DSA-1 Project

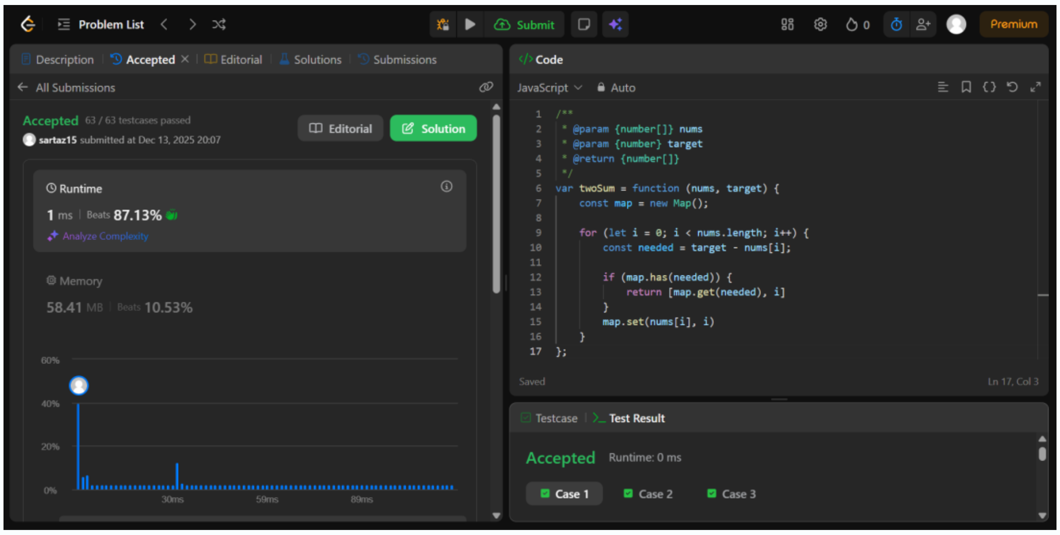
Question: Leetcode 1

Two Sum

https://leetcode.com/problems/two-sum/description/

Answer Link

https://leetcode.com/problems/two-sum/submissions/1854574291/



Accepted 63 / 63 testcases passed
sartaz15 submitted at Dec 13, 2025 20:07

Runtime
1 ms | Beats 87.13%

Memory
58.41 MB | Beats 10.53%

```javascript
/**
 * @param {number[]} nums
 * @param {number} target
 * @return {number[]}
 */
var twoSum = function (nums, target) {
    const map = new Map();

    for (let i = 0; i < nums.length; i++) {
        const needed = target - nums[i];

        if (map.has(needed)) {
            return [map.get(needed), i]
        }
        map.set(nums[i], i)
    }
};
```

Accepted Runtime: 0 ms

Time Complexity: O(n)

The algorithm traverses the array only once using a single loop. For each element, hash map lookup and insertion operations take constant time O(1). Since all n elements are processed once, the overall time complexity is O(n).

Space Complexity: O(n)

An extra hash map is used to store numbers and their indices. In the worst case, all n elements are stored in the map, resulting in O(n) additional space usage.

Detailed Description (Algorithm and Approach)

The objective is to find two indices such that the sum of their corresponding values equals the given target.

1. Initialize an empty hash map to store array elements and their indices.
2. Traverse the array from left to right.
3. For each element, calculate the required value needed to reach the target.
4. Check if this required value already exists in the map.
5. If it exists, return the stored index and the current index.
6. If it does not exist, store the current element and its index in the map.
7. Continue the process until the valid pair is found.

This approach avoids nested loops and reduces the time complexity from O(n^2) to O(n) using efficient hash map operations.

