

01_Sequences

Arrays and Strings.

<https://www.techinterviewhandbook.org/algorithms/array/>

Techniques

<https://www.techinterviewhandbook.org/coding-interview-techniques/>

- [Array Cheatsheet](#)
- [01_Arrays](#)
 - [Brute Force](#)
 - [Two Pointers: Inward Traversal](#)
 - [Two Pointers: Unidirectional Traversal](#)
 - [Two Pointers: Stage Traversal](#)
 - [Sliding Window: Fixed](#)
 - [Sliding Window: Dynamic](#)
 - [Traversing Array From The Right](#)
 - [Sorting The Array](#)
 - [Index as a Hash Key](#)
- [String Cheatsheet](#)
- [02_Strings](#)
 - [Counting Characters in a String](#)
 - [Anagram](#)
 - [Palindrome](#)

Array Cheatsheet

Arrays

Arrays hold values of the same type at contiguous memory locations. In an array, we're usually concerned about two things - the position/index of an element and the element itself.

Advantages

- Store multiple elements of the same type with one single variable name
- Accessing elements is fast ($O(1)$) as long as you have the index, as opposed to linked lists where you have to traverse from the head ($O(n)$).

Disadvantages

- Addition and removal of elements in the middle of an array is slow ($O(n)$) because the remaining elements need to be shifted to accommodate the new/missing element.
 - An exception to this is if the position to be inserted/removed is at the end of the array ($O(1)$).
- It cannot alter its size after initialization.
 - If an insertion causes the total number of elements to exceed the size, a new array has to be allocated and the existing elements have to be copied over.
 - The act of creating a new array and transferring elements over takes $O(n)$ time.

Common terms

- Subarray - A range of contiguous values within an array.
 - Example: given an array `[2, 3, 6, 1, 5, 4]`, `[3, 6, 1]` is a subarray while `[3, 1, 5]` is not a subarray.
- Subsequence - A sequence that can be derived from the given sequence by deleting some or no elements without changing the order of the remaining elements.
 - Example: given an array `[2, 3, 6, 1, 5, 4]`, `[3, 1, 5]` is a subsequence but `[3, 5, 1]` is not a subsequence.

Things to look out for during interviews

- Clarify if there are duplicate values in the array. Would the presence of duplicate values affect the answer? Does it make the question simpler or harder?
- When using an index to iterate through array elements, be careful not to go out of bounds.

- Be mindful about slicing or concatenating arrays in your code. Typically, slicing and concatenating arrays would take $O(n)$ time. Use start and end indices to demarcate a subarray/range where possible.

Corner Cases

- Empty sequence
- Sequence with 1 or 2 elements
- Sequence with repeated elements
- Duplicated values in the sequence

Time Complexity

Operation	Big-O	Note
Access	$O(1)$	
Search	$O(n)$	
Search (sorted array)	$O(\log(n))$	
Insert	$O(n)$	Insertion would require shifting all the subsequent elements to the right by one and that takes $O(n)$
Insert (at the end)	$O(1)$	Special case of insertion where no other element needs to be shifted
Remove	$O(n)$	Removal would require shifting all the subsequent elements to the left by one and that takes $O(n)$
Remove (at the end)	$O(1)$	Special case of removal where no other element needs to be

Resource

[Array in Data Structure: What is, Arrays Operations \[Examples\]](#)

01_Arrays

Brute Force

The brute force approach tries every possible combination to check for a solution, without leveraging any special properties or optimizations of the data (such as sorted order). Typically involves:

- Nested loops
 - Outer Loop traverses the array for the first element of the pair
 - Inner Loop traverses the rest of the array to find second element
- Recursion

Drawbacks:

- **Time Complexity:** Typically $O(n^2)$ or slower, impractical for large datasets.
- **Redundancy:** Many computations are repeated unnecessarily

Need to know: `Nested For Loops`, `Range`, `Len`



Target: 35. Starting brute force search...

```
def brute_force_two_sum(nums, target):  
    # Iterate through each element in the list  
    for i in range(len(nums)):  
        # For each element, check every other element that comes after it  
        for j in range(i + 1, len(nums)):  
            # Check if the current pair sums to the target  
            if nums[i] + nums[j] == target:  
                return (i, j) # Return the indices as a tuple  
    # If no pair is found that adds up to the target, return None  
    return None
```

```
# Example usage:
nums = [2, 7, 11, 15]
target = 9

result = brute_force_two_sum(nums, target)
if result:
    print("Pair found at indices:", result)
else:
    print("No pair found that adds up to the target.")
```

Great question — **no**, nested `for` loops in Python **don't always have to use** `range()`. It totally depends on **what you're looping over**.

