **window size = k**, ना कि `arr.length`
और इसी को fix रखते हुए हम calculations करते हैं।

Sliding Window में Initial Setup क्या होता है?

Sliding Window = खिड़की → जिसमें दो pointer (start, end) चलते हैं
शुरुआत में हमें:

1. कुछ pointers initialize करने होते हैं (जैसे `start = 0`, `end = 0`)
2. कुछ variables रखने होते हैं, जैसे:
   - `maxSum`, `minLen`, `windowSum`, `maxLength`, etc.
3. कुछ helper data structures अगर चाहिए हों (Set, Map, Hash, etc.)

```
let s = "abcadc"
let start = 1
let end = 3

Index :     0     1     2     3     4     5
Chars :     a     b     c     a     d     c
             ↑           ↑
           start       end

Window :        [ b   c   a ]
```

- यह है **Sliding Window**
- मतलब **window हमेशा start से लेकर end तक होता है**
- उसका length होता है: `end - start + 1`

```
Window = सारा हिस्सा जो start से लेकर end तक आता है
Window Length = end - start + 1
```

Sliding का मतलब?

- अगर duplicate मिल जाए → `start++` करके window को **left से shrink करो**
- अगर सब कुछ valid हो → `end++` करके window को **right से grow करो**

Divide & conquer

"किसी बड़े problem को छोटे-छोटे हिस्सों में बांटो (Divide करो), हर हिस्से को अलग से हल करो (Conquer करो), और फिर सबको मिलाओ (Combine करो)।"

क्यों इस्तेमाल होता है?

- बड़े problems को efficiently हल करने के लिए।
- Time complexity कम करने के लिए।
- Recursion के साथ बहुत useful होता है।

कहां इस्तेमाल होता है?

बहुत सारे famous algorithms Divide & Conquer का इस्तेमाल करते हैं:

Algorithm	Use
Merge Sort	Sorting
Quick Sort	Sorting
Binary Search	Searching
Karatsuba	Fast multiplication
Merge Intervals / Inversion Count	Array problems

Divide & Conquer का Flow कैसे याद रखें?

Step by Step Flow:

```
[ 5, 2, 1, 8, 6, 3 ]

1️⃣ Pivot चुनो (usually आखिरी element) → pivot = 3

2️⃣ दो pointer रखो:
    - i = -1 (smaller elements का pointer)
    - j = 0 से start करके j< =n-2 तक जाओ

3️⃣ Compare each element with pivot:
   अगर arr[j] < pivot हो → i++ और arr[i] ↔️ arr[j] swap

4️⃣ Loop के बाद pivot को arr[i+1] के साथ swap करो
   (pivot सही जगह आ गया)

5️⃣ अब दो हिस्सों में divide करो:
   - Left part → 3 से छोटे elements
   - Right part → 3 से बड़े elements

6️⃣ अब recursive call करो left और right दोनों हिस्सों पर
```

याद रखने की Trick

```
Pivot → Partition → Swap → Divide → Recursion
पिवट लो → बाँटो → छोटे-बड़े को swap करो → दो भाग → हर भाग को दोबारा sort करो
```

Quick Sort

```
function quickSort(arr, low, high) {
  if (low < high) {
    const pi = partition(arr, low, high); // 1. पिवट को सही जगह पर रखो
    quickSort(arr, low, pi - 1);          // 2. left side sort करो
    quickSort(arr, pi + 1, high);         // 3. right side sort करो
  }
}

function partition(arr, low, high) {
  const pivot = arr[high];
  let i = low - 1;

  for (let j = low; j < high; j++) {
    if (arr[j] < pivot) {
      i++;
      [arr[i], arr[j]] = [arr[j], arr[i]];
    }
  }

  [arr[i + 1], arr[high]] = [arr[high], arr[i + 1]];
  return i + 1;
}
```

Binary Search (Iterative version)

```
function binarySearch(arr, target) {
  let left = 0;
  let right = arr.length - 1;

  while (left < = right) {
    const mid = Math.floor((left + right) / 2);

    if (arr[mid] === target) {
      return mid; // Found target at index `mid`
    } else if (arr[mid] < target) {
      left = mid + 1; // Search in the right half
    } else {
      right = mid - 1; // Search in the left half
    }
  }
```

```
  return -1; // Target not found
}
```

Merge Sort क्या है?

Merge Sort एक Divide and Conquer algorithm है।
इसका मतलब है:
बड़े problem को छोटे-छोटे हिस्सों (sub-problems) में तोड़ना (Divide)
हर हिस्से को solve करना (Conquer)
और फिर वापस जोड़कर final result बनाना (Combine/Merge)
ये एक stable sorting algorithm है — यानी दो same value वाले items की order नहीं बदलती।

Algorithm का Logic (Behind the Scenes)

सोचो तुम्हारे पास एक array है:
`[8, 4, 2, 7, 5]`
Step 1: Divide — Array को 2 parts में divide करो

```
Left:  [8, 4]
Right: [2, 7, 5]
```

Step 2: फिर हर part को तब तक divide करते रहो जब तक हर part 1 element का न बन जाए

```
[8], [4], [2], [7], [5]
```

