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

Given an integer array nums and an integer k, return **the number of pairs** (i, j) **where** i < j **such that** |nums[i] - nums[j]| == k.

The value of |x| is defined as:

x if x >= 0.

-x if x < 0.

**Example 1:**

**Input:** nums = [1,2,2,1], k = 1

**Output:** 4

**Explanation:** The pairs with an absolute difference of 1 are:

- [**1**,**2**,2,1]

- [**1**,2,**2**,1]

- [1,**2**,2,**1**]

- [1,2,**2**,**1**]

**Example 2:**

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

**Output:** 0

**Explanation:** There are no pairs with an absolute difference of 3.

**Example 3:**

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

**Output:** 3

**Explanation:** The pairs with an absolute difference of 2 are:

- [**3**,2,**1**,5,4]

- [**3**,2,1,**5**,4]

- [3,**2**,1,5,**4**]

**Constraints:**

1 <= nums.length <= 200

1 <= nums[i] <= 100

1 <= k <= 99

**Attempt 1: 2024-01-13**

**Solution 1: Hash Table (60 min)**

**Wrong Solution (142/237)**

**Test out by**

Input: nums = [9,3,1,1,4,5,4,9,5,10], k = 1

Output = 7, Expect = 8

**错误原因：map中满足nums[i] + k的个数和满足nums[i] - k的个数不应该是if...else if的互斥关系，而应该是if...if...的并列关系，因为对于任意一个位于坐标i的数nums[i]来说，都会在遍历到它的时候具备和之前已经存入map中的(遍历过的[0, i - 1]的数)所有数建立这两种关系的可能性，两种可能性不是对立的，而是同时存在的，而且这样从左往右对数组中的每个数都检查一次有没有满足的nums[i] + k或者nums[i] - k不会有重复或者漏掉的情况，因为每次检查只针对当时的map中已经遍历过的数，这个状态是动态变化的，每个第i个数只针对前i - 1个数在map中的存储情况寻找**

class Solution {

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

        int count = 0;

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

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

            if(map.containsKey(nums[i] + k)) {

                count += map.get(nums[i] + k);

            } else if(map.containsKey(nums[i] - k)) {

                count += map.get(nums[i] - k);

            }

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

        }

        return count;

    }

}

**Correct Solution**

class Solution {

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

        int count = 0;

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

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

            if(map.containsKey(nums[i] + k)) {

                count += map.get(nums[i] + k);

            }

            if(map.containsKey(nums[i] - k)) {

                count += map.get(nums[i] - k);

            }

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

        }

        return count;

    }

}

Time Complexity: O(N)

Space Complexity: O(N)

**Solution 2: Counting Sort (10 min)**

class Solution {

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

        int count = 0;

        // From definition we know 1 <= nums[i] <= 100

        int[] buckets = new int[101];

        for(int num : nums) {

            if(num >= k) {

                count += buckets[num - k];

            }

            if(num + k <= 100) {

                count += buckets[num + k];

            }

            buckets[num]++;

        }

        return count;

    }

}

Time Complexity: O(N)

Space Complexity: O(N)

**As easy as 1,2,3 or not? From n^2 to n. Super detailed explanation.**

**Refer to**

<https://leetcode.com/problems/count-number-of-pairs-with-absolute-difference-k/solutions/1647117/as-easy-as-1-2-3-or-not-from-n-2-to-n-super-detailed-explanation/>

**101**

What's absolute difference (AD)? According to <https://en.wikipedia.org/wiki/Absolute_difference>, AD describes the **distance** on the real line between the points.

An example:

\_\_\_\_3\_\_\_\_\_

\_\_|\_\_|\_\_|\_\_|\_\_|\_\_

-2 -1 0 1 2

The AD of pair (-1,2) /\* (start, end) \*/, going from **left to right**, is |2 - (-1)| = |3| = 3. Also going backward, from **right to left**, for another pair (2,-1) is also |-1 - 2| = |-3| = 3.

Now, back to our problem:

**Problem statement**

Copy + paste with important parts highlighted (now in **bold**!)

Return the number of pairs (i, j) **where i < j** such that **|nums[i] - nums[j]| == k.**

**Approach #1: Bruteforce**

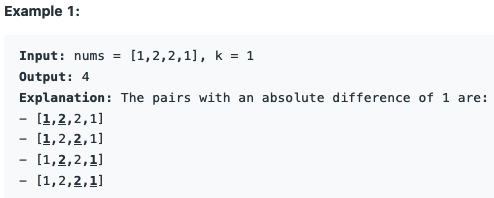
Find all possible pairs (1...n, 1...n) and check Math.abs(nums[i] - nums[j]) == k ?

