Optimal/Optimised Approach

Constraints

Constraints define the **range of values** for input variables in a problem. They help you understand:

- What size of input (n) is allowed
- What time and space efficiency is expected
- Whether brute force will work or not

Example:

$1 \le n \le 10^5$

$1 \le arr[i] \le 10^9$

Means: Input array can be of size up to 100,000 and elements up to 1 billion.

Why Are Constraints Important?

Understanding constraints allows you to:

- Predict the correct **algorithm complexity**
- Decide whether **brute force**, **DP**, or **optimized techniques** are needed
- Avoid **TLE (Time Limit Exceeded)** or **MLE (Memory Limit Exceeded)**
- Estimate how many **operations per second** are allowed

Modern computers $\rightarrow$ 10^8 ops/sec

problem $\rightarrow$ 1sec $\rightarrow$ 1.46 TLE/MLE

✅ 1. Time Limit: ≤ 10^8 operations (≈ 1s)

Acceptable Complexity: `O(n)`, `O(n log n)`

Case 1: `O(n)`

- Let's assume `n = 10⁸`
- Then number of operations = `n = 10⁸` ✔

Case 2: `O(n log n)`

Let's assume `n = 10⁶`

$$\log_2(10^6) = \frac{\log_{10}(10^6)}{\log_{10}(2)} = \frac{6}{0.3010} \approx 19.93 \approx 20$$

$$n \cdot \log_2 n = 10^6 \cdot 20 = 2 \cdot 10^7 \text{ operations}$$

✅ `2 × 10⁷` is **well below** `10⁸` → acceptable.

Time Limit vs Acceptable Complexity (With Proofs)

Time Limit	Acceptable Complexity	Proof & Reasoning
$\le 10^8$ ops (1s)	$O(n)$, $O(n \log n)$	CPU handles $\sim 10^8$ ops/sec. For $n = 10^6$, $\log n \approx 20$ → $10^6 \times 20 = 2\times10^7$ ✔
$\le 10^7$ ops	$O(n \log n)$, $O(n \sqrt{n})$	For $n = 10^5$, $\log n \approx 17$ → 1.7×10^6 ops ✔ For $n = 10^4$, $\sqrt{n} \approx 316$ → 3.1×10^7 ops ✔
$\le 10^6$ ops	$O(n^2)$	For $n = 1000$ → 10^6 operations = acceptable
$\le 10^5$ ops	$O(n^2)$, $O(n\sqrt{n})$ (small constants)	For $n = 300$ ⇒ 90,000 ops ✔ For $n = 1000$, $\sqrt{n} = 31$ → 31,000 ops ✔
$\le 10^4$ ops	$O(n^3)$, $O(n^2 \log n)$	For $n = 100$ ⇒ 10^6 ops ✔ For $n = 300$: $90{,}000 \times \log n \approx 9\times10^5$ ✔

Use this to **estimate complexity just from time limit and n**.

$10^2 \to n^3$

$10^6 \to 10^6$

$O(n)$

$\sqrt{n}$

$n \log n$

10^2

10^6

Arr = [ ], $n = 10^6$ → $O(n)$ → $\log n$ → $n \log n$

$1 \le n \le 10^6$

10^8 TLE

C_1 → $n \log n$ → $\log_2$

$(n \log_2 n)$

→ $10^6 \times \log_2(10^6)$

$n = 10^6$

$C_1 \rightarrow n\log_2 n \rightarrow 10^6 \times \overline{\log_2(10^6)}^{\,20}$

$\log_2(10^6) \longrightarrow \dfrac{\log_{10}(10^6)}{\log_{10}(2)} \Rightarrow \dfrac{6 \times \cancel{\log_{10} 10}^{\,1}}{\cancel{\log_{10}(2)}_{\,0.3010}}$

$\dfrac{6}{0.300} \Rightarrow 19.98 \Rightarrow 20$

$n = 10^6$

$n \log_2 n \longrightarrow 10^6 \times 20 \Rightarrow 2 \times 10^1 \times 10^6$

$2 \times 10^7 < 10^8$

$\geq 10^5$ to $10^8 \longrightarrow$ $n \log n$, n, $\log n$, constant, $\sqrt{n}$ $\longrightarrow$ Acceptable

