Two Pointers

1. 3Sum

Pattern: Two Pointers

Problem Statement

Given an integer array `nums`, return all the triplets `[nums[i], nums[j], nums[k]]` such that `i != j`, `i != k`, `j != k`, and `nums[i] + nums[j] + nums[k] == 0`.

Notice that the solution set must not contain duplicate triplets.

Sample Input & Output

```
Input: nums = [-1,0,1,2,-1,-4]
Output: [[-1,-1,2],[-1,0,1]]
Explanation: Two valid triplets sum to zero.
Duplicates like [-1,0,1] appearing twice are removed.
```

```
Input: nums = [0,1,1]
Output: []
Explanation: No triplet sums to zero.
```

```
Input: nums = [0,0,0]
Output: [[0,0,0]]
Explanation: Only one triplet exists, and it sums to zero.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        # STEP 1: Initialize structures
        #   - Sort to enable two pointers and skip duplicates
        #   - Use list to collect valid triplets
        nums.sort()
        result = []
        n = len(nums)

        # STEP 2: Main loop / recursion
        #   - Fix first number (i), then use two pointers for rest
        #   - Skip duplicates for i to avoid repeated triplets
        for i in range(n - 2):
            if i > 0 and nums[i] == nums[i - 1]:
                continue  # Skip duplicate i values

            left = i + 1
            right = n - 1

            # Two-pointer search for complement = -nums[i]
            while left < right:
                total = nums[i] + nums[left] + nums[right]

                if total == 0:
                    result.append([nums[i], nums[left], nums[right]])

                    # STEP 3: Update state / bookkeeping
                    #   - Skip duplicates for left and right
                    while left < right and nums[left] == nums[left + 1]:
                        left += 1
                    while left < right and nums[right] == nums[right - 1]:
                        right -= 1

                    left += 1
                    right -= 1
                elif total < 0:
                    left += 1  # Need larger sum
                else:
```

```python
                    right -= 1  # Need smaller sum

        # STEP 4: Return result
        #   - Already filtered duplicates; empty if none found
        return result

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.threeSum([-1, 0, 1, 2, -1, -4]) == [[-1, -1, 2], [-1, 0, 1]]

    #  Test 2: Edge case - no solution
    assert sol.threeSum([0, 1, 1]) == []

    #  Test 3: Tricky/negative - all zeros
    assert sol.threeSum([0, 0, 0]) == [[0, 0, 0]]

    print(" All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `threeSum([-1, 0, 1, 2, -1, -4])` step by step.

Initial State:
- Input: `nums = [-1, 0, 1, 2, -1, -4]`
- After sorting: `nums = [-4, -1, -1, 0, 1, 2]`
- `result = []`, `n = 6`

Loop 1: `i = 0` → `nums[i] = -4`

- Not a duplicate (first element).

- Set `left = 1`, `right = 5` → values: `-1` and `2`
- **While `left < right`:**
  - `total = -4 + (-1) + 2 = -3` → too small → `left += 1` → `left = 2`
  - `total = -4 + (-1) + 2 = -3` → still too small → `left += 1` → `left = 3`
  - `total = -4 + 0 + 2 = -2` → too small → `left = 4`
  - `total = -4 + 1 + 2 = -1` → too small → `left = 5` → now `left == right` → exit

→ No triplet found with `-4`.

Loop 2: `i = 1` **→** `nums[i] = -1`

- Not duplicate (`i=1`, `nums[0] = -4  -1`)
- `left = 2`, `right = 5` → values: `-1` and `2`
- **While `left < right`:**
  - `total = -1 + (-1) + 2 = 0` → Found triplet `[-1, -1, 2]`
    - Append to `result` → `result = [[-1, -1, 2]]`
    - Skip duplicates:
      - `nums[2] == nums[3]?` → `-1 != 0` → no skip
      - `nums[5] == nums[4]?` → `2 != 1` → no skip
    - Move pointers: `left = 3`, `right = 4`
  - Now `left=3`, `right=4` → values: `0` and `1`
  - `total = -1 + 0 + 1 = 0` → Found `[-1, 0, 1]`
    - Append → `result = [[-1, -1, 2], [-1, 0, 1]]`
    - No duplicates next → move: `left=4`, `right=3` → exit loop

Loop 3: `i = 2` → `nums[i] = -1`

- **Duplicate!** `i > 0` and `nums[2] == nums[1]` → `continue`

Loop 4: `i = 3` → `nums[i] = 0`

- `left = 4`, `right = 5` → `0 + 1 + 2 = 3 > 0` → `right -= 1` → `right = 4` → exit

Loops end. Return `result = [[-1, -1, 2], [-1, 0, 1]]`.

Complexity Analysis

- **Time Complexity**: `O(n²)`

  Sorting takes `O(n log n)`. The outer loop runs `O(n)` times. For each `i`, the two-pointer scan is `O(n)`. Total: `O(n log n + n²) = O(n²)`.

- **Space Complexity**: `O(1)` (or `O(n)` if counting output)

  We only use a few pointers and the result list. Sorting may use `O(log n)` stack space (Timsort), but no extra DS scales with input beyond output.

2. 3Sum Closest

Pattern: Two Pointers

Problem Statement

Given an integer array `nums` of length `n` and an integer `target`, find three integers in `nums` such that the sum is closest to `target`. Return the sum of the three integers.
You may assume that each input would have exactly one solution.

Sample Input & Output

