01_Arrays

Brute Force

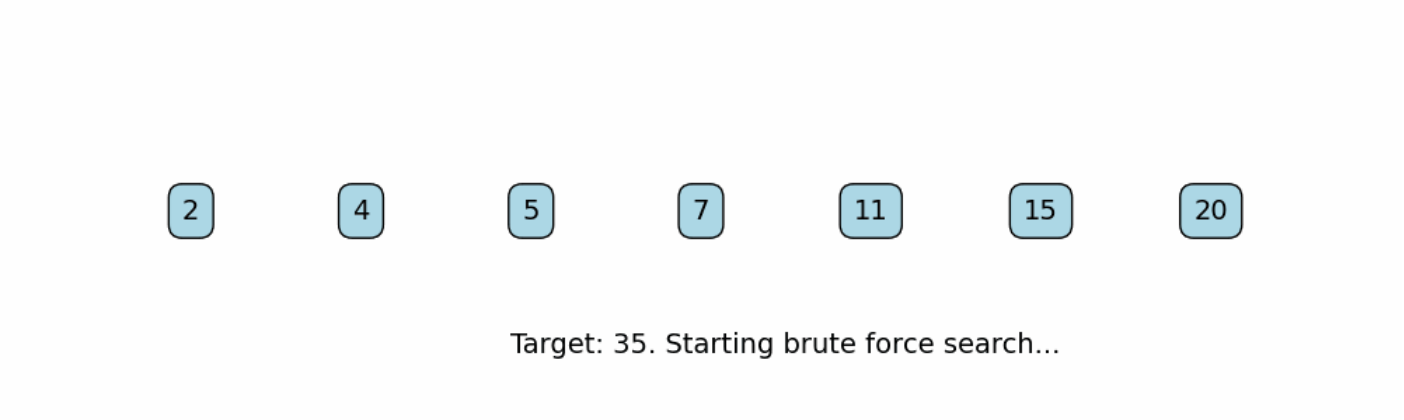
The brute force approach tries every possible combination to check for a solution, without leveraging any special properties or optimizations of the data (such as sorted order). Typically involves:

- 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`




```
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

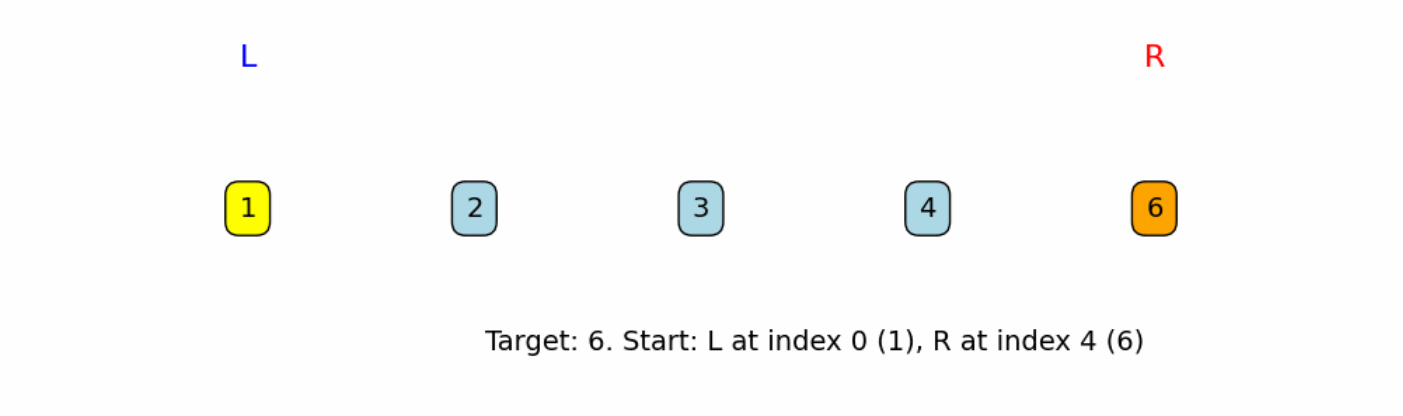
Use Case	Use `range()`	Direct Loop (`for item in list`)
You need the index □	□ Yes	□ No
You only care about the value	□ No	□ Yes
You're modifying a list in-place	□ Often	□ Risky

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 and 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.



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

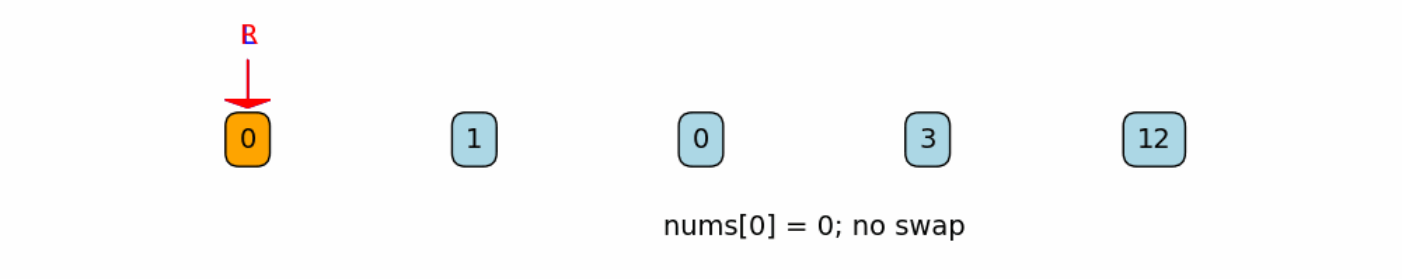
Real-World Example:

- 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
```