$n = 10^6 \longrightarrow$

$C_1 \longrightarrow (n^2)$

$(10^6)^2 \longrightarrow 10^{12} > 10^8$

(n^2) TLE

$n = 10^5$

$(10^5)^2 \rightarrow 10^{10} \Rightarrow$ TLE

Optimal Approaches

Category	Common Techniques / Examples
1. Searching	Linear Search, Binary Search, Two Pointer, Sliding Window
2. Sorting	Bubble, Selection, Insertion, Merge, Quick, (Counting, Radix)
3. Greedy	Activity Selection, Huffman Coding
4. Dynamic Programming	Knapsack, Longest Common Subsequence (LCS), Fibonacci
5. Graph Algorithms	BFS, DFS, Dijkstra, Kruskal, Prim, Topological Sort
6. Divide & Conquer	Merge Sort, Binary Search, Quick Sort
7. Backtracking	N-Queens, Sudoku Solver, Subset Sum
8. Bit Manipulation	XOR Tricks, Count Set Bits, Check Power of Two
9. Mathematical	GCD (Euclidean), Sieve of Eratosthenes, Fast Exponentiation
10. Advanced DS	Trie, Union-Find (Disjoint Set), Segment Tree, Fenwick Tree

Introduction to Two Pointer

Two Sum Problem:-

Two Sum Problem:-

n=8, k=18

i

arr

0	1	2	3	4	5	6	7
7	4	9	6	21	8	11	17

$(7,4) \neq K$

$(7,9) \neq K$

$(7,6) \neq K$

$(7,21) \neq K$

$(7,8) \neq K$

$(7,11) == K \longrightarrow$ return True

```
function twoSum(arr, target) {
    let n = arr.length;
    for(let i = 0; i < n; i++)
    {
        for(let j = i + 1; j < n; j++)
        {
            if(arr[i] + arr[j] === target)
            {
                return true;
            }
        }
    }
    return false;
}
```

arr

0	1	2	3	4	5	6	7
7	4	9	6	21	8	11	17

$n = 8 \longrightarrow 2$

$^nC_r \Rightarrow \dfrac{n!}{(n-r)! \times r!}$

$^nC_2 \Rightarrow \dfrac{n!}{(n-2)! \times 2!} \Rightarrow \dfrac{n!}{(n-2)! \times 2}$

$\Rightarrow \dfrac{n \times (n-1) \times \cancel{(n-2)!}}{\cancel{(n-2)!} \times 2}$

$\dfrac{n(n-1)}{2} \Rightarrow O(n^2)$

$$O(1) < O(\log n) < O(\sqrt{n}) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)$$

Note:-

MAX value of N	Time complexity
10^8	O(N) Border case
10^7	O(N) Might be accepted
10^6	O(N) Perfect
10^5	O(N * logN)
10^4	O(N ^ 2)
10^2	O(N ^ 3)
10^9	O(logN) or Sqrt(N)

USE TWO POINTER IN **LAST PROBLEM**

$n = 8$ $K = 18$

step1 :- sort the array (↑ order)

arr

0	1	2	3	4	5	6	7
7	4	9	6	21	8	11	17

min max

arr

0	1	2	3	4	5	6	7
4	6	7	8	9	11	17	21

L H

Two pointer

Step1 → Array must be in sorted order

$7 + 11 \rightarrow 18$

$A[L] + A[H]$ vs K

$Arr[L] + A[H] == K$ — true — 18

$A[L] + A[H] < K$ — L++ — 18

$A[L] + A[H] > K$ — H-- — 18

arr

0	1	2	3	4	5	6	7
4	6	7	8	9	11	17	21

$n = 8$

$K = 18$

```javascript
function twoSum(arr, target)
{
    arr.sort((a, b) => a - b);
    let left = 0, right = arr.length - 1;
    while (left < right)
    {
        let sum = arr[left] + arr[right];
        if (sum === target)
            return true;
        else if (sum < target)
            left++;
        else
            right--;
    }
    return false;
}
```

nlogn

L < R   L ≠ R   R > L

arr

0	1	2	3	4	5	6	7
4	6	7	8	9	11	17	21

L ⟶   ⟵ R

$n \log n < n^2$

sor(

sor(arr.bayd), arr.(nd())