```
Input: nums = [-1,2,1,-4], target = 1
Output: 2
Explanation: The sum that is closest to the target is 2
             (-1 + 2 + 1 = 2).
```

```
Input: nums = [0,0,0], target = 1
Output: 0
Explanation: Only possible sum is 0, which is closest to 1.
```

```
Input: nums = [1,1,1,0], target = -100
Output: 2
Explanation: Smallest possible sum is 1+1+0=2, which is still
             the closest to -100.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def threeSumClosest(
        self, nums: List[int], target: int
    ) -> int:
        # STEP 1: Initialize structures
        #   - Sort array to enable two pointers
        #   - Track closest sum and min diff from target
        nums.sort()
        n = len(nums)
        closest_sum = float('inf')
        min_diff = float('inf')

        # STEP 2: Main loop / recursion
        #   - Fix first number (i), then use two pointers
        #   - Invariant: for each i, we find best j,k > i
        for i in range(n - 2):
            left = i + 1
```

```
            right = n - 1

            # Two-pointer scan for best pair
            while left < right:
                curr_sum = nums[i] + nums[left] + nums[right]
                curr_diff = abs(curr_sum - target)

                # STEP 3: Update state / bookkeeping
                #   - Update if we found a closer sum
                #   - Why here? We must check every valid triplet
                if curr_diff < min_diff:
                    min_diff = curr_diff
                    closest_sum = curr_sum

                # Move pointers based on sum vs target
                if curr_sum < target:
                    left += 1
                elif curr_sum > target:
                    right -= 1
                else:
                    # Exact match - can't get closer
                    return curr_sum

        # STEP 4: Return result
        #   - Guaranteed to have found at least one triplet
        return closest_sum

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.threeSumClosest([-1,2,1,-4], 1) == 2

    #  Test 2: Edge case - all same
    assert sol.threeSumClosest([0,0,0], 1) == 0

    #  Test 3: Tricky/negative - target far from all sums
    assert sol.threeSumClosest([1,1,1,0], -100) == 2

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 1**: `nums = [-1, 2, 1, -4]`, `target = 1`.

Initial Setup: - Input: `nums = [-1, 2, 1, -4]`, `target = 1` - After sorting: `nums = [-4, -1, 1, 2]` - `n = 4` - `closest_sum = inf`, `min_diff = inf`

Step 1: `i = 0` → `nums[i] = -4`
- `left = 1`, `right = 3`
- Enter `while left < right` ($1 < 3$ → True)

Step 1.1:
- `curr_sum = -4 + (-1) + 2 = -3`
- `curr_diff = abs(-3 - 1) = 4`
- Since `4 < inf` → update:
- `min_diff = 4`
- `closest_sum = -3`
- Since `-3 < 1` → move `left` to `2`

State: `closest_sum = -3`, `min_diff = 4`

Step 1.2: `left = 2`, `right = 3`
- `curr_sum = -4 + 1 + 2 = -1`
- `curr_diff = abs(-1 - 1) = 2`
- `2 < 4` → update:
- `min_diff = 2`
- `closest_sum = -1`
- `-1 < 1` → move `left` to `3`

State: `closest_sum = -1`, `min_diff = 2`

Step 1.3: `left = 3`, `right = 3` → loop ends (`left < right` is False)

Step 2: `i = 1` → `nums[i] = -1`
- `left = 2`, `right = 3`
- Enter loop ($2 < 3 \to$ True)

Step 2.1:
- `curr_sum = -1 + 1 + 2 = 2`
- `curr_diff = abs(2 - 1) = 1`
- `1 < 2` → update:
- `min_diff = 1`
- `closest_sum = 2`
- `2 > 1` → move `right` to 2

State: `closest_sum = 2`, `min_diff = 1`

Step 2.2: `left = 2`, `right = 2` → loop ends

Step 3: `i = 2` → loop stops (`range(n-2)` = `range(2)` → i=0,1 only)

Final Return: `closest_sum = 2`

Complexity Analysis

- **Time Complexity**: `O(n²)`

  Sorting takes `O(n log n)`. The outer loop runs `O(n)` times, and the inner two-pointer scan is `O(n)` per outer iteration → total `O(n²)`. Dominates sorting.

