ChatGPT

Mastering Two Pointers: Deep Dive into Key Problems

The **two-pointers technique** is a powerful pattern for solving array and string problems efficiently. In this report, we'll explore five classic problems from the Neetcode roadmap and see how two pointers can drastically improve solutions:

- **Valid Palindrome**
- **Two Sum II – Input Array Is Sorted**
- **3Sum**
- **Container With Most Water**
- **Trapping Rain Water**

For each problem, we'll start with a **brute-force approach** (as you might in a FAANG interview discussion) and then refine it to the optimal two-pointer solution. We'll include dry-run examples with pointer movement diagrams, full JavaScript code (with comments), complexity analysis, alternative approaches (if noteworthy), and common gotchas that often trip up candidates. Finally, we'll wrap up with **memorization tips** to help you solidify these patterns for interviews.

---

1. Valid Palindrome

Problem: Determine if a given string is a palindrome, considering only alphanumeric characters and ignoring cases. In other words, after removing non-letters/numbers and normalizing to one case, does the string read the same backward and forward? [1]

Brute-Force Approach (Filtering & Reversing)

A straightforward brute-force method is: 1. **Filter and Normalize:** Remove all non-alphanumeric characters and convert the string to all lowercase. [2] 2. **Check Palindrome:** Compare the cleaned string with its reverse. If they are identical, it's a palindrome; otherwise not.

Correctness: This works because after filtering and normalizing, a palindrome reads the same forward and backward. For example, `"A man, a plan, a canal: Panama"` becomes `"amanaplanacanalpanama"` which is the same reversed (palindrome), whereas `"race a car"` becomes `"raceacar"`, which differs from its reverse [3] .

Complexity: Cleaning and reversing each take O(n) time for a string of length n, so overall time is O(n). Memory overhead is O(n) for the filtered copy of the string.

Why Not Stop Here? While this method is O(n) and acceptable, it uses extra space for the filtered string. In an interview, you might be prompted to do it **in-place with two pointers**, which is more space-efficient [4] and demonstrates grasp of the pattern.

Optimized Two-Pointer Solution (In-Place Check)

The two-pointer approach avoids extra space by checking characters in-place from both ends: - Initialize one pointer `i` at the start (index 0) and another pointer `j` at the end (index n-1) [5]. - Convert characters to lowercase for uniform comparison (or compare in a case-insensitive manner) [6]. - **Skip non-alphanumeric characters:** Move `i` forward until it points to a letter or digit; likewise, move `j` backward until it points to a letter or digit [7]. - Compare the characters at `i` and `j`. If they differ, the string is not a palindrome (return false). If they match, move `i` forward and `j` backward and continue [7]. - Continue until the two pointers meet or cross. If all compared characters matched, it's a palindrome (return true).

This approach checks the palindrome condition on the fly without creating a new string. It leverages two pointers to scan inward from both ends, naturally handling the reversal comparison.

Dry Run Example:
Consider `s = "A man, a plan, a canal: Panama"`: 1. Lowercase everything: `"a man, a plan, a canal: panama"`. 2. Start with `i = 0` (at `'a'`) and `j = (length-1)` (at `'a'`). Both are alphanumeric and match (`'a' == 'a'`), so move inward (`i→1`, `j→length-2`) [8]. 3. `i` at index 1 points to space (non-alnum), so skip it (`i→2`). Continue skipping any non-alphanumeric (there's a space and comma to skip, etc.) [6]. 4. `j` from the end moves left skipping non-alnum (`j` skips the trailing space or punctuation if any). 5. Compare next pair: `'m'` and `'m'` → match. Continue. 6. Continue this process for each pair: all letters "a", "n", "a", ... match in order. 7. If we reach the middle with all matches, the function returns true [9]. (For the given example, it returns **true**.) 8. For a counterexample, `s = "race a car"`: lowercase -> `"race a car"`. Compare outward letters: `'r' == 'r'`, `'a' == 'a'`, `'c' == 'c'`, then compare `'e'` vs `'a'` (after skipping spaces) which **mismatch**, so return false at that point [8].

Why Two Pointers? It allows us to compare the string from both ends in one pass (O(n) time) and skip irrelevant characters efficiently. We avoid building a reversed string or extra arrays, achieving O(1) space usage [10].

Common Pitfalls: - **Skipping logic:** Make sure to skip non-alphanumeric characters on both pointers *before* comparing. A common bug is forgetting one of the skips or not using `while` loops to skip sequences of invalid characters [11]. - **Case sensitivity:** Always normalize case (e.g., using `toLowerCase()` or converting char codes) before comparing [6]. Forgetting this will cause false mismatches (like 'A' vs 'a'). - **Unicode and alphanumeric check:** In JavaScript, using a regex like `/[a-z0-9]/i` or `Character.isLetterOrDigit` in other languages helps identify alphanumeric chars [11]. Edge cases include accented letters or non-English letters – depending on problem assumptions, focusing on [A-Za-z0-9] is typically sufficient.

Example Dry Run Diagram

Let's visualize the two-pointer checks with a shorter example: `s = "ab?ca"`. This is not a palindrome because of the middle character difference, but let's see how pointers catch it:

- Initial string: `"ab?ca"`
  Positions: 0 1 2 3 4
  Characters: **a** b ? c **a**
  Pointers: `i` → a (index 0) , `j` → a (index 4) – compare 'a' vs 'a' (match).
- Move inward: `i` →1 (`b`), `j` →3 (`c`).
- Check if pointers are on non-alphanumeric:
- `i` at index 1 is 'b' (valid).
- `j` at index 3 is 'c' (valid).
- Compare `s[i]` vs `s[j]`: 'b' vs 'c' – they **do not match**, so we conclude early that the string isn't a palindrome (return false).

No need to check further once a mismatch is found [12]. This approach would identify the palindrome failure as soon as the first unequal pair is encountered, which is optimal.

Full JavaScript Solution: Valid Palindrome

```
/**
 * Determine if a string is a palindrome, considering only alphanumeric chars.
 * @param {string} s
 * @return {boolean}
 */
function isPalindrome(s) {
  // 1. Normalize the string: lowercase for case-insensitivity
  s = s.toLowerCase();
  let i = 0;
  let j = s.length - 1;
  // 2. Use two pointers to compare from both ends
  while (i < j) {
    // Move i right if not alphanumeric
    while (i < j && !isAlphaNumeric(s[i])) {
      i++;
    }
    // Move j left if not alphanumeric
    while (i < j && !isAlphaNumeric(s[j])) {
      j--;
    }
    // Compare characters at i and j
    if (s[i] !== s[j]) {
      return false; // mismatch found, not a palindrome
    }
    i++;
    j--;
```

```
  }
  // If we complete the loop without mismatches, it is a palindrome
  return true;
}

// Helper to check alphanumeric character (letter or digit)
function isAlphaNumeric(ch) {
  return /[0-9a-z]/i.test(ch);
}
```

Time Complexity: O(n), where n is the length of the string. We potentially touch each character at most once (skipping invalid ones and comparing valid ones) [10] .

Space Complexity: O(1) auxiliary space – the checks happen in-place with just a few pointer/index variables [10] . (Note: Converting to lowercase may create a new string in some languages, but in an interview, that is usually acceptable. You can also convert characters on the fly to truly achieve O(1) extra space.)

Alternate Approach: The filter-and-reverse method is a valid alternative to remember. It's conceptually simple: `cleaned = s.toLowerCase().replace(/[^0-9a-z]/gi, "")` then check `cleaned === cleaned.split("").reverse().join("")`. This uses O(n) time and O(n) space. It's fine for practice, but the two-pointer method is more memory-efficient and often preferred in interviews to demonstrate algorithmic thinking [5] .