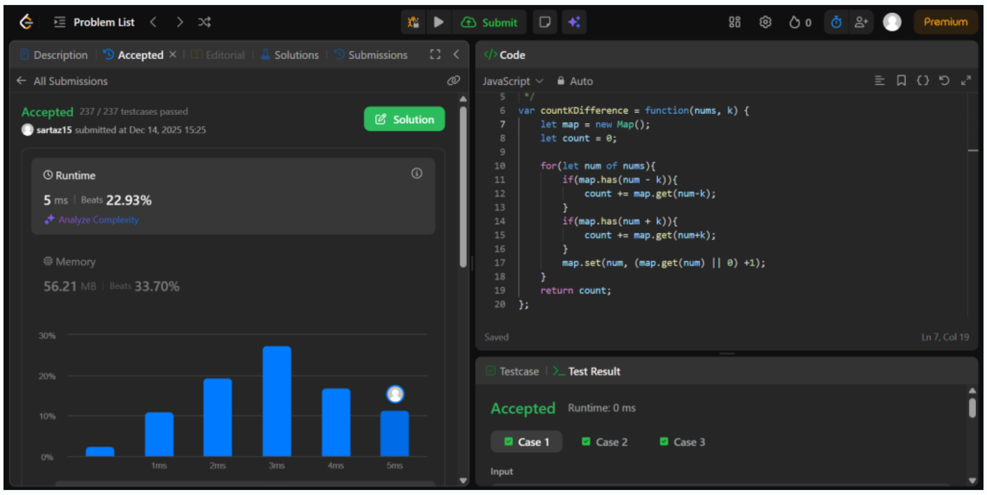
Question: Leetcode 2006

Count Number of Pairs with Absolute Difference K

https://leetcode.com/problems/count-number-of-pairs-with-absolute-difference-k/description/

Answer Link

https://leetcode.com/problems/count-number-of-pairs-with-absolute-difference-k/submissions/1855268331/



Accepted 237 / 237 testcases passed
sartaz15 submitted at Dec 14, 2025 15:25

Runtime
5 ms | Beats 22.93%

Memory
56.21 MB | Beats 33.70%

```javascript
 */
var countKDifference = function(nums, k) {
    let map = new Map();
    let count = 0;

    for(let num of nums){
        if(map.has(num - k)){
            count += map.get(num-k);
        }
        if(map.has(num + k)){
            count += map.get(num+k);
        }
        map.set(num, (map.get(num) || 0) +1);
    }
    return count;
};
```

Saved
Ln 7, Col 19

Accepted Runtime: 0 ms

Time Complexity: O(n)

The algorithm iterates through the array once using a single loop. For each element, constant time hash map operations such as lookup and update are performed. Since every element is processed once, the total time complexity is O(n).

Space Complexity: O(n)

A hash map is used to store the frequency of elements encountered during traversal. In the worst case, all (n) elements may be stored in the map, leading to O(n) additional space usage.

Detailed Description (Algorithm and Approach)

The task is to count the number of index pairs (i, j) such that i < j and the absolute difference between nums[i] and nums[j] equals k.

1. Initialize an empty hash map to store frequencies of elements.

2. Initialize a counter to store the number of valid pairs.

3. Traverse the array from left to right.

4. For the current element, check if (current − k) exists in the map:

   - If it exists, add its frequency to the counter.

5. Check if (current + k) exists in the map:

   - If it exists, add its frequency to the counter.

6. Update the frequency of the current element in the map.

7. Continue until all elements are processed.

8. Return to the final count.

This optimized approach avoids nested loops and improves the time complexity from $O(n^2)$ to O(n) by using hash map-based frequency counting.

