NAME - PRATEEK SARASWAT

DSA-1 Project

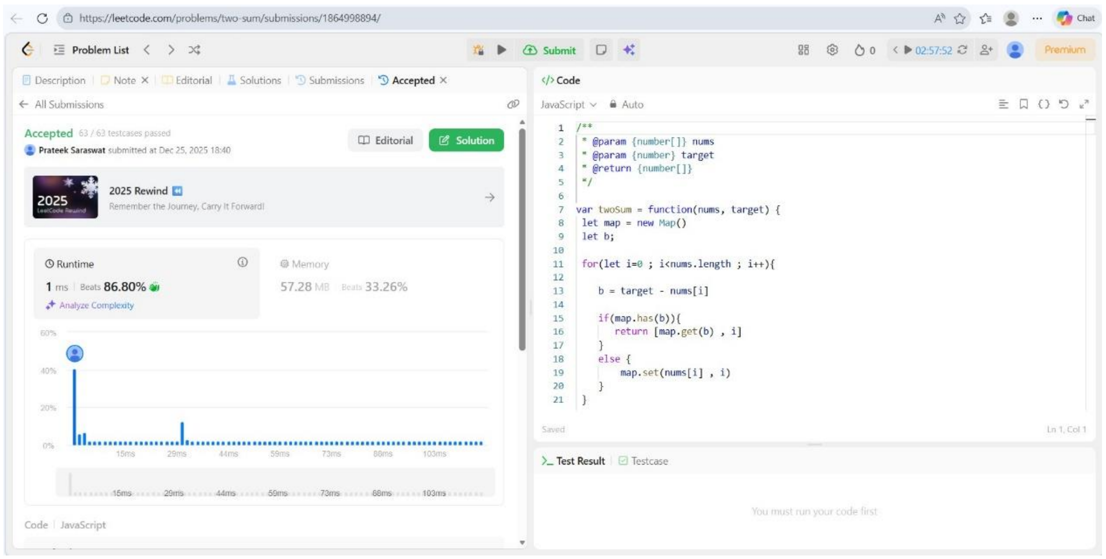
Question: Leetcode 1

Two Sum

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

Answer Link

https://leetcode.com/problems/two-sum/submissions/1864998894/


https://leetcode.com/problems/two-sum/submissions/1864998894/

Accepted 63 / 63 testcases passed
Prateek Saraswat submitted at Dec 25, 2025 18:40

2025 Rewind
Remember the Journey, Carry It Forward!

Runtime: 1 ms | Beats 86.80%
Memory: 57.28 MB | Beats 33.26%

Code | JavaScript

```javascript
/**
 * @param {number[]} nums
 * @param {number} target
 * @return {number[]}
 */

var twoSum = function(nums, target) {
 let map = new Map()
 let b;

 for(let i=0 ; i<nums.length ; i++){

    b = target - nums[i]

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

Test Result | Testcase

You must run your code first

Time Complexity: O(n)

The algorithm processes each element of the array exactly once using a single loop. For every element, it performs hash map operations such as lookup and insertion, which run in constant time **O(1)** on average. Since these operations are executed for all **n** elements, the total time complexity of the algorithm is **O(n)**.

Space Complexity: O(n)

The algorithm uses an additional hash map to store elements along with their corresponding indices. In the worst case, all **n** elements are inserted into the map, which leads to an extra space complexity of **O(n)**.

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.

NAME - PRATEEK SARASWAT

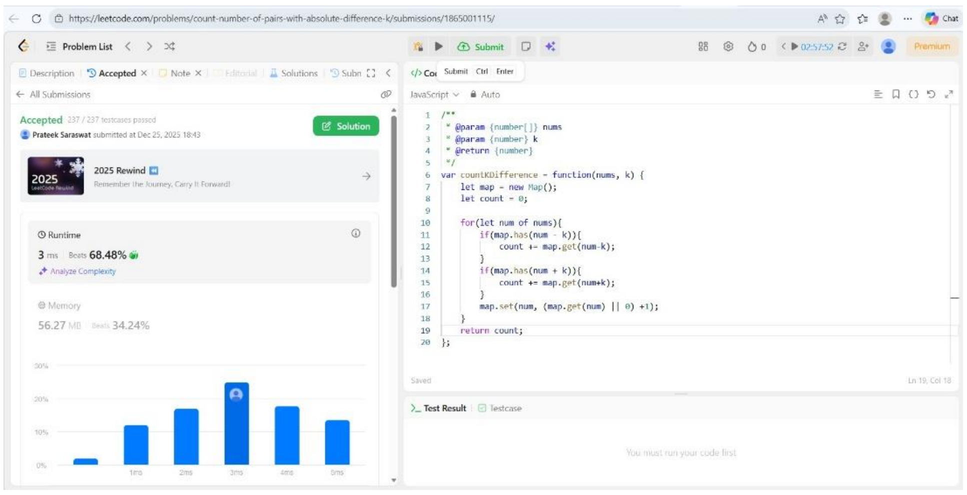
Question: Leetcode 2006

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/1865001115/



Accepted 237 / 237 testcases passed

Prateek Saraswat submitted at Dec 25, 2025 18:43

2025 Rewind

Remember the Journey, Carry It Forward!

Runtime

3 ms | Beats 68.48%

Memory

56.27 MB | Beats 34.24%

