Two Pointers and Sliding Windows

Presented by Alexander Mak

Two Pointers and Sliding Windows

- Two pointer and sliding window strategies allow us to avoid nested for loops that check every possible combination.
- These patterns generally allow us to achieve O(n)
- Two pointer
  - A strategy we employ when we are concerned with 2 specific values within a list.
- Sliding window
  - A strategy we employ when we are concerned with the CONTENTS of the subarray between the two pointers.

Two Pointer Analogy

Imagine you're online shopping with a budget and want to find two pieces of clothing within that budget. Instead of checking every possible pair, which would be time-consuming ($O(n^2)$), a smarter approach is to sort the results by price.

Using a two-pointer technique, you start with the highest and lowest priced items.

- If their sum exceeds your budget, you decrease the total by removing the higher priced item. (Decrementing right pointer)
- If the total is within budget, it's a valid pair. By incrementing or decrementing pointers accordingly, you efficiently explore options without redundant comparisons. (Incrementing left pointer)

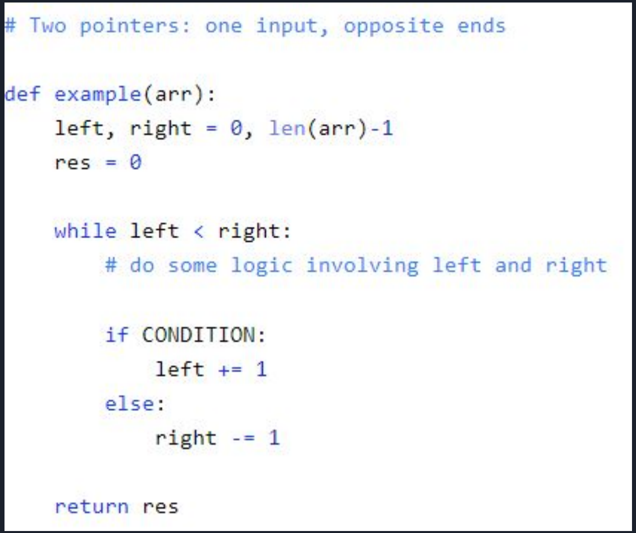
Sorting the list optimizes the process. Moving the right pointer eliminates the need to compare with more expensive items, and moving the left pointer ensures comparisons with all cheaper items were already made. This way, you streamline the search for clothing within your budget.

How to recognize a two pointer problem?

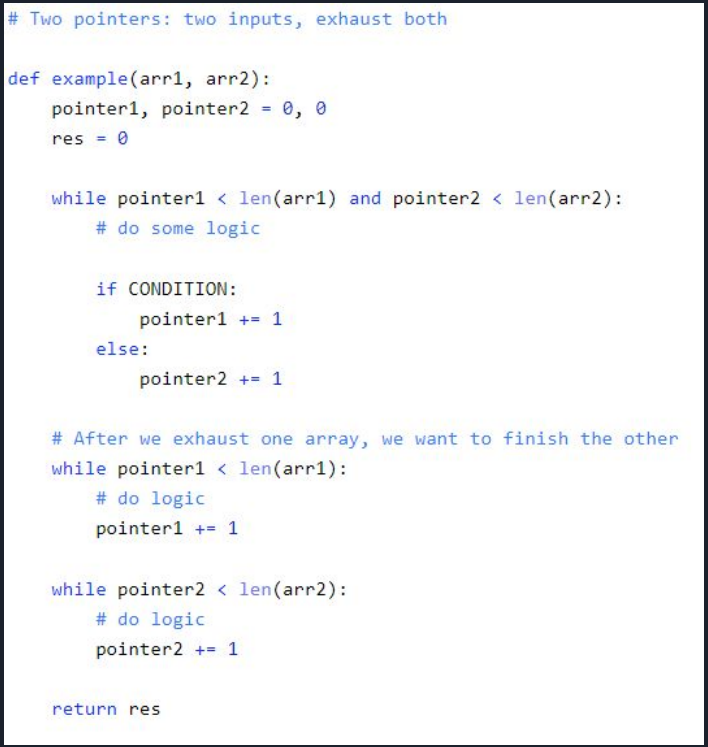
Two pointers is a very abstract idea and can be applicable for many different problems. Here are a few examples of what could hint at a two pointer approach.