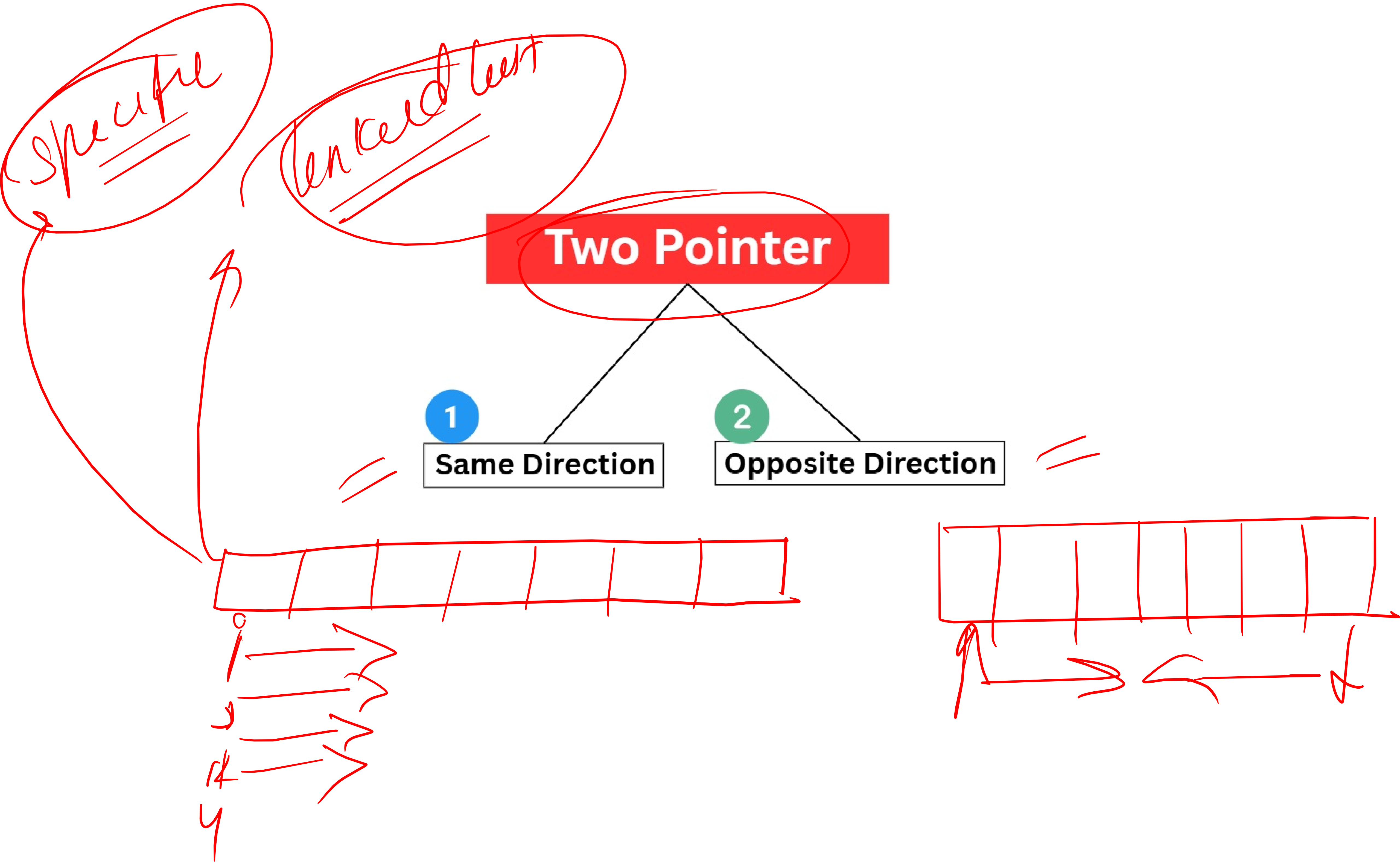
$n \log n + n$

Remove Duplicates from Sorted Array

Input: *arr[] = [2, 2, 2, 2, 2]*
Output: *[2]*
Explanation: *All the elements are 2, So only keep one instance of 2.*

Input: *arr[] = [1, 2, 2, 3, 4, 4, 4, 5, 5]*
Output: *[1, 2, 3, 4, 5]*

Input*: arr[] = [1, 2, 3]*
Output*: [1, 2, 3]*
Explanation *: No change as all elements are distinct.*