Gotchas to Watch Out For: - **Empty string or single char:** These should trivially return true (palindrome by default). Our two-pointer loop naturally handles these (loop won't run long if `i >= j` immediately). - **All non-alphanumeric string:** e.g. `"!!.."`. After skipping all chars, our loop would exit and return true (which is correct, as an empty string is a palindrome). Just be mindful that skipping logic covers this (our function would return true by reaching the end of while loop without finding a false condition). - **Numeric palindromes:** e.g. `"0P0"`. Ensure that digits are handled in the same logic (they are alphanumeric by definition). Our regex `/[0-9a-z]/i` covers digits [11] . - **Unicode letters:** If the input can contain Unicode letters (like é, α, 中), be aware that `\w` or `[a-z]` won't match them. JavaScript's regex with `\p{L}` (Unicode property escapes) can handle letters in general categories, but that's beyond typical interview scope. Generally, assume only basic alphanumeric.

---

2. Two Sum II – Input Array Is Sorted

Problem: We have a sorted array of integers and a target sum. We need to find **two numbers** such that they add up to the target, and return their 1-indexed positions (with the guarantee that exactly one valid solution exists) [13] [14] . This is essentially the classic two-sum problem with the twist that the input is sorted and we want the pair's indices.

Brute-Force Approach (Double Loop)

The brute force approach tries every possible pair of numbers to check if they sum to the target: - Use two loops: the outer loop picks the first number (at index i), the inner loop picks a second number (at index j > i).

For each pair, check if `numbers[i] + numbers[j] == target`. - If a matching pair is found, return their indices (adjusted to 1-indexing as required).

Complexity: This checks O(n^2) pairs in the worst case for an array of length n. That's fine for very small n, but not scalable (n can be up to 30,000 in this problem [15]; 30k^2 ~ 900 million checks, which is far too slow).

Why suboptimal? We're not exploiting the sorted order at all. Sorted order is a *huge* hint that a more efficient approach is possible.

Optimized Two-Pointer Solution (Opposite Ends)

Because the array is sorted, we can use the classic two-pointer trick from opposite ends [16]: - Set one pointer `left` at the **start** of the array and another pointer `right` at the **end** of the array [17]. - Compute the current sum: `S = numbers[left] + numbers[right]`. - If `S == target`, we've found the solution. Return `[left+1, right+1]` (using 1-based indexing as required) [18]. - If `S < target`, the sum is too small. Since the array is sorted, increasing the sum means moving the `left` pointer to the right (to a larger number) [19]. So do `left++` [20]. - If `S > target`, the sum is too large. We can decrease it by moving the `right` pointer to the left (to a smaller number) [19]. So do `right--` [20]. - Continue this process until the sum matches the target. The guarantee of one solution means we will find it without needing additional checks for exhausting possibilities (the two pointers will meet when `left >= right` if no solution, but here solution exists) [21].

Why does this work? Since the array is sorted, when the current sum is lower than the target, moving the left index rightwards increases the sum (because we replace a smaller addend with a larger one). Conversely, if the sum is too high, moving the right index leftwards decreases the sum [22] [23]. This way, we systematically narrow down to the correct pair without checking every combination.

Dry Run Example:
Input: `numbers = [2, 7, 11, 15]`, `target = 9`. The correct answer is indices `[1, 2]` (1-indexed) because 2 + 7 = 9.

- Start: `left = 0` (points to 2), `right = 3` (points to 15).
  Sum `S = 2 + 15 = 17`. This is greater than 9 (too large) [24]. To reduce the sum, move the right pointer left.
- Now: `left = 0` (still 2), `right = 2` (points to 11).
  Sum `S = 2 + 11 = 13`. Still too large. Move `right` left again [24].
- Now: `left = 0` (2), `right = 1` (points to 7).
  Sum `S = 2 + 7 = 9`. Bingo – found the target sum [24]. Return `[left+1, right+1] = [1, 2]`.

These steps show how the pointers "homed in" on the correct pair by eliminating pairs that were too large. We didn't have to check any pair involving the 15 again after the first step, for example.

Another example: `numbers = [2, 3, 4]`, `target = 6`. - Start: `left=0` (2), `right=2` (4), `S=6` → found immediately, output `[1, 3]`. If the first sum wasn't correct, we'd adjust pointers accordingly. The approach cleanly covers all possibilities by moving inward.

Corner cases: If the numbers can be negative or zero, the logic still holds as long as the array is sorted. For instance, `numbers = [-1, 0]`, `target = -1` should return `[1, 2]` (because -1 + 0 = -1) [25]. Our algorithm would handle that: `left` at -1, `right` at 0, sum = -1, which matches directly.

Common Pitfalls: - **Off-by-one and Indexing:** Remember the problem expects 1-indexed results. It's easy to forget adding 1 to each index when returning [26] [27]. In code, do `return [left+1, right+1]`. - **Not using the sorted property:** Occasionally, candidates try using a hash map (like the classic unsorted two-sum solution) which works in O(n) time but isn't necessary here. The two-pointer is simpler and uses constant space [28]. Using a hash in this sorted scenario would be a wasted opportunity to show understanding of sorted array patterns. - **Skipping duplicate handling:** In this problem, there is exactly one solution by constraint [13], so we don't need to worry about multiple pairs or skipping duplicates. But in variations where multiple pairs might be possible, ensure to decide if you need all pairs or just one. Here just one pair is needed, so we can return as soon as we find it. - **When pointers cross:** If no solution existed (contrary to this problem's guarantee), the loop would end when `left >= right` without finding a sum. In that case, one might return an indicator like `[-1, -1]` or throw an exception. But given the problem guarantee, we don't implement that case in code – still, it's good to know for completeness.

Full JavaScript Solution: Two Sum II

```
/**
 * Given a sorted array and a target, finds two numbers such that they add to
target.
 * Returns 1-indexed positions of the two numbers.
 * @param {number[]} numbers - sorted array of integers
 * @param {number} target
 * @return {number[]} - [index1, index2] (1-indexed)
 */
function twoSumSorted(numbers, target) {
  let left = 0;
  let right = numbers.length - 1;
  while (left < right) {
    const sum = numbers[left] + numbers[right];
    if (sum === target) {
      // Found the pair; return 1-indexed positions
      return [left + 1, right + 1];
    } else if (sum < target) {
      // Sum too low, move left pointer to increase sum
      left++;
    } else {
      // Sum too high, move right pointer to decrease sum
      right--;
    }
  }
  // If no solution (though one is guaranteed in this problem), we could return
[-1, -1]
}
```

Time Complexity: O(n). In the worst case, each element will be visited at most once by either pointer, so it's a linear scan through the array [28]. This is a significant improvement over O(n^2). For n=30,000, this is on the order of 30k steps instead of 900 million!

Space Complexity: O(1) auxiliary. We only use a couple of integer variables for indices and sum, and the output array of size 2.

Alternate Approaches:
- *Binary Search Variant:* For each element `numbers[i]`, use binary search to find `target - numbers[i]`. This yields O(n log n) time. It's more complex and still slower than two-pointer's O(n) in practice. - *Hash Map:* As mentioned, we can use a hash map to store seen values and look for complements (`target - x`) as we iterate. However, building a hash map would be O(n) space, and since the array is sorted and indices (1-indexed) are required, the two-pointer approach is cleaner.

Those alternatives are usually brought up to compare approaches, but **two-pointers** is the idiomatic solution given the sorted input. It's a pattern worth memorizing: **for sorted arrays and a two-sum goal, think two pointers from ends.**

Gotchas and Tips:
- Remember that this problem uses **1-indexed positions** (LeetCode 167's peculiarity). Always double-check indexing in the problem statement. Our solution correctly does `+1` on the indices [27]. - The input is sorted non-decreasing (could have equal elements). If there were a scenario with, say, `numbers = [2, 2, 7, ...]` and target 4, the first and second elements would be the answer. Our method would find it because left starts at first 2 and right would move inward to second 2, sum = 4. - There is exactly one solution, meaning we don't need to consider cases of multiple pairs or no pairs. This simplifies the reasoning – once we find a match, we return immediately. If you ever use this approach in a context where multiple solutions are possible (like the 3Sum problem next), you'd modify it to collect all solutions or continue searching appropriately. - Emphasize the sorted usage in explanation: many interviewers want to see that you **identify sorted order as a cue for two-pointer techniques**. Saying something like "Since the array is sorted, I can use a two-pointer approach to find the complement efficiently" is a good way to articulate your thought process [29].