Let's break this down:

When You Use `range()` in Nested Loops

You use `range()` when you're looping over **indexes**, usually for:

- Arrays
- Matrices
- Iterations with index math

```
for i in range(len(matrix)):
    for j in range(len(matrix[0])):
        print(matrix[i][j])
```

When You Don't Need `range()`

You can also loop directly over values — nested or not — when index access isn't needed.

Example: Nested Lists

```
for row in matrix:
    for val in row:
```

```
print(val)
```

Example: Nested Dictionaries

```
for category, items in asset_dict.items():  
    for item in items:  
        print(f"{category}: {item}")
```

Example: In Tech Art (Blender)

```
for obj in bpy.data.objects:  
    for modifier in obj.modifiers:  
        print(f"{obj.name} has modifier {modifier.name}")
```

☐☐ When to Use `range()` vs Direct Loop

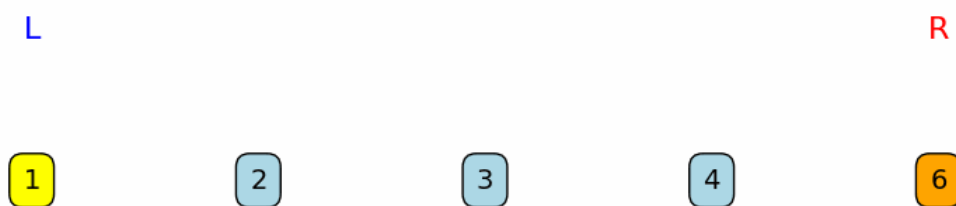
Use Case	Use <code>range()</code>	Direct Loop (<code>for item in list</code>)
You need the index ☐	☐ Yes	☐ No
You only care about the value	☐ No	☐ Yes
You're modifying a list in-place	☐ Often	☐ Risky

So nested loops **can** use `range()`, but they **only should when** you need an index or are doing some kind of math-based traversal (e.g., comparing elements, jumping steps, etc).

Want me to show a nested loop example with `range()` from a LeetCode matrix problem or a nested asset traversal from a DCC pipeline?

Two Pointers: Inward Traversal

A pointer is a variable that represents an index or position within a data structure, such as an Array or Linked List. With Two pointers we can make comparisons, with a pointer at two different positions, and infer a decision based on that.



Target: 6. Start: L at index 0 (1), R at index 4 (6)

When to use:

- Data Structure: Linear such as Array, Linked List
- Sorted Array or Palindrome
- Result asks for Pair of Values
- One Result is decided by a Pair of Values

Real-World Example:

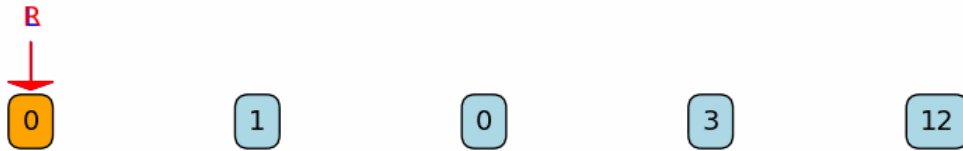
- Garbage Collection Algorithm

```
def pair_sum_sorted(nums: List[int], target: int) -> List[int]:  
    left, right = 0, len(nums)-1  
    while left < right:  
        sums = nums[left] + nums[right]  
        # if the sum is smaller, increment the left pointer, aiming  
        # to increase the sum towards the target value  
        if sums < target:
```



```
    left += 1
# if the sum is larger, decrement the right pointer, aiming
# to decrease the sum towards the target value
elif sums > target:
    right -= 1
# If target pair is found return its indices
else:
    return [left, right]
return []
```

Two Pointers: Unidirectional Traversal



nums[0] = 0; no swap

```
def shift_zeros_to_the_end(nums: List[int]) -> None:
    # The 'left' pointer is used to position non-zero elements.
    left = 0
    # Iterate through the array using a 'right' pointer to locate non-zero
    # elements.
    for right in range(len(nums)):
        if nums[right] != 0:
            nums[left], nums[right] = nums[right], nums[left]
            # Increment 'left' since it now points to a position already occupied
            # by a non-zero element.
            left += 1
```

Two Pointers: Stage Traversal



Start partitioning by parity: even numbers (L) and odd numbers (R)

Problem: Partition Array by Parity

Description:

Given an integer array `nums`, rearrange the array **in-place** such that all even numbers appear before all odd numbers. The order of the elements within the even or odd group does not matter.

Return the array after rearrangement.

