**Leetcode Problems & Solutions with Explanation**

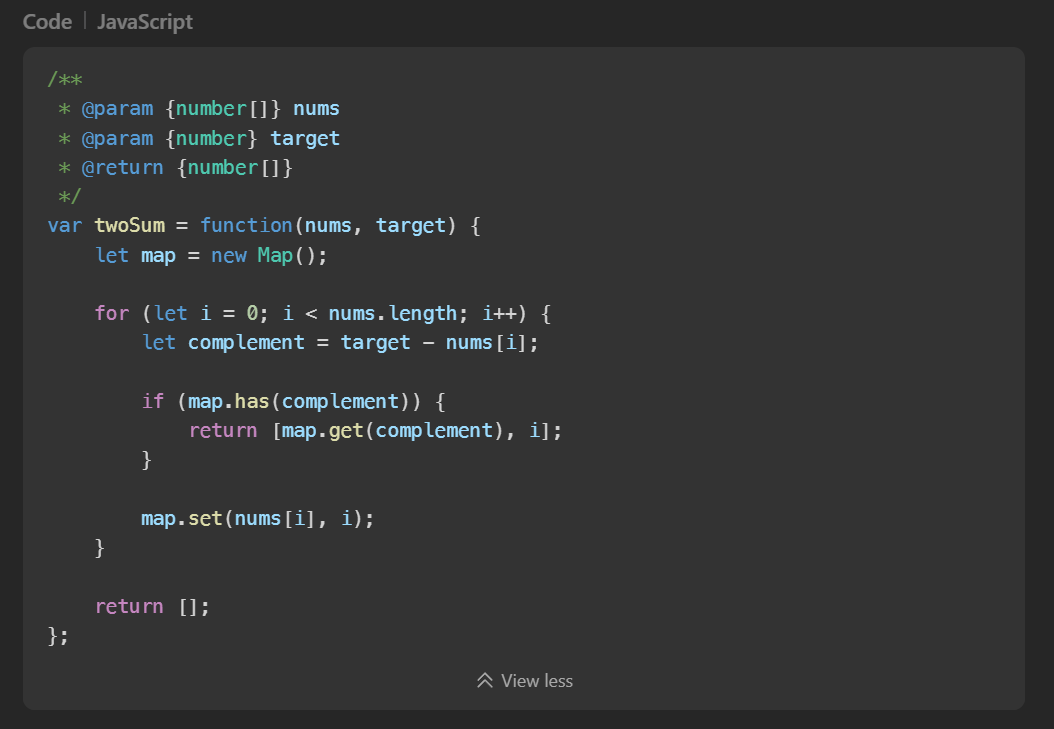
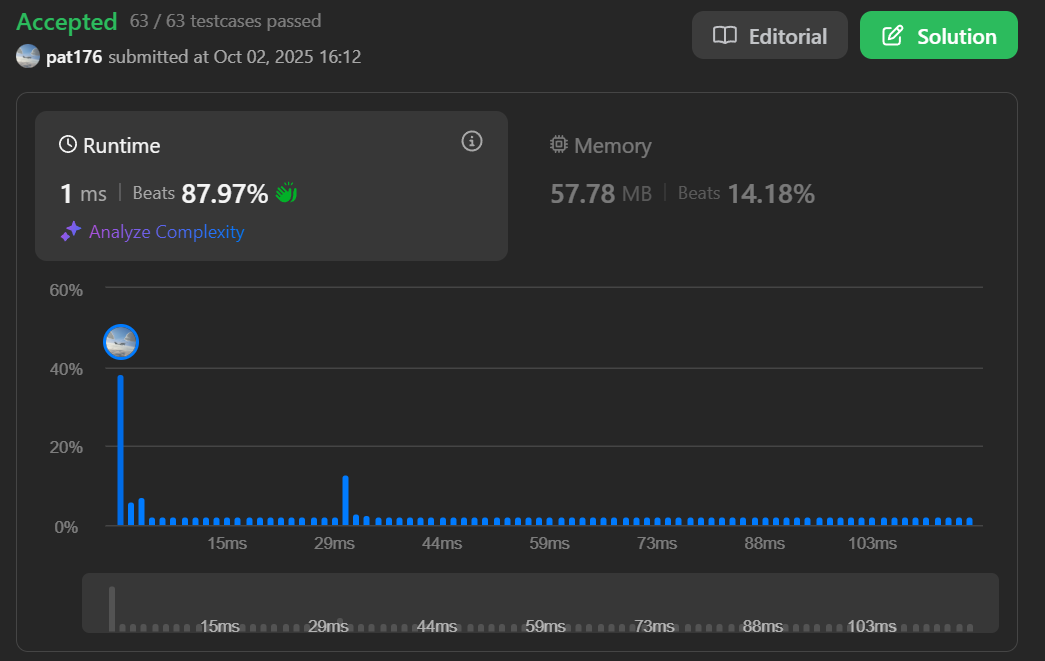
1. **Two Sum**

