2200. Find All K-Distant Indices in an Array

Easy Array Leetcode Link

Problem Description

You're provided with an array `nums` and two integers `key` and `k`. The task is to find all indices in the array that are within a distance of `k` from any occurrence of `key` in the array. An index `i` is considered *k-distant* if there is at least one occurrence of `key` at index `j` such that the absolute difference between `i` and `j` is less than or equal to `k`. The challenge is to return all such *k-distant* indices in ascending order.

The problem focuses on finding these indices efficiently and ensuring that the distance condition (`|i - j| <= k`) is met for each index related to the `key` value. In essence, you're creating a list of indices where each index is not too far from any position in the array that contains the key you're interested in.

Intuition

To solve this problem, one straightforward approach is to check each index and compare it with every other index to determine if it meets the k-distance condition with respect to the `key`.

Here are the steps involved in this approach:

1. Loop through each index `i` of the array.
2. For each index `i`, loop through the entire array again and check every index `j`.
3. As soon as you find an index `j` where `nums[j]` equals `key` and `|i - j| <= k`, you know that `i` is a valid k-distant index.
4. Add the index `i` to the answer list (`ans`) as soon as the condition is satisfied and stop the inner loop to avoid duplicates.
5. Once the loops complete, you will have a list of all k-distant indices.
6. Since we begin the search from the start of the array and never revisit indices we've already determined to be k-distant, the resulting list is naturally sorted in increasing order.

The intuition behind this method relies on a brute-force strategy, checking every possible pair to ensure no potential k-distant index is missed. The `break` statement after adding an index to `ans` ensures we're not adding the same index multiple times.

Solution Approach

The Reference Solution Approach provided above implements a brute-force method to identify all k-distant indices. The algorithm does not use any complex data structures or patterns but a simple approach to comparing indices with straightforward nested for-loops. Here's an explanation of how the implementation works:

1. We start by initializing an empty list `ans` that will store our final list of k-distant indices.

2. We determine the length of the `nums` array and store it in variable `n`, which is used to control the loop boundaries.

3. A for-loop runs with `i` ranging from 0 to `n - 1`, iterating over each index in the array. For each iteration (for each index `i`), we want to check if it is a k-distant index.

4. Inside the outer loop, another for-loop runs with `j` ranging from 0 to `n - 1`. This inner loop scans the entire array to check for occurrences of `key`.

5. For every index `j`, if `nums[j]` equals `key` and the absolute difference between `i` and `j` (`abs(i - j)`) is less than or equal to `k`, we've found that index `i` is k-distant from an occurrence of `key`.

6. As soon as the condition is met (`nums[j] == key` and `abs(i - j) <= k`), we append the current index `i` to the `ans` list, ensuring that each index is only added once due to the `break` statement that follows the append operation. The `break` ensures the algorithm moves to the next index `i` once it confirms that `i` is k-distant.

7. After both loops have completed their execution, the list `ans` contains all of the k-distant indices. The outer loop's sequential nature guarantees that `ans` is sorted in increasing order, as indices are checked starting from the smallest `i` to the largest.

This solution has a time complexity of `O(n^2)` due to the nested for-loops each ranging from 0 to `n-1`. There is no additional space complexity aside from the output array `ans`, and thus the space complexity is `O(n)` in the worst case, where `n` is the number of k-distant indices found.

While this brute-force method guarantees a correct solution by exhaustively checking all conditions, it may not be the most efficient for larger arrays due to its quadratic time complexity. Optimizations can be considered using different algorithms or data structures if efficiency is critical.

Example Walkthrough

To illustrate the solution approach, let's consider an example. Suppose we have the array `nums = [1, 2, 3, 2, 4]`, with `key = 2` and `k = 2`.

We are tasked with finding all indices in the array that are at most 2 steps away from any occurrence of the number 2. Following the steps of the solution approach:

1. We start with an empty list `ans`.
2. The length of `nums` is 5 (n = 5).
3. We start an outer loop with `i` ranging from 0 to 4 (since n - 1 = 4).
4. Now, for each `i`, we will check every other index `j` for an occurrence of `key` that is k-distant.

Let's see how this unfolds step by step:

- For `i = 0`, we check every `j`. We find that `j = 1` satisfies `nums[j] == key` and `abs(i - j) <= k` (1 - 0 <= 2), so we add `i = 0` to `ans` and break the inner loop.
- For `i = 1`, `key` is present at this same index `j = 1` and `abs(i - j) = 0` which is within k distance, so we add `i = 1` to `ans` and break the inner loop.
- For `i = 2`, the next occurrence of `key` is at `j = 1`. The condition `nums[1] == key` and `abs(2 - 1) <= k` is true, so add `i = 2` to `ans` and break the inner loop.
- For `i = 3`, we find that `j = 3` satisfies `nums[j] == key` as well, and `abs(i - j) = 0` is within k distance, so we add `i = 3` to `ans` and break the inner loop.
- For `i = 4`, the closest key is at `j = 3`, but `abs(4 - 3) <= k` is true, so we also add `i = 4` to `ans` and break the inner loop.

