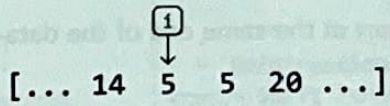
1

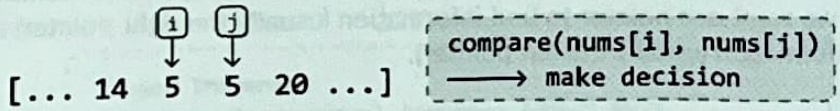
Two Pointers

Introduction to Two Pointers

As the name implies, a two-pointer pattern refers to an algorithm that utilizes two pointers. But what is a pointer? It's a variable that represents an index or position within a data structure, like an array or linked list. Many algorithms just use a single pointer to attain or keep track of a single element:



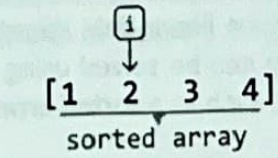
Introducing a second pointer opens a new world of possibilities. Most importantly, we can now make **comparisons**. With pointers at two different positions, we can compare the elements at those positions and make decisions based on the comparison:



In many cases, such comparisons are made using two nested for-loops, which takes $O(n^2)$ time, where n denotes the length of the data structure. In the code snippet below, `i` and `j` are two pointers used to compare every two elements of an array:

```
for i in range(n):
    for j in range(i + 1, n):
        compare(nums[i], nums[j])
```

Often, this approach does not take advantage of **predictable dynamics** that might exist in a data structure. An example of a data structure with predictable dynamics is a sorted array: when we move a pointer in a sorted array, we can predict whether the value being moved to is greater or smaller. For example, moving a pointer to the right in an ascending array guarantees we're moving to a value greater than or equal to the current one:



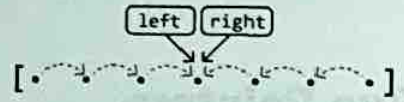

```
prediction: if nums[i] == 2, then nums[i + 1] ≥ 2
```

As you can see, data structures with predictable dynamics let us move pointers in a logical way. Taking advantage of this predictability can lead to improved time and space complexity, which we'll illustrate with real interview problems in this chapter.

Two-pointer Strategies

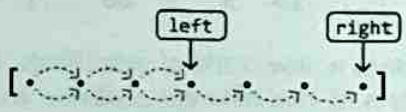
Two-pointer algorithms usually take only $O(n)$ time by eliminating the need for nested for-loops. There are three main strategies for using two pointers.

Inward traversal
This approach has pointers starting at opposite ends of the data structure and moving inward toward each other:



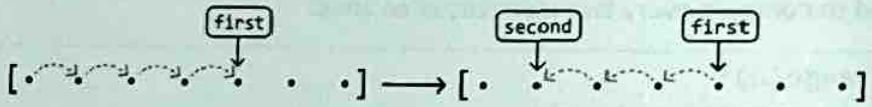
The pointers move toward the center, adjusting their positions based on comparisons, until a certain condition is met, or they meet/cross each other. This is ideal for problems where we need to compare elements from different ends of a data structure.

Unidirectional traversal
In this approach, both pointers start at the same end of the data structure (usually the beginning) and move in the same direction:



These pointers generally serve two different but supplementary purposes. A common application of this is when we want one pointer to find information (usually the right pointer) and another to keep track of information (usually the left pointer).

Staged traversal
In this approach, we traverse with one pointer, and when it lands on an element that meets a certain condition, we traverse with the second pointer:



Similar to unidirectional traversal, both pointers serve different purposes. Here, the first pointer is used to search for something, and once found, a second pointer finds additional information concerning the value at the first pointer.

We discuss all of these techniques in detail throughout the problems in this chapter.

When To Use Two Pointers?

A two-pointer algorithm usually requires a linear data structure, such as an array or linked list. Otherwise, an indication that a problem can be solved using the two-pointer algorithm, is when the input follows a predictable dynamic, such as a **sorted array**.

Predictable dynamics can take many forms. Take, for instance, a palindromic string. Its symmetrical pattern allows us to logically move two pointers toward the center. As you work through the

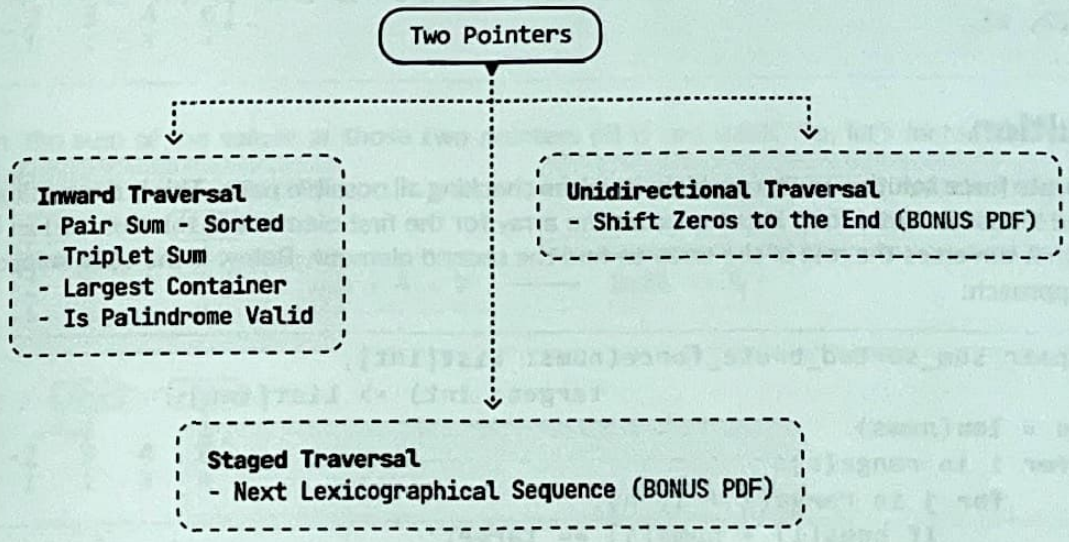
problems in this chapter, you'll learn to recognize these predictable dynamics more easily.