---

3. 3Sum

Problem: Given an integer array (not necessarily sorted), find all unique triplets `[a, b, c]` in the array such that `a + b + c = 0`. We need to output **unique** triplets, meaning no duplicate combinations (order of numbers in a triplet doesn't matter, and each triplet's numbers should be in non-decreasing order typically) [30].

This is a classic interview problem that extends two-sum to three numbers. It's significantly trickier because of the need to avoid duplicates and because a brute-force approach is cubic time.

Brute-Force Approach (Triple Loop with Duplicate Check)

Brute force would entail checking every possible triplet: - Use three nested loops (i, j, k) to pick all combinations of three distinct indices and check if they sum to zero. - To avoid outputting duplicate triplets, you'd need a structure (like a `Set` or sorting each triplet before adding to a set) to ensure uniqueness of output.

Complexity: O(n^3) time just to enumerate triplets for an array of length n. For n=200 (typical LeetCode limit ~300), that's 8 million combinations – borderline or likely too slow in practice (and certainly too slow for larger n). Also, handling duplicates correctly adds complexity.

Why it's not ideal: O(n^3) is usually unacceptable for n > ~200. Plus, writing the logic to skip duplicates in the brute force is cumbersome. We need a better way.

Two-Pointer Optimized Solution (After Sorting)

A well-known optimal solution for 3Sum is O(n^2), achieved by: 1. **Sort the array.** This helps systematically avoid duplicates and use two-pointer technique for the two-sum part [31] . 2. **Traverse with a fixed pointer:** Iterate `i` from 0 to n-3 (each `i` is a candidate for the first number of the triplet). - If the array is sorted, and you want unique triplets, it's common to **skip duplicate values for** `i` (if `nums[i]` is the same as `nums[i-1]`, skip to avoid duplicate triplets) [32] . - For each `i`, we want to find two numbers `nums[j]` and `nums[k]` (with `j > i` and `k` at end) such that `nums[i] + nums[j] + nums[k] = 0`. 3. **Two-sum for the remaining part:** Now this reduces to a two-sum problem for the subarray `nums[i+1 ... end]` with target `-nums[i]` (because we need `nums[j] + nums[k] = -nums[i]` to make the total zero). - Set `left = i+1` and `right = n-1` (start and end of the remaining subarray). - While `left < right`, compute `sum = nums[i] + nums[left] + nums[right]`. - If `sum == 0`, we found a triplet. Add it to results [33] . - Then move `left` and `right` inward to look for another pair. **Skip duplicates:** after finding a valid triplet, increment `left` while `nums[left]` is the same as the previous value to avoid duplicate triplets; similarly decrement `right` while `nums[right]` is the same as previous [34] . This ensures uniqueness in results. - If `sum < 0`: the sum is too low (since array is sorted, if we need a bigger sum, we should increment `left` to get a larger number) [35] . - If `sum > 0`: the sum is too high, decrement `right` to get a smaller number [35] . - Continue until `left >= right`. Then move to the next `i`.

This approach leverages sorting to both manage duplicates and enable the two-pointer sweep for the two-sum subproblem.

Why sorting? - Sorting the array doesn't change the existence of triplets, but it allows us to **avoid duplicates easily** by skipping over equal elements in sequence [36] [37] . - It also allows the two-pointer technique for two-sum: with a sorted subarray, we can move `left` and `right` intelligently based on the sum.

Example Dry Run:
Consider `nums = [-1, 0, 1, 2, -1, -4]`. The unique triplets summing to 0 are expected to be `[-1, -1, 2]` and `[-1, 0, 1]`.

- **Sort first:** `nums` becomes `[-4, -1, -1, 0, 1, 2]`.
- Initialize result list `res = []`.
- Iterate `i` from 0 to 3 (inclusive, since we need room for two more numbers after i):
- `i = 0`: `nums[i] = -4`. Target for two-sum is `0 - (-4) = 4`.
  - Set `left = 1` (at `-1`), `right = 5` (at `2`).
  - sum = `-4 + (-1) + 2 = -3` (too low). Increment `left` (we need a larger sum) [35].
  - `left = 2` (at `-1`), `right = 5` (2). Now sum = `-4 + (-1) + 2 = -3` (still too low, left++ again).
  - `left = 3` (at `0`), `right = 5` (2). sum = `-4 + 0 + 2 = -2` (too low, left++).
  - `left = 4` (at `1`), `right = 5` (2). sum = `-4 + 1 + 2 = -1` (too low, left++).
  - `left = 5` which is not < right anymore. Loop ends. No triplet found with i=0.
- `i = 1`: `nums[i] = -1`. (Important: `i=1` is the first -1; we'll allow this, but we *will skip i=2* because that's a duplicate -1.)
  - Target for two-sum is `0 - (-1) = 1`.
  - `left = 2` (at the next `-1`), `right = 5` (at `2`).
  - sum = `-1 + (-1) + 2 = 0`. Found a triplet `[-1, -1, 2]` [33]. Add it to results.
  - Now, move `left` and `right`: `left++` -> index 3 (value `0`), `right--` -> index 4 (value `1`).
  - Also skip duplicates: after incrementing left, `nums[left]` is now `0` which is different from previous `-1`, so that's fine. `right` moved to `1` which is different from previous `2`, so also fine.
  - Now `left = 3`, `right = 4`.
  - sum = `-1 + 0 + 1 = 0`. Found another triplet `[-1, 0, 1]`. Add it.
  - Move `left++` -> 4, `right--` -> 3.
  - Now `left >= right`, loop ends for i=1.
  - We have two triplets so far: `[-1, -1, 2]` and `[-1, 0, 1]`.
- `i = 2`: `nums[i] = -1` (a duplicate of the previous value at i=1). We **skip** this i to avoid generating duplicate triplets that start with -1 again [37].
- `i = 3`: `nums[i] = 0`. Target is `0 - 0 = 0`.
  - `left = 4` (at `1`), `right = 5` (at `2`).
  - sum = `0 + 1 + 2 = 3` (too high). Decrement `right` (need smaller sum).
  - `left = 4`, `right = 4` (they meet, end loop). No triplet with 0 as first element (aside from the one we already found).
- We can stop at i=3 because beyond that i=4 or 5 would not have two others after it.

Result: `res = [[-1, -1, 2], [-1, 0, 1]]`, which matches expectation (order of triplets or elements within doesn't matter as long as they're sorted in each triplet).

Diagram – Pointer Movement for 3Sum:

Diagram: Using the two-pointer technique for 3Sum. We fix pointer `i` *for each first element and then use* `j` *(left) and* `k` *(right) to find pairs that sum with* `nums[i]` *to zero. If the sum is too low,* `j` *moves right to*

increase it; if too high, `k` *moves left to decrease it* [35]*. Each time a valid triplet is found (*`sum == 0`*), both pointers move inward and duplicates are skipped to avoid repeating the same triplet.*

In the figure above, imagine `i` scanning through the array indices on the horizontal axis, while `j` and `k` approach from either side for each `i`. This method efficiently explores combinations without backtracking every possibility.