1. Searching for a pair of elements
2. Counting pairs
3. Swapping or reversing elements
4. Sorting based problems
5. Moving boundaries
6. Detecting cycles (will discuss in the future with linked lists and graphs)

Two pointer examples



```python
# Two pointers: one input, opposite ends

def example(arr):
    left, right = 0, len(arr)-1
    res = 0

    while left < right:
        # do some logic involving left and right

        if CONDITION:
            left += 1
        else:
            right -= 1

    return res
```



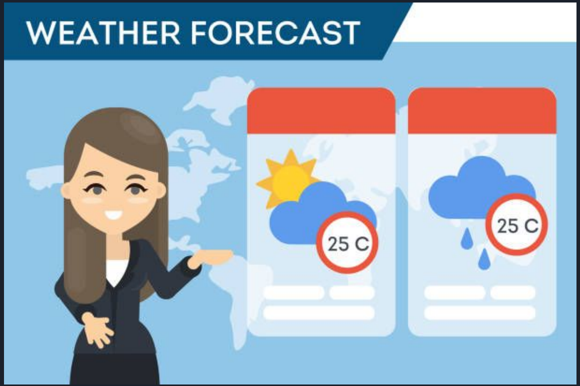
```python
# Two pointers: two inputs, exhaust both

def example(arr1, arr2):
    pointer1, pointer2 = 0, 0
    res = 0

    while pointer1 < len(arr1) and pointer2 < len(arr2):
        # do some logic

        if CONDITION:
            pointer1 += 1
        else:
            pointer2 += 1

    # After we exhaust one array, we want to finish the other
    while pointer1 < len(arr1):
        # do logic
        pointer1 += 1

    while pointer2 < len(arr2):
        # do logic
        pointer2 += 1

    return res
```

Fixed Sliding Window Analogy

You are a meteorologist who is interested in temperature trends throughout the year. You have a massive dataset with temperatures for every day of the last year. Instead of looking at all the data all at once, you decide to use a “sliding window approach.”

You analyze the first 7 days of the year (Jan 1-7). 7 days is your “fixed sized window”. You then add on the next day (Jan 8) into your analysis but remove the earliest day (Jan 1). You continue doing this until you reach the end of the year.

This approach allows you to make observations on weather patterns within the context of a 7-day period.



Variable Sliding Window Analogy

You are a software development manager who is tasked with leading a team in developing a new feature. However, your team already has other tasks, so you want to find the minimum number of consecutive days where they have at least a total of 7 hours to work on this project. After you survey your team, you find the free time available for your team to be in the table below.

Mon	Tue	Wed	Thu	Fri
2	1	4	3	1

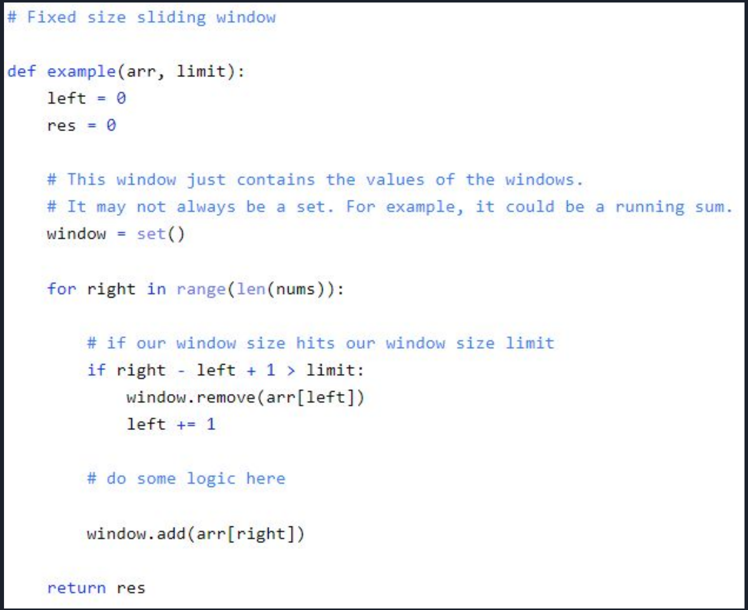
- Start the window with just Monday.
- 2 hours is not enough, so expand window to Mon-Tue.
- 3 hours is not enough, so expand window to Mon-Wed.
- 7 hours is enough! Mon-Wed is a window of 3
- See if we can make a smaller window. Check Tue-Wed
- 5 is not enough, so expand window to Tue-Thu
- 8 is enough! Tue-Thu is a window of 3 (same as before)
- See if we can make a smaller window. Check Wed-Thu
- 7 is enough! Wed-Thu is a window of 2 (new smallest)
- See if we can make a smaller window. Check Thu
- 3 is not enough, so expand to Thu-Fri
- 4 is not enough, and we have reached the end of our week

How to recognize a sliding window problem?