```javascript
/**
 * @param {number[]} nums
 * @param {number} k
 * @return {number}
 */
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 19, Col 18

You must run your code first

Time Complexity: O(n)

The algorithm scans the array only once using a single loop. For each element, it performs hash map operations such as lookup and insertion, which take **O(1)** time on average. As each of the **n** elements is processed exactly once, the overall 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 are stored in the map, resulting in an additional space complexity of **O(n)**.

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.

NAME - PRATEEK SARASWAT

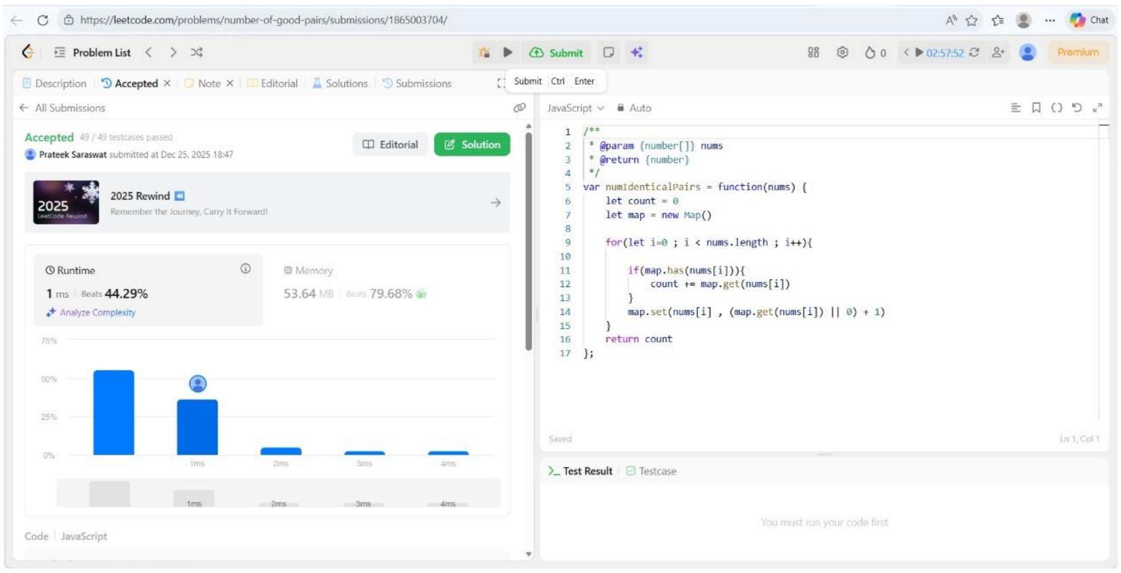
Question: Leetcode 1512

Number of Good Pairs

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

Answer Link

https://leetcode.com/problems/number-of-good-pairs/submissions/1865003704/



Accepted 49 / 49 testcases passed

Prateek Saraswat submitted at Dec 25, 2025 18:47

Runtime 1 ms | Beats 44.29%

Memory 53.64 MB | Beats 79.68%

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

    for(let i=0 ; i < nums.length ; i++){

        if(map.has(nums[i])){
            count += map.get(nums[i])
        }
        map.set(nums[i] , (map.get(nums[i]) || 0) + 1)
    }
    return count