- **Space Complexity**: `O(1)`

  Only a few extra variables (`closest_sum`, `min_diff`, pointers). Sorting is in-place (Python's Timsort uses `O(n)` worst-case, but we consider auxiliary space → `O(1)` for algorithm logic).

3. Container With Most Water

Pattern: Two Pointers

Problem Statement

You are given an integer array `height` of length `n`. There are `n` vertical lines drawn such that the two endpoints of the `i`th line are `(i, 0)` and `(i, height[i])`.

Find two lines that together with the x-axis form a container, such that the container contains the most water.

Return the maximum amount of water a container can store.

Note: You may not slant the container.

Sample Input & Output

```
Input: height = [1,8,6,2,5,4,8,3,7]
Output: 49
Explanation: The max area is between index 1 (height 8) and
index 8 (height 7): width = 7, height = min(8,7)=7 → area = 7*7=49.
```

```
Input: height = [1,1]
Output: 1
Explanation: Only two lines; width=1, height=1 → area=1.
```

```
Input: height = [4,3,2,1,4]
Output: 16
Explanation: Leftmost (4) and rightmost (4) → width=4,
height=4 → area=16 (better than inner pairs).
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def maxArea(self, height: List[int]) -> int:
        # STEP 1: Initialize structures
        #   - Use two pointers at ends to maximize width
        #   - Track max_area seen so far
        left = 0
        right = len(height) - 1
        max_area = 0

        # STEP 2: Main loop / recursion
        #   - While pointers haven't crossed
        #   - Compute current area using min height * width
        while left < right:
            width = right - left
            h = min(height[left], height[right])
            current_area = width * h
            max_area = max(max_area, current_area)

            # STEP 3: Update state / bookkeeping
            #   - Move pointer with smaller height inward
            #   - Why? Larger height might yield bigger area
            if height[left] < height[right]:
                left += 1
            else:
                right -= 1

        # STEP 4: Return result
        #   - max_area is updated throughout; safe for all cases
        return max_area

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #   Test 1: Normal case
    assert sol.maxArea([1,8,6,2,5,4,8,3,7]) == 49, "Normal case failed"

    #   Test 2: Edge case
```

```python
assert sol.maxArea([1,1]) == 1, "Edge case (two elements) failed"

#   Test 3: Tricky/negative
assert sol.maxArea([4,3,2,1,4]) == 16, "Tricky symmetric case failed"

print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `maxArea([1,8,6,2,5,4,8,3,7])` step by step.

Initial state:
- `height = [1,8,6,2,5,4,8,3,7]`
- `left = 0`, `right = 8`, `max_area = 0`

Step 1:
- Line: `while left < right` → `0 < 8` → enter loop
- `width = 8 - 0 = 8`
- `h = min(height[0], height[8]) = min(1, 7) = 1`
- `current_area = 8 * 1 = 8`
- `max_area = max(0, 8) = 8`
- Since `height[0] (1) < height[8] (7)`, move `left` → `left = 1`
- **State**: `left=1`, `right=8`, `max_area=8`

Step 2:
- `1 < 8` → continue
- `width = 8 - 1 = 7`
- `h = min(8, 7) = 7`
- `current_area = 7 * 7 = 49`
- `max_area = max(8, 49) = 49`
- `height[1] (8) > height[8] (7)` → move `right` → `right = 7`
- **State**: `left=1`, `right=7`, `max_area=49`

Step 3:
- `1 < 7` → continue
- `width = 6`
- `h = min(8, 3) = 3`
- `current_area = 6 * 3 = 18`
- `max_area` remains 49
- `8 > 3` → move `right` → `right = 6`
- **State**: `left=1`, `right=6`, `max_area=49`

Step 4:
- `1 < 6` → continue
- `width = 5`
- `h = min(8, 8) = 8`
- `current_area = 5 * 8 = 40`
- `max_area` still 49
- Heights equal → move either; code moves `right` (since `else` branch) → `right = 5`
- **State**: `left=1`, `right=5`, `max_area=49`

Step 5:
- `1 < 5` → continue
- `width = 4`
- `h = min(8, 4) = 4`
- `current_area = 16` → no change
- Move `right` → `right = 4`

Continue similarly… all subsequent areas are 40.

Eventually, `left` and `right` meet → loop ends.

Final return: `49`

Key insight: By always moving the shorter pointer, we never miss a better area — because keeping the shorter one limits height, and reducing width further can't help unless we find a taller line.

Complexity Analysis

- **Time Complexity**: `O(n)`

    We traverse the array once with two pointers moving toward each other — exactly `n-1` iterations.

- **Space Complexity**: `O(1)`

    Only a few integer variables (`left`, `right`, `max_area`, etc.) are used — no extra space proportional to input size.

4. Valid Palindrome

Pattern: Two Pointers

Problem Statement

> A phrase is a **palindrome** if, after converting all uppercase letters into lowercase letters and removing all non-alphanumeric characters, it reads the same forward and backward. Alphanumeric characters include letters and numbers.
>
> Given a string `s`, return `true` if it is a **palindrome**, or `false` otherwise.

Sample Input & Output