Common Pitfalls: - **Duplicate triplets:** The bane of this problem. Always skip duplicate values for `i` (outer loop) to avoid repeating the same triplet base [36]. Similarly, after finding a valid triplet, skip over any duplicate values at `left` and `right` to avoid duplicate triplets in the result [34]. Forgetting these skips leads to repeated triplets in the output, which will likely be marked wrong. - **Triplet uniqueness vs element uniqueness:** Note we can use the same number in different triplets as long as the combination is unique. The skipping is just to avoid outputting the *same combination* more than once. E.g., if the array has multiple -1s, we just ensure triplets `[-1, -1, 2]` is output once, not multiple times. - **Sorting changes index order:** Since we sort, the original indices don't matter anymore (we output values, not original indices, in this problem). If a problem variant asked for original indices, sorting would complicate that. But here we return triplets of values, so sorting is fine. - **All-positive or all-negative arrays:** Quick checks can avoid unnecessary work. If after sorting, the first number is > 0, you can break early (no way to sum to 0 if smallest is > 0). If the last number is < 0, similarly no solution. Not strictly needed but a small optimization. More importantly, ensure your logic naturally handles cases where no triplets exist – you would just return an empty list. - **Two-pointer within triple loop boundary conditions:** The inner two-pointer runs from `i+1` to end. Ensure you reset `left` and `right` correctly for each `i`. A common mistake is reusing `left` / `right` from a previous iteration incorrectly or forgetting to reinitialize one of them.

Full JavaScript Solution: 3Sum

```
/**
 * Finds all unique triplets in the array which sum up to 0.
 * @param {number[]} nums
 * @return {number[][]} array of triplets
 */
function threeSum(nums) {
  const res = [];
  // Sort the array to enable two-pointer approach and easy duplicate skip
  nums.sort((a, b) => a - b);

  for (let i = 0; i < nums.length - 2; i++) {
    // Skip duplicate values for i to avoid duplicate triplets
    if (i > 0 && nums[i] === nums[i - 1]) {
      continue;
    }
    let left = i + 1;
    let right = nums.length - 1;
    const target = -nums[i];

    while (left < right) {
```

```
      const sum = nums[i] + nums[left] + nums[right];
      if (sum === 0) {
        // Found a valid triplet
        res.push([nums[i], nums[left], nums[right]]);
        // Move both pointers and skip over duplicates
        left++;
        right--;
        while (left < right && nums[left] === nums[left - 1]) {
          left++;  // skip duplicate left
        }
        while (left < right && nums[right] === nums[right + 1]) {
          right--; // skip duplicate right
        }
      } else if (sum < 0) {
        // Sum too low, increase it by moving left pointer to the right
        left++;
      } else {
        // Sum too high, decrease it by moving right pointer to the left
        right--;
      }
    }
  }

  return res;
}
```

Time Complexity: O(n^2). Sorting takes O(n log n), and the two-pointer sweep for each `i` takes O(n) in the worst case, and that inside an O(n) loop for `i` gives O(n^2) overall [38]. For an array of size ~200, n^2 = 40k, which is fine. Even for n=1000, 1e6 operations is okay. O(n^3) would be 1e9 which is not. Thus, O(n^2) is the practical upper bound for 3Sum; no known algorithm does significantly better for the general case (it's a well-known challenging problem in terms of time complexity).

Space Complexity: O(1) extra (not counting the output). We sort in-place and use a few pointers/variables. The output list of triplets could be large in theory (worst-case O(n^3) triplets for contrived inputs with many zeros), but usually we consider output separately. Auxiliary space for the algorithm itself is constant.

Alternative Approaches:
- *Hash Set (Two-Sum) Approach:* We can fix `i` and then use an *un-sorted* two-sum with a hash set to find pairs that sum to `-nums[i]`. This is another O(n^2) approach. For each `i`, you iterate j from i+1 to end, and use a set to check if `target - nums[j]` has been seen. This avoids sorting and two-pointer, but you must be careful to avoid duplicates. As an example, you could use a set to store seen pairs and only add a triplet when you encounter a complement that hasn't been added before [39] [40]. However, managing duplicates can get tricky. In practice, the sorting + two-pointer method is cleaner. - *No-sort approach:* It's possible to avoid sorting and still find unique triplets using a set of sets or other data structures to enforce uniqueness, but that tends to be more complex and slower in practice due to the overhead of maintaining those sets. Sorting simplifies the duplicate logic greatly. - *kSum generalization:* It's useful to note that this two-pointer approach is a specific case of a general kSum algorithm. For 3Sum, we fixed one element and

did two-sum. For 4Sum, one could fix two elements and then do two-sum on the remainder, etc. Those are also often asked in interviews. The patterns of sorting and skipping duplicates extend naturally to those, with complexity O(n^(k-1)) typically.

Gotchas: - **Don't forget to sort.** Without sorting, it's very hard to efficiently ensure triplets are unique without extra storage. Sorting is the key enabling factor for the two-pointer technique here [31]. - **Integer overflow (in other languages):** In some languages, adding three numbers could overflow if they are extreme (though many languages have big integer types or the constraints are limited). For most interview contexts with JavaScript or Python, not an issue. - **Output format:** Make sure you output a list of triplet *lists*. Each triplet should be sorted internally if the array is sorted (ours naturally is). And no duplicates in the list. It's easy to accidentally include a duplicate if skip logic isn't perfect – be sure to test on cases like `[0,0,0,0]` which should output just `[[0,0,0]]` (our code would handle this: i=0 yields one triplet [0,0,0], then skip all subsequent 0s for i). - **Performance on large input:** n can often be around 3000 for this problem on LeetCode. O(n^2) = 9e6 which is borderline but usually passes in optimized languages. In JS, 9 million operations might be okay if efficient, but any higher and you might worry. Always consider if any micro-optimizations are needed (like breaking early if sums get too large or small, but above we implicitly do that by pointer checks). Our approach is already quite optimal.

---

4. Container With Most Water

Problem: We're given an array of non-negative integers where each represents a vertical line's height on an X-axis graph (the index is the x-coordinate, height is the y-coordinate). We want to select two lines that, together with the x-axis, would form a container that holds the most water [41]. In other words, find two indices `i` and `j` such that `min(height[i], height[j]) * (j - i)` is maximized.

This is a classic two-pointer problem (LeetCode #11). It's often described as finding the pair of lines that can trap the most water between them.

Brute-Force Approach (Evaluate All Pairs)

The brute force solution checks every possible pair of lines: - For each index `i`, for each index `j > i`, compute the area formed by lines at `i` and `j`: `width = j - i`, `height = min(height[i], height[j])`, so `area = width * height` [42]. - Track the maximum area found. - Return that maximum area.

This correctly finds the answer but at O(n^2) time, which is too slow for large arrays (n could be 10^5, making n^2 = 10^10 comparisons – impossible to do in reasonable time).

Observing the problem structure: If we increase width, we might decrease height (because the limiting height is the shorter of the two lines). Conversely, focusing on tall lines might sacrifice width. We need a strategy to find the optimum without brute force.

Optimal Two-Pointer Solution (Opposite Ends, Greedy Move)

A well-known greedy strategy for this problem uses two pointers at the ends of the array and moves inward: - Start with `left = 0` and `right = n-1`, the widest possible container (using the first and last line). - Compute the area: `currentArea = min(height[left], height[right]) * (right - left)` [43]. Keep track of the max area seen. - Now, **move the pointer which has the smaller height** inward by one [44]: - If `height[left] < height[right]`, increment `left` (hoping to find a taller line to increase area) [44]. - Else if `height[right] <= height[left]`, decrement `right` [44]. - Continue calculating area at each step and updating max, until `left` meets `right`.

Why move the smaller side? This is the key to the greedy choice: If we have two lines at `left` and `right`, the area is limited by the **shorter line** (because water can't go higher than the shorter wall). Let's say `height[left] < height[right]`. Moving the `right` pointer inward (to maybe find a taller line on the right side) would reduce the width and still be limited by `height[left}` or something shorter – it can't increase area because you've reduced width and you haven't increased the limiting height (the left side remains the bottleneck) [45]. On the other hand, moving the `left` pointer inward might find a taller line that could overcome the loss of width. In short, **moving the taller side inward can never yield a larger area** than what you already have, because the area is constrained by the shorter side [45]. So you move the shorter side, hoping to find a taller line to increase the min(height) term of the area formula [45].

