Sorting, Arrays & Matrices

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

Two Pointer Approach

Definition:

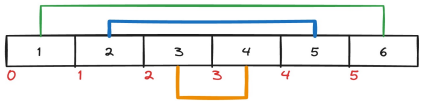
It involves two indices to traverse the data structure, simplifying search and comparison operations.

Use Cases:

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

l = [1, 2, 3, 4]
rev_l = [4, 3, 2, 1]

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:

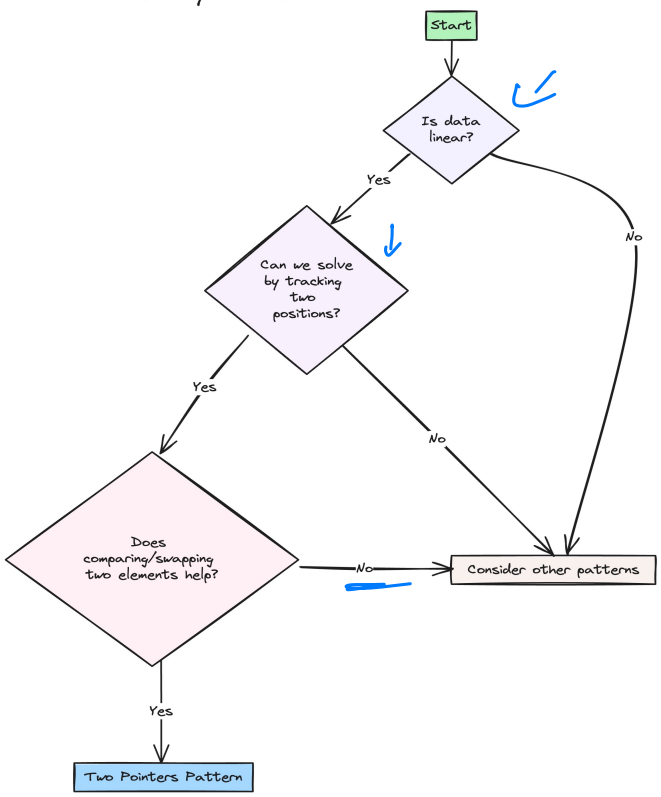
Memory Management: Allocation / Deallocation

Advantages:

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

Space complexity: avoiding auxiliary data structures.

Does My Problem Match This Pattern?



Sliding Window Approach

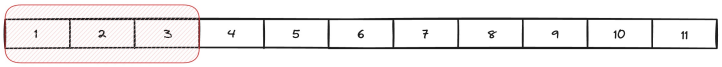
Definition:

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.

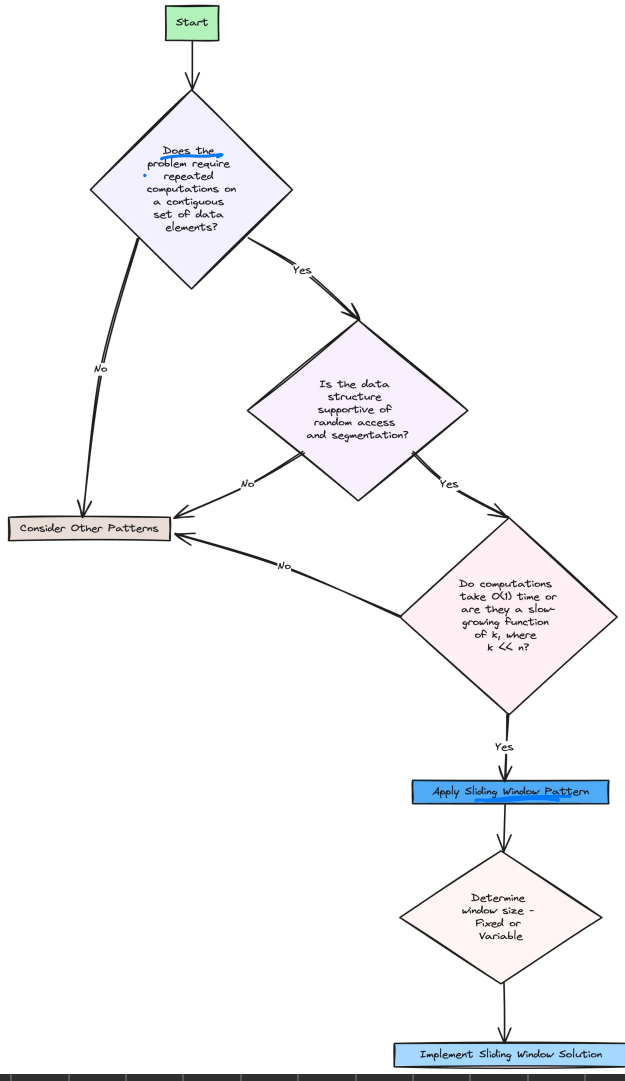
Example



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?



Problem - 1 : Move Zeroes

283. Move Zeroes

Easy | Topics | Companies | Hint

Given an integer array `nums`, move all `0`'s to the end of it while maintaining the relative order of the non-zero elements.

