DSA patterns

1. Two pointers:

what is two pointers?

The two pointer technique involves using two indices (pointers) to iterate over a data structure (usually an array or a string) to solve problems efficiently by avoiding nested loops.

when to use?

* when you need to find pairs, triplets or subarrays meeting certain conditions
* when the data is sorted or can be sorted.
* when you want to optimize brute force solutions that use nested loops ($O(n^2)$) to linear or near linear time ($O(n)$).

26. Remove Duplicates from sorted array:

I/P: nums = [1,1,2]

O/P: 2, nums = [1,2,_]

Explanation: your function should return k = 2 with the first two elements of nums being 1 and 2 respectively. It doesn't matter what you leave beyond the returned k (hence they are underscores).

Brute force:

```
class Solution {
    public int removeDuplicates(int [] num) {
        if (nums.length = = 0) return 0;

            int k = 1;

        for (int i = 1 ; i < nums.length ; i++) ;
            boolean isDuplicate = false;
for (int j = 0; j < k ; j++) {
    if (nums [i] = = nums[j]){
        isDuplicate = true;
        break;
    }
  }
  if (!isDuplicate) {
     nums[k] = nums[i];
     k++;
  }
 }
 return k;
}
}
```

But since we want in-place and no extra array, we need pointers.

Two pointer :

Since the array is sorted, duplicates are next to each other.

we two pointers.

i → keeps track of the position of last unique number.

j → moves through the array.

step :

1. start with i = 0 → first element is always unique.

2. Loop j from 1 to end of array:
* If nums [j] ! = nums [i] → found a new uniquenumber
  * Increment i → i = i + 1
  * copy nums [j] to nums [i] → nums [i] = nums [j]

3. At end, i + 1 is the count of unique numbers (k).

```
class Solution {
    public int removeDuplicates (int [] nums) {
        if (nums. length = = 0) return 0;
        int i = 0;
        for (int j = 1 ; j < nums . length ; j++) {
            if (nums [j] ! = nums [i]) {
                i++ ;
                nums [i] = nums [j] ;
            }
        }
        return i + 1;
    }
}
```

for printing the array elements :

```
int k = i + 1
return Arrays. copyOfRange (nums,
                   0, k);
```

Dry run:

nums = [1, 1, 2]

k = 1 (first element 1 is unique)

i = 1 → nums [1] = 1
check previous k = 1 element → duplicate → skip.

i = 2 → nums [2] = 2
check previous k = 1 element → not duplicate → nums [k] = 2 → k = 2

Final array : [1, 2 , _] → return k = 2.

Time → $O(n^2)$ space → $O(1)$ → in-place