Step 3: अब दो-दो parts को merge करो — और merge करते वक्त उन्हें **sorted way** में जोड़ो

```
[8] + [4] → [4, 8]
[2] + [7] → [2, 7]
अब [2, 7] + [5] → [2, 5, 7]
अब [4, 8] + [2, 5, 7] → [2, 4, 5, 7, 8]
```

3. Flowchart Type Steps (Algorithm Flow):

```
function mergeSort(arr):
 अगर arr.length <= 1:
   return arr

 middle = arr.length / 2
 left = arr.slice(0, middle)
 right = arr.slice(middle)

 sortedLeft = mergeSort(left)
 sortedRight = mergeSort(right)
```

```
  return merge(sortedLeft, sortedRight)
```

4. Merge Function कैसे काम करता है? (Core Magic)

```
function merge(left, right):
  result = []

  दो pointer रखो — leftBox, rightBox = 0

  जब तक दोनों arrays में elements हैं:
    अगर left[leftBox] छोटा है:
      result.push(left[leftBox])
      leftBox++
    वरना:
      result.push(right[rightBox])
      rightBox++

  बचे हुए elements (किसी एक array में बच सकते हैं):
    result.push( ... left बचा हो तो)
    result.push( ... right बचा हो तो)

  return result
```

Time & Space Complexity:

Case	Time Complexity
Best Case	O(n log n)
Average	O(n log n)
Worst Case	O(n log n)
Space	O(n) → क्योंकि हर बार नया array बनता है

> ***Quick Sort की तरह ही divide करता है**, लेकिन इसमें pivot नहीं होता। Merge Sort हर बार पूरे array को divide करता है और फिर merge करता है।*

कहाँ Use होता है?

- जब stable sort ज़रूरी हो
- जब linked list या external sorting (file sorting) करनी हो

> **Important**

(टुकड़े टुकड़े करो → sort करो → फिर जोड़ो)

```
function merge(left, right) {
  const result = [];
  let leftIndex = 0;
  let rightIndex = 0;
  // दोनों arrays के elements को compare करके result में push करो
  while (leftIndex < left.length && rightIndex < right.length) {
    if (left[leftIndex] < right[rightIndex]) {
      result.push(left[leftIndex]);
      leftIndex++;
    } else {
      result.push(right[rightIndex]);
      rightIndex++;
    }
  }
  // जो elements बच गए हैं उन्हें भी जोड़ दो
  return result
    .concat(left.slice(leftIndex))
    .concat(right.slice(rightIndex));
}
function mergeSort(arr) {
  // base case: अगर array में 1 या 0 elements हैं, वो already sorted है
  if (arr.length <= 1) {
    return arr;
  }
  const middle = Math.floor(arr.length / 2);
  const left = arr.slice(0, middle);
  const right = arr.slice(middle);
  // पहले left और right को recursively sort करो
  const sortedLeft = mergeSort(left);
  const sortedRight = mergeSort(right);
  // फिर उन्हें merge करो
  return merge(sortedLeft, sortedRight);
}
```

Insertion Sort

- मान लो हमारे पास एक array है।
- हम assume करते हैं कि पहला element पहले से sorted है।
- अब हम दूसरा element उठाते हैं (`current`) और उसे left side में जितने भी elements हैं उनसे compare करते हैं:
  - अगर वो current से **बड़े हैं** → तो उन्हें **एक जगह right में shift** कर दो।
  - जब कोई ऐसा element मिले जो current से छोटा है या ख़त्म हो जाए — **बस वहीं current को रख दो**।

```
function insertionSort(arr) {
  for (let i = 1; i < arr.length; i++) {
    let current = arr[i];  // 🔸 1. current element उठा लो
    let j = i - 1;         // 🔸 2. current से पहले वाला index

    // 🔁 3. left side के elements से compare करो
    // जब तक arr[j] current से बड़ा है → उसे right में shift करो
    while (j >= 0 && arr[j] > current) {
      arr[j + 1] = arr[j]; // 🔸 4. बड़ा element एक जगह right में shift करो
      j--;                 // 🔸 5. एक step और पीछे जाओ
    }

    // 🔸 6. अब जो जगह खाली हुई है (j+1), वहाँ current को डाल दो
    arr[j + 1] = current;
  }

  return arr; // 🔸 7. sorted array return कर दो
}
let arr = [5, 3, 8, 2, 1];
insertionSort(arr);
console.log(arr); // [1, 2, 3, 5, 8]
```

Visualization (Step-by-step):

1. `i = 1` → current = `3`, left = `[5]`
   - 5 > 3 → shift 5 to right → insert 3 → `[3, 5, 8, 2, 1]`
2. `i = 2` → current = `8`, left = `[3, 5]`
   - कोई shift नहीं → `[3, 5, 8, 2, 1]`
3. `i = 3` → current = `2`, left = `[3, 5, 8]`
   - 8 > 2 → shift
   - 5 > 2 → shift
   - 3 > 2 → shift → insert 2 → `[2, 3, 5, 8, 1]`
4. `i = 4` → current = `1`, left = `[2, 3, 5, 8]`
   - सब shift → insert 1 → `[1, 2, 3, 5, 8]`