Sorting, Arrays & Matrices

- Exercism: Python Lists
- Python.org: Sorting HOW TO

```
      0  1  2  3
l = [1, 2, 3, 4]
    -4 -3 -2 -1
```

l[0] (or) l[-4]

l[0:2] = [1, 2]

Two Pointer Approach

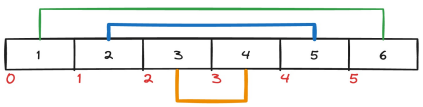
Definition:

It involves two indices to traverse the data structure, simplifying search and comparison operations.

Use Cases:

Finding pairs with a given sum, reversing arrays, and detecting palindromes

Example:



To track elements from two ends or positions in the data structure, which allows us to compare, sum, or swap elements efficiently.
This technique is particularly useful in problems where a brute force approach would require nested loops.

Real-world:

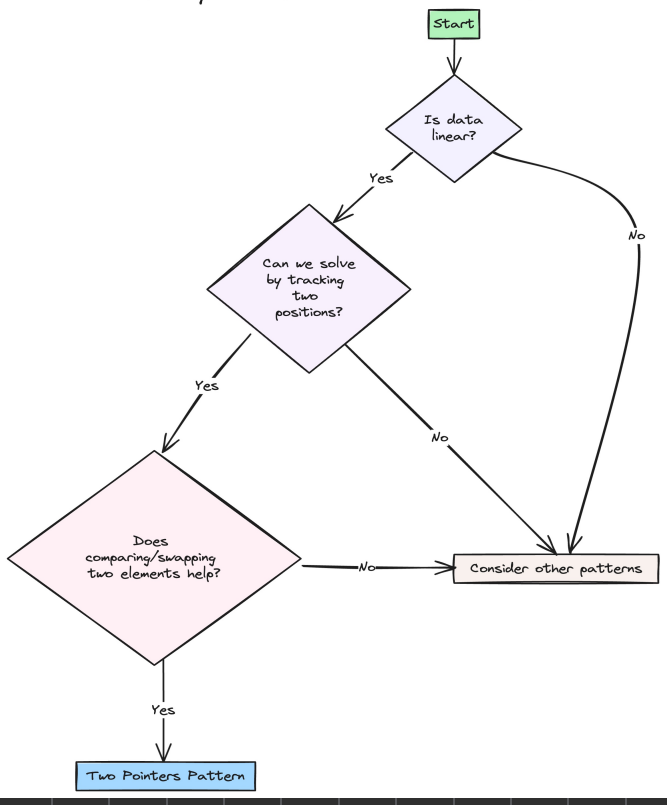
Memory Management: Allocation / Deallocation

Advantages:

Time complexities: O(n^2) vs. O(n).

Space complexity: avoiding auxiliary data structures.

Does My Problem Match This Pattern?



Sliding Window Approach

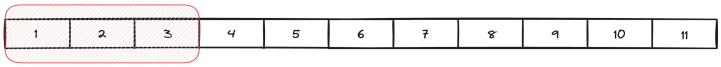
Definition:

Moving a subset of data (the "window") across a data structure to examine different segments efficiently.

Use Cases:

Calculating maximum sums, finding longest substrings without repeating characters.