```
Input: s = "A man, a plan, a canal: Panama"
Output: true
Explanation: After cleaning: "amanaplanacanalpanama" - reads
same forwards/backwards.
```

```
Input: s = "race a car"
Output: false
Explanation: Cleaned string is "raceacar", which is not a palindrome.
```

```
Input: s = " "
Output: true
Explanation: Empty string (after cleaning) is considered a palindrome.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def isPalindrome(self, s: str) -> bool:
        # STEP 1: Initialize two pointers at start and end
        #   - Left pointer starts at beginning
        #   - Right pointer starts at end
        left, right = 0, len(s) - 1

        # STEP 2: Move pointers toward center while valid
        #   - Skip non-alphanumeric chars from both ends
        #   - Compare lowercase versions when both are valid
        while left < right:
            # Skip non-alphanumeric from left
            while left < right and not s[left].isalnum():
                left += 1
            # Skip non-alphanumeric from right
            while left < right and not s[right].isalnum():
                right -= 1

            # Compare characters (case-insensitive)
            if s[left].lower() != s[right].lower():
                return False

            # Move both pointers inward
            left += 1
            right -= 1

        # STEP 3: If loop completes, it's a palindrome
        #   - All valid char pairs matched
        return True
```

```
# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.isPalindrome("A man, a plan, a canal: Panama") == True

    #  Test 2: Edge case - empty or whitespace
    assert sol.isPalindrome(" ") == True

    #  Test 3: Tricky/negative - non-palindrome with mixed case/symbols
    assert sol.isPalindrome("race a car") == False

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace **`s = "A man, a plan, a canal: Panama"`** step by step:

1. **Initialize pointers**:
   - `left = 0` (points to `'A'`)
   - `right = 29` (points to `'a'` — last char)
2. **First outer loop (`left=0`, `right=29`)**:
   - Both `'A'` and `'a'` are alphanumeric → no skipping.
   - Compare `'A'.lower()` (`'a'`) vs `'a'.lower()` (`'a'`) → match.
   - Update: `left = 1`, `right = 28`.
3. **Now `left=1` (`' '`), `right=28` (`'m'`)**:
   - `' '` is not alphanumeric → `left` increments to 2 (`'m'`).
   - `'m'` and `'m'` → match.

- Update: `left = 3`, `right = 27`.

4. **Continue this process**:

   - Skip commas, spaces, colons as needed.

   - Always compare only alphanumeric chars in lowercase.

   - Pointers move inward symmetrically.

5. **Eventually**:

   - Pointers cross (`left >= right`) → loop ends.

   - Return `True`.

Key Insight:
We never create a new cleaned string — we **simulate** it by skipping invalid chars on the fly. This saves space and aligns with the **Two Pointers** pattern: two indices converging while maintaining a palindrome invariant.

Complexity Analysis

- **Time Complexity**: `O(n)`

  We scan each character at most once (each pointer moves inward monotonically). `isalnum()` and `lower()` are O(1) per char.

- **Space Complexity**: `O(1)`

  Only two integer pointers used — no extra storage proportional to input size. In-place comparison.

5. Move Zeroes

Pattern: Two Pointers

Problem Statement

Given an integer array `nums`, move all `0`'s to the end of it while maintaining the relative order of the non-zero elements.
You must do this **in-place** without making a copy of the array.
Minimize the total number of operations.