- Solution Link: [<https://leetcode.com/problems/two-sum/submissions/1789103212>]

- Time Complexity: O(n)

- Space Complexity: O(n)

- Explanation: Used a hash map to store values and indices for constant-time lookups.



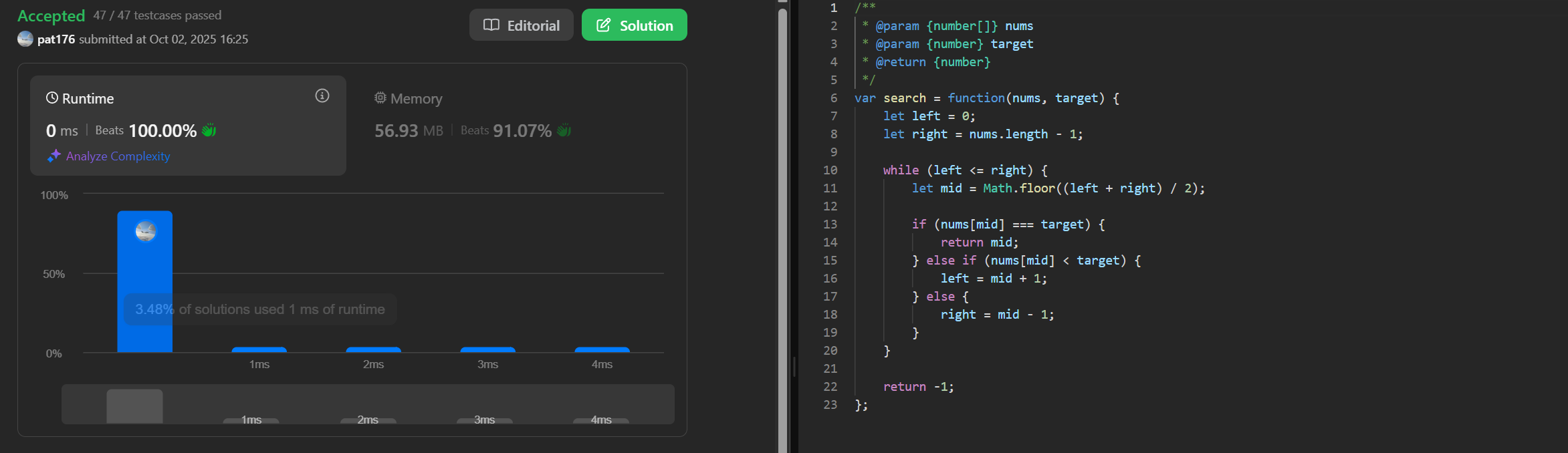
1. **Binary Search (704)**

- Solution Link: [<https://leetcode.com/problems/binary-search/submissions/1789112460>]

- Time Complexity: O(n)

- Space Complexity: O(n)

