1679. Max Number of K-Sum Pairs

Two Pointers

Sorting

Hash Table

# **Problem Description**

Array

Medium

You're provided with an integer array called nums and another integer k. The goal is to determine how many pairs of numbers you can find and remove from the array such that the sum of each pair equals k. The operation of picking and removing such a pair is counted as one operation. The task is to return the maximum number of such operations that you can perform on the given array.

Intuition To solve this problem, we use a two-pointer technique, which is a common strategy in problems involving sorted arrays or

the end (r) of the array.

• If the sum of the values at the two pointers is exactly k, we've found a valid pair that can be removed from the array. We increment our operation count (ans), and then move the left pointer to the right (1 + 1) and the right pointer to the left (r - 1) to find the next potential pair. • If the sum is greater than k, we need to decrease it. Since the array is sorted, the largest sum can be reduced by moving the right pointer to the

sequences. First, we sort the array in ascending order. After sorting, we position two pointers: one at the beginning (1) and one at

left (r - 1). • If the sum is less than k, we need to increase it. We do this by moving the left pointer to the right (1 + 1).

We repeat this process, scanning the array from both ends towards the middle, until the two pointers meet. This approach

ensures that we find all valid pairs that can be formed without repeating any number, as each operation requires removing the paired numbers from the array.

The reason this approach works efficiently is that sorting the array allows us to make decisions based on the sum comparison,

**Solution Approach** 

The solution provided uses a two-pointer approach to implement the logic that was described in the previous intuition section.

ensuring that we do not need to reconsider any previous elements once a pair is found or the pointers have been moved.

# Below is a step-by-step walkthrough of the algorithm, referencing the provided Python code.

nums.sort()

1. Sort the nums list. This is a crucial step as it allows for the two-pointer approach to work efficiently. We need the array to be ordered so we can target sums that are too high or too low by moving the appropriate pointer.

## 2. Initialize two pointers, 1 (left) and r (right), at the start and end of the array, respectively. Also, initialize an ans variable to count the number of operations.

while l < r:

l, r, ans = 0, len(nums) - 1, 03. Enter a while loop that will continue to execute as long as the left pointer is less than the right pointer, ensuring we do not cross pointers and

4. Within the loop, calculate the sum s of the elements at the pointers' positions.

recheck the same elements.

s = nums[l] + nums[r]5. Check if the sum s equals k. If it does:

 Move the left pointer one step to the right to seek the next potential pair. Move the right pointer one step to the left.

if s == k:

elif s > k:

r -= 1

l += 1

ans += 1 l, r = l + 1, r - 1

6. If the sum s is more significant than k, the right pointer must be decremented to find a smaller pair sum.

• Increment the ans variable because we found a valid operation.

pairs with the sum k that were removed from the array.

Let's walk through the algorithm step-by-step:

nums = [2, 3, 4, 5, 6] // Sorted array

ans = 0 // Number of pairs found

2. We initialize our pointers and answer variable:

```
7. If the sum s is less than k, the left pointer must be incremented to find a greater pair sum.
else:
```

**Example Walkthrough** 

nums and an integer k = 10:

1. First, we sort the nums array:

This approach only uses the sorted list and two pointers without additional data structures. The space complexity of the algorithm is O(log n) due to the space required for sorting, with the time complexity being O(n log n) because of the sorting step;

8. After the while loop concludes, return the ans variable, which now contains the count of all operations performed — the maximum number of

the scanning of the array using two pointers is O(n), which does not dominate the time complexity.

return ans

nums = [3, 5, 4, 6, 2]

Let's take an example to see how the two-pointer solution approach works. Assume we have the following integer array called

## l = 0 // Left pointer index r = 4 // Right pointer index (nums.length - 1)

6 = 8.

Since 8 is less than k, we increment the left pointer 1 to try and find a larger sum. The pointers are now 1 = 1 and r = 4.

At the first iteration, the sum of the elements at the pointers' positions is s = nums[1] + nums[r] = nums[0] + nums[4] = 2 +

Start the loop with while l < r. Our initial pointers are at positions nums [0] and nums [4].

Since 9 is still less than k, we increment 1 again. The pointers are now 1 = 2 and r = 4.

Now, s = nums[l] + nums[r] = nums[1] + nums[4] = 3 + 6 = 9.

Now, s = nums[l] + nums[r] = nums[2] + nums[4] = 4 + 6 = 10.

def max\_operations(self, nums: List[int], k: int) -> int:

# Iterate through the list with two pointers

operations\_count += 1

left += 1

right -= 1

} else if (sum > k) {

// Return the total number of operations

--right;

++left;

} else {

return answer;

// If the sum is greater than k, we need to decrease the sum

// We do this by moving the right pointer to the left

// We do this by moving the left pointer to the right

// If the sum is less than k, we need to increase the sum

# Sort the array first to apply the two-pointer technique

# Initialize a counter to keep track of valid operations

# Increment the count of valid operations

# Move both pointers towards the center

# Initialize two pointers, one at the start and one at the end

Since 1 is no longer less than r, the loop ends.

