<https://leetcode.com/problems/k-diff-pairs-in-an-array/description/>

Given an array of integers nums and an integer k, return **the number of unique k-diff pairs in the array**.

A **k-diff** pair is an integer pair (nums[i], nums[j]), where the following are true:

0 <= i, j < nums.length

i != j

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

**Notice** that |val| denotes the absolute value of val.

**Example 1:**

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

**Output:** 2

**Explanation:** There are two 2-diff pairs in the array, (1, 3) and (3, 5).

Although we have two 1s in the input, we should only return the number of **unique** pairs.

**Example 2:**

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

**Output:** 4

**Explanation:** There are four 1-diff pairs in the array, (1, 2), (2, 3), (3, 4) and (4, 5).

**Example 3:**

**Input:** nums = [1,3,1,5,4], k = 0

**Output:** 1

**Explanation:** There is one 0-diff pair in the array, (1, 1).

**Constraints:**

1 <= nums.length <= 10^4

-10^7 <= nums[i] <= 10^7

0 <= k <= 10^7

**Attempt 1: 2024-01-14**

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

**这道题的难点在于k == 0这个特殊情况需要单独处理，比如当nums = [1,3,1,5,4], k = 0或者nums = [1,3,1,1,5,4], k = 0时，答案都是1，因为[1,1]或者[1,1,1]都被视为唯一满足k = 0的一对，而剩下k != 0的情况就相对简单，在累计过程中会把任何一对都算2次，比如当nums = [1,3,1,5,4], k = 2时，在map处理过后key set是[1,3,1,5,4]，遍历过程中1找到[1,3]和3找到[3,1]会被算作两对，但实际是一对，其他数也同理，所以必须有count / 2的处理**

class Solution {

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

        int count = 0;

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

        for(int num : nums) {

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

        }

        for(int num : map.keySet()) {

            // Special handle for k == 0

            if(k == 0) {

                if(map.get(num) > 1) {

                    count++;

                }

            } else {

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

                    count++;

                }

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

                    count++;

                }

            }

        }

        // When k != 0, all pair count twice, need count/2

        return k == 0 ? count : count >> 1;

    }

}

Time Complexity: O(N)

Space Compleixty: O(N)

**Solution 2: Two Hash Sets (60 min)**

**这个解法的难点在于uniquePairs set中存的一直是一个pair中较小的那个值，比如num - k和num是一个pair，那么就存入num - k，而如果num和num + k是一个pair，那么就存入num，这样统一用一个pair中的较小值代表一个pair，这个uniqueParis set的size就是结果**

class Solution {

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

        int count = 0;

        Set<Integer> visited = new HashSet<>();

        Set<Integer> uniquePairs = new HashSet<>();

        for(int num : nums) {

            // Notice we only add the smaller value of the pair to the ans set

            if(visited.contains(num - k)) {

                uniquePairs.add(num - k);

            }

            if(visited.contains(num + k)) {

                uniquePairs.add(num);

            }

            visited.add(num);

        }

        return uniquePairs.size();

    }

}

Time Complexity: O(N)

Space Compleixty: O(N)

**Refer to**

<https://algo.monster/liteproblems/532>

**Problem Description**

The challenge is to count the distinctive pairs (nums[i], nums[j]) in a given array of integers nums where each pair meets certain criteria. These criteria are that i and j are distinct indices in the array (they are not the same), and the absolute difference between the values at these indices is exactly k. The absolute difference is denoted as |nums[i] - nums[j]| and must equal k. A further constraint is that each pair must be unique; that is, even if multiple pairs have the same numbers, they should only be counted once.

**Intuition**

The solution rests on using a set data structure, which naturally only stores unique elements, thus eliminating duplicate pairs. The main idea is to iterate through nums and at each step determine if there is a corresponding value that, when added or subtracted by k, matches the current element. Two sets are used:

vis (visited): This set keeps track of the elements we have seen so far. As we traverse the array, we add elements to this set.

ans (answer): This set stores the unique elements that form part of a pair with the current element that satisfies the k-diff condition.

For each value v in nums, we perform two checks:

First, we check if v - k is in vis. If it is, it means we've already seen an element which can form a pair with v that has a difference of k (since v - (v - k) = k). We add v - k to the ans set to account for this unique pair.

Secondly, we check if v + k is in vis. If it is, it indicates that v can form a pair with this previously seen element satisfying the k-diff condition. In this case, we add v to the ans set.

After completing the loop, the size of the ans set reflects the total count of unique k-diff pairs, because we've only stored one element from each unique pair, and duplicates are not allowed by the set property. This is the number we return.

**Solution Approach**

The implementation makes use of Python sets and straightforward conditional statements within a loop. Here's a step-by-step breakdown:

We first define two sets: vis to keep track of the integers we have encountered so far as we iterate through the array, and ans to store the unique values that form valid k-diff pairs.