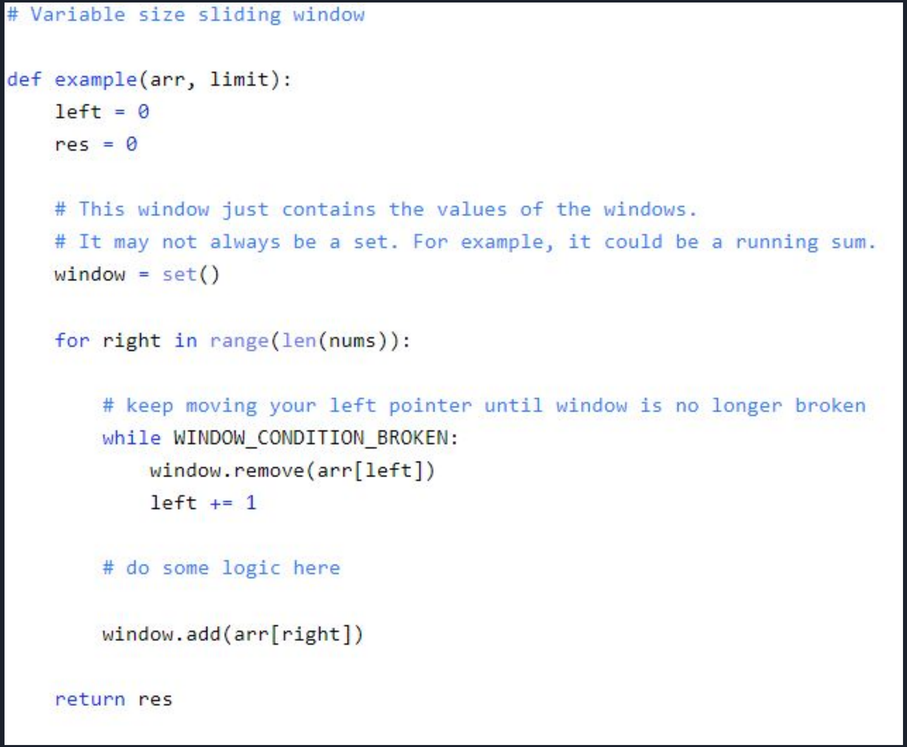
There are two criteria that you can use to identify a sliding window problem.

1. The problem will explicitly or implicitly define criteria that make a subarray "valid". Two components make up a "valid subarray".
   a. A constraint metric concerning elements within the subarray. This could be a sum, number of unique elements, frequency of elements, etc.
   b. A threshold or restriction for the subarray. This is what the constraint metric needs to be valid.
2. The problem asks for subarrays in some way.
   a. Finding the "best" subarray that meets conditions. Examples of what "best" could mean:
      i. Longest or shortest subarray
      ii. Greatest or least sum
      iii. Most or fewest frequency of an element
   b. Finding the number of valid subarrays.

Sliding window examples



```python
# Fixed size sliding window

def example(arr, limit):
    left = 0
    res = 0

    # This window just contains the values of the windows.
    # It may not always be a set. For example, it could be a running sum.
    window = set()

    for right in range(len(nums)):

        # if our window size hits our window size limit
        if right - left + 1 > limit:
            window.remove(arr[left])
            left += 1

        # do some logic here

        window.add(arr[right])

    return res
```



```python
# Variable size sliding window

def example(arr, limit):
    left = 0
    res = 0

    # This window just contains the values of the windows.
    # It may not always be a set. For example, it could be a running sum.
    window = set()

    for right in range(len(nums)):

        # keep moving your left pointer until window is no longer broken
        while WINDOW_CONDITION_BROKEN:
            window.remove(arr[left])
            left += 1

        # do some logic here

        window.add(arr[right])

    return res
```

Questions?

Let’s practice!

https://github.com/Dijkstra-LLC/dsa_live_pro/tree/main/W02D01/classwork