Example: [1,2]

**We need to be careful here, as just by going (1...n, 1...n) we will count (1,2) and (2,1) which is double counting** (<https://brilliant.org/wiki/double-counting-definition/>). According to the problem statement "pairs (i, j) **where i < j**", so we cannot just returt all |pair| == k.

How do we filter pairs with i < j?

Let's looks at provided in the definition example:



**The first idea coming to your mind might be (you will see later why it's not the one we need):**

i = 0 ... n-1

j = i+1 ... n-1

The double loop will run in n^2 and will work and we are done. Right?

According to the constraints section: **1 <= nums.length <= 200** (4\*10^4, OJ accepts it)

var n = nums.length;

var ans = 0;

for (int i = 0; i < n; i++) {

for (int j = i + 1; j < n; j++) {

if (Math.abs(nums[i] - nums[j]) == k) {

ans++;

}

}

}

return ans;

**Interviewer: Can you do better?** ¯\_(ツ)\_/¯

**Approach #1.5: Another bruteforce**

So what is wrong with the logic above? It works and was accepted by OJ.

It is not wrong, it is just not sutable for the future optimization to O(n) because it leads to nowhere in the current state. Why?

**When we are going to "transition" from n^2 to n we will not have an opportunity to look for "a future" element in the array. Eventually we will have only 1 pass (1 ... n), left-to-right.**

Let's rework approach #1.

The second idea:

i = 1 ... n-1

j = i-1 ... 0

var n = nums.length;

var ans = 0;

for (int i = 1; i < n; i++) {

for (int j = i - 1; j >= 0; j--) {

if (Math.abs(nums[i] - nums[j]) == k) {

ans++;

}

}

}

return ans;

It also works and is accepted by OJ. What did we improve here?

**Now we do not depend on "a future" element and work only with known so far elements.**

**Approach #2: from n^2 to 1-pass n**

Let's start with a few simple examples to demonstrate how absolute difference (AD) works:

Example: **[1,2], k=1**

Let's expand Math.abs(nums[i] - nums[j]) == k for the example above:

According to "101" and when nums[i] > nums[j] we know that |2-1| = 1 expands to 2-1 = 1. k cannot be negative because the distance between (start, end) or (end, start) is always a positive value (you go forward or backward the same distance, just a different direction). So we just drop "|"s and get the following condition:

when nums[i] > nums[j] and (nums[i] - nums[j]) == k. Also in "approach 1.5" **j < i** is always true.

Let's test it:

var n = nums.length;

var ans = 0;

for (int i = 1; i < n; i++) {

for (int j = i - 1; j >= 0; j--) {

if (nums[i] > nums[j] && nums[i] - nums[j] == k) {

ans++;

}

}

}

return ans;

Works fine. So far, so good.

Example: **[2,1], k=1**

Code above does not work for the case. Why?

Because according to our logic when nums[i] > nums[j] and (nums[i] - nums[j]) == k we do not count the pair as a valid one. Remember **k** cannot be negative.

How to resolve the issue?

We know that the distance between |1-2| = 1 is the same as |2-1| = 1. which is when nums[i] < nums[j] and (nums[j] - nums[i]) == k.

Let's add the new logic and test the code for [1,2] and [2,1], k=1:

var n = nums.length;

var ans = 0;

for (int i = 1; i < n; i++) {

for (int j = i - 1; j >= 0; j--) {

if (nums[i] > nums[j] && nums[i] - nums[j] == k) {

ans++;

}

if (nums[i] < nums[j] && nums[j] - nums[i] == k) {

ans++;

}

}

}

return ans;

Now we have the code that works for [1,2] and [2,1].

**Interviewer: Can you do better now?!** (▰˘◡˘▰)

Remember **we will have an opportunity to look for an element from the past** ? Let's utilizae this oportunity for optimization:

**How can we remove the inner loop for (int j = i - 1; j >= 0; j--) ?**

**We can replace it with a HashSet to reduce O(n) traversals to O(1) lookup.**

What should we store in the hashset? What can we store? No nums[j], k is a constant. Left - nums[i], our variable number.

From the 3 variables in the expressions: nums[i] - nums[j] == k and nums[j] - nums[i] == k we know 2 - nums[i] and k.

How to calculate nums[j] ?