We then enter a loop over each value v in nums. For each v, we perform two important checks:

We check if v - k is in vis. Since set elements are unique, this check is constant time on average, O(1). If this condition is true, it means there is another number in the array such that the difference between it and v is exactly k. **We then add v - k to the ans set, which ensures we're counting the lower number of the pair only once.**

We also check if v + k is in vis for the same reasons as above, but this time if the condition holds true, we add v to the ans set, considering v as the higher number in the pair.

Each iteration also involves adding v to vis set, thus expanding the set of seen numbers and preparing for the subsequent iterations.

After the loop finishes, we have accumulated all unique numbers that can form k-diff pairs in ans. Finally, the solution function returns the size of ans, which is the count of all unique k-diff pairs in the array.

The algorithm's time complexity is O(n) where n is the number of elements in nums, because it goes through the list once and set operations like adding and checking for existence are O(1) on average. The space complexity is also O(n) since at most, the vis and ans sets can grow to the size of the entire array in the worst-case scenario (no duplicates).

**Example Walkthrough**

Let's use the array nums = [3, 1, 4, 1, 5] and k = 2 to illustrate the solution approach:

Initialize two empty sets: vis = set() and ans = set().

Start iterating through the array nums:

Take the first element v = 3. Since vis is empty, there's nothing to check, so simply add 3 to vis.

The next element is v = 1. Check 1 - 2 (which is -1) and 1 + 2 (which is 3) against the vis set. The value 3 is in vis, thus we can form a pair (1, 3). Add 1 to the ans set and 1 to the vis set.

Continue with v = 4, and check 4 - 2 = 2 and 4 + 2 = 6 against vis. Neither is in the set, so simply add 4 to vis.

Now v = 1 again. As 1 is already in vis, no new pairs can be formed that haven't already been counted. Therefore, continue to the next number without making any changes.

Lastly, v = 5. Check 5 - 2 = 3 and 5 + 2 = 7 against vis. The value 3 is there; therefore, the pair (3, 5) can be formed. Add 3 to the ans set.

Final sets after iteration:

vis = {3, 1, 4, 5}

ans = {1, 3}. **Notice we do not have 5 in ans because we only add the smaller value of the pair to the ans set.**

Count the elements in the ans set, which gives us 2. Thus, there are 2 distinct pairs with an absolute difference of 2: these are (1, 3) and (3, 5).

This walkthrough demonstrates the implementation of the solution approach where we end up with the distinct pairs and the count of these unique pairs is the final answer. The time and space complexity for this approach is linear with respect to the number of elements in the nums array.

**Java Solution**

class Solution {

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

// Initialize a hash set to store the unique elements we've seen

Set<Integer> seen = new HashSet<>();

// Initialize a hash set to store the unique pairs we've found

Set<Integer> uniquePairs = new HashSet<>();

// Loop through all elements in the array

for (int num : nums) {

// Check if there's a number in the array such that num - k is already in 'seen'

if (seen.contains(num - k)) {

// If so, add the smaller number of the pair to 'uniquePairs'

uniquePairs.add(num - k);

}

// Check if there's a number in the array such that num + k is already in 'seen'

if (seen.contains(num + k)) {

// If so, add the current number to 'uniquePairs'

uniquePairs.add(num);

}

// Add the current number to the set of seen numbers

seen.add(num);

}

// The number of unique pairs that have a difference of k is the size of 'uniquePairs'

return uniquePairs.size();

}

}

**Time and Space Complexity**

The provided Python code entails iterating through the given list of numbers and checking for the existence of certain values within a set. Here is an analysis of its time complexity and space complexity:

**Time Complexity:**

The time complexity is O(n), where n is the length of the input list nums. This is because the code consists of a single loop that iterates through each element in the list exactly once. The operations within the loop (checking for existence in a set, adding to a set, and adding to another set) all have constant time complexity, i.e., O(1).

**Space Complexity:**

The space complexity of the code is also O(n). Two sets, vis and ans, are used to keep track of the numbers that have been visited and the unique pairs that satisfy the condition, respectively. In the worst-case scenario, the vis set could potentially contain all the elements from the input list nums, resulting in O(n) space usage. The ans set might contain at most min(n, n/2) elements, since it stores the smaller value of each pair, leading to O(n) space requirement as well. Hence, the overall space complexity is O(n) due to these two sets.

Therefore, the complete complexity analysis of the code snippet is O(n) time and O(n) space.