Sample Input & Output

```
Input: [0,1,0,3,12]
Output: [1,3,12,0,0]
Explanation: All zeros are moved to the end; non-zeros retain order.
```

```
Input: [0]
Output: [0]
Explanation: Single zero remains in place.
```

```
Input: [1,2,3]
Output: [1,2,3]
Explanation: No zeros to move - array unchanged.
```

LeetCode Editorial Solution + Inline Tests

```
from typing import List

class Solution:
    def moveZeroes(self, nums: List[int]) -> None:
        # STEP 1: Initialize structures
        #   - `left` tracks the position to place the next non-zero
        #   - Starts at 0: first valid spot for non-zero element
        left = 0

        # STEP 2: Main loop / recursion
```

```
        #   - Iterate through every element with `right`
        #   - Invariant: all elements before `left` are non-zero
        #   - When we find a non-zero, swap it to `left`
        for right in range(len(nums)):
            if nums[right] != 0:
                # STEP 3: Update state / bookkeeping
                #   - Swap non-zero to front; increment `left`
                #   - Ensures stable order and minimal writes
                nums[left], nums[right] = nums[right], nums[left]
                left += 1

        # STEP 4: Return result
        #   - Function modifies `nums` in-place; no return needed
        #   - Edge cases (empty, all zeros, no zeros) handled
        #     naturally by loop bounds and condition

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    nums1 = [0, 1, 0, 3, 12]
    sol.moveZeroes(nums1)
    print(f"Test 1:{nums1} == [1, 3, 12, 0, 0] → {nums1 == [1, 3, 12, 0, 0]}")

    #  Test 2: Edge case
    nums2 = [0]
    sol.moveZeroes(nums2)
    print(f"Test 2: {nums2} == [0] → {nums2 == [0]}")

    #  Test 3: Tricky/negative
    nums3 = [1, 2, 3]
    sol.moveZeroes(nums3)
    print(f"Test 3: {nums3} == [1, 2, 3] → {nums3 == [1, 2, 3]}")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 1**: `nums = [0, 1, 0, 3, 12]`

Initial State:
- `nums = [0, 1, 0, 3, 12]`
- `left = 0`

Step 1: `right = 0`
- Check `nums[0] != 0` → `0 != 0` → **False**
- Do nothing.
- **State**: `left = 0`, `nums = [0, 1, 0, 3, 12]`

Step 2: `right = 1`
- Check `nums[1] != 0` → `1 != 0` → **True**
- Swap `nums[0]` and `nums[1]`:
- Before: `[0, 1, ...]`
- After: `[1, 0, 0, 3, 12]`
- Increment `left` → `left = 1`
- **State**: `left = 1`, `nums = [1, 0, 0, 3, 12]`

Step 3: `right = 2`
- Check `nums[2] != 0` → `0 != 0` → **False**
- Do nothing.
- **State**: `left = 1`, `nums = [1, 0, 0, 3, 12]`

Step 4: `right = 3`
- Check `nums[3] != 0` → `3 != 0` → **True**
- Swap `nums[1]` and `nums[3]`:
- Before: `[1, 0, 0, 3, ...]`
- After: `[1, 3, 0, 0, 12]`
- Increment `left` → `left = 2`
- **State**: `left = 2`, `nums = [1, 3, 0, 0, 12]`

Step 5: `right = 4`
- Check `nums[4] != 0` → `12 != 0` → **True**
- Swap `nums[2]` and `nums[4]`:
- Before: `[1, 3, 0, 0, 12]`
- After: `[1, 3, 12, 0, 0]`
- Increment `left` → `left = 3`
- **State**: `left = 3`, `nums = [1, 3, 12, 0, 0]`

Loop Ends (`right` reaches 5)

Final Output: `[1, 3, 12, 0, 0]`

Key Takeaway:
The `left` pointer always points to the **next position** where a non-zero should go. Every time we find a non-zero, we **swap it forward**, preserving order and pushing zeros backward **in one pass**.

Complexity Analysis

- **Time Complexity**: `O(n)`

  We iterate through the array exactly once with the `right` pointer. Each swap is `O(1)`. Total operations scale linearly with input size.

- **Space Complexity**: `O(1)`

  Only two integer pointers (`left`, `right`) are used. The array is modified in-place — no extra space proportional to input.

6. Squares of a Sorted Array

Pattern: Two Pointers

Problem Statement

Given an integer array `nums` sorted in **non-decreasing order**, return an array of the squares of each number sorted in **non-decreasing order**.

Sample Input & Output