};
```

Time Complexity: O(n)

The algorithm iterates through the array exactly once. For each element, constant-time hash map operations such as lookup and insertion are performed. As all **n** elements are processed a single time, the overall time complexity is **O(n)**.

Space Complexity: O(n)

An auxiliary hash map is used to store the frequency of each element. In the worst case, all **n** elements are distinct and stored in the map, leading to an additional space complexity of **O(n)**.

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 $O(n)$.

NAME - PRATEEK SARASWAT

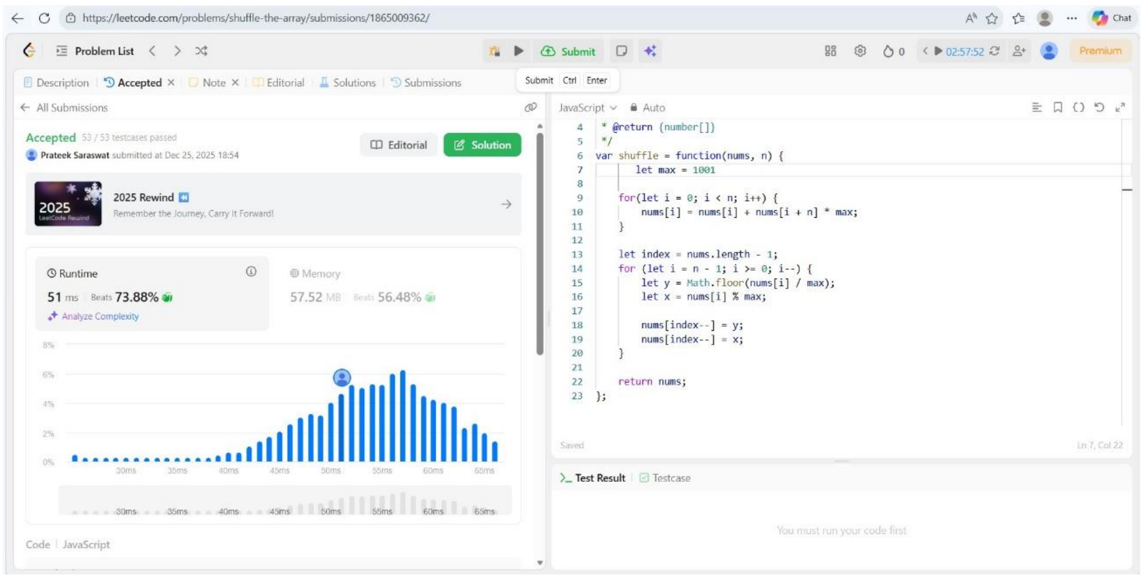
Question: Leetcode 1470

Shuffle the Array

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

Answer Link

https://leetcode.com/problems/shuffle-the-array/submissions/1865009362/



Accepted 53 / 53 testcases passed
Prateek Saraswat submitted at Dec 25, 2025 18:54

Runtime 51 ms | Beats 73.88%
Memory 57.52 MB | Beats 56.48%

```javascript
 * @return {number[]}
 */
var shuffle = function(nums, n) {
        let max = 1001

    for(let i = 0; i < n; i++) {
        nums[i] = nums[i] + nums[i + n] * max;
    }

    let index = nums.length - 1;
    for (let i = n - 1; i >= 0; i--) {
        let y = Math.floor(nums[i] / max);
        let x = nums[i] % max;