**Solution 3: Hash Set + Binary Search (30 min)**

class Solution {

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

        Arrays.sort(nums);

        Set<Integer> uniquePairs = new HashSet<>();

        for(int i = 0; i < nums.length - 1; i++) {

            if(binarySearch(nums, i + 1, nums.length - 1, nums[i] + k) > 0) {

                // Notice we only add the smaller value of the pair to the ans set

                uniquePairs.add(nums[i]);

            }

        }

        return uniquePairs.size();

    }

    // Find lower boundary

    private int binarySearch(int[] nums, int lo, int hi, int val) {

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

            if(nums[mid] >= val) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        if(lo == nums.length || nums[lo] != val) {

            return -1;

        }

        return lo;

    }

}

Time Complexity: O(N\*logN)

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/k-diff-pairs-in-an-array/solutions/1756933/an-explanation-going-from-o-nlogn-o-n/>

So, this problem is very similar to a very famous problem Two Sum problem. But a slightly different, there we have to only check wether a pair exists or not which has 2 sum equals to the target. But here we have to count those such pairs & only consider the unique one.

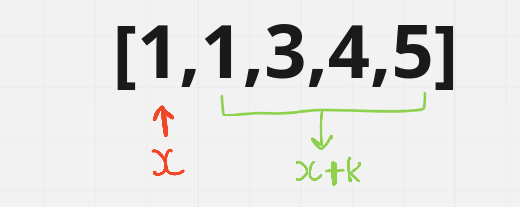
**Okay, so how we will solve this problem?**

One of the first idea came in mind that first-of all we sort this array. Let's take an example :-

**Input:** nums = [3,1,4,1,5], k = 2

**Output:** 2

First we sort this array & it becomes :- [1,1,3,4,5]. After sorting what we will do is, start from the starting place & check the 'x + k' exists on to right place or not!



So, first we sort the array & then we will look for the binary search. There is a method in Java library called **Arrays.binarySearch** & it may be available in C++ & Python. In this method we will pass the array [nums], start index [i + 1], end [n] "size of the array" & the value which we have to search [x + k]. **If we find that we, then we will store minimum value of 'x' into a set. Because we need only the** **Unique Pair**.

So, we will get :- **{1,3} & {3,5}** so these are the 2 pairs we will get.

**Let's look at the code you will understand more clearly then,**

class Solution {

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

Arrays.sort(nums); // sorted the array

Set<Integer> set = new HashSet<>(); // Declare the HashSet to only consider unique one's

int n = nums.length; // length of the array

for(int i = 0; i < nums.length - 1; i++){

// searching for binary index for the no from the i + 1 index to n

// and check if we are getting nums[i] + k, where nums[i] is our 'x'

if(Arrays.binarySearch(nums, i + 1, n, nums[i] + k) > 0){

set.add(nums[i]);

}

}

return set.size();

}

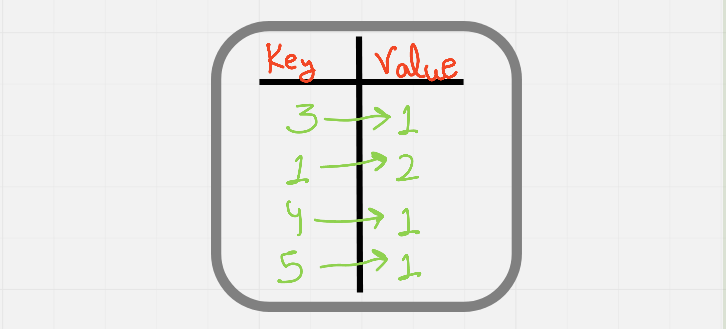
}

**Time Complexity :-** O(NlogN) + O(N \* logN) = O(NlogN) + O(NlogN) = O(2NlogN) = **BigO(NlogN)**

**Now, you we will ask**. Can we further improve it's time complexity? **I'll say yes. Using HashMap.**

Okay, so considering the same example : nums[3,1,4,1,5]

First we will build our HashMap. In Map we will keep the no. as a key & value as a count of occurence



Now, there are 2 cases :-

If k > 0, then in this case we just need to check wether the counter part exists or not. So, if we are iterating 'x + k' in our map, then we can increment our count

If k == 0, then we just need to check if x is more then 1 or not [x > 1] in our map.

But another thing we need to note that here as we iterate from the array after doing counting. Then we will get "1" two times. So, to avoid this instead of iterating over the array, we will iterate over the keyset of this map, which will give us the unique no. i.e. (3,1,4,5).

Alright, so now **I hope approach is clear.**

**Let's code it:**

class Solution {

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

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

for(int num : nums){

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

}

int count = 0;

for(int x : map.keySet()){

if(k > 0 && map.containsKey(x + k) || k == 0 && map.get(x) > 1) count++;

}

return count;

}

}

ANALYSIS :-

Time Complexity :- BigO(N)

Space Complexity :- BigO(N)