You must solve the problem without using extra space (i.e. in $O(1)$ extra space) and in one pass if possible.

Example 1:

Input: `nums = [3, 8, 5, 12, 7, 4, 6]` Output: `[8, 12, 4, 6, 3, 5, 7]` Explanation: The even numbers [8, 12, 4, 6] appear before the odd numbers [3, 5, 7]. Note that the order within each group does not matter.

Example 2:

Input: `nums = [1, 3, 5, 7]` Output: `[1, 3, 5, 7]` Explanation: Since there are no even numbers, the array remains unchanged.

Constraints:

- $1 \leq \text{nums.length} \leq 10^5$
- $0 \leq \text{nums}[i] \leq 10^5$

1. Initialization:

The script starts with two pointers:

- `left` at the beginning (index 0)
- `right` at the end (last index)

2. Stage Traversal:

In each loop iteration:

- It prints the current array and the positions (and values) of the two pointers.
- If the number at the left pointer is even, that element is already in the correct half, so the left pointer is moved right.
- If the number at the right pointer is odd, that element is in the correct half, so the right pointer is moved left.
- If neither of those conditions holds (meaning the left number is odd and the right number is even), the two values are swapped. This moves the even number toward the left and the odd number toward the right.

3. Termination:

The loop ends when the left pointer is no longer less than the right pointer. At that point, the array is partitioned with evens on the left and odds on the right.

```
def partition_by_parity(nums):
    left = 0                # Initialize the left pointer at the beginning.
    right = len(nums) - 1   # Initialize the right pointer at the end.
    print("Initial array:", nums)

    while left < right:
        print("\nCurrent array:", nums)
        print(f"Left pointer at index {left} with value {nums[left]}")
        print(f"Right pointer at index {right} with value {nums[right]}")

        # If the left element is even, it's already in the correct partition.
        if nums[left] % 2 == 0:
            print(f"{nums[left]} is even, so move the left pointer right.")
            left += 1

        # If the right element is odd, it's already in the correct partition.
        elif nums[right] % 2 == 1:
            print(f"{nums[right]} is odd, so move the right pointer left.")
            right -= 1

        # Otherwise, the left element is odd and the right element is even.
        # In that case, swap them.
```

else:

```
    print(f"Swapping {nums[left]} (odd) and {nums[right]} (even).")
```

```
    nums[left], nums[right] = nums[right], nums[left]
```

```
    left += 1
```

```
    right -= 1
```

```
print("\nFinal partitioned array:", nums)
```

Example usage:

```
nums = [3, 8, 5, 12, 7, 4, 6]
```

```
partition_by_parity(nums)
```

Sliding Window: Fixed

A subset of the Two Pointer Method, but uses left and right pointers to define the bounds of a "window" in iterable data structures like arrays. The window defines the subcomponent, like subarray or substring, and it slides across the data structure in one direction, searching for a subcomponent that meets a certain requirement.



Window indices: [0, 2] with values: [1, 2, 3]

When to use:

- Data Structure: Linear such as Array, Linked List
- Find a Subcomponent of a length

Brute Force:

- Finding all possible subcomponents for an answer using a Nested Loop
 - Outer Loop traverses the array for the first element of the pair
 - Inner Loop traverses the rest of the array to find second element
- Time Complexity is $O(n^2)$ where n is length of the loop (Two Loops)

```
def sliding_window_fixed(nums, window_size):  
    n = len(nums)  
    # Slide the window from the start of the array until the end.  
    for i in range(n - window_size + 1):  
        window = nums[i:i + window_size]  
        print(f"Window indices: [{i}, {i + window_size - 1}] -> Values: {window}")  
  
# Example usage:  
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
window_size = 3  
sliding_window_fixed(nums, window_size)
```

Real-World Example:

Buffering in Video Streaming

Sliding Window: Dynamic

Problem: Given an array of positive integers and a target sum, find a contiguous subarray whose sum is as close as possible to, but does not exceed, the target.