```
Input: [-4,-1,0,3,10]
Output: [0,1,9,16,100]
Explanation: After squaring, the array becomes [16,1,0,9,100].
             Sorting gives [0,1,9,16,100].
```

```
Input: [-7,-3,2,3,11]
Output: [4,9,9,49,121]
```

```
Input: [-5]
Output: [25]
Explanation: Single negative element → square is positive.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def sortedSquares(self, nums: List[int]) -> List[int]:
        # STEP 1: Initialize structures
        #   - n: length of input (for indexing)
        #   - result: pre-allocated output array (same size)
        #   - left, right: two pointers at start and end
        n = len(nums)
        result = [0] * n
        left, right = 0, n - 1
```

```python
        # STEP 2: Main loop / recursion
        #   - Fill result from END to START (largest square first)
        #   - Compare abs values via squares (no need for abs())
        #   - Invariant: result[i+1:] is sorted; we fill result[i]
        for i in range(n - 1, -1, -1):
            left_sq = nums[left] * nums[left]
            right_sq = nums[right] * nums[right]

            # STEP 3: Update state / bookkeeping
            #   - Pick larger square → place at current end
            #   - Move corresponding pointer inward
            if left_sq > right_sq:
                result[i] = left_sq
                left += 1
            else:
                result[i] = right_sq
                right -= 1

        # STEP 4: Return result
        #   - Always valid: input non-empty per constraints
        return result

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.sortedSquares([-4, -1, 0, 3, 10]) == [0, 1, 9, 16, 100]

    #  Test 2: Edge case
    assert sol.sortedSquares([-5]) == [25]

    #  Test 3: Tricky/negative
    assert sol.sortedSquares([-7, -3, 2, 3, 11]) == [4, 9, 9, 49, 121]

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `sortedSquares([-4, -1, 0, 3, 10])` step by step.

Initial State:
- `nums = [-4, -1, 0, 3, 10]`
- `n = 5`
- `result = [0, 0, 0, 0, 0]`
- `left = 0`, `right = 4`

We fill `result` **from the last index (4) to 0**, placing the **largest remaining square** at each step.

Iteration 1: `i = 4`
- `left_sq = (-4)² = 16`
- `right_sq = 10² = 100`
- Since `16 < 100` → pick `right_sq`
- `result[4] = 100`
- `right` moves to `3`
- **State**: `result = [0, 0, 0, 0, 100]`, `left=0`, `right=3`

Iteration 2: `i = 3`
- `left_sq = (-4)² = 16`
- `right_sq = 3² = 9`
- `16 > 9` → pick `left_sq`
- `result[3] = 16`
- `left` moves to `1`
- **State**: `result = [0, 0, 0, 16, 100]`, `left=1`, `right=3`

Iteration 3: `i = 2`
- `left_sq = (-1)² = 1`
- `right_sq = 3² = 9`
- `1 < 9` → pick `right_sq`
- `result[2] = 9`
- `right` moves to `2`
- **State**: `result = [0, 0, 9, 16, 100]`, `left=1`, `right=2`

Iteration 4: `i = 1`
- `left_sq = (-1)² = 1`
- `right_sq = 0² = 0`
- `1 > 0` → pick `left_sq`
- `result[1] = 1`
- `left` moves to 2
- **State**: `result = [0, 1, 9, 16, 100]`, `left=2`, `right=2`

Iteration 5: `i = 0`
- `left_sq = 0² = 0`
- `right_sq = 0² = 0`
- Equal → pick `right_sq` (or left; doesn't matter)
- `result[0] = 0`
- `right` moves to 1
- **Final State**: `result = [0, 1, 9, 16, 100]`

Output: `[0, 1, 9, 16, 100]` — correctly sorted squares!

Key Insight:
Because the input is sorted, the **largest square must come from one of the ends** (most negative or most positive). The two-pointer technique exploits this by comparing ends and working inward.

Complexity Analysis

- **Time Complexity**: `O(n)`

  We iterate through the array **exactly once** (`n` steps). Each step does constant-time operations (squaring, comparison, assignment).

- **Space Complexity**: `O(1)` **auxiliary space** (or `O(n)` total if counting output)

  We use only a few extra variables (`left`, `right`, `i`, `left_sq`, `right_sq`). The output array `result` is **required** by the problem, so auxiliary space is constant.

7. Sort Colors

Pattern: Two Pointers

Problem Statement

> Given an array `nums` with `n` objects colored red, white, or blue, sort them **in-place** so that objects of the same color are adjacent, with the colors in the order red, white, and blue.

