532. K-diff Pairs in an Array

Medium | Array | Hash Table | Two Pointers | Binary Search | Sorting

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:

1. `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.
2. `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:
  1. 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.
  2. 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:

1. Initialize two empty sets: `vis = set()` and `ans = set()`.
2. 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.
3. 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.
4. 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.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def findPairs(self, nums: List[int], k: int) -> int:
        # Set to keep track of unique numbers seen so far.
        visited = set()

        # Set to keep track of unique pairs that satisfy the condition.
        # Pairs are identified by the smaller number in the pair.
        unique_pairs = set()

        for number in nums:
            # If the current number minus k was already seen,
            # add the smaller number of the pair (number - k) to the set of unique pairs.
            if number - k in visited:
                unique_pairs.add(number - k)

            # If the current number plus k was already seen,
            # add the current number to the set of unique pairs as it
            # represents the smaller number in the pair (current number, number + k).
            if number + k in visited:
                unique_pairs.add(number)

            # Mark the current number as seen.
            visited.add(number)

        # The number of unique pairs is the size of the set.
        return len(unique_pairs)

# Example usage:
# sol = Solution()
# print(sol.findPairs([3, 1, 4, 1, 5], 2))  # Output: 2
```

Java

```java
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();
    }
}
```

C++

```cpp
class Solution {
public:
    int findPairs(vector<int>& nums, int k) {
        unordered_set<int> visited; // Set to keep track of visited numbers
        unordered_set<int> foundPairs; // Set to store unique pairs that satisfy the condition

        for (int& number : nums) { // Iterate over each number in the given array
            // Check if there's a number in visited set such that the difference between
            // the current number and that number equals k
            if (visited.count(number - k)) {
                // If such a number is found, insert the smaller number of the pair into foundPairs
                foundPairs.insert(number - k);
            }

            // Check if there's a number in visited set such that the difference between
            // that number and the current number equals k
            if (visited.count(number + k)) {
                // If such a number is found, insert the current number into foundPairs
                foundPairs.insert(number);
            }

            // Mark the current number as visited
            visited.insert(number);
        }

        // The result is the number of unique pairs found, which is the size of foundPairs set
        return foundPairs.size();
    }
};
```

TypeScript

```typescript
let visited = new Set<number>(); // Set to keep track of visited numbers
let foundPairs = new Set<number>(); // Set to store unique indices whose elements satisfy the condition

function findPairs(nums: number[], k: number): number {
  visited.clear(); // Clear sets before use
  foundPairs.clear();

  // Iterate over each number in the given array nums
  nums.forEach((number) => {
    // Check if there is a number in the visited set such that
    // the difference between the current number and that number equals k
    if (visited.has(number - k)) {
      // If such a number is found, insert the smaller number of the pair into foundPairs
      foundPairs.add(number - k);
    }

    // Check if there is a number in the visited set such that
    // the difference between that number and the current number equals k
    if (visited.has(number + k)) {
      // If such a number is found, insert the current number into foundPairs
      foundPairs.add(number);
    }

    // Mark the current number as visited
    visited.add(number);
  });

  // The result is the number of unique pairs found, which is the size of the foundPairs set
  return foundPairs.size();
}

// Example usage:
// let numPairs = findPairs([3, 1, 4, 1, 5], 2);
// console.log(numPairs); // Should log the number of unique pairs found with difference k
```


```python
from typing import List

class Solution:
    def findPairs(self, nums: List[int], k: int) -> int:
        # Set to keep track of unique numbers seen so far.
        visited = set()

        # Set to keep track of unique pairs that satisfy the condition.
        # Pairs are identified by the smaller number in the pair.
        unique_pairs = set()

        for number in nums:
            # If the current number minus k was already seen,
            # add the smaller number of the pair (number - k) to the set of unique pairs.
            if number - k in visited:
                unique_pairs.add(number - k)

            # If the current number plus k was already seen,
            # add the current number to the set of unique pairs as it
            # represents the smaller number in the pair (current number, number + k).
            if number + k in visited:
                unique_pairs.add(number)

            # Mark the current number as seen.
            visited.add(number)

        # The number of unique pairs is the size of the set.
        return len(unique_pairs)

# Example usage:
# sol = Solution()
# print(sol.findPairs([3, 1, 4, 1, 5], 2))  # Output: 2
```


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.