Example: nums = [1, 3, 2, 5, 1, 1, 2, 1, 4], target = 8



Window: [0, -1] Sum: 0 Target: 8
Start dynamic sliding window

```
def find_all_subarrays(nums, target):  
    """  
    Finds and prints all contiguous subarrays whose sum equals the target.  
    Assumes that all numbers in 'nums' are positive.  
    """  
  
    print("Problem: Given an array of positive integers and a target sum,")  
    print("find all contiguous subarrays whose sum equals the target.\n")  
    print(f"Example: nums = {nums}, target = {target}\n")  
    print("-" * 70)  
  
    left = 0  
    current_sum = 0  
    n = len(nums)  
  
    # Iterate with 'right' pointer to expand the window.  
    for right in range(n):  
        current_sum += nums[right]  
        print(f"Expanding window: added nums[{right}] = {nums[right]}")  
        print(f"Current window (indices [{left} to {right}]): {nums[left:right+1]}")  
        print(f"Current sum: {current_sum}\n")
```



```
# If the sum exceeds the target, contract the window from the left.
```

```
while current_sum > target and left <= right:
```

```
    print(f" Sum {current_sum} exceeds target {target}.")
```

```
    print(f" Removing nums[{left}] = {nums[left]} from window.")
```

```
    current_sum -= nums[left]
```

```
    left += 1
```

```
    print(f" After contraction, window (indices [{left} to {right}]): {nums[left:right+1]}")
```

```
    print(f" Current sum: {current_sum}\n")
```

```
# If the current sum equals the target, print the subarray.
```

```
if current_sum == target:
```

```
    print(f"Found subarray with sum {target}: indices [{left}, {right}] -> {nums[left:right+1]}\n")
```

```
print("-" * 70)
```

```
print("Done searching for subarrays.")
```

```
# Example usage:
```

```
nums = [1, 3, 2, 5, 1, 1, 2, 1, 4]
```

```
target = 8
```

```
find_all_subarrays(nums, target)
```

Traversing Array From The Right



Target: 4. Start right-to-left traversal.

Find Last Occurrence of Target in Array

Description:

Given an integer array `nums` and an integer `target`, return the index of the last occurrence of `target` in `nums`. If the target is not found, return -1.

You must solve this problem with an efficient $O(n)$ time solution by traversing the array from right to left.

Example 1:

Input: `nums = [1, 3, 5, 3, 2]`, `target = 3`

Output: 3

Explanation: The target 3 appears at indices 1 and 3, but its last occurrence is at index 3.

Example 2:

Input: `nums = [1, 2, 3, 4]`, `target = 5`

Output: -1

Explanation: The target 5 is not present in the array.

Constraints:

- $1 \leq \text{nums.length} \leq 10^5$
- $-10^9 \leq \text{nums}[i] \leq 10^9$

- $-10^9 \leq \text{target} \leq 10^9$

```
from typing import List
```

```
def find_last_occurrence(nums: List[int], target: int) -> int:
```

```
    """
```

```
    Returns the index of the last occurrence of target in nums.
```

```
    If the target is not found, returns -1.
```

```
    """
```

```
    # Traverse from rightmost index to left
```

```
    for i in range(len(nums) - 1, -1, -1):
```

```
        if nums[i] == target:
```

```
            return i
```

```
    return -1
```

```
# Example usage:
```

```
nums = [1, 3, 5, 3, 2]
```

```
target = 3
```

```
result = find_last_occurrence(nums, target)
```

```
print("Last occurrence of", target, "is at index:", result)
```

- **Right-to-Left Traversal:**

The function iterates over the array starting from the last index down to 0. This guarantees that the first time the target is found (when traversing from right to left) is its last occurrence in the array.

- **Time Complexity:**

The traversal takes $O(n)$ time in the worst-case scenario (when the target is at the beginning of the array or not present), which meets the requirements.

Sorting The Array

When you receive an unsorted array and decide to sort it before applying the two-pointer technique, the overall time complexity is dominated by the sorting step.

- **Sorting:** This typically takes $O(n \log n)$ time.
- **Two Pointer Traversal:** Once sorted, scanning the array with two pointers takes $O(n)$ time.

Thus, the overall time complexity becomes:

$$O(n \log n) + O(n) = O(n \log n)$$

If the array is already sorted, then you only pay the $O(n)$ cost for the two-pointer traversal, without the additional $O(n \log n)$ sorting cost.

Index as a Hash Key

String Cheatsheet

A string is a sequence of characters. Many tips that apply to Arrays also apply to Strings.

Time complexity

A String is an array of characters, so the time complexities of basic string operations will closely resemble that of array operations.

Operation	Big-O
Access	$O(1)$
Search	$O(n)$
Insert	$O(n)$
Remove	$O(n)$

Operations involving another String

Here we assume the other string is of length m .

Operation	Big-O
Find substring	$O(n.m)$
Concatenating strings	$O(n + m)$
Slice	$O(m)$
Split (by token)	$O(n + m)$
Strip (remove leading and trailing whitespaces)	$O(n)$

Things to look out for during interviews

- Ask about input character set and case sensitivity.