Two Pointer
1
Same Direction
2
Opposite Direction
Specific

Opposite Direction

1) Find a pair whose sum is equal to k [a+b=k]

arr

0	1	2	3	4	5	6	7
7	4	9	6	21	8	11	17

$n=8$

$k=16$.

```
function chkPair(arr,n,k)
{

        for (i = 0; i < n-1; i++)
        {
            for (j = i + 1; j < n; j++)
            {

                if (arr[i] + arr[j] == k)
                {
                    return true;
                }
            }
        }

        return false;
}
```

arr

0	1	2	3	4	5	6	7
7	4	9	6	21	8	11	17

$n = 8$

$K = \underline{46}$

0	1	2	3	4	5	6	7

```
function chkPair(arr,n,k)
{
        arr.sort();
        l=0, r=n-1;
        while(l<r)
        {
              if(arr[l]+arr[r]==k)
                    return true
              else if(arr[l]+arr[r]<k)
                    l++
              else
                    r--;
         }
         return false;
}
```

2) Find a triplet whose sum is equal to k [a+b+c=k]

arr

0	1	2	3	4	5	6	7
7	4	9	6	21	8	11	17

i j k

$n = 8 - 3$ 5

$K = 33$

$(7, 4, 9) == K$

$O(n^3)$

L1 for($i \rightarrow n-2$)

L2 for($j \rightarrow n-1$)

L3 for($k \rightarrow n$)

arr

0	1	2	3	4	5	6	7
7	4	9	6	21	8	11	17

0	1	2	3	4	5	6	7

```
function chkPair(arr,n,k)
{
        arr.sort();
        for(i=0;i<n-2;i++)
        {
            l=i+1
            r=n-1
            while(l<r)
            {
                if(arr[i]+arr[l]+arr[r]==k)
                    return true
                else if(arr[i]+arr[l]+arr[r]<k)
                    l++
                else
                    r--;
            }
        }
        return false;
}
```

(n)

(n)

0	1	2	3	4	5	6	7
7	4	9	6	21	8	11	17

0	1	2	3	4	5	6	7
4	6	7	8	9	11	12	21

i L → ← R

$10^3 \rightarrow 1000$

$O(n^2) \rightarrow 100$

$n \log n + n^2$

3) Seperate 0's and 1's

$M_1 \rightarrow$ 2ptr (opposite-dir)

arr

0	1	2	3	4	5	6	7	8	9	10	11	12
1	1	0	0	0	1	1	0	0	1	0	0	0

n = 13

L L   R R

o/p arr

0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	0	0	1	1	1	1	1

App1 $\rightarrow$ sort

$(n \log n)$

App2 $\rightarrow$ Extra [0]

$\rightarrow O(n)$ $O(n)$

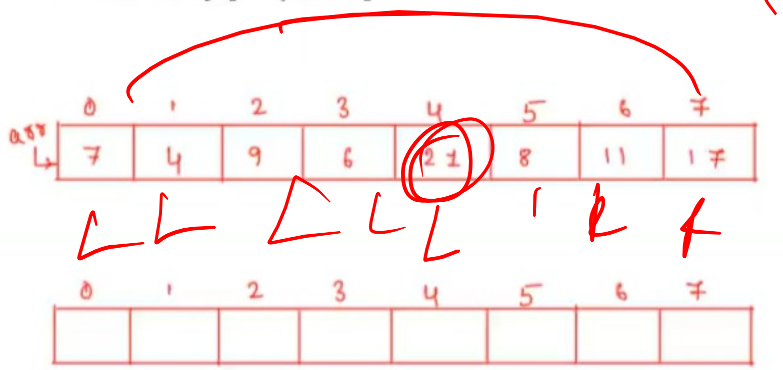
App3 $\rightarrow O(n)$

If array is already sorted?