We return the ans variable, which stands at 1, indicating we have found one pair (4, 6) that sums up to k.

Hence, using this approach, the maximum number of operations (pairs summing up to k) we can perform on nums is 1.

Since 10 is equal to k, we increment ans to 1 and move both pointers inward: 1 becomes 3, and r becomes 3.

**Python** 

### # Calculate the sum of elements pointed by left and right sum\_of\_pair = nums[left] + nums[right] # If the sum equals k, we found a valid pair if sum\_of\_pair == k:

Solution Implementation

from typing import List

nums.sort()

left, right = 0, len(nums) - 1

operations\_count = 0

while left < right:</pre>

class Solution:

```
# If the sum is too large, move the right pointer to the left
           elif sum_of_pair > k:
                right -= 1
           # If the sum is too small, move the left pointer to the right
           else:
                left += 1
       # Return the total count of valid operations
        return operations_count
Java
class Solution {
    public int maxOperations(int[] nums, int k) {
       // Sort the array to use two pointers approach
       Arrays.sort(nums);
       // Initialize two pointers, one at the start (left) and one at the end (right) of the array
       int left = 0, right = nums.length - 1;
       // Initialize the answer variable to count the number of operations
       int answer = 0;
       // Use a while loop to move the two pointers towards each other
       while (left < right) {</pre>
           // Calculate the sum of the two-pointer elements
           int sum = nums[left] + nums[right];
           // Check if the sum is equal to k
           if (sum == k) {
                // If it is, increment the number of operations
                ++answer;
                // Move the left pointer to the right and the right pointer to the left
                ++left;
                --right;
```

**}**;

// Example usage of the class:

// std::vector<int> nums = {3, 1, 3, 4, 3};

// int result = sol.maxOperations(nums, k);

const countMap = new Map<number, number>();

countMap.set(num, currentCount);

// Solution sol;

// int k = 6;

**TypeScript** 

C++ #include <vector> // Include necessary header for vector #include <algorithm> // Include algorithm header for sort function class Solution { public: int maxOperations(std::vector<int>& nums, int k) { // Sort the vector to make two-pointer technique applicable std::sort(nums.begin(), nums.end()); int count = 0; // Initialize count of operations int left = 0; // Initialize left pointer int right = nums.size() - 1; // Initialize right pointer // Use two-pointer technique to find pairs that add up to k while (left < right) {</pre> // When the sum of the current pair equals k if (nums[left] + nums[right] == k) { left++; // Move left pointer to the right right--; // Move right pointer to the left count++; // Increment the count of valid operations } else if (nums[left] + nums[right] > k) { // If the sum is greater than k, move right pointer to the left right--; } else { // If the sum is less than k, move left pointer to the right left++;

```
let operationsCount = 0;
// Iterate over each number in the array
for (const num of nums) {
   const complement = targetSum - num; // Calculate the complement of the current number
   // If the complement is already in the map,
   // we can form a pair whose sum is equal to targetSum
   if (countMap.get(complement) > 0) {
        countMap.set(complement, countMap.get(complement) - 1); // Decrement the count of complement in map
        operationsCount++; // Increment the count of valid operations
   } else {
```

const currentCount = (countMap.get(num) || 0) + 1;

return operationsCount; // Return the total number of operations

# Move both pointers towards the center

# If the sum is too large, move the right pointer to the left

# If the sum is too small, move the left pointer to the right

// If the complement is not there, store/update the count of the current number

return count; // Return the total count of operations

function maxOperations(nums: number[], targetSum: number): number {

// std::cout << "Maximum operations to reach sum k: " << result << std::endl;

from typing import List class Solution: def max\_operations(self, nums: List[int], k: int) -> int: # Sort the array first to apply the two-pointer technique nums.sort() # Initialize two pointers, one at the start and one at the end left, right = 0, len(nums) - 1# Initialize a counter to keep track of valid operations operations\_count = 0 # Iterate through the list with two pointers while left < right:</pre> # Calculate the sum of elements pointed by left and right sum\_of\_pair = nums[left] + nums[right] # If the sum equals k, we found a valid pair if sum of pair == k: # Increment the count of valid operations operations\_count += 1

## The given code has a time complexity of $O(n \log n)$ . Here's the breakdown:

Sorting the nums list takes 0(n log n) time.

# Return the total count of valid operations

left += 1

right -= 1

right -= 1

left += 1

return operations\_count

Time and Space Complexity

else:

**Time Complexity** 

elif sum\_of\_pair > k:

• The while loop runs in O(n) time because it iterates through the list at most once by moving two pointers from both ends towards the center. In each iteration, one of the pointers moves, ensuring that the loop cannot run for more than n iterations. The operations inside the while loop are all constant time checks and increments, each taking 0(1).

**Space Complexity** The space complexity of the code is 0(1) provided that the sorting algorithm used in place.

Therefore, the combined time complexity is dominated by the sorting step, giving us  $0(n \log n)$ .

• Extra variables 1, r, and ans are used, but they occupy constant space.

No additional data structures are used that depend on the input size of nums.