Corner cases

- Empty string
- String with 1 or 2 characters

- String with repeated characters
- Strings with only distinct character

02_Strings

Counting Characters in a String

h e l l o w o r l d

Initial state (no characters processed yet)

Current counts: {}

Need to know: Dictionary, for loop

```
def count_characters(string):
    counts = {}
    for char in string:
        if char in counts:
            counts[char] = counts[char] + 1
        else:
            counts[char] = 1
    return counts

# Example usage:
input_string = "hello world"
result = count_characters(input_string)

print("Character counts:")
for char, count in result.items():
    print(f"{char}: {count}")
```

If you need to keep a counter of characters, a common mistake is to say that the space complexity required for the counter is $O(n)$. The space required for a counter of a string of latin characters is

$O(1)$ not $O(n)$. This is because the upper bound is the range of characters, which is usually a fixed constant of 26. The input set is just lowercase Latin characters.

Anagram

An anagram is word switch or word play. It is the result of rearranging the letters of a word or phrase to produce a new word or phrase, while using all the original letters only once.

In interviews, usually we are only bothered with words without spaces in them.

Let's break down the three common approaches in very simple terms:

1. Sorting Both Strings

- **How it works:**

Rearrange (sort) the letters of both strings alphabetically. If the sorted versions are exactly the same, then the strings are anagrams.

- **Time Cost:**

Sorting takes about $O(n \log n)$ time, where n is the number of characters.

- **Space Cost:**

It typically uses extra space proportional to the depth of recursion (about $O(\log n)$).

- **Simple Analogy:**

Imagine you have two mixed-up decks of cards. If you sort them by suit and number, and they end up in the same order, then both decks originally had the same cards.

2. Prime Multiplication Method

- **How it works:**

Assign each letter a unique prime number (like a secret code). For each string, multiply the primes corresponding to its letters. Thanks to the unique nature of prime factors, two strings will have the same product if and only if they have the exact same letters.

- **Time Cost:**

You only go through the string once, so it takes $O(n)$ time.

- **Space Cost:**

It uses constant extra space, $O(1)$, since the mapping of letters to primes is fixed.

- **Simple Analogy:**

Think of it like each letter is a unique ingredient with a special "number." Mix them all together (multiply), and if two recipes (strings) yield the same "flavor number," they contain exactly the same ingredients.

3. Frequency Counting

- **How it works:**

Count how many times each letter appears in each string using a hash (dictionary). Then, compare these counts. If they match for every letter, the strings are anagrams.

- **Time Cost:**

This also takes $O(n)$ time, as you just make one pass through each string.

- **Space Cost:**

It uses constant extra space, $O(1)$, because the number of letters (for example, 26 for the English alphabet) is fixed.

- **Simple Analogy:**

Imagine you have two baskets of fruit. If you count the number of apples, oranges, etc., in both baskets and the counts match exactly, then both baskets have the same mix of fruits.

Each method has its tradeoffs between speed and space:

- **Sorting** is straightforward but slightly slower due to the sorting process.
- **Prime multiplication** is fast and space-efficient, but it can run into issues with very long strings because the multiplication might produce huge numbers.
- **Frequency counting** is both fast and efficient, and it's often the simplest and most reliable method.

Palindrome

Palindrome

Worse approaches

- **Reversing the String:**

You can reverse the string (using slicing like `s[::-1]` in Python) and then compare it to the original. This is very concise but uses $O(n)$ additional space.

- **Recursion:**

A recursive approach can check the first and last characters and then recurse on the substring that excludes them. However, this method uses extra space for the call stack and is generally less efficient.

Overall, the two-pointer approach is typically considered the best way to solve the palindrome problem due to its efficiency in both time and space.



Starting palindrome check.

A palindrome is a word, phrase, number, or other sequence of characters which reads the same backward as forward, such as `madam` or `racecar`.

Here are ways to determine if a string is a palindrome:

- Reverse the string and it should be equal to itself.
- Have two pointers at the start and end of the string. Move the pointers inward till they meet. At every point in time, the characters at both pointers should match.

The order of characters within the string matters, so hash tables are usually not helpful.

When a question is about counting the number of palindromes, a common trick is to have two pointers that move outward, away from the middle.

- Note that palindromes can be even or odd length. For each middle pivot position, you need to check it twice - once that includes the character and once without the character.
- For substrings, you can terminate early once there is no match

```
def is_palindrome(s: str) -> bool:
    left, right = 0, len(s) - 1
    while left < right:
        if s[left] != s[right]:
            return False
        left += 1
        right -= 1
    return True

# Example usage:
input_string = "racecar"
if is_palindrome(input_string):
    print(f'{input_string} is a palindrome.')
else:
    print(f'{input_string} is not a palindrome.')
```