$O(n)$

```
function segregate0and1(arr, n)
{

    let left = 0, right = n-1;

    while (left < right)
    {
        /* Increment left index while we see 0 at left */
        while (arr[left] == 0 && left < right)
            left++;

        /* Decrement right index while we see 1 at right */
        while (arr[right] == 1 && left < right)
            right--;

        /* If left is smaller than right then there is a 1 at left
        and a 0 at right. Exchange arr[left] and arr[right]*/
        if (left < right)
        {
            arr[left] = 0;
            arr[right] = 1;
            left++;
            right--;
        }
    }
}
```

4. Reverse the array [in-place]



L < R

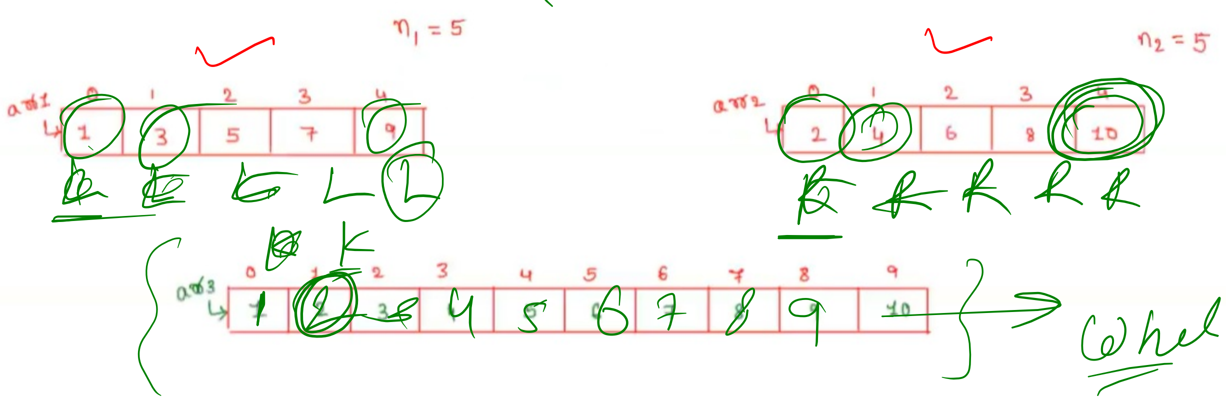
Interview

O(n)

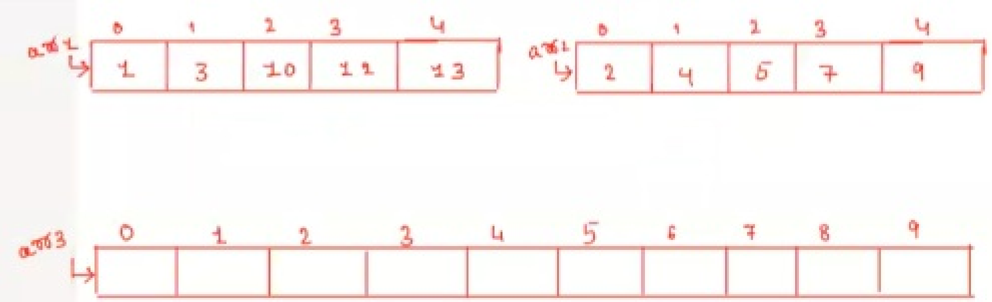
Same Direction Pointer

5) Merge Two Sorted Arrays

O(n)



```
function mergeTwoSortedArrays(arr1,n1,arr2,n2,arr3,n)
{
      i=0, j=0, k=0
      while(i<n1 && j<n2)
      {
            if(arr1[i]<arr2[j])
            {
                arr3[k]=arr1[i]
                i++
                k++
            }
            else
            {
                arr3[k]=arr2[j]
                j++
                k++
            }
      }
      while(i<n1)
      {
            arr3[k]=arr1[i]
            i++
            k++
      }
      while(j<n2)
      {
           arr3[k++]=arr2[j++]
      }
}
```



6) Remove Duplicates from Sorted array

Assignment

n = 9

arr

0	1	2	3	4	5	6	7	8
1	1	1	2	2	3	4	4	4

L

R

Two Pointer

n = 9

arr

0	1	2	3	4	5	6	7	8
1	1	1	2	3	4	4	4	5

```
function removeDupSortedArray(arr, n)
{
        j=0
        for(i=0;i<=n-2;i++)
        {
              if(arr[i]!=arr[i+1])
              {
                    arr[j]=arr[i]
                    j++
              }
        }
        arr[j]=arr[n-1]

        for(i=0;i<=j;i++)
        {
            print(arr[i])
        }
}
```

arr

0	1	2	3	4	5	6	7	8
1	1	1	2	2	3	4	4	4