Another potential indicator that a problem can be solved using two pointers is if the problem asks for a **pair of values** or a result that can be generated from two values.

Real-world Example

Garbage collection algorithms: In memory compaction – which is a key part of garbage collection – the goal is to free up contiguous memory space by eliminating gaps left by deallocated (aka dead) objects. A two-pointer technique helps achieve this efficiently: a 'scan' pointer traverses the heap to identify live objects, while a 'free' pointer keeps track of the next available space to where live objects should be relocated. As the 'scan' pointer moves, it skips over dead objects and shifts live objects to the position indicated by the 'free' pointer, compacting the memory by grouping all live objects together and freeing up continuous blocks of memory.

Chapter Outline



</>

Practice these problems on our online code editor
⟶ bit.ly/run-code

To receive the bonus PDF, sign up for our Coding Interview Patterns newsletter by using the link below:

bit.ly/coding-patterns-pdf

The two-pointer pattern is very versatile and, consequently, quite broad. As such, we want to cover more specialized variants of this algorithm in separate chapters, such as *Fast and Slow Pointers* and *Sliding Windows*.

Pair Sum – Sorted

Given an array of integers sorted in ascending order and a target value, return the indexes of any pair of numbers in the array that sum to the target. The order of the indexes in the result doesn't matter. If no pair is found, return an empty array.

Example 1:

Input: `nums = [-5, -2, 3, 4, 6], target = 7`
Output: `[2, 3]`
Explanation: `nums[2] + nums[3] = 3 + 4 = 7`

Example 2:

Input: `nums = [1, 1, 1], target = 2`
Output: `[0, 1]`
Explanation: other valid outputs could be `[1, 0]`, `[0, 2]`, `[2, 0]`, `[1, 2]` or `[2, 1]`.

Intuition

The brute force solution to this problem involves checking all possible pairs. This is done using two nested loops: an outer loop that traverses the array for the first element of the pair, and an inner loop that traverses the rest of the array to find the second element. Below is the code snippet for this approach:

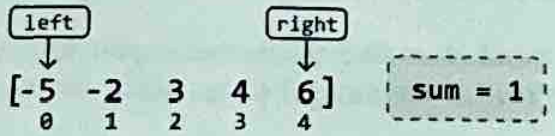
```
def pair_sum_sorted_brute_force(nums: List[int],
                                target: int) -> List[int]:
    n = len(nums)
    for i in range(n):
        for j in range(i + 1, n):
            if nums[i] + nums[j] == target:
                return [i, j]
    return []
```

This approach has a time complexity of $O(n^2)$, where n denotes the length of the array. Here, we do not take into account that the input array is sorted. Could we use this fact to come up with a more efficient solution?

A **two-pointer approach** is worth considering here because a sorted array allows us to move the pointers in a logical way. Let's see how this works in the example below:

```
[-5  -2   3   4   6],   target = 7
 0   1    2   3   4
```

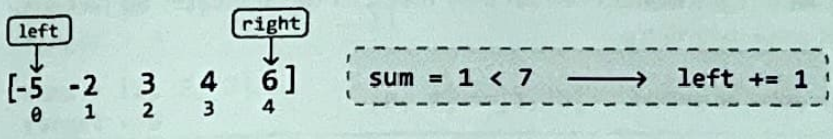
A good place to start is by looking at the smallest and largest values: the first and last elements, respectively. The sum of these two values is 1.

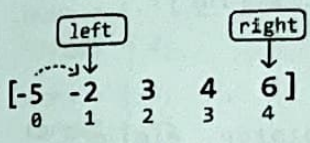


Since 1 is less than the target, we need to **move one of the pointers** to find a new pair with a larger sum.

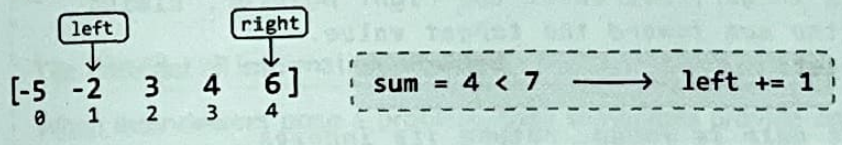
- **Left pointer:** The left pointer will always point to a value less than or equal to the value at the right pointer because the array is sorted. Incrementing it would result in a sum greater than or equal to the current sum of 1.
- **Right pointer:** Decrementing the right pointer would result in a sum that's less than or equal to 1.

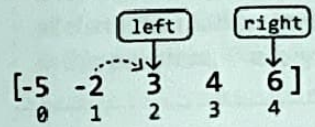
Therefore, we should increment the left pointer to find a larger sum:



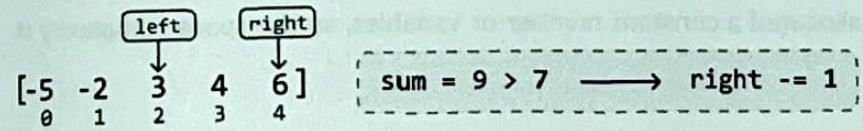


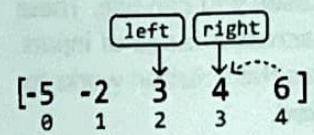
Again, the sum of the values at those two pointers (4) is too small. So, let's increment the left pointer:



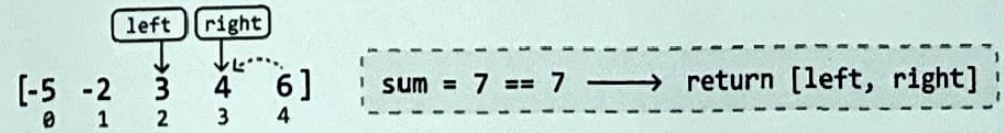


Now, the sum (9) is too large. So, we should decrement the right pointer to find a pair of values with a smaller sum:





Finally, we found two numbers that yield a sum equal to the target. Let's return their indexes:



Above, we've demonstrated a two-pointer algorithm using inward traversal. Let's summarize this logic. For any pair of values at `left` and `right`:

- If their sum is less than the target, increment `left`, aiming to increase the sum toward the target value.
- If their sum is greater than the target, decrement `right`, aiming to decrease the sum toward the target value.
- If their sum is equal to the target value, return `[left, right]`.