        nums[index--] = y;
        nums[index--] = x;
    }

    return nums;
};
```

Time Complexity: O(n)

The solution processes the array using two loops, each iterating **n** times. In every iteration, only constant-time arithmetic and assignment operations are performed. Since a total of **n** elements are handled in each loop, the overall time complexity is **O(n)**.

Space Complexity: O(1)

The algorithm performs all operations **in place** on the given array. Only a few extra variables (max, index, x, y) are used, and no additional data structures are required. Therefore, the extra space complexity is **O(1)**.

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. Uses a large constant max to **encode two numbers into one** array element.
2. First loop **packs** nums[i] and nums[i + n] at index i.
3. Second loop **decodes** the packed values using division and modulo.
4. Elements are placed in the correct shuffled order **from the end of the array**.
5. The array is modified **in place** and returned as the result.

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

NAME - PRATEEK SARASWAT

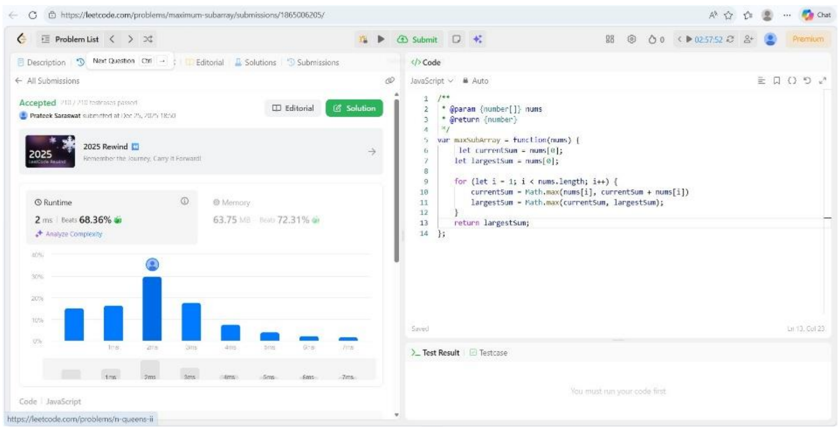
<u>Question: Leetcode 53</u>

<u>Maximum Subarray</u>

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

<u>Answer Link</u>

https://leetcode.com/problems/maximum-subarray/submissions/1865006205/



Accepted 210 / 210 testcases passed
Prateek Saraswat submitted at Dec 25, 2025 18:50

Runtime
2 ms | Beats 68.36%

Memory
63.75 MB | Beats 72.31%

```javascript
/**
 * @param {number[]} nums
 * @return {number}
 */
var maxSubArray = function(nums) {
     let currentSum = nums[0];
    let largestSum = nums[0];

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

<u>Time Complexity: O(n)</u>

The algorithm scans the array exactly once. At each iteration, it performs constant-time operations to update the current sum and the maximum sum. Since all **n** elements are processed a single time, the overall time complexity is **O(n)**.

<u>Space Complexity: O(1)</u>

The algorithm uses only a small, fixed number of variables to maintain the current sum and the maximum sum. Since no additional data structures are required, the space complexity is **O(1)**.

<u>Detailed Description (Algorithm and Approach)</u>

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.

NAME - PRATEEK SARASWAT

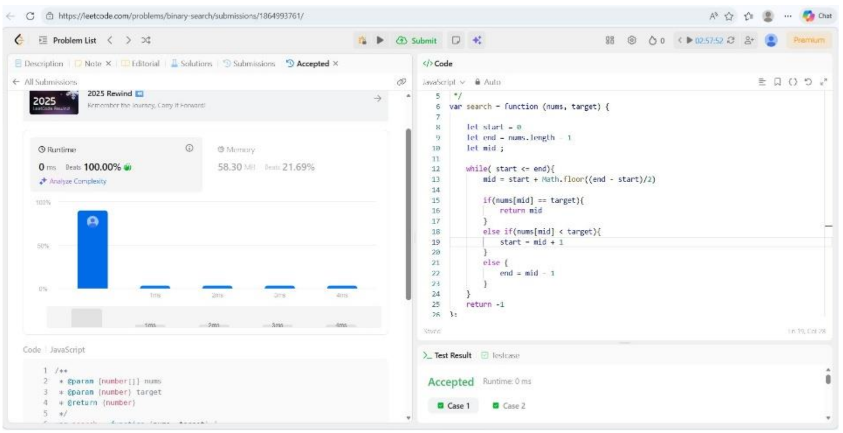
Question: Leetcode 704

Binary Search

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

Answer Link

https://leetcode.com/problems/binary-search/submissions/1864993761/



```
/**
 * @param {number[]} nums
 * @param {number} target
 * @return {number}
 */
var search = function (nums, target) {

    let start = 0
    let end = nums.length - 1
    let mid ;

    while( start <= end){
        mid = start + Math.floor((end - start)/2)

        if(nums[mid] == target){
            return mid
        }
        else if(nums[mid] < target){
            start = mid + 1
        }
        else {
            end = mid - 1
        }
    }
    return -1
};
```

Accepted Runtime: 0 ms

Time Complexity: O(log n)

Binary search works by repeatedly dividing the search space in half. Because the array is sorted, each comparison eliminates half of the remaining elements. As a result, the algorithm runs in logarithmic time, with a time complexity of **O(log n)**.

Space Complexity: O(1)

The iterative binary search approach uses only a fixed number of variables such as **low**, **high**, and **mid**. Since no extra data structures are required, 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:

  - start pointing to the first index of the array.

  - end 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 end pointer to mid − 1.

  - If the target is larger, move the start 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.