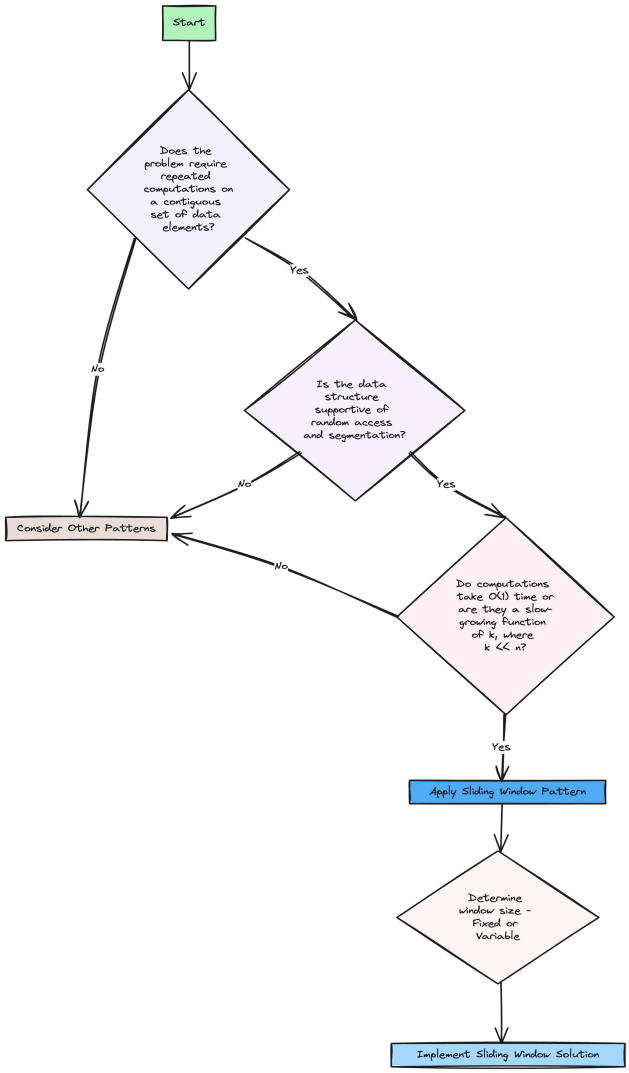
Example



How do you decide to use the Two Pointers or Sliding Window pattern?

By focusing on the nature of the problem, such as the need for contiguous segments or pairs meeting a condition.

Does My Problem Match This Pattern?



1.) Move Zeroes:

nums = [0, 1, 0, 3, 12]

[1, 3, 12, 0, 0]

ans = [ ]

[1, 3, 12, 0, 0]

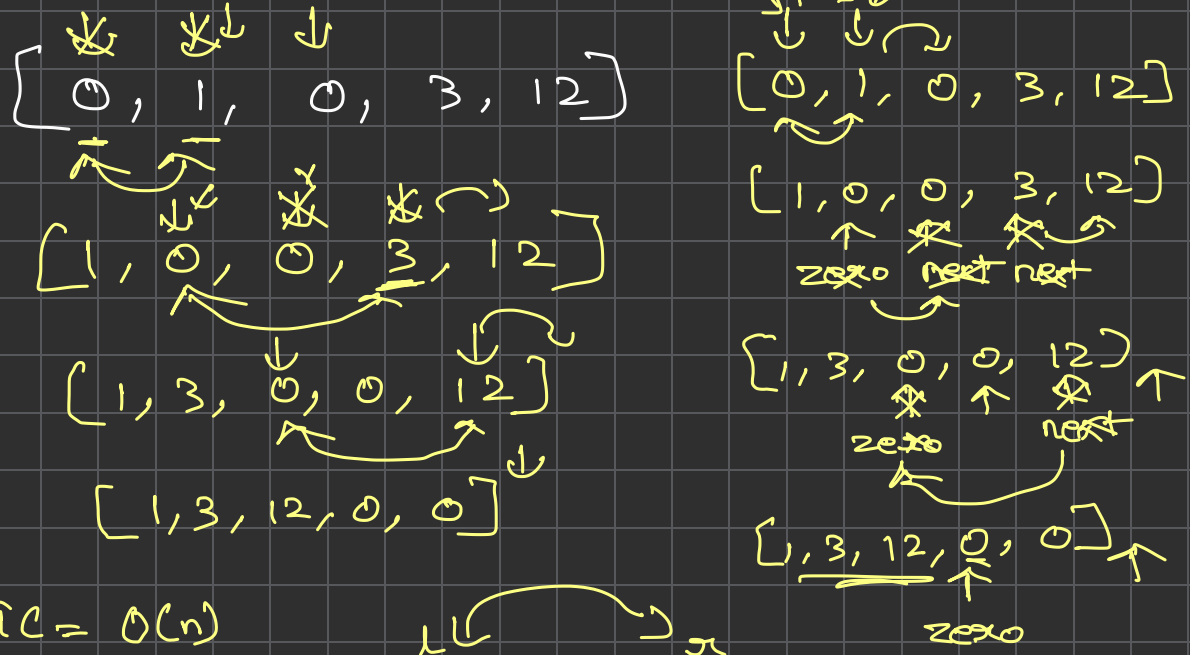
TC = O(n) + O(k) } O(n)

SC = O(n)

nums = [0, 1, 0, 3, 12]

