532. K-diff Pairs in an Array

July 7, 2020 | 1.4K views

Average Rating: 5 (5 votes)

Given an array of integers and an integer **k**, you need to find the number of **unique** k-diff pairs in the array. Here a **k-diff** pair is defined as an integer pair (i, j), where **i** and **j** are both numbers in the array and their absolute difference is **k**.

Example 1:

```
Input: [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:[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,
```

Example 3:

```
Input: [1, 3, 1, 5, 4], k = 0
Output: 1
Explanation: There is one 0-diff pair in the array, (1, 1).
```

Note:

1. The pairs (i, j) and (j, i) count as the same pair.
2. The length of the array won't exceed 10,000.
3. All the integers in the given input belong to the range: [-1e7, 1e7].

Solution

Overview: Approach 1 exhibits a naive way to tackle this problem by checking all possible pairs. Approach 2 improves the time complexity of approach 1 by using left and right pointers. Approach 3 uses Hashmap and is the fastest of all three approaches.

Approach 1: Brute Force

Intuition

The most naive way to tackle this problem is to sort the array and check every possible pair. We can have two loops, one loop fixing at one number while the other looping going over every number after that fixed number, to check every possible pair. One thing that we have to be aware of is to make sure that we don't repeatedly count the duplicate pairs. To do so, we will have to check whether the current number that we are looking at is the same as the previous number. If the current number is the same as the previous number, whether we are in the outer loop or the inner loop, we can just continue to the next number.

If the difference between the pair that we are looking is the same as `k`, we increment our placeholder variable, `result`.

For `nums = [2,5,1,2,8,1,3,5,7,1]` and `k = 2`:

Implementation

Java Python3 Copy

```
class Solution:
    def findPairs(self, nums, k):

        s_nums = sorted(nums)

        result = 0

        for i in range(len(s_nums)):
            if (i > 0 and s_nums[i] == s_nums[i - 1]):
                continue
            for j in range(i + 1, len(s_nums)):
                if (j > i + 1 and s_nums[j] == s_nums[j - 1]):
                    continue

                if abs(s_nums[j] - s_nums[i] == k):
                    result += 1

        return result
```

Complexity Analysis

- Time complexity : $O(N^2)$ where N is the size of `nums`. The time complexity for sorting is $O(N \log N)$ while the time complexity for going through ever pair in the `nums` is $O(N^2)$. Therefore, the final time complexity is $O(N \log N) + O(N^2) \approx O(N^2)$.

- Space complexity : $O(N)$ where N is the size of `nums`. This space complexity is incurred by the sorting algorithm. Space complexity is bound to change depending on the sorting algorithm you use. There is no additional space required for the part with two `for` loops, apart from a single variable `result`. Therefore, the final space complexity is $O(N) + O(1) \approx O(N)$.

Addendum: We can also approach this problem using brute force without sorting `nums`. First, we have to create a hash set which will record pairs of numbers whose difference is `k`. Then, we look for every possible pair. As soon as we find a pair (say `i` and `j`) whose difference is `k`, we add `(i, j)` and `(j, i)` to the hash set and increment our placeholder `result` variable. Whenever we encounter another pair which is already in the hash set, we simply ignore that pair. By doing so we have a better practical runtime (since we are eliminating the sorting step) even though the time complexity is still $O(N^2)$ where N is the size of `nums`.

Approach 2: Two Pointers

Intuition

We can do better than quadratic runtime in Approach 1. Rather than checking for every possible pair, we can have two pointers to point the left number and right number that should be checked in a sorted array.

First, we have to initialize the left pointer to point the first element and the right pointer to point the second element of `nums` array. The way we are going to move the pointers is as follows:

Take the difference between the numbers which left and right pointers point.

1. If it is less than `k`, we increment the right pointer.
   - If left and right pointers are pointing to the same number, we increment the right pointer too.
2. If it is greater than `k`, we increment the left pointer.
3. If it is exactly `k`, we have found our pair, we increment our placeholder `result` and increment left pointer.

The idea behind the behavior of incrementing left and right pointers in the manner above is similar to:

- Extending the range between left and right pointers when the difference between left and right pointers is less than `k` (i.e. the range is too small).
  - Therefore, we extend the range (by incrementing the right pointer) when left and right pointer are pointing to the same number.
- Contracting the range between left and right pointers when the difference between left and right pointers is more than `k` (i.e. the range is too large).