This greedy argument is crucial and can be proven more formally [45], but intuitively: - If left side is shorter, even if you move the right pointer left, any new area will still have height limited by left (or something even shorter if you encounter smaller lines), and width is less – so area can only decrease. - Thus, you only potentially gain area by increasing the limiting height, which means moving the shorter side inward to find a taller line.

Dry Run Example:
Consider `height = [1, 8, 6, 2, 5, 4, 8, 3, 7]` (a common example). The maximum area here is 49, achieved by using heights 8 and 7 at distance 7 apart (the lines at index 1 and index 8, 0-based) [46].

- Start: `left=0` (`height[0]=1`), `right=8` (`height[8]=7`).
  Width = 8, minHeight = min(1,7) = 1, area = 8 * 1 = 8. Max = 8 so far.
  Left side is shorter (1 < 7), so move `left` inwards.
- Now: `left=1` (`height[1]=8`), `right=8` (`height[8]=7`).
  Width = 7, minHeight = min(8,7) = 7, area = 7 * 7 = 49. Max = 49 now (new max).
  Right side is shorter (7 <= 8), move `right` inwards.
- Now: `left=1` (`8`), `right=7` (`height[7]=3`).
  Width = 6, minHeight = min(8,3) = 3, area = 6 * 3 = 18. Max stays 49.
  Right side (3) is shorter, move `right` inwards.
- Now: `left=1` (`8`), `right=6` (`height[6]=8`).
  Width = 5, minHeight = min(8,8) = 8, area = 5 * 8 = 40. Max stays 49.
  Both sides equal height (8 == 8). Typically, you can move either; our algorithm will move the right in this case (since `height[right] <= height[left]` triggers right--).
- Now: `left=1` (`8`), `right=5` (`height[5]=4`).
  Width = 4, minHeight = min(8,4) = 4, area = 4 * 4 = 16. Max still 49.
  Right side shorter, move `right`.

- `left=1` (`8`), `right=4` (`height[4]=5`).
  Width = 3, minHeight = min(8,5) = 5, area = 3 * 5 = 15. Right side shorter, move `right`.
- `left=1` (`8`), `right=3` (`height[3]=2`).
  Width = 2, minHeight = min(8,2) = 2, area = 2 * 2 = 4. Right side shorter, move `right`.
- `left=1` (`8`), `right=2` (`height[2]=6`).
  Width = 1, minHeight = min(8,6) = 6, area = 6 * 1 = 6. Right side shorter (6 < 8? Actually 6 is not less than 8, but let's say we move right if equal or less).
- `left=1`, `right=1` terminates loop.

Max area found was 49. Notice we didn't check every pair; we smartly navigated the space of pairs using the two pointers and the rule to move the shorter side.

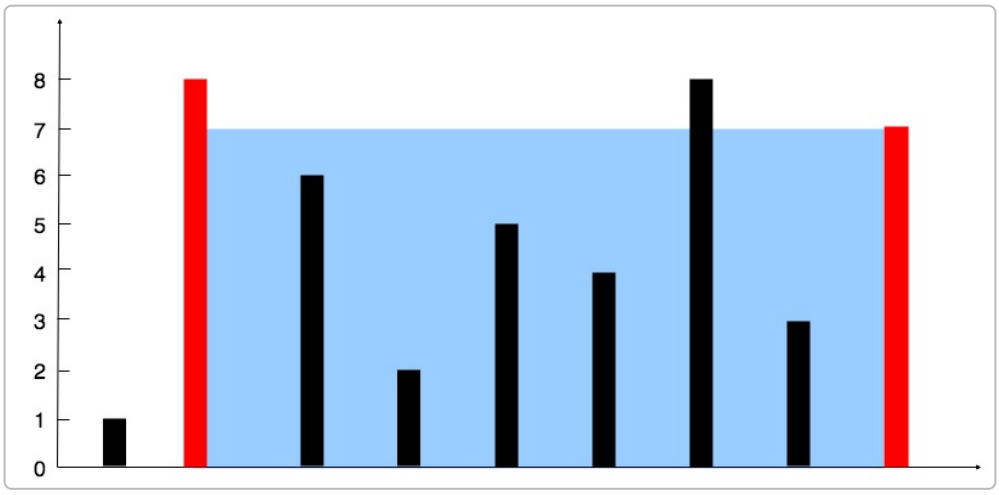

Figure: Example of the container problem with vertical lines. The maximum water (blue shaded area) is trapped between the line of height 8 at index 1 and the line of height 7 at index 8, yielding area 49.

In the above diagram, each vertical bar corresponds to an element of the array `[1,8,6,2,5,4,8,3,7]`. The optimal container is highlighted, demonstrating the widest distance (index 1 to 8) combined with a reasonably tall smaller side (height 7) [46]. Any other combination would either have a shorter width or a shorter min-height (or both), resulting in less area.

Why is this optimal? The two-pointer approach essentially narrows down the search by eliminating suboptimal pairs en route: - Starting with the widest pair ensures we consider maximum width initially. Then we gradually narrow the width. - By always moving the smaller height, we're giving ourselves a chance to find a taller line that might increase the limiting height, while only slightly reducing width [45]. - A more formal reasoning: assume the optimal solution is at indices (*i, j*). If our current left-right pair is (L, R), any pair between L and R has less width, so to beat the current area, it must have a significantly taller min-height. By moving the smaller side, we are searching for that taller height. If we moved the taller side, we'd only lose width without increasing height, potentially missing the optimal.

Common Pitfalls: - **Moving the wrong pointer:** A frequent mistake is to try moving the pointer at the taller line, which, as explained, does not help. Always move the pointer at the shorter line [44] (if equal, you can move either one step; it doesn't matter for correctness, though some implementations move both in case of equal heights). - **Calculating area correctly:** Use a long type if needed (in JS, number is fine). The width is

`right - left`, height is `Math.min(height[left], height[right])`. Simple, but mixing up left/right or using the wrong indices for height can happen under pressure. - **Edge cases:** If the array is very small (length 2), the answer is trivially the area between those two lines. Our loop handles that (it will compute once and exit). - **All lines of equal height:** E.g., `[5,5,5,5]`. The max area would be between the first and last (width 3 * height 5 = 15). The algorithm: - left=0, right=3, area=15. height[left] == height[right], say we move right. - left=0, right=2, area= min(5,5)*2 = 10. - right moves, left=0, right=1, area=5*1=5. - It finds 15 as max, which is correct. If instead left moved first or both moved, we'd still eventually find 15 as initial or later. - **Not updating max properly:** Always compare and store the max area at each step.

Full JavaScript Solution: Container With Most Water

```
/**
 * Find the maximum water container area formed by any two lines.
 * @param {number[]} height - array of non-negative integers
 * @return {number} max area
 */
function maxArea(height) {
  let left = 0;
  let right = height.length - 1;
  let maxArea = 0;

  while (left < right) {
    const width = right - left;
    const minHeight = height[left] < height[right] ? height[left] :
height[right];
    const area = width * minHeight;
    if (area > maxArea) {
      maxArea = area;
    }
    // Move the pointer at the shorter line inward
    if (height[left] < height[right]) {
      left++;
    } else {
      right--;
    }
  }

  return maxArea;
}
```

Time Complexity: O(n). We perform a single pass, with one pointer moving inward from each end. Each index is visited at most once by each pointer, so linear time overall [47] [48] .

Space Complexity: O(1) extra. Just a few variables for indices and area calculations.

There are no fancy data structures needed – this is an in-place calculation.

Alternate Approaches:
There's essentially no better approach than two-pointers for this problem. Brute force is O(n^2) and unacceptable for large n. Some people attempt divide-and-conquer or other strategies, but they end up similar to brute force or complicated without improving complexity. The two-pointer greedy is the optimal solution pattern here and is one of the reasons this problem is famous (it showcases a non-intuitive greedy step that you kind of have to reason out).