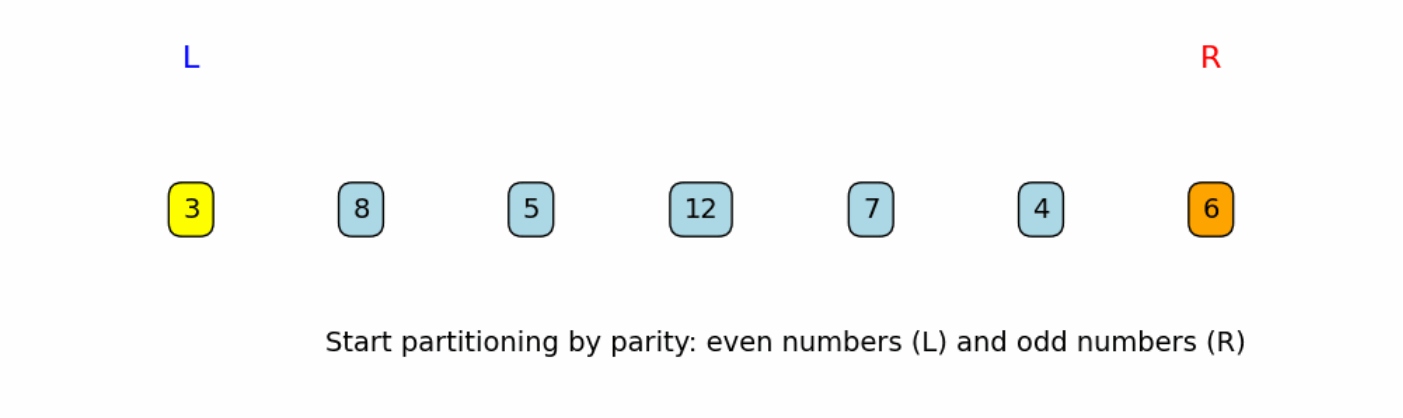
```
    # if the sum is larger, decrement the right pointer, aiming
    # to decrease the sum towards the target value
    elif sums > target:
      right -=1
    # If target pair is found return its indices
    else:
      return [left,right]
  return []
```