We will use the integers `0`, `1`, and `2` to represent the color red, white, and blue, respectively.

You must solve this problem without using the library's sort function.

Sample Input & Output

```
Input: [2,0,2,1,1,0]
Output: [0,0,1,1,2,2]
Explanation: All 0s (red) come first, then 1s (white), then 2s (blue).
```

```
Input: [2,0,1]
Output: [0,1,2]
Explanation: One of each color, sorted in correct order.
```

```
Input: [0]
Output: [0]
Explanation: Single-element edge case - already sorted.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def sortColors(self, nums: List[int]) -> None:
        # STEP 1: Initialize pointers
        #   - left: boundary for 0s (exclusive right end)
        #   - right: boundary for 2s (exclusive left end)
        #   - i: current index being examined
        left = 0
        right = len(nums) - 1
        i = 0

        # STEP 2: Main loop - process until i crosses right
        #   - Invariant: [0:left] = 0s, [right+1:] = 2s
        #   - Middle section [left:i] = 1s, [i:right+1] = unprocessed
        while i <= right:
            if nums[i] == 0:
                # STEP 3a: Move 0 to left section
                nums[i], nums[left] = nums[left], nums[i]
                left += 1
                i += 1  # safe to move forward (swapped 0 or 1)
            elif nums[i] == 2:
                # STEP 3b: Move 2 to right section
                nums[i], nums[right] = nums[right], nums[i]
                right -= 1
                # do NOT increment i - new nums[i] is unprocessed
            else:
                # STEP 3c: nums[i] == 1 → already in correct zone
                i += 1

        # STEP 4: Return result
        #   - Modification is in-place; no return needed (None)

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    nums1 = [2, 0, 2, 1, 1, 0]
    sol.sortColors(nums1)
    print(f"Test 1: {nums1}")  # Expected: [0, 0, 1, 1, 2, 2]
```

```python
#  Test 2: Edge case
nums2 = [0]
sol.sortColors(nums2)
print(f"Test 2: {nums2}")  # Expected: [0]

#  Test 3: Tricky/negative
nums3 = [2, 0, 1]
sol.sortColors(nums3)
print(f"Test 3: {nums3}")  # Expected: [0, 1, 2]
```

> **How to use**: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace **Test 3**: `nums = [2, 0, 1]`

Initial state:
- `nums = [2, 0, 1]`
- `left = 0`, `right = 2`, `i = 0`

Step 1: `i=0`, `nums[0]=2`
→ Enter `elif nums[i] == 2`
→ Swap `nums[0]` and `nums[2]`:
  `nums` becomes `[1, 0, 2]`
→ `right` decrements to `1`
→ `i` stays at `0` (because new `nums[0]=1` must be checked)

State: `nums=[1,0,2]`, `left=0`, `right=1`, `i=0`

Step 2: `i=0`, `nums[0]=1`
→ Enter `else` (value is 1)
→ `i` increments to `1`

State: `nums=[1,0,2]`, `left=0`, `right=1`, `i=1`

Step 3: `i=1`, `nums[1]=0`
→ Enter `if nums[i] == 0`
→ Swap `nums[1]` and `nums[left=0]`:
  `nums` becomes `[0, 1, 2]`
→ `left` increments to `1`
→ `i` increments to `2`

State: `nums=[0,1,2]`, `left=1`, `right=1`, `i=2`

Step 4: Check loop condition `i <= right` → `2 <= 1`?
→ Loop ends.

Final output: `[0, 1, 2]`

Key insight:
- When we swap a `2` to the right, the element swapped *into* position `i` might be a `0` or `1`, so we **must not advance `i`** until it's processed.
- But when we swap a `0` from the left, we know the element coming in is either a `1` (from middle) or another `0` (already processed), so it's safe to move `i` forward.

Complexity Analysis

- **Time Complexity**: `O(n)`

  We traverse the array at most once. Each element is swapped at most twice (once for 0, once for 2), so total operations are linear.

- **Space Complexity**: `O(1)`

  Only three integer pointers (`left`, `right`, `i`) are used — constant extra space. The sort is in-place.

8. Rotate Array

Pattern: Two Pointers

Problem Statement

> Given an integer array `nums`, rotate the array to the right by `k` steps, where `k` is non-negative.

You must do this **in-place** with **O(1)** extra space.