However, as a thought exercise: - One could imagine starting with the global highest bar as one side, and then scanning for the best partner to that bar (either leftward or rightward) and then possibly another highest bar, etc. But this still ends up being O(n^2) in worst case if done naively. - The two-pointer method cleverly eliminates a large number of possibilities with each move (because when we move left++ after comparing left vs right, we effectively discard all pairs that included the old left with any other right, as they can't be optimal) [45] .

Gotchas and Insights: - A common interview discussion: *"Why can we move the shorter pointer? Could we be eliminating the global optimal pair by doing that?"* You should be prepared to explain or at least intuit that we're not missing the optimal because any optimal that involved the shorter line would have been considered and any other pair with that shorter line would only be worse (due to even less width) [45] . This is a critical insight for convincing the interviewer you understand the greedy choice. - If heights can be zero, the algorithm still works (zero height just means area 0 with that line). If one side is zero, we'll move that side inward for sure, seeking a taller line. - This problem doesn't require outputting which indices achieve the max, but just the area. If you needed the indices, you could easily store them when you find a new max. - The container problem is a good example of the **"Opposite Ends" two-pointer pattern**: whenever you have to pick two elements from an array to optimize some function and the array is involved in that function (here length between indices and min of values), consider a two-pointer approach.

---

5. Trapping Rain Water

Problem: Given an array representing heights of bars, imagine water pouring on top. How much water gets trapped between the bars? Each element of the array is a non-negative integer, and water at a position i is trapped by the taller bars on its left and right. We need to compute total trapped rainwater volume (in units) [49] [50] .

For example, for heights `[3, 0, 1, 0, 4, 0, 2]`, the water trapped is 10 units (as water fills the dips between bars) [51] .

This is a classic problem that can be solved with dynamic programming or stacks, but also optimally with two pointers.

Brute-Force Approach (Compute Water Above Each Bar)

The simplest way to think of it: For each index `i`, find: - `leftMax[i]`: the highest bar to its left (including itself). - `rightMax[i]`: the highest bar to its right (including itself). - The water that can be trapped on top of bar i is `min(leftMax[i], rightMax[i]) - height[i]`, if that quantity is positive (otherwise 0) [50] .

Brutishly, to get leftMax and rightMax for each i, you could scan left and scan right for each i (two inner loops): - For each i, scan leftwards to find the tallest bar on the left. - Scan rightwards to find the tallest bar on the right [52] [53] . - Then compute water at i.

This results in O(n^2) time in the worst case (for each of n bars, scanning up to n in worst case).

Better Approach (Precompute with DP arrays)

Before jumping to two pointers, it's worth noting a common improvement: We can precompute the `leftMax` array and `rightMax` array in O(n) time each: - `leftMax[i]` = max(height[0..i]) (highest bar from the left up to i). This can be built in one pass from left to right. - `rightMax[i]` = max(height[i..end]) (highest bar from the right up to i). This can be built with one pass from right to left [54] .

Then water at i = min(leftMax[i], rightMax[i]) - height[i] if positive [50] . Sum all those up for the answer.

This yields O(n) time and O(n) extra space (for the two arrays). It's a good approach, and easier to reason about. However, there's an even more space-optimized method using two pointers: We can achieve the result in O(n) time and O(1) extra space by essentially doing those two passes simultaneously with two pointers.

Optimal Two-Pointer Solution (Using LeftMax and RightMax on the Fly)

The two-pointer approach for trapping rain water is a bit tricky but elegant: - Use two pointers: `left = 0` at start, `right = n-1` at end. - Maintain two variables `leftMax` and `rightMax` to track the highest wall seen so far from the left side and right side, respectively [55] [56] . - While `left < right`: - We look at the shorter side (similar strategy to container, but now we use the side's max height): - If `height[left] <= height[right]`: - This means the right side is taller (or equal). So the water level on the left side is bounded by `leftMax` (from left side) and something on right. Actually, if left side is shorter, the water trapped at `left` is determined by the left side's highest boundary. - We know `rightMax` is at least `height[right]` (there is a tall boundary on right) which is >= height[left]. So the limiting factor for water at `left` is `leftMax` (if we have seen a taller bar on left already) or the current right side – whichever is smaller. - In this case, ensure `leftMax` is updated: `leftMax = max(leftMax, height[left])`. - Then move `left` pointer to the right by 1. - Before moving, calculate water at the current `left` index: - If `leftMax` is higher than the current height, water can be trapped = `leftMax - height[left]` (because we know there's a taller bar on the right, guaranteeing a boundary) [57] [58] . - Add that to result. - Else (height[right] < height[left]): - Symmetrically, the right side is shorter. So we consider the right index: water there is limited by `rightMax` (tallest to its right) and the fact that left side is taller means there is a boundary on left. - Update `rightMax = max(rightMax, height[right])`. - Calculate water at `right`: `rightMax - height[right]` (if positive) [59] [60] . - Move `right` pointer left by 1. - Continue until left meets right.

In essence, we always process the lower side because that side's water trapping is limited by its own side's max boundary. The higher side will serve as a boundary for the lower side. We move inward from the lower side, updating boundaries as we go [59] [60] .

Another intuition: water trapped at a position is determined by the *min(leftMax, rightMax)* [50]. The two-pointer method keeps track of these effectively. When `height[left] < height[right]`, we know `leftMax < rightMax` (not always strictly, but let's think conceptually) [59]. So the water at `left` is limited by `leftMax`. When `height[right] < height[left]`, then `rightMax <= leftMax`, and the water at `right` is limited by `rightMax`. By processing the smaller side first, we ensure we have both a left boundary and right boundary in place.

Dry Run Example:
Take `height = [3, 0, 1, 0, 4, 0, 2]`. We know the answer is 10 (let's verify through steps):

- Initialize: `left=0, right=6` (pointing to heights 3 and 2). `leftMax = 0, rightMax = 0`. `water = 0`.
- Compare `height[left]=3` vs `height[right]=2`. Right is smaller, so we'll process the `right` side.
- Update `rightMax = max(0, height[6]=2) = 2`.
- Water at index 6 = `rightMax - height[6] = 2 - 2 = 0`. (No water on top of the last bar, obviously, since it's at right boundary).
- Move `right--` to 5.
- Now `left=0 (h=3)`, `right=5 (h=0)`. Compare 3 vs 0, right is smaller.
- Update `rightMax = max(2, height[5]=0) = 2`. (rightMax stays 2).
- Water at index 5 = `rightMax - height[5] = 2 - 0 = 2`. We add 2 units of water (a puddle on top of index 5 bar).
- Move `right--` to 4.
- Now `left=0 (3)`, `right=4 (h=4)`. Compare 3 vs 4, left is smaller this time.
- Update `leftMax = max(0, height[0]=3) = 3`.
- Water at index 0 = `leftMax - height[0] = 3 - 3 = 0`. (No water on the leftmost bar edge).
- Move `left++` to 1.
- Now `left=1 (h=0)`, `right=4 (h=4)`. Compare 0 vs 4, left is smaller.
- Update `leftMax = max(3, height[1]=0) = 3`. (leftMax remains 3, which is the bar at index 0).
- Water at index 1 = `leftMax - height[1] = 3 - 0 = 3`. We trap 3 units at index 1.
- Move `left++` to 2.
- Now `left=2 (h=1)`, `right=4 (h=4)`. Compare 1 vs 4, left is smaller.
- Update `leftMax = max(3, height[2]=1) = 3`.
- Water at index 2 = `leftMax - height[2] = 3 - 1 = 2`. (Between the left boundary of height 3 and some right boundary, index 2 can hold 2 units).
- Move `left++` to 3.
- Now `left=3 (h=0)`, `right=4 (h=4)`. Compare 0 vs 4, left smaller.
- Update `leftMax = max(3, height[3]=0) = 3`.
- Water at index 3 = `leftMax - height[3] = 3 - 0 = 3`. Trap 3 units at index 3.
- Move `left++` to 4.
- Now `left=4, right=4`, loop ends.