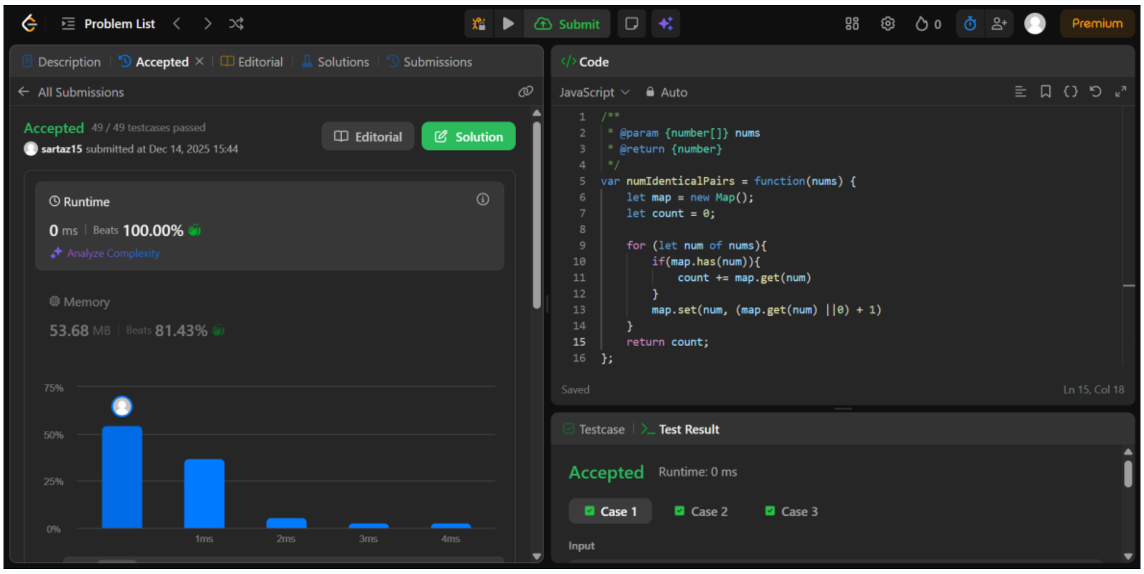
Question: Leetcode 1512

Number of Good Pairs

https://leetcode.com/problems/number-of-good-pairs/description/

Answer Link

https://leetcode.com/problems/number-of-good-pairs/submissions/1855279986/



Accepted 49 / 49 testcases passed
sartaz15 submitted at Dec 14, 2025 15:44

Runtime
0 ms | Beats 100.00%

Memory
53.68 MB | Beats 81.43%

```javascript
/**
 * @param {number[]} nums
 * @return {number}
 */
var numIdenticalPairs = function(nums) {
    let map = new Map();
    let count = 0;

    for (let num of nums){
        if(map.has(num)){
            count += map.get(num)
        }
        map.set(num, (map.get(num) ||0) + 1)
    }
    return count;
};
```

Saved
Ln 15, Col 18

Accepted Runtime: 0 ms

Time Complexity: O(n)

The algorithm traverses the array once. For each element, hash map operations such as lookup and update are performed in constant time O(1). Since every element is processed exactly once, the overall time complexity is O(n).

Space Complexity: O(n)

An additional hash map is used to store the frequency of each number. In the worst case, all n elements are distinct and stored in the map, resulting in O(n) extra space usage.

Detailed Description (Algorithm and Approach)

The objective is to count the number of good pairs (i, j) such that nums[i] == nums[j] and i < j.

1. Initialize an empty hash map to store the frequency of numbers encountered.
2. Initialize a counter variable to count good pairs.
3. Traverse the array from left to right.
4. For each element, check if it already exists in the map:
   - If it exists, add the current frequency of that element to the counter.
5. Update the frequency of the current element in the map.
6. Continue this process until all elements are processed.
7. Return the total count of good pairs.

This approach counts valid pairs efficiently by using frequency counting and avoids nested loops, reducing the time complexity from $O(n^2)$ to O(n).