Simple math:

nums[i] - nums[j] == k and nums[j] - nums[i] == k

=>

nums[i] - k == nums[j] and nums[j] == k + nums[i]

So all we have to do now is to check if the new conditons above "work for us" and do "ans++".

Let's code it:

var n = nums.length;

var ans = 0;

var set = new HashSet<Integer>();

for (int i = 0; i < n; i++) { // changed from i=1 to i=0 to add the first element to the set

if (set.contains(nums[i] - k)) {

ans++;

}

if (set.contains(k + nums[i])) {

ans++;

}

set.add(nums[i]);

}

return ans;

This one works fine for [1,2], [2,1], [1,2,1], k=1 and is O(n).

Example: **[1,1,2], k=1**

The code above works fine for pairs i < j when there are no duplicate pairs. For the example above the pairs should be: [2,1] and [2,1], but we got only [2,1]. How come? This is because we use the hashset data structure. It keeps only 1 copy of an element.

Okay. How can we keep values with duplicates count? Like (nums[i], count) ? HashMap to the rescue.

Also since now we will need to count duplicates too and thus ans++ will need to be changed to ans += # of duplicates

Let's code it:

var n = nums.length;

var ans = 0;

var counts = new HashMap<Integer, Integer>();

for (int i = 0; i < n; i++) {

if (counts.containsKey(nums[i] - k)) {

ans += counts.get(nums[i] - k);

}

if (counts.containsKey(k + nums[i])) {

ans += counts.get(k + nums[i]);

}

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

}

return ans;

**Further optimizations and code encryption:**

From the problem's constraints section we know that 1 <= nums[i] <= 100

**Thus we can replace the hashmap with an array. The algo looks quite similar to the first part of Counting Sort algo.**

var n = nums.length;

var ans = 0;

var counts = new int[100 + 1];

for (var num : nums) {

if (num > k) ans += counts[num - k]; // [1,2], k=1

if (num + k < 101) ans += counts[k + num]; // [2,1], k=1

counts[num]++;

}

return ans;

As you can see, there is no **magic**.

**Refer to**

<https://algo.monster/liteproblems/2006>

**Problem Description**

The goal of this problem is to find the total number of unique pairs (i, j) in a given integer array nums wherein the absolute difference between the numbers at positions i and j is exactly k. The condition is that i should be less than j, which implies that we're looking at pairs where the first element comes before the second element in the array order. The absolute value here means that if the result of the subtraction is negative, we consider its positive value instead.

To put it simply, we iterate over the array, and for each element, we check how many elements that come after it have a difference of k.

**Intuition**

The solution makes use of a hash map to efficiently track the counts of numbers we have seen so far. This is a common strategy in problems where we need to quickly access the count or existence of elements based on their value, which is often referred to as the frequency map pattern.

When we look at an element num in the array, there are two numbers that could form a valid pair with it: num + k and num - k. For each num, the solution checks if num + k and num - k have been seen before (i.e., they are in the hash map). If they are, it adds the count of how many times they've been seen to our answer because each of those instances forms a valid pair with our current num.

We then update the count of the current num in the hash map, increasing it by 1, to keep track of how many times it has been seen for future iterations.

This approach works because by increasing the count of the current number after checking for pairs, we ensure that we only count pairs where i < j. Hence, we are systematically building and utilizing a frequency map to keep count of potential complements for every element as we iterate through the array.

**Solution Approach**

The solution makes use of the Counter data structure from Python's collections module, which is essentially a hash map (or dictionary) designed for counting hashable objects. The keys in this hash map are the distinct elements from nums and the values are the counts of how many times they appear. Here's how the solution is implemented:

Initialize a variable ans to count the number of valid pairs found. It starts at 0.

Create a Counter object cnt which will store the frequency of each number encountered in nums.

Iterate over each number num in the nums array:

For the current number num, check if num - k is in the counter. If it is, it means there are numbers previously seen that, when subtracted from num, give k. We add the count of num - k to ans.

Similarly, check if num + k is in the counter. If it is, add the count of num + k to ans. This counts the cases where the previous numbers were smaller than num and had a difference of k.

After checking for pairs, increment the count of num in the cnt Counter to account for its occurrence.

After finishing the loop, return the value of ans, which now contains the total number of valid pairs.

The algorithm operates in O(n) time complexity, where n is the number of elements in nums. This is because the operation of checking and updating the counter is O(1), and we only iterate through the array once.