Summing trapped water: index5:2, index1:3, index2:2, index3:3. Total = 2+3+2+3 = 10, which matches the expected result [61].

We effectively did what the DP approach does (min of leftMax and rightMax at each position) but in one pass with constant space by using the two-pointer technique.

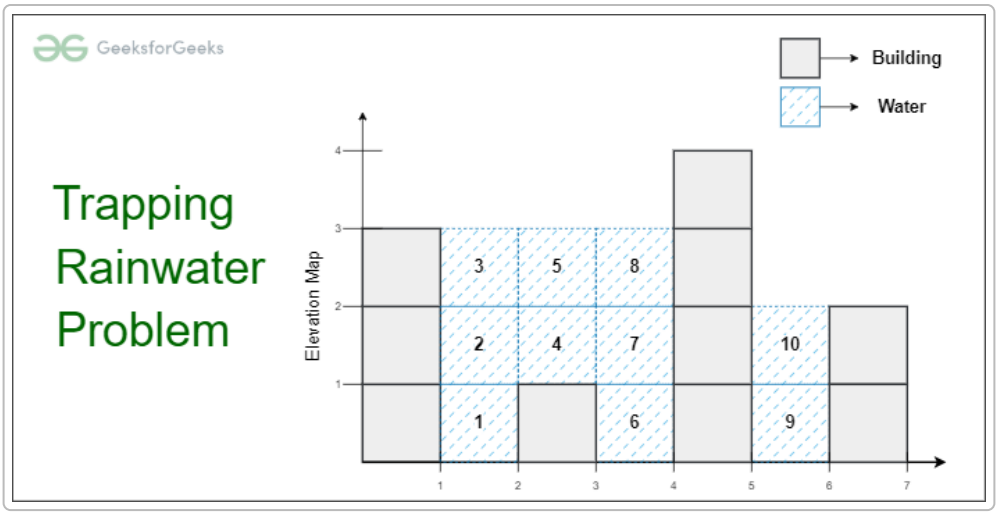

Diagram: Visualization of trapping rain water. The blue sections represent trapped water above bars. The water at any position is limited by the shorter of the tallest bars on its left and right [50] *. Using two pointers (one from left, one from right), we accumulate water in one pass by always filling from the lower side inward.*

In the above image, you can see how each "valley" between taller bars accumulates water. The numbers on top of water blocks often indicate how many units of water are above each bar. The two-pointer method essentially calculates these numbers by maintaining the current leftMax and rightMax as it sweeps inward.

Common Pitfalls: - **Mixing up leftMax/rightMax logic:** This approach can be confusing. Some forget to update the max *before* calculating water or vice versa. Notice in our pseudocode above, we updated `leftMax` / `rightMax` first, then computed water = (currentMax - height). Another valid approach is to update after computing water depending on how you set it up. The important part is that when, say, `height[left] < height[right]`, we know leftMax is the limiting factor. If `leftMax` is greater than the current height, then `leftMax - height[left]` is water. If not, then `leftMax` becomes that height (no water on that bar). In code, it's often written as:

```
if(height[left] < height[right]) {
    if(height[left] >= leftMax) leftMax = height[left];
    else water += (leftMax - height[left]);
    left++;
} else { ... symmetric for right ... }
```

Make sure the logic is correctly capturing the min(maxLeft, maxRight). - **Off-by-one in loop:** Typically use `while(left < right)` because when they meet, that position is either a peak or a valley but it doesn't matter — you could include equality but then you'd just process the same bar from one side or the other with zero water. - **All increasing or all decreasing height:** If the array is strictly increasing or strictly decreasing, no water will be trapped. The algorithm should handle that: one side pointer will move all the way without adding any water because always update max and height is never below current max (for strictly monotonic). E.g., `[1,2,3,4]`: - leftMax goes updating and rightMax starts as 4 at end. The pointer that moves will be left (since left height < right height each time), and water will be 0 at each because leftMax equals the height of left each time. Ends with 0 total, which is correct. - **Multiple highest bars:** If

the highest bar appears multiple times, water can accumulate in between. Our method handles it – basically leftMax and rightMax will meet at the global max points. - **Negative heights?** Not in this problem – heights are non-negative. If negative were allowed conceptually, it breaks the trapping model (and physically doesn't make sense), so not applicable.

Full JavaScript Solution: Trapping Rain Water

```
/**
 * Calculate total trapped rainwater given bar heights.
 * @param {number[]} height
 * @return {number} total trapped water
 */
function trap(height) {
  let left = 0;
  let right = height.length - 1;
  let leftMax = 0;
  let rightMax = 0;
  let totalWater = 0;

  while (left < right) {
    if (height[left] <= height[right]) {
      // Process left side
      if (height[left] >= leftMax) {
        // Update leftMax if current bar is taller
        leftMax = height[left];
      } else {
        // Current bar is lower than the leftMax, so water can accumulate
        totalWater += (leftMax - height[left]);
      }
      left++;
    } else {
      // Process right side (height[right] < height[left])
      if (height[right] >= rightMax) {
        rightMax = height[right];
      } else {
        totalWater += (rightMax - height[right]);
      }
      right--;
    }
  }

  return totalWater;
}
```

Time Complexity: O(n). We perform a single pass with two pointers, each moving n steps in total at most [62] [63]. This is linear and optimal for this problem.

Space Complexity: O(1) extra. We only use a few variables (`leftMax`, `rightMax`, pointers, total). This is better than the DP array approach which uses O(n) extra space for the arrays.

Alternate Approaches:
- The **dynamic programming approach** with precomputed `leftMax[]` and `rightMax[]` is a good alternative to know. It's easier to implement correctly on the first try and also O(n) time (but O(n) space) [54]. If an interviewer isn't specifically looking for the two-pointer trick, the DP approach is perfectly fine. For completeness: - Compute `leftMax[i]` for i from 0 to n-1. - Compute `rightMax[i]` for i from n-1 down to 0. - Then loop through i and add `min(leftMax[i], rightMax[i]) - height[i]` if positive. - The **monotonic stack** approach: Another technique uses a stack to keep track of indices of bars and calculates water when a bar "fills" a gap between taller bars. It's more complex but also O(n) time, O(n) space. Typically, one would push indices onto a stack until a taller bar comes that traps water with a previous bar, etc. While neat, it's not necessary if you know the two-pointer or DP solutions. - *Brute force* as described is O(n^2) and not feasible for large n (n can be 100k in some versions of this problem).

Gotchas & Insights: - Understand that **water at position i = min(max height to left, max height to right) - height[i]** [50]. This fundamental formula must hold in any method. The two-pointer method is essentially dynamically figuring out that min of maxes by maintaining `leftMax` and `rightMax`. - The two-pointer approach might feel similar to the container problem approach but it's doing a different calculation. A big difference is we are **adding up water at every index**, not just finding two indices that maximize something. But the technique of moving the lower side inward is analogous – here, whichever side is lower dictates the water trapping on that side. - When implementing under pressure, many choose the DP arrays because it's straightforward. But the two-pointer is worth practicing because it's elegant and shows you can optimize space. In some interviews, after giving the O(n) time + O(n) space solution, they might follow up asking if you can reduce space, leading to this approach. - Edge cases: if the array length is less than 3, result is obviously 0 (since at least 3 bars are needed to trap any water in between). Our loop naturally handles length 0,1,2 because left<right will be false or only one iteration with no addition. - Another edge: flat areas like `[0,0,0]` just return 0, which we'd get because leftMax stays 0 and no water added. - Very tall at edges and short in middle, e.g. `[100, 0, 100]` yields min(100,100)-0 = 100 at the middle. Our method: - leftMax updates to 100 at index0, then as left moves, we add 100-0 for index1. - rightMax updates to 100 at end, etc. Yes, result 100.