We can stop moving the `left` and `right` pointers when they meet, as this indicates no pair summing to the target was found.

Implementation

```
def pair_sum_sorted(nums: List[int], target: int) -> List[int]:
    left, right = 0, len(nums) - 1
    while left < right:
        sum = nums[left] + nums[right]
        # If the sum is smaller, increment the left pointer, aiming
        # to increase the sum toward the target value.
        if sum < target:
            left += 1
        # If the sum is larger, decrement the right pointer, aiming
        # to decrease the sum toward the target value.
        elif sum > target:
            right -= 1
        # If the target pair is found, return its indexes.
        else:
            return [left, right]
    return []
```

Complexity Analysis

Time complexity: The time complexity of `pair_sum_sorted` is $O(n)$ because we perform approximately n iterations using the two-pointer technique in the worst case.

Space complexity: We only allocated a constant number of variables, so the space complexity is $O(1)$.

Test Cases

In addition to the examples already discussed, here are some other test cases you can use. These extra test cases cover different contexts to ensure the code works well across a range of inputs. Testing is important because it helps identify mistakes in your code, ensures the solution works for uncommon inputs, and brings attention to cases you might have overlooked.

Input	Expected output	Description
`nums = []` `target = 0`	`[]`	Tests an empty array.
`nums = [1]` `target = 1`	`[]`	Tests an array with just one element.
`nums = [2, 3]` `target = 5`	`[0, 1]`	Tests a two-element array that contains a pair that sums to the target.
`nums = [2, 4]` `target = 5`	`[]`	Tests a two-element array that doesn't contain a pair that sums to the target.
`nums = [2, 2, 3]` `target = 5`	`[0, 2]` or `[1, 2]`	Testing an array with duplicate values.
`nums = [-1, 2, 3]` `target = 2`	`[0, 2]`	Tests if the algorithm works with a negative number in the target pair.
`nums = [-3, -2, -1]` `target = -5`	`[0, 1]`	Tests if the algorithm works with both numbers of the target pair being negative.

Interview Tip

Tip: Consider all information provided.

When interviewers pose a problem, they sometimes provide only the minimum amount of information required for you to start solving it. Consequently, it's crucial to thoroughly evaluate all that information to determine which details are essential for solving the problem efficiently. In this problem, the key to arriving at the optimal solution is recognizing that the input is sorted.

Triplet Sum

> Given an array of integers, return all triplets `[a, b, c]` such that `a + b + c = 0`. The solution must not contain duplicate triplets (e.g., `[1, 2, 3]` and `[2, 3, 1]` are considered duplicate triplets). If no such triplets are found, return an empty array.
>
> Each triplet can be arranged in any order, and the output can be returned in any order.
>
> **Example:**
>
> Input: `nums = [0, -1, 2, -3, 1]`
> Output: `[[-3, 1, 2], [-1, 0, 1]]`

Intuition

A brute force solution involves checking every possible triplet in the array to see if they sum to zero. This can be done using three nested loops, iterating through each combination of three elements.

Duplicate triplets can be avoided by sorting each triplet, which ensures identical triplets with different representations (e.g., `[1, 3, 2]` and `[3, 2, 1]`) are ordered consistently (`[1, 2, 3]`). Once sorted, we can add these triplets to a hash set. This way, if the same triplet is encountered again, the hash set will only keep one instance. Below is the code snippet for this approach:

```
def triplet_sum_brute_force(nums: List[int]) -> List[List[int]]:
    n = len(nums)
    # Use a hash set to ensure we don't add duplicate triplets.
    triplets = set()
    # Iterate through the indexes of all triplets.
    for i in range(n):
        for j in range(i + 1, n):
            for k in range(j + 1, n):
                if nums[i] + nums[j] + nums[k] == 0:
                    # Sort the triplet before including it in the
                    # hash set.
                    triplet = tuple(
                        sorted([nums[i], nums[j], nums[k]])
                    )
                    triplets.add(triplet)
    return [list(triplet) for triplet in triplets]
```

This solution is quite inefficient with a time complexity of $O(n^3)$, where n denotes the length of the input array. How can we do better?

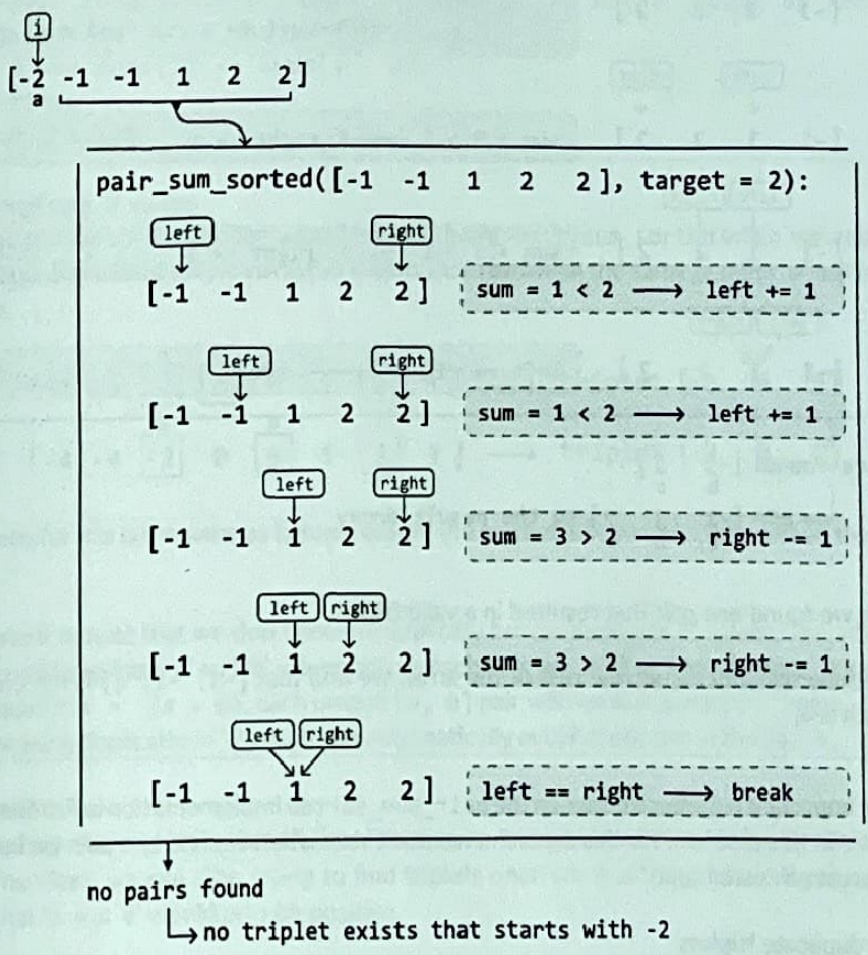
Let's see if we can find at least one triplet that sums to 0. Notice that if we fix one of the numbers in a triplet, the problem can be reduced to finding the other two. This leads to the following observation:

> For any triplet `[a, b, c]`, if we fix '`a`', we can focus on finding a pair `[b, c]` that sums to '`-a`' (`a + b + c = 0` → `b + c = -a`).

Sound familiar? That's because the problem of finding a pair of numbers that sum to a target has already been addressed by *Pair Sum - Sorted*. However, we can only use that algorithm on a sorted array. So, the first thing we should do is **sort the input**. Consider the following example:

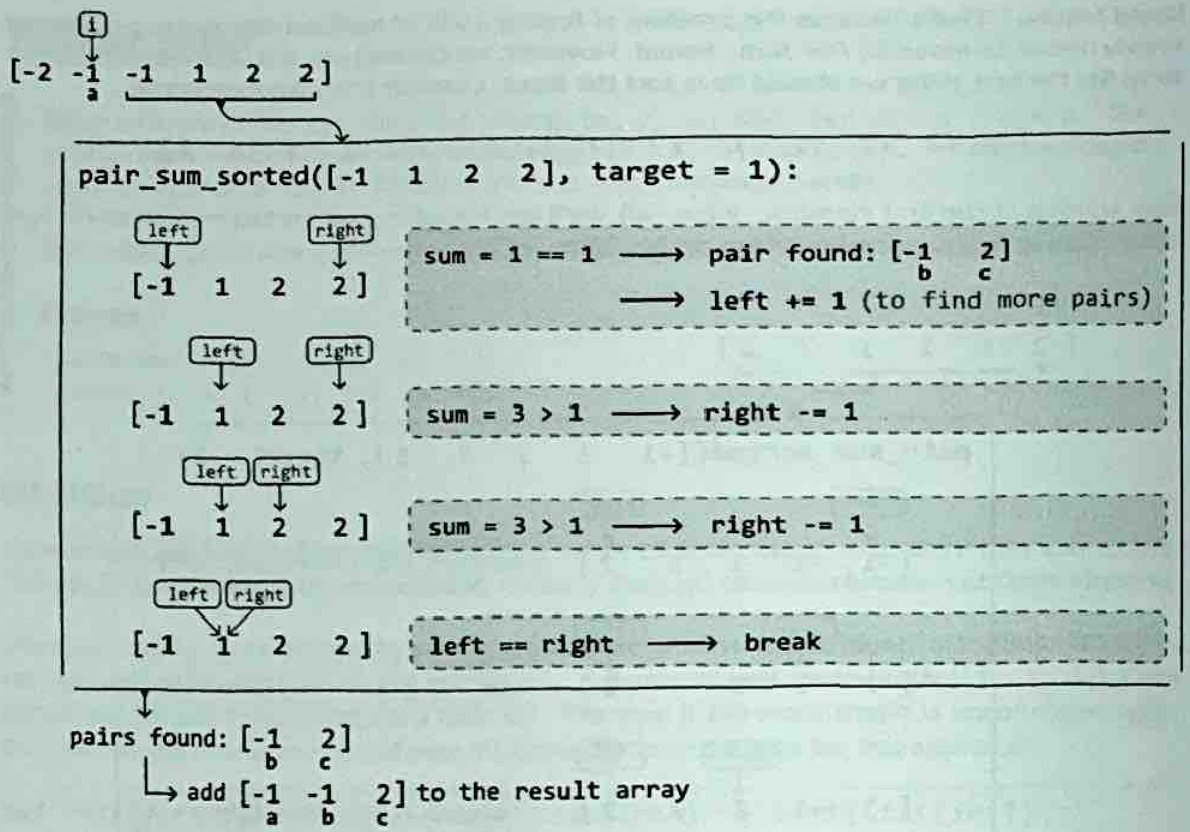
```
[-1  2 -2  1 -1  2] --sort--> [-2 -1 -1  1  2  2]
```

Now, starting at the first element, -2 (i.e., 'a'), we'll use the `pair_sum_sorted` method on the rest of the array to find a pair whose sum equals 2 (i.e., '-a'):



As you can see, when we called `pair_sum_sorted`, we did not find a pair with a sum of 2. This indicates there are no triplets starting with -2 that add up to 0.

So, let's increment our main pointer, `i`, and try again.



This time, we found one pair that resulted in a valid triplet.

If we continue this process for the rest of the array, we find that `[-1, -1, 2]` is the only triplet whose sum is 0.

There's an important difference between the `pair_sum_sorted` implementation in *Pair Sum - Sorted* and the one in this problem: for this problem, we don't stop when we find one pair, we keep going until all target pairs are found.