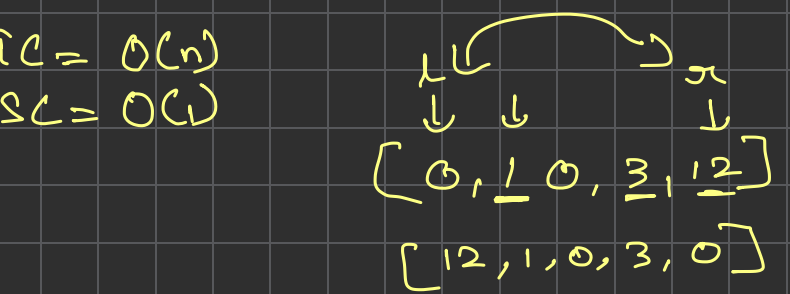
zero_index = 1st zero in array

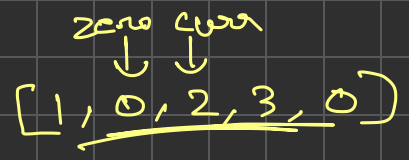
next_ptr = next element of zero index



TC = O(n)

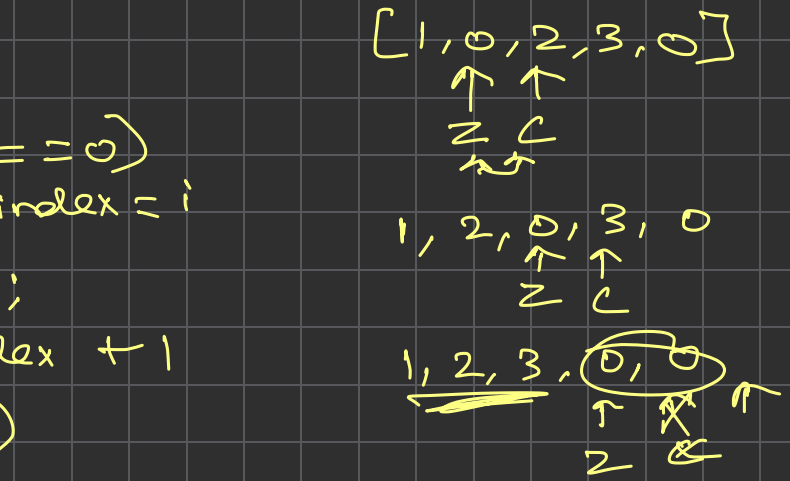
SC = O(1)





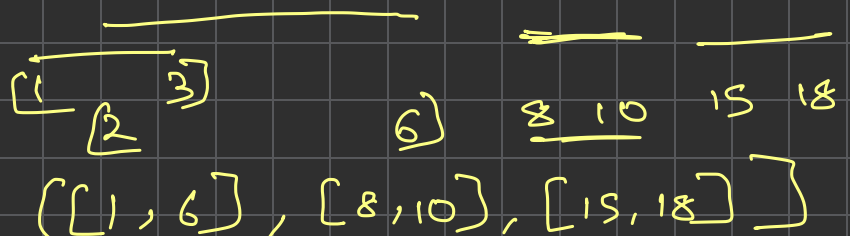
zero_index

```
for(i -> 0 - n-1)
    if (arr[i] == 0)
        zero_index = i
        break;
curr = zero_index + 1
while (curr < n)
{
    if arr[curr] != 0
        swap(arr[zero_ind], arr[curr])
        zero_ind += 1

    curr += 1

}
```



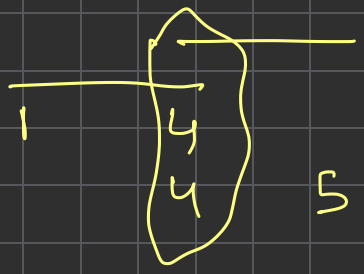
2) Merge Intervals:

intervals = [[1,3], [2,6], [8,10], [15,18]]



[[1,6], [8,10], [15,18]]

[[1,4], [4,5]]



[[1,5]]

[[4,7], [1,4]]

=> [[1,7]]

arr = [[1,3], [2,6], [8,10], [15,18]]

[1,6]

[8,10]

[15,18]

n } no. of lists in the parent list

ans = [[1,6], [8,10], [15,18]]

arr = [[4,7], [1,4]]

sort_arr = [[1,4], [4,7]]