Memorization Tips & Key Takeaways

Mastering these patterns is easier when you distill the key ideas and practice them. Here are some flashcard-style pointers and mnemonics to help memorize the two-pointer techniques for these problems:

- **Valid Palindrome – "Two Ends, Skip Non-Letters":** Remember to use two pointers from each end, and *skip* characters that aren't letters or digits [7]. **Rule:** Move inward comparing case-insensitively. *(Flashcard: Q: How to handle spaces/punctuation in palindrome check? A: Skip them using two-pointer iteration.)* Also, think *"normalize and compare"*.

- **Two Sum II (Sorted) – "Left++ or Right--":** In a sorted array, use two pointers at ends. **If sum is too low, move left pointer right**; **if sum too high, move right pointer left** [18]. This is a fundamental pattern for any two-sum in sorted data. *Mnemonic:* **"Two Sum Sorted: Two Pointers Shorten**

Range" (shrink the window from either end based on sum). Always remember to return indices +1 for this problem [27].

- **3Sum – "Sort and Two-Sum": Sort first** (critical!). Then for each number, do two-sum with two pointers [31]. Key invariants: skip duplicate `i` values [37]; after finding a triplet, skip duplicate `left` and `right` values [34]. *Flashcard: What's the approach for 3Sum?* **Sorted + loop i + two-pointer (left, right)**. Keep in mind 3Sum's complexity is O(n^2) – that's expected.

- **Container With Most Water – "Move Shorter Wall":** Start with the widest container and move the side with smaller height inward [44]. This maximizes our chances to find a taller wall to increase area. *Mantra:* **"Short side moves in."** You can even memorize the rationale: *Moving the taller side can't help, so move the shorter.* Also recall the formula `area = width * min(h_left, h_right)` which guides this movement [43].

- **Trapping Rain Water – "Two Pointers for Boundaries":** Think in terms of leftMax and rightMax boundaries. **Rule:** Move the pointer on the smaller height side inward, accumulating water as you go [59] [60]. *Mnemonic:* **"Fill from the low side."** At each step, if left side is lower, water on that side = leftMax - height[left]; if right side is lower, water = rightMax - height[right]. Update the max boundaries as you move. Alternatively, remember the DP formula: *water[i] = min(maxLeft[i], maxRight[i]) - height[i]* [50], and that two-pointer is a way to get that without extra space.

- **General Two-Pointer Patterns:** Many problems use one of two patterns:

- **Opposite Ends** (like Two Sum II, Container, Valid Palindrome) – pointers start at both ends and move towards center based on conditions.
- **Sliding Window / Adjacent Pointers** (not covered in this question, but e.g. Longest Substring Without Repeat uses two pointers moving in one direction).

For the ones we did: - Opposite-ends pointers leverage sorted order or a criteria to decide which pointer to move. - If array is sorted and you seek a target sum or pair condition → think opposite-ends (Two Sum II, 3Sum's inner loop, etc.). - If problem involves maximizing/minimizing with pairs from ends (Container, or checking symmetry as in Palindrome) → opposite-ends.

- **When to Sort:** If the input isn't sorted but you need two-sum or multi-sum combinations (like 3Sum), you **almost always sort first** to use two-pointer efficiently [31]. Sorting is O(n log n) which is usually fine for n up to 10^5. Remember, for 3Sum and beyond, sorting is your friend for avoiding duplicates and simplifying logic.

- **Avoiding Duplicates:** For problems like 3Sum (and 4Sum, etc.), memorize the idiom:

```
if (i > 0 && nums[i] == nums[i-1]) continue;  // skip duplicate first
element
...
while(left < right && nums[left] == nums[left-1]) left++;   // skip dupes
after finding a valid triple
```

```
while(left < right && nums[right] == nums[right+1]) right--; // skip dupes
on right side
```

Skipping duplicates is crucial for correctness [64] [34].

- **Two-pointer vs Other Approaches:** It helps to know alternatives:

- For Two Sum II, hash map could solve unsorted two-sum but uses extra space.
- For 3Sum, a hash set approach exists but is trickier and often slower in practice.
- For Trapping Rain Water, the DP arrays or stack can be alternatives. But if asked for O(1) space, two-pointer is the way.

- Recognize when a problem is a variant of these classics – interviewers often tweak these patterns.

- **Flashcard Q&A Recap:**

- Q: *"What's the pointer movement rule for the container (most water) problem?"*
  A: *Always move the pointer at the shorter line inward* [44].
- Q: *"How do we find triplets summing to zero efficiently?"*
  A: *Sort the array, then use a loop for the first number and two-pointer for the other two (avoiding duplicates)* [31].
- Q: *"How to compute trapped water at a given index?"*
  A: *It's* `min(max height to left, max height to right) - height[index]` [50]. *Two-pointer method maintains those maxima dynamically.*
- Q: *"Why does two-pointer work for Two Sum II?"*
  A: *Because the array is sorted; moving pointers inward adjusts the sum in a predictable way to find the target* [18].
- Q: *"Do I always need two pointers from opposite ends for two-pointer technique?"*
  A: *Not always; sometimes two pointers start together or at fixed interval (like slow/fast pointer problems), but for these typical problems, yes, opposite ends or one fixed & one scanning (3Sum inner loop) are common patterns.*

Practice these patterns: Write out the two-pointer strategy in plain English for each problem: - *Palindrome:* "Set one pointer at start, one at end, move towards middle skipping non-alnum and comparing." - *Two Sum II:* "Start at both ends of sorted array, adjust pointers inward to reach target sum." - *3Sum:* "Sort array, loop i, then two-pointer (left=i+1, right=end) to find complements for -nums[i]." - *Container:* "Start at ends (max width), compute area, move the side with smaller height inward to try for larger height." - *Trapping Water:* "Use two pointers to maintain leftMax and rightMax; move the lower side inward, adding water when a pointer sees a height lower than the known max on that side."

Having these narratives in your head (or on flashcards) will help you quickly recall the approach during an interview. Remember, the more problems you solve with two pointers, the more intuitive these movements will become. Good luck, and happy pointer-ing!

---

1 2 3 4 5 7 125. Valid Palindrome - In-Depth Explanation
https://algo.monster/liteproblems/125

6 8 9 10 11 12 Valid Palindrome | Approach 2 Two Pointers - NamasteDev Blogs
https://namastedev.com/blog/valid-palindrome-approach-2-two-pointers/

13 14 15 17 20 24 25 27 28 29 Efficiently Solving Two Sum II - Input Array Is Sorted - DEV Community
https://dev.to/abhivyaktii/efficiently-solving-two-sum-ii-input-array-is-sorted-323k

16 18 19 21 26 Two Sum - Pair sum in sorted array - GeeksforGeeks
https://www.geeksforgeeks.org/dsa/pair-with-given-sum-in-sorted-array-two-sum-ii/

22 23 Two Sum II - Input Array Is Sorted — Omniverse
https://www.gaohongnan.com/dsa/two_pointers/questions/two_pointers/167-two-sum-ii-input-array-is-sorted.html

30 3-Sum | Hello Interview
https://www.hellointerview.com/learn/code/two-pointers/3-sum

31 32 33 34 35 36 37 38 39 40 64 3sum Solution - Top Interview Questions and Tricks #1 - 3sum and the 2 pointer technique | FizzBuzzed
https://fizzbuzzed.com/top-interview-questions-1/

41 42 43 44 45 47 48 Container with Most Water - GeeksforGeeks
https://www.geeksforgeeks.org/dsa/container-with-most-water/

46 LeetCode #11: Container With Most Water — Solved in Java
https://alexanderobregon.substack.com/p/leetcode-11-container-with-most-water

49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 Trapping Rain Water Problem - GeeksforGeeks
https://www.geeksforgeeks.org/dsa/trapping-rain-water/