- Explanation: Classic Binary Search Algorithm



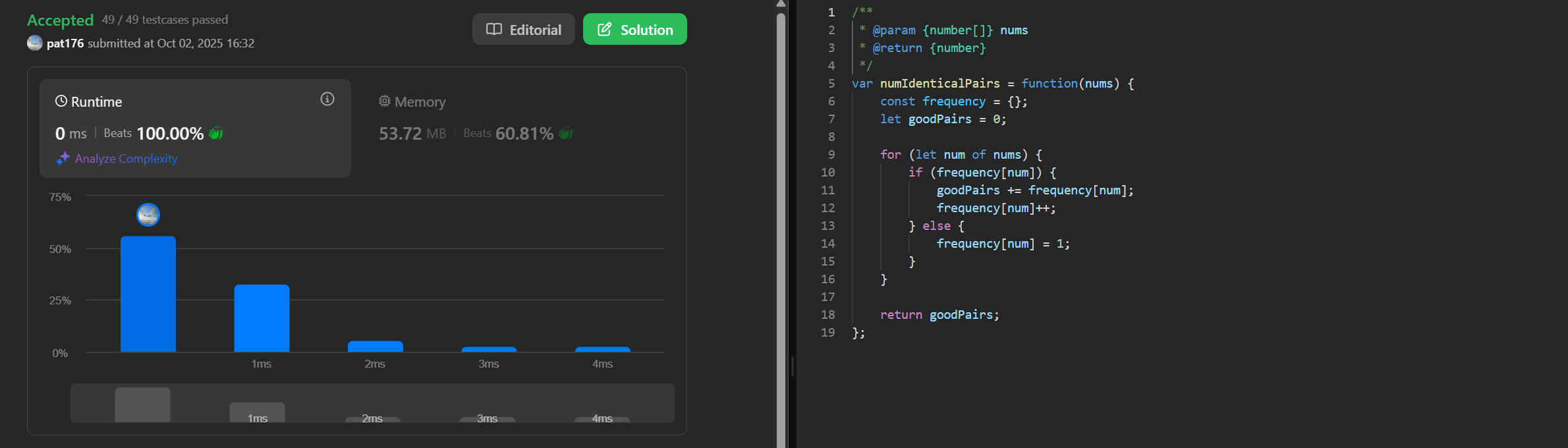
1. **Process**:
   * Start with left = 0 and right = nums.length - 1
   * Calculate mid index
   * If nums[mid] equals target, return mid
   * If target is greater, search right half (left = mid + 1)
   * If target is smaller, search left half (right = mid - 1)
   * Continue until left > right (target not found)
2. **Number of Good Pairs (1512)**

- Solution Link: [<https://leetcode.com/problems/number-of-good-pairs/submissions/1789117904>]

- Time Complexity: O(n)

- Space Complexity: O(n)

1. Explanation: Approach: Use frequency counting with combinatorial mathematics
2. Key Insight:
   * If a number appears k times, it can form k\*(k-1)/2 good pairs
   * Instead of calculating at the end, we **accumulate as we go**:
     + When we see a number for the n-th time, it can pair with all previous (n-1) occurrences



1. **Shuffle the Array (1470)**

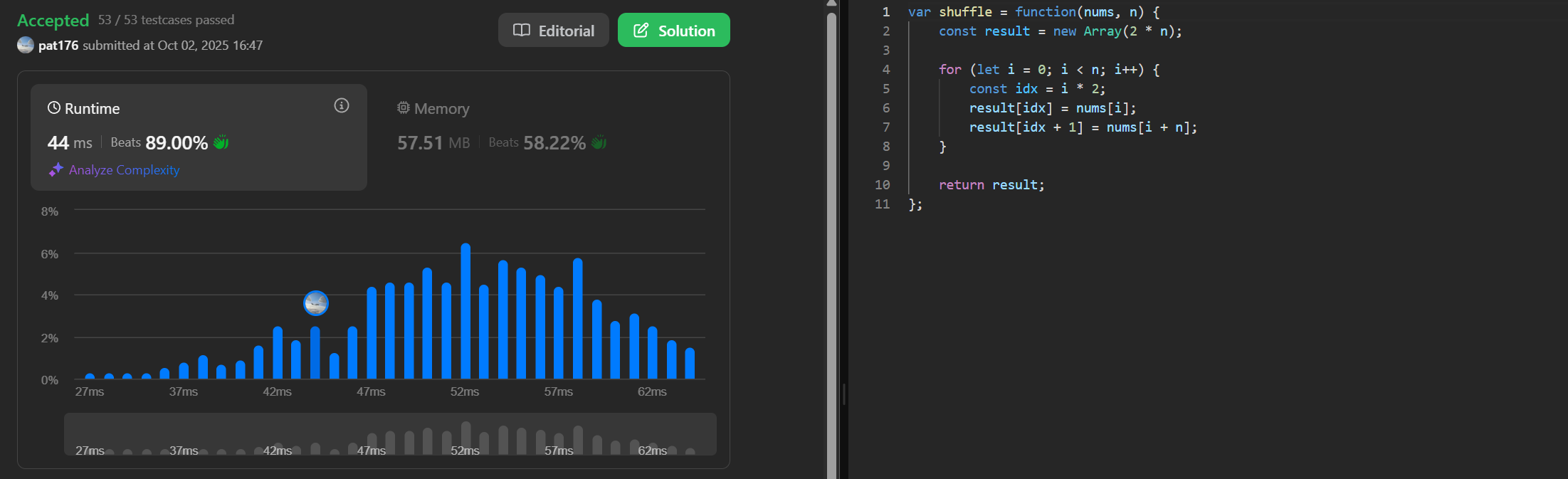
- Solution Link: [<https://leetcode.com/problems/shuffle-the-array/submissions/1789128248>]