[[1,7]]

arr = [[2,5], [3,4]]

Brute force

```
for (i -> 0 - n-2)
    for (j -> i+1 - n-1)
        arr[i]
        //overlap logic
ans.append([,])
```

=> O(2n) + O(n log(n))

=> O(k) -

sort_arr = [[1,3], [2,6], [8,10], [15,18]]

ans = [ ]

curr_int = [1,3]

[1,6]

ans = [[1,6], [8,10], [15,18]]

curr_int = [8,10]

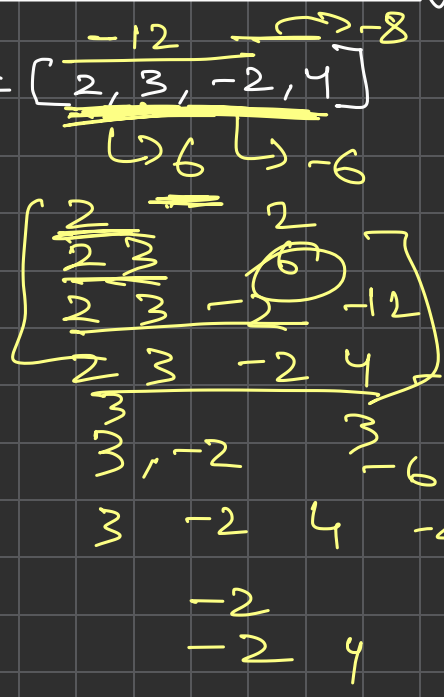
[15,18]

T.C = O(n) + O(n log(n)) → sorting the array

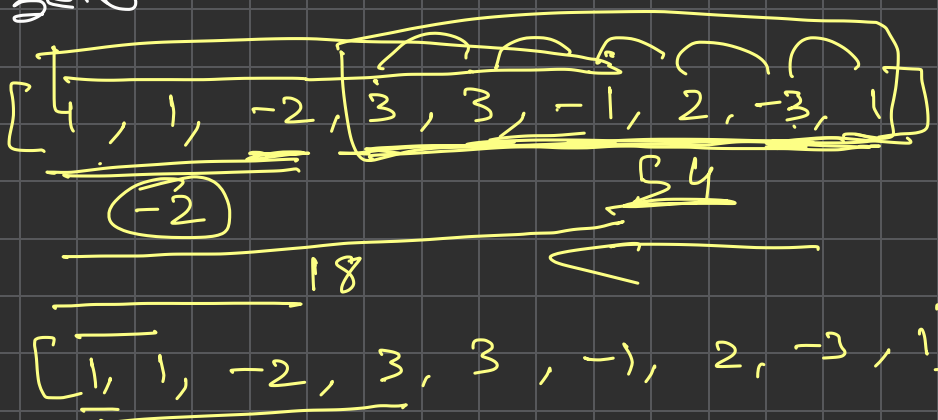
S.C = O(k)

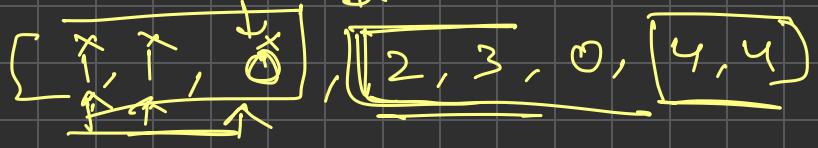
3.) Max Product Subarray:

nums = [2, 3, -2, 4]



1.) all +ve no.'s → [1, 2, 3, 4] = 24
2.) even -ve no.'s → [-1, -2, 3, 4] = 24
3.) odd -ve no.'s → [1, -2, 3, 4] = 12
4.) zero





⇒ Pseudo code for brute force:

Brute

T.C = $O(n^3)$

S.C = 1

```
n = len(nums)
max_product = -∞
for start = 0 to (n-1):
    for end = start to (n-1):
        product = 1
        for k = start to end:
            product = product x nums[k]
        max_product = max(max_product, product)
return max_product
```

Better

```
n = len(nums)
max_prod = -∞
for i = 0 to n-1:
    running_prod = 1
    for j = i to n-1:
        running_prod = running_prod * nums[j]
        max_prod = max(max_prod, running_prod)
return max_prod
```