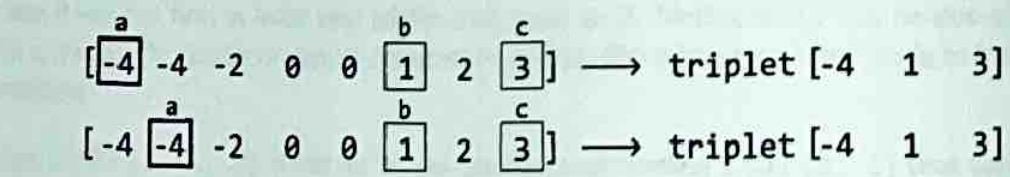
Handling duplicate triplets

Something we previously glossed over is how to avoid adding duplicate triplets. There are two cases in which this happens. Consider the example below:

```
[-4 -4 -2  0  0  1  2  3]
```

Case 1: duplicate 'a' values

The first instance where duplicates may occur is when seeking pairs for triplets that start with the same 'a' value:



Since `pair_sum_sorted` would look for pairs that sum to `'-a'` in both instances, we'd naturally end

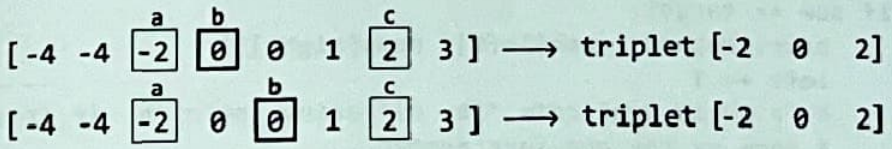
up with the same pairs and, hence, the same triplets.

To avoid picking the same 'a' value, we keep increasing `i` (where `nums[i]` represents the value 'a') until it reaches a different number from the previous one. We do this before we start looking for pairs using the `pair_sum_sorted` method. This logic works because the array is sorted, meaning equal numbers are next to each other. The code snippet for checking duplicate 'a' values looks like this:

```
# To prevent duplicate triplets, ensure 'a' is not a repeat of the
# previous element in the sorted array.
if i > 0 and nums[i] == nums[i - 1]:
    continue
... Find triplets ...
```

Case 2: duplicate 'b' values

As for the second case, consider what happens during `pair_sum_sorted` when we encounter a similar issue. For a fixed target value ('-a'), pairs that start with the same number 'b' will always be the same:



The remedy for this is the same as before: ensure the current 'b' value isn't the same as the previous value.

It's important to note that we don't need to explicitly handle duplicate 'c' values. The adjustments made to avoid duplicate 'a' and 'b' values ensure each pair `[a, b]` is unique. Since 'c' is determined by the equation `c = -(a + b)`, each unique `[a, b]` pair will result in a unique 'c' value. Therefore, by just avoiding duplicates in 'a' and 'b', we automatically avoid duplicates in the `[a, b, c]` triplets.

Optimization

An interesting observation is that triplets that sum to `0` cannot be formed using positive numbers alone. Therefore, we can stop trying to find triplets once we reach a positive 'a' value since this implies that 'b' and 'c' would also be positive.

Implementation

From the above intuition, we know we need to slightly modify the `pair_sum_sorted` function to avoid duplicate triplets. We also need to pass in a `start` value to indicate the beginning of the subarray on which we want to perform the pair-sum algorithm. Otherwise, the two-pointer logic remains nearly identical to that of *Pair Sum - Sorted*.

```
def triplet_sum(nums: List[int]) -> List[List[int]]:
    triplets = []
    nums.sort()
    for i in range(len(nums)):
        # Optimization: triplets consisting of only positive numbers
        # will never sum to 0.
```

```
        if nums[i] > 0:
            break
        # To avoid duplicate triplets, skip 'a' if it's the same as
        # the previous number.
        if i > 0 and nums[i] == nums[i - 1]:
            continue
        # Find all pairs that sum to a target of '-a' (-nums[i]).
        pairs = pair_sum_sorted_all_pairs(nums, i + 1, -nums[i])
        for pair in pairs:
            triplets.append([nums[i]] + pair)
    return triplets

def pair_sum_sorted_all_pairs(nums: List[int],
                              start: int, target: int) -> List[int]:
    pairs = []
    left, right = start, len(nums) - 1
    while left < right:
        sum = nums[left] + nums[right]
        if sum == target:
            pairs.append([nums[left], nums[right]])
            left += 1
            # To avoid duplicate '[b, c]' pairs, skip 'b' if it's the
            # same as the previous number.
            while left < right and nums[left] == nums[left - 1]:
                left += 1
        elif sum < target:
            left += 1
        else:
            right -= 1
    return pairs
```

Complexity Analysis

Time complexity: The time complexity of `triplet_sum` is $O(n^2)$. Here's why:

- We first sort the array, which takes $O(n \log(n))$ time.
- Then, for each of the n elements in the array, we call `pair_sum_sorted_all_pairs` at most once, which runs in $O(n)$ time.

Therefore, the overall time complexity is $O(n \log(n)) + O(n^2) = O(n^2)$.

Space complexity: The space complexity is $O(n)$ due to the space taken up by Python's sorting algorithm. It's important to note that this complexity does not include the output array `triplets` because we're only concerned with the additional space used by the algorithm, not the space needed for the output itself.

If the interviewer asks what the space complexity would be if we included the output array, it would be $O(n^2)$. This is because the `pair_sum_sorted_all_pairs` function, in the worst case, can add approximately n pairs to the output. Since this function is called approximately n times, the overall space complexity would be $O(n^2)$.

Test Cases

In addition to the examples already covered in this explanation, below are some others to consider when testing your code.

Input	Expected output	Description
`nums = []`	`[]`	Tests an empty array.
`nums = [0]`	`[]`	Tests a single-element array.
`nums = [1, -1]`	`[]`	Tests a two-element array.
`nums = [0, 0, 0]`	`[0, 0, 0]`	Tests an array where all three of its values are the same.
`nums = [1, 0, 1]`	`[]`	Tests an array with no triplets that sum to 0.
`nums = [0, 0, 1, -1, 1, -1]`	`[-1, 0, 1]`	Tests an array with duplicate triplets.

Is Palindrome Valid

A palindrome is a sequence of characters that reads the same forward and backward.

Given a string, determine if it's a palindrome after removing all non-alphanumeric characters. A character is alphanumeric if it's either a letter or a number.

Example 1:

Input: `s = "a dog! a panic in a pagoda."`
Output: `True`

Example 2:

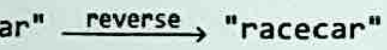
Input: `s = "abc123"`
Output: `False`

Constraints:

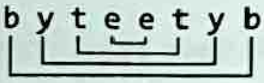
The string may include a combination of lowercase English letters, numbers, spaces, and punctuations.