The key algorithms and data structures used in this solution include:

Looping through Arrays: The for loop iterates through each element in nums to check for possible pairs.

Hash Map (Counter): Utilizes the Counter data structure to store and access frequency of elements in constant time (O(1)).

Incremental Counting: Maintains the count of valid pairs in variable ans as the array is processed.

By employing the Counter, we are able to maintain a running total of pair counts as the nums array is iterated over, thus avoiding the need for nested loops that would significantly increase the computational complexity (potential O(n^2) if using brute force approach).

**Example Walkthrough**

Let's assume we are given a small integer array nums = [1, 5, 3, 4, 2] and we must find the number of unique pairs (i, j) such that the absolute difference between nums[i] and nums[j] is k = 2. Following the solution approach:

Initialize ans to 0 as no pairs have been counted yet.

Create a Counter object cnt which is initially empty.

Now, let's iterate over each number num in nums:

For the first number 1, we check if 1 - 2 (which is -1) and 1 + 2 (which is 3) are in the counter. Neither are because the counter is empty, so we don't change ans. Then we add 1 to the counter, so cnt becomes Counter({1: 1}).

Moving to the second number 5, we look for 5 - 2 (equals 3) and 5 + 2 (equals 7). Neither are in the counter, so ans remains 0. Then we update cnt, now Counter({1: 1, 5: 1}).

Next, 3 is checked against the counter. We look for 3 - 2 (which is 1) and 3 + 2 (which is 5). We find 1 in the counter with a count of 1. So we increment ans by 1. We do not find 5 because we only count pairs where i < j, to avoid re-counting. Update the counter with 3, now Counter({1: 1, 5: 1, 3: 1}).

For 4, we do the same. We find 4 - 2 = 2 is not in the counter but 4 + 2 = 6 isn't in the counter either. So, ans is still 1. Update cnt to Counter({1: 1, 5: 1, 3: 1, 4: 1}).

Lastly, for 2, 2 - 2 equals 0 (not present in the counter) but 2 + 2 equals 4 which is in the counter with a count of 1. Thus, we increment ans by 1 making it 2. Final update to the counter leaves it as Counter({1: 1, 5: 1, 3: 1, 4: 1, 2: 1}).

After finishing the iteration, ans is 2, implying there are two unique pairs where the difference is exactly k = 2: these are (1, 3) and (2, 4) based on the original positions in the array (nums[0] and nums[2], nums[4] and nums[3] respectively).

The solution has efficiently counted the pairs without re-counting or using nested loops, showcasing the advantage of using a Counter to keep track of frequencies and significantly simplifying the search process for complements that result in the required difference k.

**Java Solution**

class Solution {

/\*\*

\* Counts the number of unique pairs in the array with a difference of k.

\*

\* @param nums The array of integers to process.

\* @param k The difference to look for between pairs of numbers.

\* @return The count of pairs with the specified difference.

\*/

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

// Initialize answer to 0 to keep count of pairs

int countPairs = 0;

// Array to store counts of each number, considering the constraint 1 <= nums[i] <= 100

int[] countNumbers = new int[110];

// Iterate through each number in the input array

for (int num : nums) {

// If current number minus k is non-negative, add the count of that number to the total

// as it represents a pair where num - (num - k) = k

if (num >= k) {

countPairs += countNumbers[num - k];

}

// If current number plus k is within the allowed range (less than or equal to 100),

// add the count of that number to the total as it represents a pair where (num + k) - num = k

if (num + k <= 100) {

countPairs += countNumbers[num + k];

}

// Increment the count for the current number

++countNumbers[num];

}

// Return total count of pairs

return countPairs;

}

}

**Time and Space Complexity**

The given Python code implements a function countKDifference to count pairs of elements in an array nums that have a difference of k.

**Time Complexity**

The time complexity of the given solution can be analyzed as follows:

The function iterates over each element in the array nums exactly once.

For each element num, it performs a constant-time operation to check and update the counts in the Counter, which is an implementation of a hash map.

Therefore, the time complexity is linear with regard to the number of elements in the list, which is O(n) where n is the length of the nums list.

**Space Complexity**

The space complexity of the solution can be analyzed as follows:

A Counter is used to keep track of the occurrences of each number in the list.

In the worst case, if all elements in the list are unique, the size of the Counter will grow linearly with the number of elements.

Therefore, the space complexity of the solution is O(n) where n is the number of unique elements in nums.

In summary, both the time complexity and the space complexity of the given code are O(n).