This is the core of the idea but there is another issue which we have to take care of to make everything work correctly. We have to make sure duplicate pairs are not counted repeatedly. In order to do so, whenever we have a pair whose difference matches with `k`, we keep incrementing the left pointer as long as the incremented left pointer points to the number which is equal to the previous number.

For `nums = [2,5,1,2,8,1,3,5,7,1]` and `k = 2`:

Implementation

Java Python3 Copy

```
class Solution:
    def findPairs(self, nums: List[int], k: int) -> int:

        nums = sorted(nums)

        left = 0
        right = 1

        result = 0

        while (left < len(nums) and right < len(nums)):
            if (left == right or nums[right] - nums[left] < k):
                # List item 1 in the text
                right += 1
            elif nums[right] - nums[left] > k:
                # List item 2 in the text
                left += 1
            else:
                # List item 3 in the text
                left += 1
                result += 1
                while (left < len(nums) and nums[left] == nums[left - 1]):
                    left += 1

        return result
```

Complexity Analysis

- Time complexity : $O(N \log N)$ where N is the size of `nums`. The time complexity for sorting is $O(N \log N)$ while the time complexity for going through `nums` is $O(N)$. One might mistakenly think that it should be $O(N^2)$ since there is another `while` loop inside the first `while` loop. The `while` loop inside is just incrementing the pointer to skip numbers which are the same as the previous number. The animation should explain this behavior clearer. Therefore, the final time complexity is $O(N \log N) + O(N) \approx O(N \log N)$.

- Space complexity : $O(N)$ where N is the size of `nums`. Similar to approach 1, this space complexity is incurred by the sorting algorithm. Space complexity is bound to change depending on the sorting algorithm you use. There is no additional space required for the part where two pointers are being incremented, apart from a single variable `result`. Therefore, the final space complexity is $O(N) + O(1) \approx O(N)$.

Approach 3: Hashmap

Intuition

This method removes the need to sort the `nums` array. Rather than that, we will be building a frequency hash map. This hash map will have every unique number in `nums` as keys and the number of times each number shows up in `nums` as values.

For example:

```
nums = [2,4,1,3,5,3,1], k = 3
hash_map = {1: 2,
            2: 1,
            3: 2,
            4: 1,
            5: 1}
```

Next, we look at a key (let's call `x`) in the hash map and ask whether:

- There is a key in the hash map which is equal to `x+k` **IF** `k > 0`.
  - For example, if a number in `nums` is 1 `(x=1)` and `k` is 3, you would need to have 4 to satisfy this condition (thus, we need to look for `1+3 = 4` in the hash map). Using addition to look for a complement pair has the advantage of not double-counting the same pair, but in reverse order (i.e. if we have found a pair (1,4), we won't be counting (4,1)).
- There is more than one occurrence of `x` **IF** `k = 0`.
  - For example, if we have `nums = [1,1,1,1]` and `k = 0`, we have one unique (1,1) pair. In this case, our hash map will be `{1: 4}`, and this condition is satisfied since we have more than one occurrence of number `1`.

If we can satisfy either of the above conditions, we can increment our placeholder `result` variable.

Then we look at the next key in the hash map.

Implementation

Java Python3 Copy

```
from collections import Counter

class Solution:
    def findPairs(self, nums, k):
        result = 0

        counter = Counter(nums)

        for x in counter:
            if k > 0 and x + k in counter:
                result += 1
            elif k == 0 and counter[x] > 1:
                result += 1
        return result
```

Complexity Analysis

- Time complexity : $O(N)$ where N is numbers in `nums`. It takes $O(N)$ to create an initial frequency hash map and another $O(N)$ to traverse the keys of that hash map. One thing to note about is the hash key lookup. The time complexity for hash key lookup is $O(1)$ but if there are hash key collisions, the time complexity will become $O(N)$. However those cases are rare and thus, the amortized time complexity is $O(2N) \approx O(N)$.

- Space complexity : $O(M)$ where M is the number of unique numbers in `nums`.

Rate this article:

Comments: 4 Sort By

Type comment here... (Markdown is supported)

Preview Post

LeetCoding_Master 195 2 days ago

Since k is absolute difference, just return 0 when k < 0;

0 Share Reply

anarky24 36 2 days ago

Surprisingly, it took me longer than expected to solve this through two pointers.

0 Share Reply

jyotiprakashrout434 24 July 10, 2020 1:08 AM

My Intuitive Solution :

```
class Solution {
    public int findPairs(int[] nums, int k) {
```