Intuition

Identifying palindromes
A string is a palindrome if it remains identical when read from left to right or right to left. In other words, if we reverse the string, it should still read the same, disregarding spaces and punctuation:



An important observation is that if a string is a palindrome, the first character would be the same as the last, the second character would be the same as the second-to-last, etc:



A palindrome of odd length is different because it has a middle character. In this case, the middle character can be ignored since it has no "mirror" character elsewhere in the string.

Palindromes provides an ideal scenario for using **two pointers** (`left` and `right`). By initially setting the pointers at the beginning and end of the string, we can compare the characters at these positions. Ignoring non-alphanumeric characters for the moment, the logic can be summarized as follows:

- If the alphanumeric characters at `left` and `right` are the same, move both pointers inward to process the next pair of characters.
- If not, the string is not a palindrome: return false.

If we successfully compare all character pairs without returning false, the string is a palindrome, and we should return true.

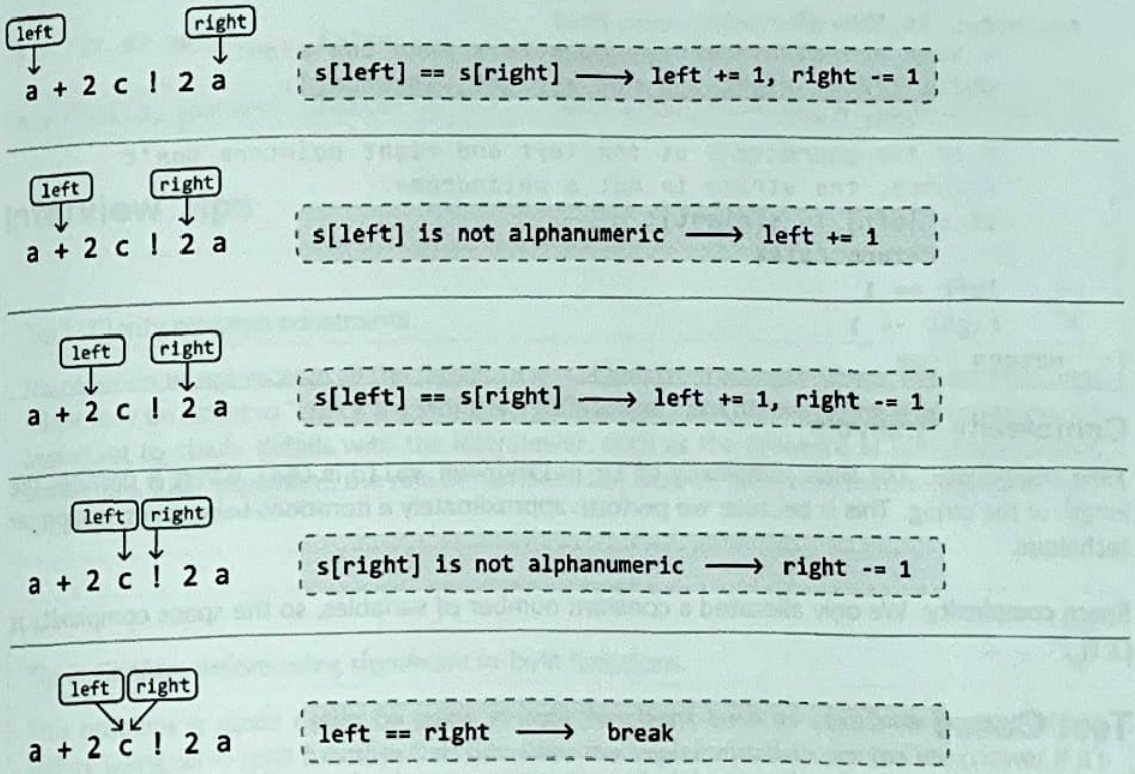
Processing non-alphanumeric characters

Now, let's explore how to find palindromes that include non-alphanumeric characters.

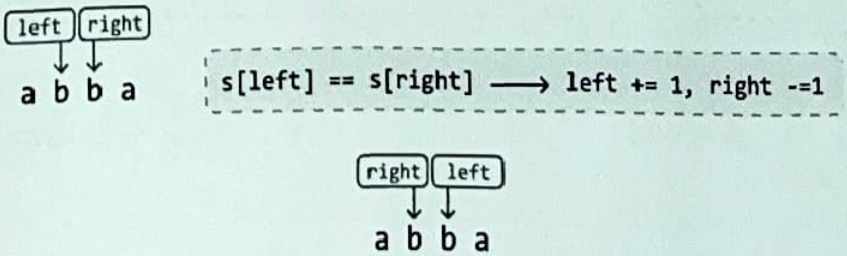
Since non-alphanumeric characters don't affect whether a string is a palindrome, we should skip them. This can be achieved with the following approach, which ensures the left and right pointers are adjusted to focus only on alphanumeric characters:

- Increment `left` until the character it points to is alphanumeric.
- Decrement `right` until the character it points to is alphanumeric.

With this in mind, let's check if the string below is a palindrome using all the information we know so far:



As shown above, when the left and right pointers meet, it signals our exit condition. When these pointers meet, we've reached the middle character of the palindrome, at which point we can exit the loop since the middle character doesn't need to be evaluated. However, we need to keep in mind that exiting when left equals right won't always be sufficient as an exit condition. For example, if the number of alphanumeric characters is even, the pointers won't meet. This can be observed below:



Therefore, we need to ensure we exit the loop when `left` equals `right`, or when `left` passes `right`. In other words, the algorithm continues while `left` is less than `right`:

```
while left < right:
```

Implementation

In Python, we can use the inbuilt `isalnum` method to check if a character is alphanumeric.

```
def is_palindrome_valid(s: str) -> bool:
    left, right = 0, len(s) - 1
    while left < right:
        # Skip non-alphanumeric characters from the left.
        while left < right and not s[left].isalnum():
            left += 1
        # Skip non-alphanumeric characters from the right.
        while left < right and not s[right].isalnum():
            right -= 1
        # If the characters at the left and right pointers don't
        # match, the string is not a palindrome.
        if s[left] != s[right]:
            return False
        left += 1
        right -= 1
    return True
```

Complexity Analysis

Time complexity: The time complexity of `is_palindrome_valid` is $O(n)$, where n denotes the length of the string. This is because we perform approximately n iterations using the two-pointer technique.