Note that you must do this in-place without making a copy of the array.

Example 1:

```
Input: nums = [0,1,0,3,12]
Output: [1,3,12,0,0]
```

Example 2:

```
Input: nums = [0]
Output: [0]
```

Constraints:

- `1 <= nums.length <= 10^4`
- `-2^31 <= nums[i] <= 2^31 - 1`

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

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

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

[1, 3, 12]

[1, 3, 12, 0, 0]

T.C = O(n) + O(x) + O(n-x)

O(x): no. of non zeroes

O(n-x): no. of zeroes

T.C = O(2n) → ✗

S.C = O(n)

n = len of list
zeroindex = 0
for i : 0 → n :
  if l[i] == 0 :
    zeroind = i
    break

curr_ind = zero_ind + 1
while (curr_ind < n)
{
  if (l[curr_ind] != 0)
  {
    swap(zeroind, curr_ind)
    zero_ind += 1
  }
  currentind + 1
}

i=1 j=1

l = [0, 1, 2, 3, 4]
[0, 1, 0, 3, 12]

z = 5

z = 0

[1, 1, 1, 3, 12]

i = 6

l = [0, 1, 2, 3, 4, 5]
[1, 0, 3, 0, 2, 12]

[1, 3, 0, 0, 2, 12]

[1, 3, 2, 0, 0, 12]

[1, 3, 2, 12, 0, 0]

T.C = O(n) ✓

S.C = O(1) ✓

Problem 2: Merge Intervals

56. Merge Intervals

Medium | Topics | Companies

Given an array of `intervals` where `intervals[i] = [start_i, end_i]`, merge all overlapping intervals, and return *an array of the non-overlapping intervals that cover all the intervals in the input.*

Example 1:

```
Input: intervals = [[1,3],[2,6],[8,10],[15,18]]
Output: [[1,6],[8,10],[15,18]]
Explanation: Since intervals [1,3] and [2,6] overlap, merge them into [1,6].
```

Example 2:

```
Input: intervals = [[1,4],[4,5]]
Output: [[1,5]]
Explanation: Intervals [1,4] and [4,5] are considered overlapping.
```

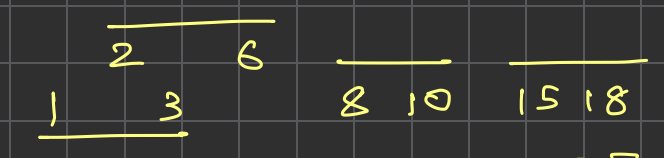
Example 3:

```
Input: intervals = [[4,7],[1,4]]
Output: [[1,7]]
Explanation: Intervals [1,4] and [4,7] are considered overlapping.
```

Constraints:

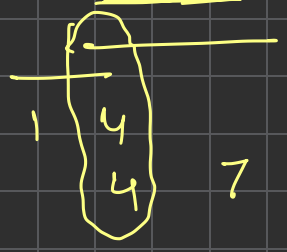
- `1 <= intervals.length <= 10^4`
- `intervals[i].length == 2`
- `0 <= start_i <= end_i <= 10^4`

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



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

l = [[4,7],[1,4]]



=> [[1,7]]

=> Sort the intervals } O(n log(n))

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

[[1,6],

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

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

[8,10] [15,18]

ans = [ ]

for(ind1 : 0 -> n)

if ind1 element overlaps with last ans.interval:

continue

start = l[ind1][0]

end = l[ind1][1]

for(ind2 : ind1 + 1 -> n)

if (l[ind2][0] <= end)

end = max(end, l[ind2][1])