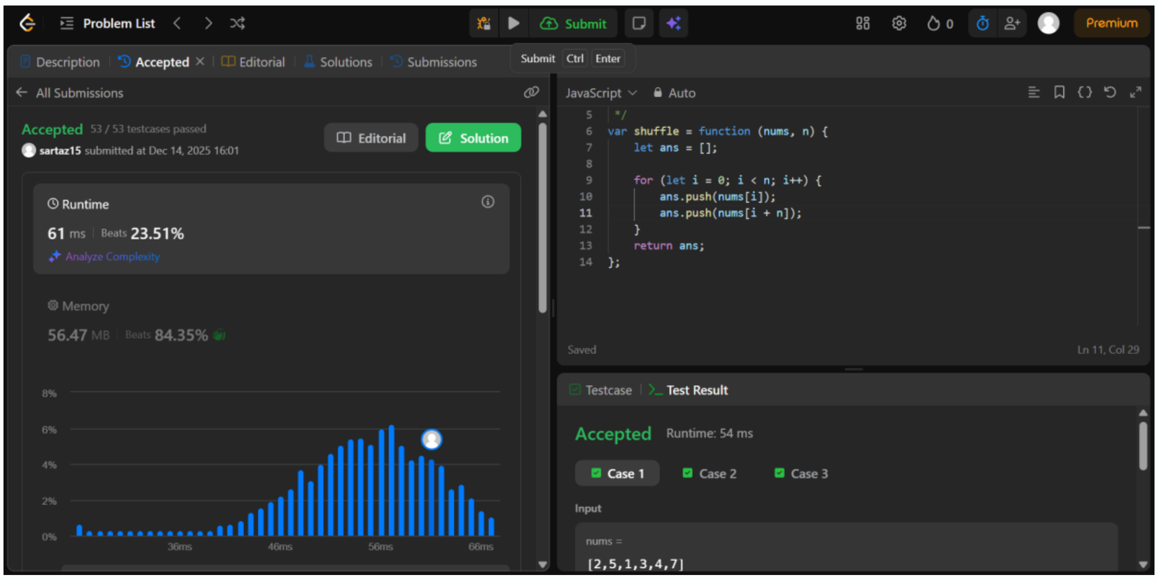
Question: Leetcode 1470

Shuffle the Array

https://leetcode.com/problems/shuffle-the-array/description/

Answer Link

https://leetcode.com/problems/shuffle-the-array/submissions/1855290281/



Accepted 53 / 53 testcases passed
sartaz15 submitted at Dec 14, 2025 16:01

Runtime
61 ms | Beats 23.51%

Memory
56.47 MB | Beats 84.35%

JavaScript

```javascript
*/
var shuffle = function (nums, n) {
    let ans = [];

    for (let i = 0; i < n; i++) {
        ans.push(nums[i]);
        ans.push(nums[i + n]);
    }
    return ans;
};
```

Saved   Ln 11, Col 29

Test Result

Accepted   Runtime: 54 ms

Case 1   Case 2   Case 3

Input

nums =
[2,5,1,3,4,7]

Time Complexity: O(n)

The algorithm iterates through the array once to construct the shuffled result. Each iteration performs constant time operations such as accessing array elements and pushing values into a new array. Since all 2n elements are processed once, the total time complexity is O(n).

Space Complexity: O(n)

A new array is created to store the shuffled result. This array stores all 2n elements, which requires O(n) additional space.

Detailed Description (Algorithm and Approach)

The problem requires rearranging the given array of the form [x1, x2, ..., xn, y1, y2, ..., yn] into [x1, y1, x2, y2, ..., xn, yn].

1. Initialize an empty result array.
2. Use a loop that runs from index 0 to n − 1.
3. For each index i:
   - Push nums[i] (element from the first half).
   - Push nums[i + n] (corresponding element from the second half).
4. Continue until all elements are added to the result array.
5. Return the result array.

This approach ensures correct ordering of elements and achieves optimal performance with linear time complexity.

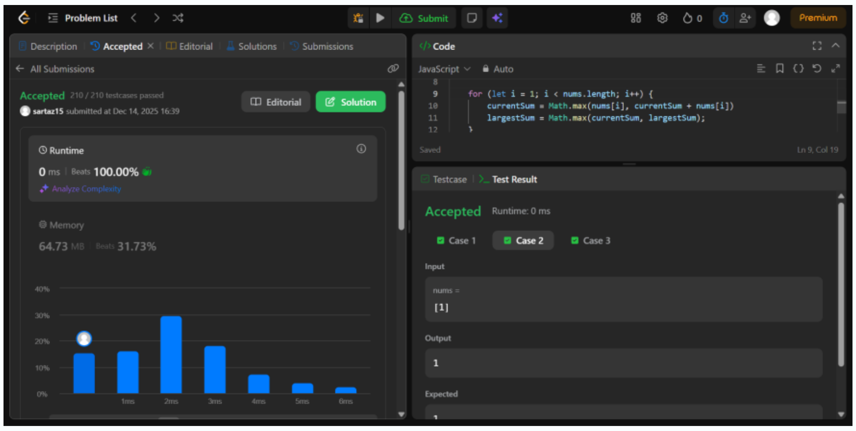
Question: Leetcode 53

Maximum Subarray

https://leetcode.com/problems/maximum-subarray/description/

Answer Link

https://leetcode.com/problems/maximum-subarray/submissions/1855314393/



Accepted 210 / 210 testcases passed
sartaz15 submitted at Dec 14, 2025 16:39

Runtime
0 ms | Beats 100.00%

Memory
64.73 MB | Beats 31.73%

```
for (let i = 1; i < nums.length; i++) {
    currentSum = Math.max(nums[i], currentSum + nums[i])
    largestSum = Math.max(currentSum, largestSum);
}
```

Saved   Ln 9, Col 19

Accepted Runtime: 0 ms
Case 1 Case 2 Case 3

Input
nums =
[1]

Output
1

Expected
1

Time Complexity: O(n)

The algorithm traverses the array only once. At each step, constant time operations are performed to update the current sum and the maximum sum. Since all (n) elements are processed once, the overall time complexity is O(n).