- Time Complexity: O(n)

- Space Complexity: O(n) – 2n

- Explanation:

1. Approach: Use two pointers - one for x-elements (0 to n-1) and one for y-elements (n to 2n-1)



1. Process:
   * For each index i from 0 to n-1:
     + Add nums[i] (x-element)
     + Add nums[i + n] (y-element)
   * This creates the pattern: x1, y1, x2, y2, ..., xn, yn
2. **Count Number of Pairs with Absolute Difference K (2006)**

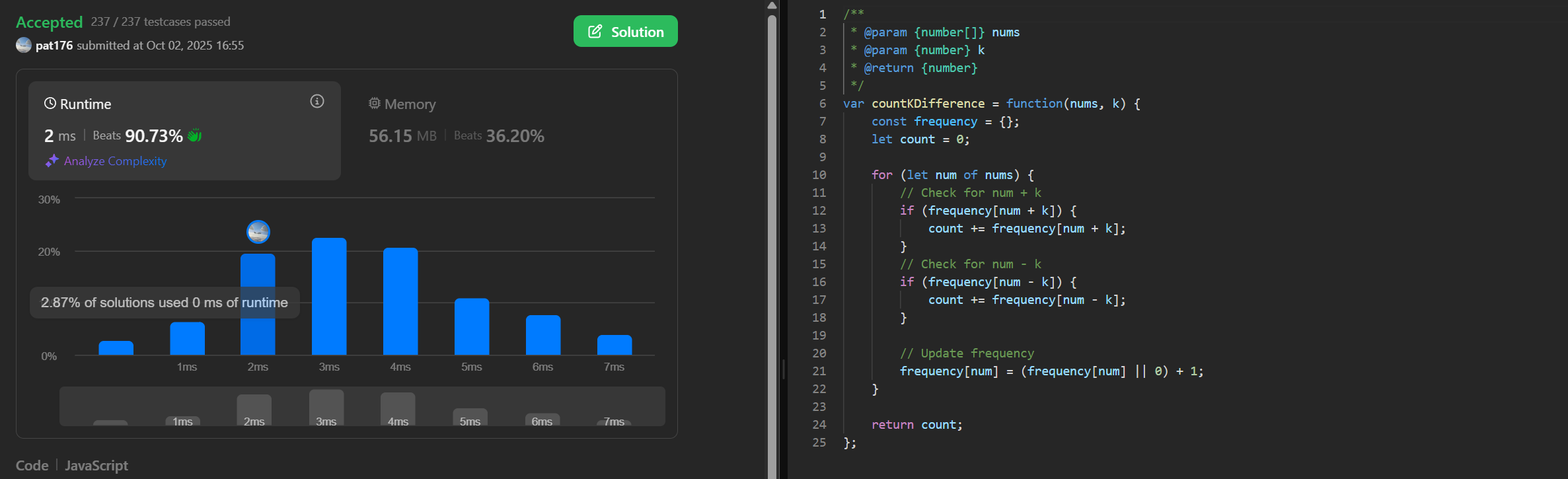
- Solution Link: [<https://leetcode.com/problems/count-number-of-pairs-with-absolute-difference-k/submissions/1789133888>]

- Time Complexity: O(n), map operation is O(1)

- Space Complexity: O(n)

- Explanation: Approach: Use frequency map to count occurrences

1. Key Insight:
   * For each number num, we look for:
     + num + k (if num is the smaller number in the pair)
     + num - k (if num is the larger number in the pair)
   * We accumulate pairs as we process each element
   * Then we update the frequency of the current number



1. **Maximum Subarray (53)**

- Solution Link: [<https://leetcode.com/problems/maximum-subarray/submissions/1789136390>]

Time Complexity: O(n)

* Single pass through the array
* Constant time operations per element

Space Complexity: O(1)

* Only using a few variables, no extra data structures

- Explanation **(Kadane's Algorithm):**

1. **Key Insight**: At each position, decide whether to:
   * Extend the previous subarray, OR
   * Start fresh from the current element
2. **Process**:
   * currentSum tracks the maximum sum ending at the current position
   * maxSum tracks the global maximum found so far
   * For each element: currentSum = max(nums[i], currentSum + nums[i])
   * Update maxSum if currentSum is larger

