<https://leetcode.com/problems/max-number-of-k-sum-pairs/description/>

You are given an integer array nums and an integer k.

In one operation, you can pick two numbers from the array whose sum equals k and remove them from the array.

Return **the maximum number of operations you can perform on the array**.

**Example 1:**

**Input:** nums = [1,2,3,4], k = 5

**Output:** 2

**Explanation:** Starting with nums = [1,2,3,4]:

- Remove numbers 1 and 4, then nums = [2,3]

- Remove numbers 2 and 3, then nums = []

There are no more pairs that sum up to 5, hence a total of 2 operations.

**Example 2:**

**Input:** nums = [3,1,3,4,3], k = 6

**Output:** 1

**Explanation:** Starting with nums = [3,1,3,4,3]:

- Remove the first two 3's, then nums = [1,4,3]

There are no more pairs that sum up to 6, hence a total of 1 operation.

**Constraints:**

1 <= nums.length <= 10^5

1 <= nums[i] <= 10^9

1 <= k <= 10^9

**Attempt 1: 2024-01-12**

**Solution 1: Sorting + Two Pointers (10min)**

class Solution {

    public int maxOperations(int[] nums, int k) {

        Arrays.sort(nums);

        int count = 0;

        int i = 0;

        int j = nums.length - 1;

        while(i < j) {

            if(nums[i] + nums[j] == k) {

                i++;

                j--;

                count++;

            } else if(nums[i] + nums[j] > k) {

                j--;

            } else {

                i++;

            }

        }

        return count;

    }

}

Time Complexity: O(N\*logN)

Space Complexity: O(1)

**Solution 2: Hash Table (10min)**

class Solution {

    public int maxOperations(int[] nums, int k) {

        Map<Integer, Integer> map = new HashMap<>();

        int count = 0;

        for(int num : nums) {

            if(map.containsKey(k - num)) {

                count++;

                if(map.get(k - num) == 1) {

                    map.remove(k - num);

                } else {

                    map.put(k - num, map.get(k - num) - 1);

                }

            } else {

                map.put(num, map.getOrDefault(num, 0) + 1);

            }

        }

        return count;

    }

}

Time Complexity: O(N)

Space Complexity: O(N)

**Going from O(N^2) -> O(NlogN) -> O(N) + MEME**

**Refer to**

<https://leetcode.com/problems/max-number-of-k-sum-pairs/solutions/2005922/going-from-o-n-2-o-nlogn-o-n-meme/>

**BRUTE**

The brute force way to solve this problem is very easy, have 2 pointers one start from very begining & another start just after the second pointer. Means the 2nd pointer is depends on 1st pointer. And now you say if that's the case it wont gonna visit the same element over & over again.?

And my answer is Yes it'll, but we;ll gonna play mind games. So, the element which i & j pointer has already visited flag them as visited with any value, let's say i'll gonna mark them with -1. So, by that we can handle this case easily. Now Happy :)

class Solution {

public int maxOperations(int[] nums, int k) {

int count = 0;

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

if(nums[i] == -1) continue;

for(int j = i + 1; j < nums.length; j++){

if(nums[j] == -1) continue;

if(nums[i] + nums[j] == k){

count++;

nums[i] = -1;

nums[j] = -1;

break;

}

}

}

return count;

}

}

ANALYSIS :-

Time Complexity :- BigO(N^2)

Space Complexity :- BigO(1)

**BETTER**

Now, Interviewer won't be happy with your brute solution, so just do some acting & behave like ya, I can improve it more. Tell him, why dont if we Sort the array & have 2 pointers one start from very begining & another from the end of the array.

Now we gonna get the sum,

if sum == k :- Increment our count & move our I & J pointer

if sum > k, Move our J pointer

if sum < k, Move our I pointer

class Solution {

public int maxOperations(int[] nums, int k) {

Arrays.sort(nums);

int count = 0;

int i = 0;

int j = nums.length - 1;

while(i < j){

int sum = nums[i] + nums[j];

if(sum == k) {

count++;

i++;

j--;

}

else if(sum > k) j--;

else i++;

}

return count;

}

}

ANALYSIS :-

Time Complexity :- BigO(NlogN) + O(N) => O(NlogN)

Space Complexity :- BigO(1)

**BEST**

But, by looking at that, interviewer will say. I'm still not happy, so give him a ~~flying kiss~~ I mean give him a better solution 😅

Now again, perform a beautiful acting, and say why dont we use MAP. So, tell him by using map, we can optimise it more beautifully.

So, what we'll gonna do is & may be you know this one, it is similar to **Two Sum**, so if u dont know then listen to me.

We gonna fill our map as frequency map. And we gonna get the result by subtracting current value from k & whatever result we get, we gonna check in our map. If that is present increment the count & remove it from the map now. That's how we'll get our answer in just O(1) for searching & as we are using a loop thus, O(N). But it's a good approach.

class Solution {

public int maxOperations(int[] nums, int k) {

Map<Integer, Integer> map = new HashMap<>();

int count = 0;

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

int res = k - nums[i];

if(map.containsKey(res)){

count++;

if(map.get(res) == 1) map.remove(res);

else map.put(res, map.get(res) - 1);

}

else{

map.put(nums[i], map.getOrDefault(nums[i], 0) + 1);

}

}

return count;

}

}

ANALYSIS :-

Time Complexity :- BigO(N)

Space Complexity :- BigO(N)

**Refer to**

<https://algo.monster/liteproblems/1679>

**Problem Description**

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 sequences. First, we sort the array in ascending order. After sorting, we position two pointers: one at the beginning (l) and one at 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 (l + 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 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 (l + 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, ensuring that we do not need to reconsider any previous elements once a pair is found or the pointers have been moved.

**Solution Approach**

The solution provided uses a two-pointer approach to implement the logic that was described in the previous intuition section. Below is a step-by-step walkthrough of the algorithm, referencing the provided Python code.

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.

nums.sort()

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

l, r, ans = 0, len(nums) - 1, 0

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 recheck the same elements.

while l < r:

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

s = nums[l] + nums[r]

Check if the sum s equals k. If it does:

Increment the ans variable because we found a valid operation.

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:

ans += 1

l, r = l + 1, r - 1

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

elif s > k:

r -= 1

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

else:

l += 1

After the while loop concludes, return the ans variable, which now contains the count of all operations performed — the maximum number of pairs with the sum k that were removed from the array.

return ans

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

**Example Walkthrough**

Let's take an example to see how the two-pointer solution approach works. Assume we have the following integer array called nums and an integer k = 10:

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

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

First, we sort the nums array:

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

We initialize our pointers and answer variable:

l = 0 // Left pointer index

r = 4 // Right pointer index (nums.length - 1)

ans = 0 // Number of pairs found

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

At the first iteration, the sum of the elements at the pointers' positions is s = nums[l] + nums[r] = nums[0] + nums[4] = 2 + 6 = 8.

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

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

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

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

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

Since l 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.

**Java Solution**

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) {

            // 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;

            } else if (sum > k) {

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

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

                --right;

            } else {

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

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

                ++left;

            }

        }

        // Return the total number of operations

        return answer;

    }

}

**Time and Space Complexity**

**Time Complexity**

The given code has a time complexity of O(n log n).

Here's the breakdown:

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

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 O(1).

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

**Space Complexity**

The space complexity of the code is O(1) provided that the sorting algorithm used in place.

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

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