Sample Input & Output

```
Input: nums = [1,2,3,4,5,6,7], k = 3
Output: [5,6,7,1,2,3,4]
Explanation: Rotate 3 steps to the right:
  [1,2,3,4,5,6,7] → [7,1,2,3,4,5,6] →
  [6,7,1,2,3,4,5] → [5,6,7,1,2,3,4]
```

```
Input: nums = [-1,-100,3,99], k = 2
Output: [3,99,-1,-100]
Explanation: Rotate 2 steps right:
  [-1,-100,3,99] → [99,-1,-100,3] → [3,99,-1,-100]
```

```
Input: nums = [1], k = 0
Output: [1]
Explanation: No rotation needed (k=0). Edge case: single element.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def rotate(self, nums: List[int], k: int) -> None:
        # STEP 1: Normalize k to avoid redundant full rotations
        #   - If k >= len(nums), only k % n matters
        n = len(nums)
        k = k % n
```

```
        if k == 0:
            return  # No rotation needed

        # STEP 2: Reverse entire array
        #   - This brings last k elements to front (but reversed)
        self.reverse(nums, 0, n - 1)

        # STEP 3: Reverse first k elements
        #   - Fixes order of the rotated part
        self.reverse(nums, 0, k - 1)

        # STEP 4: Reverse remaining n-k elements
        #   - Fixes order of the original front part
        self.reverse(nums, k, n - 1)

    def reverse(self, nums: List[int], left: int,
                right: int) -> None:
        # Helper: reverse subarray using two pointers
        while left < right:
            nums[left], nums[right] = nums[right], nums[left]
            left += 1
            right -= 1

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #   Test 1: Normal case
    nums1 = [1, 2, 3, 4, 5, 6, 7]
    sol.rotate(nums1, 3)
    print(f"Test 1: {nums1}")  # Expected: [5,6,7,1,2,3,4]

    #   Test 2: Edge case (k=0, single element)
    nums2 = [1]
    sol.rotate(nums2, 0)
    print(f"Test 2: {nums2}")  # Expected: [1]

    #   Test 3: Tricky/negative (k > n, duplicates)
    nums3 = [-1, -100, 3, 99]
    sol.rotate(nums3, 2)
    print(f"Test 3: {nums3}")  # Expected: [3,99,-1,-100]
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 1**: `nums = [1,2,3,4,5,6,7]`, `k = 3`.

Initial Setup

- `nums = [1, 2, 3, 4, 5, 6, 7]`
- `n = 7`
- `k = 3 % 7 = 3` → not zero, so proceed.

Step 1: Reverse entire array → `reverse(nums, 0, 6)`

- **Two pointers**: `left=0`, `right=6`
- Swap `nums[0]` and `nums[6]` → `[7,2,3,4,5,6,1]`
- `left=1`, `right=5` → swap → `[7,6,3,4,5,2,1]`
- `left=2`, `right=4` → swap → `[7,6,5,4,3,2,1]`
- `left=3`, `right=3` → stop (`left >= right`)
- **Result**: `[7,6,5,4,3,2,1]`

Why? The last `k=3` elements (`5,6,7`) are now at the front — but reversed as `7,6,5`.

Step 2: Reverse first `k=3` elements → `reverse(nums, 0, 2)`

- `left=0`, `right=2`
- Swap `nums[0]` and `nums[2]` → `[5,6,7,4,3,2,1]`
- `left=1`, `right=1` → stop
- **Result**: `[5,6,7,4,3,2,1]`

Now the rotated part (`5,6,7`) is in correct order!

Step 3: Reverse remaining `n-k = 4` elements → `reverse(nums, 3, 6)`

- `left=3`, `right=6`
- Swap `nums[3]` and `nums[6]` → `[5,6,7,1,3,2,4]`
- `left=4`, `right=5` → swap → `[5,6,7,1,2,3,4]`
- `left=5`, `right=4` → stop
- **Final Result**: `[5,6,7,1,2,3,4]` → matches expected!

Summary of State Changes

Step	Operation	Array State
0	Start	[1,2,3,4,5,6,7]
1	Reverse all	[7,6,5,4,3,2,1]
2	Reverse first 3	[5,6,7,4,3,2,1]
3	Reverse last 4	[5,6,7,1,2,3,4]

Key Insight: Three reverses simulate rotation without extra space.
This is a classic **two pointers** trick: use symmetry to reposition elements.

Complexity Analysis

- **Time Complexity**: `O(n)`

  We reverse the array 3 times. Each reverse visits at most `n` elements.
  Total operations `n/2 + k/2 + (n-k)/2 = n` → linear.

- **Space Complexity**: `O(1)`

  Only a few integer variables (`left`, `right`, `n`, `k`) are used.
  No extra arrays or recursion stack — truly in-place.