Space complexity: We only allocated a constant number of variables, so the space complexity is $O(1)$.

Test Cases

In addition to the examples discussed, below are more examples to consider when testing your code.

Input	Expected output	Description
`s = ""`	`True`	Tests an empty string.
`s = "a"`	`True`	Tests a single-character string.
`s = "aa"`	`True`	Tests a palindrome with two characters.
`s = "ab"`	`False`	Tests a non-palindrome with two characters.
`s = "!, (?)"`	`True`	Tests a string with no alphanumeric characters.
`s = "12.02.2021"`	`True`	Tests a palindrome with punctuation and numbers.
`s = "21.02.2021"`	`False`	Tests a non-palindrome with punctuation and numbers.
`s = "hello, world!"`	`False`	Tests a non-palindrome with punctuation.

Interview Tips

Tip 1: Clarify problem constraints.

It's common to not receive all the details of a problem from an interviewer. For example, you might only be asked to "check if a string is a palindrome." But before diving into a solution, it's important to clarify details with the interviewer, such as the presence of non-alphanumeric characters, their treatment, the role of numbers, the case sensitivity of letters, and other relevant details.

Tip 2: Confirm before using significant in-built functions.

This problem is made easier by using in-built functions such as `.isalnum` (or equivalent). Before using an in-built function that simplifies the implementation, ask the interviewer if it's okay to use it, or if they would prefer you implement it yourself.

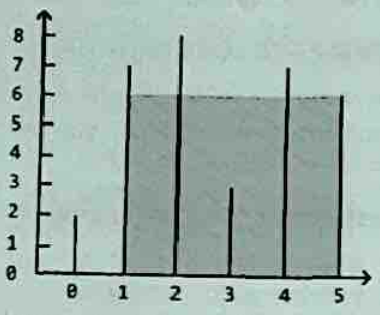
The interviewer will most likely allow the use of an in-built function, or ask you to implement it as an exercise for later in the interview. If you use an in-built function, make sure you understand its time and space complexity.

Remember that interviewers are looking for team players, and this shows them you're considerate of their preferences and can adapt your approach based on the requirements.

Largest Container

You are given an array of numbers, each representing the height of a vertical line on a graph. A container can be formed with any pair of these lines, along with the x-axis of the graph. Return the amount of water which the largest container can hold.

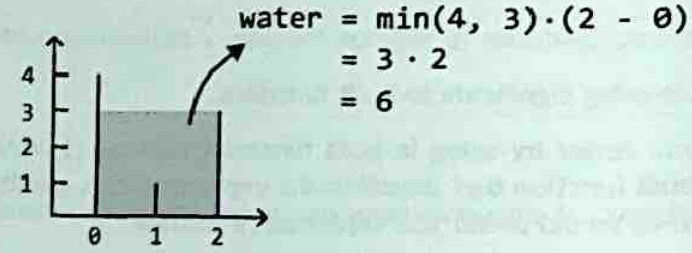
Example:



Input: `heights = [2, 7, 8, 3, 7, 6]`
Output: 24

Intuition

If we have two vertical lines, `heights[i]` and `heights[j]`, the amount of water that can be contained between these two lines is `min(heights[i], heights[j]) * (j - i)`, where `j - i` represents the width of the container. We take the minimum height because filling water above this height would result in overflow.



In other words, the area of the container depends on two things:

- The width of the rectangle.
- The height of the rectangle, as dictated by the shorter of the two lines.

The brute force approach to this problem involves checking all pairs of lines, and returning the largest area found between each pair:

```
def largest_container_brute_force(heights: List[int]) -> int:
    n = len(heights)
    max_water = 0
    # Find the maximum amount of water stored between all pairs of
    # lines.
    for i in range(n):
        for j in range(i + 1, n):
```

```
            water = min(heights[i], heights[j]) * (j - i)
            max_water = max(max_water, water)
    return max_water
```

Searching through all possible pairs of values takes $O(n^2)$ time, where n denotes the length of the array. Let's look for a more efficient solution.

We would like both the height and width to be as large as possible to have the largest container.

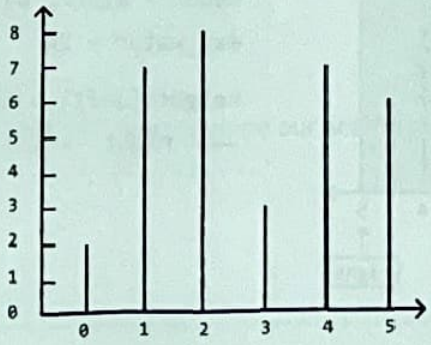
It's not immediately obvious how to find the container with the largest height, as the heights of the lines in the array don't follow a clear pattern. However, we do know the container with the maximum width: the one starting at index 0 and ending at index `n - 1`.

So, we could start by maximizing the width by setting a pointer at each end of the array. Then, we can gradually reduce the width by moving these two pointers inward, hoping to find a container with a larger height that potentially yields a larger area. This suggests we can use the two-pointer pattern to solve the problem.

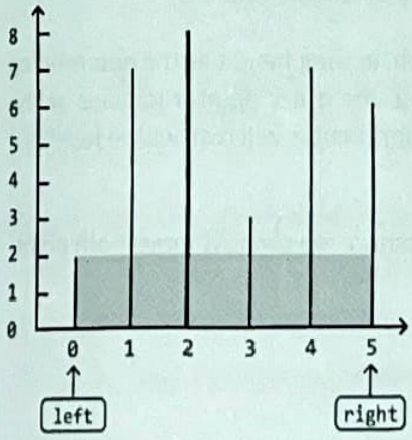
> Moving a pointer inward means shifting either the left pointer to the right, or the right pointer to the left, effectively narrowing the gap between them.