Two Pointers: Unidirectional Traversal




```
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



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 <= nums.length <= 10^5

- 0 <= nums[i] <= 10^5

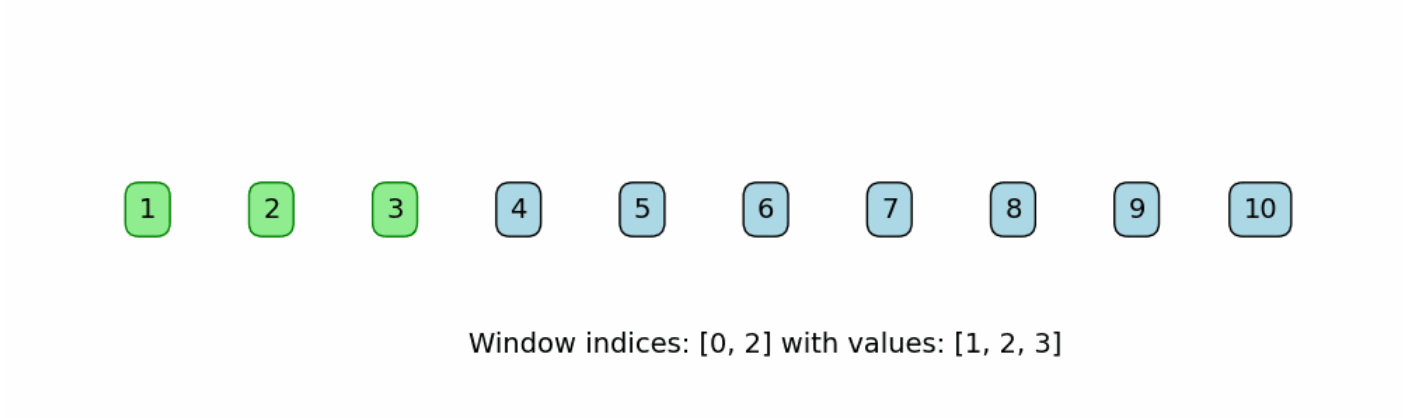
1. **Initialization:**
   The script starts with two pointers:
   - `left` at the beginning (index 0)
   - `right` at the end (last index)
2. **Stage Traversal:**
   In each loop iteration:
   - It prints the current array and the positions (and values) of the two pointers.
   - If the number at the 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.



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)

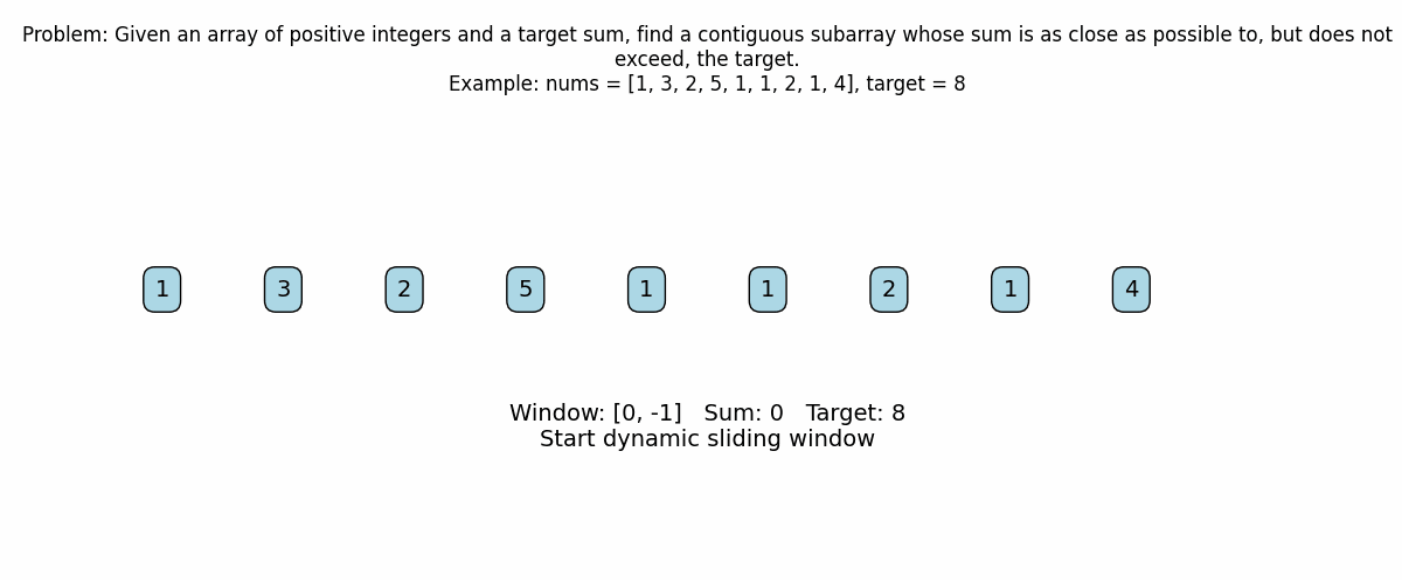
```
def sliding_window_fixed(nums, window_size):
    n = len(nums)
    # Slide the window from the start of the array until the end.
    for i in range(n - window_size + 1):
        window = nums[i:i + window_size]
        print(f"Window indices: [{i}, {i + window_size - 1}] -> Values: {window}")

# 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

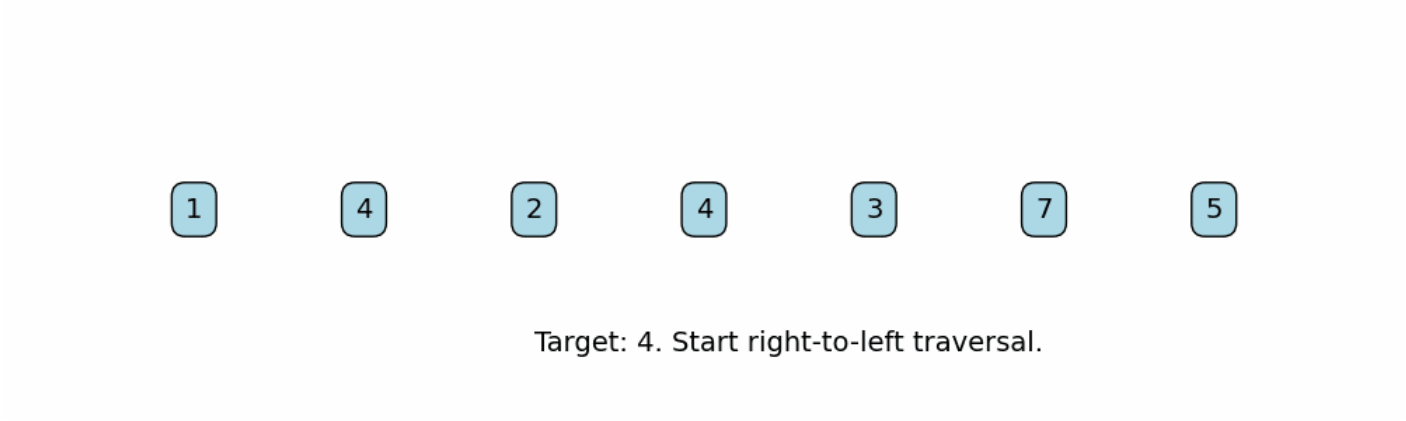
Sliding Window: Dynamic




```
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



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 <= nums.length <= 10^5
- -10^9 <= nums[i] <= 10^9
- -10^9 <= target <= 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