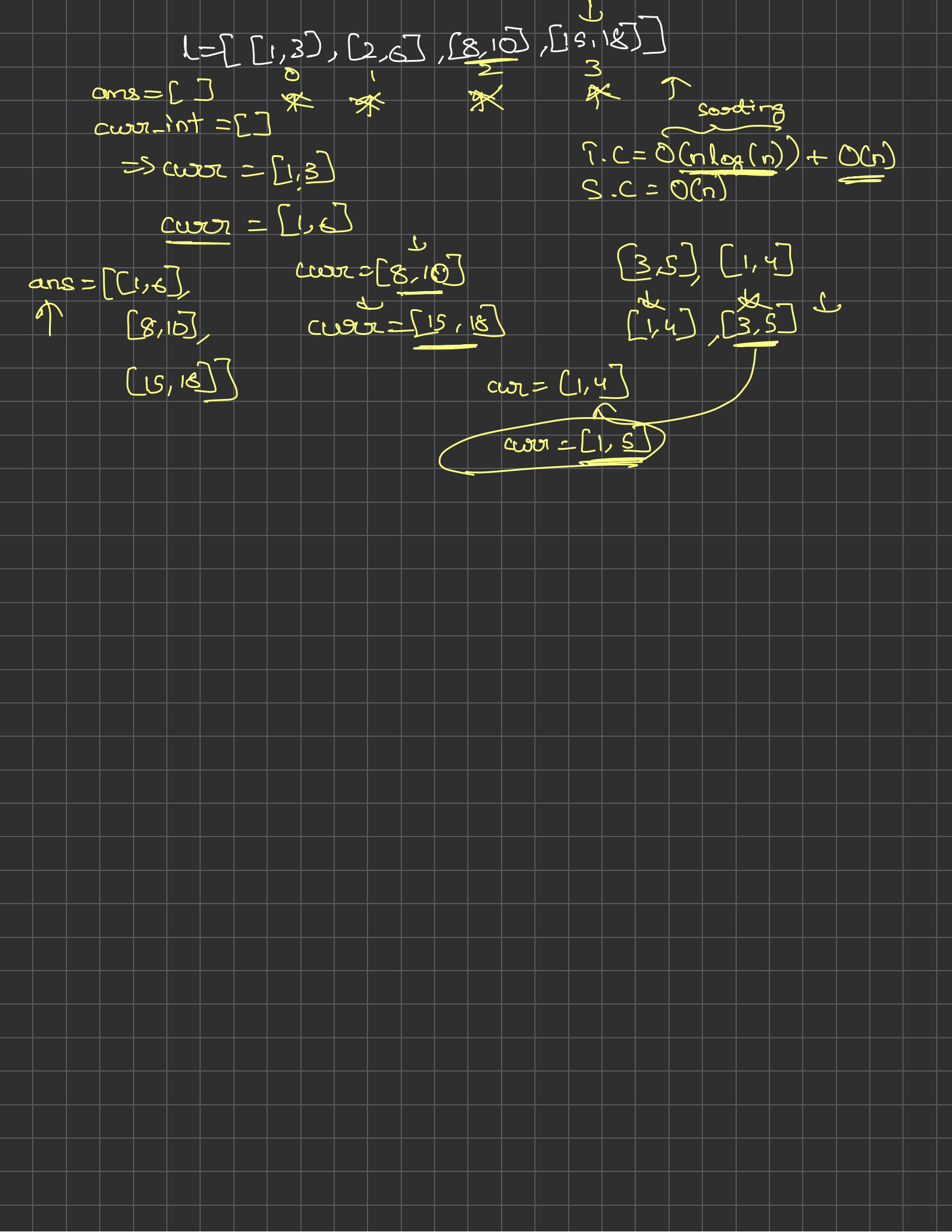
else

break

ans.apped([start, end])

T.C = O(2n) + O(n log(n))

S.C = O(n)


[[1,3],[2,6],[8,10],[15,18]]
0
1
2
3
ans=[ ]
curr_int=[ ]
=> curr=[1,3]
curr=[1,6]
ans=[[1,6],
[8,10],
[15,18]]
curr=[8,10]
curr=[15,18]
sorting
T.C=O(n log(n)) + O(n)
S.C=O(n)
[3,5], [1,4]
[1,4], [3,5]
curr=[1,4]
curr=[1,5]

Problem 3: Max Product Subarray

152. Maximum Product Subarray

Medium | Topics | Companies

Given an integer array `nums`, find a subarray that has the largest product, and return *the product*.

The test cases are generated so that the answer will fit in a **32-bit** integer.

Note that the product of an array with a single element is the value of that element.

Example 1:

```
Input: nums = [2,3,-2,4]
Output: 6
Explanation: [2,3] has the largest product 6.
```

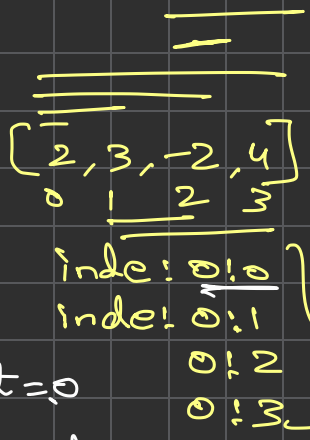
Example 2:

```
Input: nums = [-2,0,-1]
Output: 0
Explanation: The result cannot be 2, because [-2,-1] is not a subarray.
```

Constraints:

- `1 <= nums.length <= 2 * 10^4`
- `-10 <= nums[i] <= 10`
- The product of any subarray of `nums` is **guaranteed** to fit in a **32-bit** integer.

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



⇒ Pseudocode for brute force:

Brute

T.C = O(n³)

S.C = 1

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

i = 0

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

T.C = $O(n^2)$

S.C = $O(1)$

⇒ all elements are +ve

⇒ even no. of -ve elements

⇒ odd no. of -ve elements

[1, 2, 3, 4] => 24

[1, -2, 3, -4] => 24

[1, 2, -3, 1, 2, -4, 3, -1, 2]

⇒ when we have zero's

[1, 2, 0, 3, 4, 0, 5, 6]

0 1 2 3 4 5 6 7

pre = 1, suf = 1

ans = NEG_MAX

for i: 0 -> n

if pre == 0:

pre = 1

if suf == 0:

suf = 1

pre *= l[i]

suf *= l[n-i-1]

ans = max(pre, suf, ans)

pre = 1

suf = 6

ans = 6

pre = 2

suf = 30

ans = 30

pre = 6

suf = 5

ans = 30

pre = 3

suf = 4

30