Consider the following example:

`heights = [2 7 8 3 7 6]`:



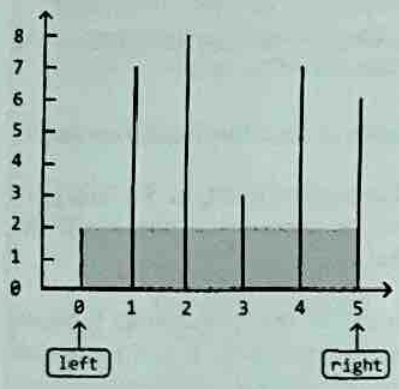
The widest container can store an area of water equal to 10. Since this is the largest container we've found so far, let's set `max_water` to 10.




```
water = min(2, 6)·(5 - 0) = 10
max_water = 10
```

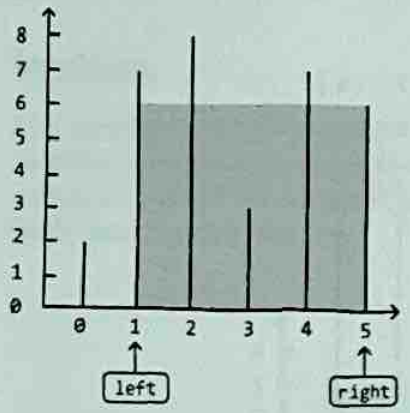
How should we proceed? Moving either pointer inward yields a container with a shorter width.

This leaves height as the determining factor. In this case, the left line is shorter than the right line, which means that the left line limits the water's height. Therefore, to find a larger container, let's move the left pointer inward:




```
heights[left] < heights[right]
⟶ left += 1
```

The current container can hold 24 units of water, the largest amount so far. So, let's update `max_water` to 24. Here, the right line is shorter, limiting the water's height. To find a larger container, move the right pointer inward:



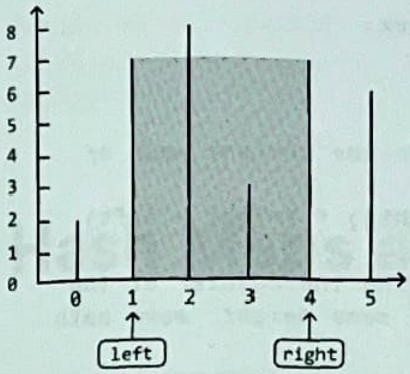

```
water = min(7, 6)·(5 - 1) = 24
max_water = 24

heights[left] > heights[right]
⟶ right -= 1
```

After this, we encounter a situation where **the height of the left and right lines are equal.** In this situation, which pointer should we move inward? Well, regardless of which one, the next container is guaranteed to store less water than the current one. Let's try to understand why.

Moving either pointer inward yields a container of shorter width, leaving height as the determining factor. However, regardless of which pointer we move inward, the other pointer remains at the same line. So, even if a pointer is moved to a taller line, the other pointer will restrict the height of the water, as we take the minimum of the two lines.

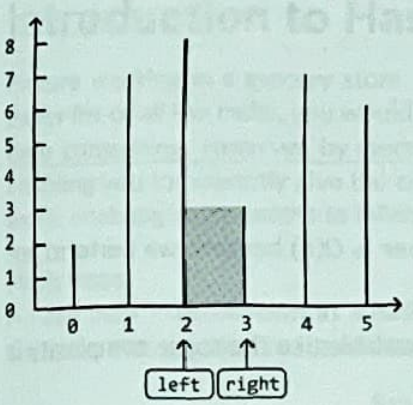
Therefore, since we can't increase height by moving just one pointer, we can just move both pointers inward:




```
water = min(7, 7)·(4 - 1) = 21
max_water = 24

heights[left] == heights[right]
⟶ left += 1 and right -= 1
```

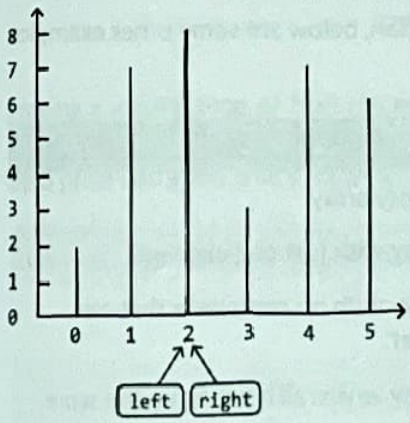
Now, the right line is limiting the height of the water. So, we move the right pointer inward:




```
water = min(8, 3)·(3 - 2) = 3
max_water = 24

heights[left] > heights[right]
⟶ right -= 1
```

Finally, the left and right pointers meet. We can conclude our search here and return `max_water`:




```
left == right ⟶ break
```

Based on the decisions taken in the example, we can summarize the logic:

1. If the left line is smaller, move the left pointer inward.
2. If the right line is smaller, move the right pointer inward.
3. If both lines have the same height, move both pointers inward.

Implementation

```
def largest_container(heights: List[int]) -> int:
    max_water = 0
    left, right = 0, len(heights) - 1
    while (left < right):
        # Calculate the water contained between the current pair of
        # lines.
        water = min(heights[left], heights[right]) * (right - left)
        max_water = max(max_water, water)
        # Move the pointers inward, always moving the pointer at the
        # shorter line. If both lines have the same height, move both
        # pointers inward.
        if (heights[left] < heights[right]):
            left += 1
        elif (heights[left] > heights[right]):
            right -= 1
        else:
            left += 1
            right -= 1
    return max_water
```

Complexity Analysis

Time complexity: The time complexity of `largest_container` is $O(n)$ because we perform approximately n iterations using the two-pointer technique.

Space complexity: We only allocated a constant number of variables, so the space complexity is $O(1)$.

Test Cases

In addition to the examples discussed throughout this explanation, below are some other examples to consider when testing your code.

Input	Expected output	Description
`heights = []`	`0`	Tests an empty array.
`heights = [1]`	`0`	Tests an array with just one element.
`heights = [0, 1, 0]`	`0`	Tests an array with no containers that can contain water.
`heights = [3, 3, 3, 3]`	`9`	Tests an array where all heights are the same.
`heights = [1, 2, 3]`	`2`	Tests an array with strictly increasing heights.
`heights = [3, 2, 1]`	`2`	Tests an array with strictly decreasing heights.