Space Complexity: O(1)

Only a few variables are used to store the current sum and maximum sum. No extra data structures are required, so the space complexity remains constant at O(1).

Detailed Description (Algorithm and Approach)

The problem is solved using Kadane's Algorithm, which efficiently finds the maximum sum of a contiguous subarray.

1. Initialize two variables:
   - currentSum to store the sum of the current subarray.
   - maxSum to store the maximum subarray sum found so far.
2. Start traversing the array from the first element.
3. For each element:
   - Add it to currentSum.
   - Update maxSum if currentSum is greater than maxSum.
   - If currentSum becomes negative, reset it to 0.
4. Continue this process until all elements are processed.
5. Return maxSum as the result.

Kadane's Algorithm works by discarding subarrays with negative sums since they cannot contribute to a maximum subarray in the future.

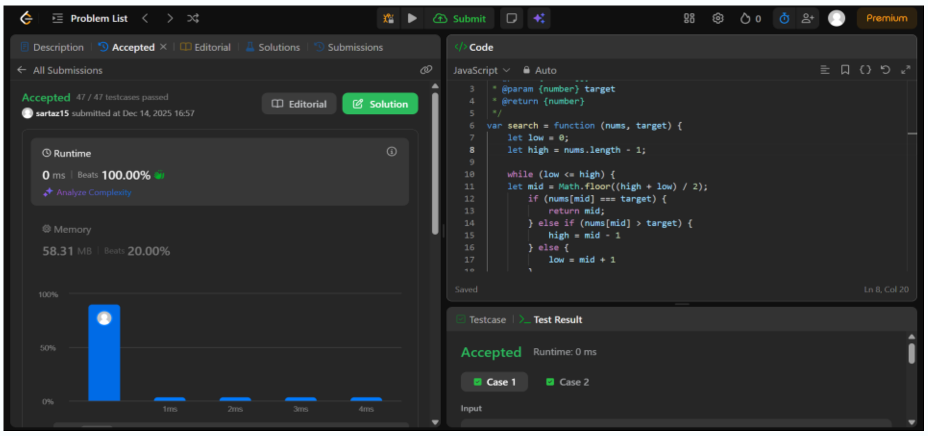
<u>**Question: Leetcode 704**</u>

Binary Search

https://leetcode.com/problems/binary-search/description/

Answer Link

https://leetcode.com/problems/binary-search/submissions/1855324924/



Accepted 47 / 47 testcases passed
sartaz15 submitted at Dec 14, 2025 16:57

Runtime
0 ms | Beats 100.00%
Analyze Complexity

Memory
58.31 MB | Beats 20.00%

```javascript
 * @param {number} target
 * @return {number}
 */
var search = function (nums, target) {
    let low = 0;
    let high = nums.length - 1;

    while (low <= high) {
    let mid = Math.floor((high + low) / 2);
        if (nums[mid] === target) {
            return mid;
        } else if (nums[mid] > target) {
            high = mid - 1
        } else {
            low = mid + 1
```

Saved
Ln 8, Col 20

Accepted Runtime: 0 ms

Time Complexity: O(log n)

Binary search repeatedly divides the search space into half in each iteration. Since the array is sorted, each comparison reduces the remaining elements by half, resulting in logarithmic time complexity O(log n).

Space Complexity: O(1)

The iterative binary search approach uses only a few variables such as low, high, and mid. No additional data structures are used, so the space **complexity** remains constant at O(1).

Detailed Description (Algorithm and Approach)

The objective is to search for a target value in a sorted array and return its index if found, otherwise return −1.

1. Initialize two pointers:

   - low pointing to the first index of the array.

   - high pointing to the last index of the array.

2. While low is less than or equal to high:

   - Calculate the middle index.

3. Compare the middle element with the target:

   - If equal, return the middle index.

   - If the target is smaller, move the high pointer to mid − 1.

   - If the target is larger, move the low pointer to mid + 1.

4. Repeat the process until the target is found, or the search space becomes empty.

5. If the target is not found, return −1.

Binary search is efficient because it eliminates half of the remaining elements in each step, making it ideal for searching in sorted arrays.