After the loops complete, our `ans` list contains `[0, 1, 2, 3, 4]`. Each index is within a distance of `k` from an occurrence of the `key`. Thus, we have successfully found all k-distant indices using the brute-force approach.

Python Solution

```python
class Solution:
    def find_k_distant_indices(self, nums: List[int], key: int, k: int) -> List[int]:
        # Initialize an empty list to store the answer
        result_indices = []
        # Get the length of the input list 'nums'
        num_count = len(nums)
        # Loop over each element in 'nums'
        for i in range(num_count):
            # Loop over the elements in 'nums' again for each 'i'
            for j in range(num_count):
                # Check if the current element is the 'key' and its index 'j' is within 'k' distance from 'i'
                if abs(i - j) <= k and nums[j] == key:
                    # If the condition is met, add the index 'i' to 'result_indices'
                    result_indices.append(i)
                    # Stop checking other 'j's for the current 'i' as we've found a qualifying 'j'
                    break
        # Return the list of indices satisfying the condition
        return result_indices

```

Java Solution

```java
class Solution {
    public List<Integer> findKDistantIndices(int[] nums, int key, int k) {
        // The length of the array 'nums'
        int n = nums.length;
        // Initialize the list to store the answer
        List<Integer> kDistantIndices = new ArrayList<>();
        // Iterate over all elements of 'nums'
        for (int i = 0; i < n; ++i) {
            // Check elements again to find indices within distance 'k' of 'key' in 'nums'
            for (int j = 0; j < n; ++j) {
                // If the absolute difference between indices 'i' and 'j' is less than or equal to 'k'
                // and the current element nums[j] is equal to 'key', the condition is met
                if (Math.abs(i - j) <= k && nums[j] == key) {
                    // Add the current index 'i' to the list of results
                    kDistantIndices.add(i);
                    // Break from the inner loop since we've found the key at this 'i' index
                    break;
                }
            }
        }
        // Return the list of indices that are within distance 'k' from the elements equal to 'key'
        return kDistantIndices;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <cstdlib> // Include for std::abs

class Solution {
public:
    // Function to find all indices within 'k' distance from elements equal to 'key'
    vector<int> findKDistantIndices(vector<int>& nums, int key, int k) {
        int n = nums.size(); // Get the size of the input vector
        vector<int> resultIndices; // Vector to store result indices

        // Iterate over all elements in 'nums'
        for (int i = 0; i < n; ++i) {

            // Check the distance of current 'i' to all elements in 'nums'
            for (int j = 0; j < n; ++j) {

                // If the absolute difference between indices 'i' and 'j' is less than or equal to 'k'
                // and the element at 'j' is equal to 'key'
                if (std::abs(i - j) <= k && nums[j] == key) {
                    resultIndices.push_back(i); // Add 'i' to the result indices
                    break; // Stop inner loop since 'i' is within the distance 'k' from an element equal to 'key'
                }
            }
        }

        return resultIndices; // Return the vector with the result indices
    }
};

```

Typescript Solution

```typescript
function findKDistantIndices(nums: number[], key: number, k: number): number[] {
    const numsLength = nums.length; // Holds the length of the nums array
    let distantIndices = []; // Array to store the result

    // Iterate over each element in nums array
    for (let index = 0; index < numsLength; index++) {
        // Check if the current element is equal to the key
        if (nums[index] === key) {
            // For each element matching the key, compute the range of indices within k distance
            for (let i = index - k; i <= index + k; i++) {
                // Ensure the computed index is within array bounds and not already included in the result
                if (i >= 0 && i < numsLength && !distantIndices.includes(i)) {
                    distantIndices.push(i); // Add the index to the result
                }
            }
        }
    }

    return distantIndices; // Return the array of k-distant indices
}

```

Time and Space Complexity

Time Complexity

The given code consists of two nested loops, each iterating over the array `nums` which has `n` elements.

- The outer loop runs `n` times for every element `i` in the array `nums`.
- For each iteration of `i`, the inner loop also checks every element `j` over the entire array.

When checking the condition `if abs(i - j) <= k and nums[j] == key:`, it performs constant time checks which can be considered as `O(1)`.

Hence, the time complexity of the code is `O(n^2)` since for each element of `nums`, the code iterates over the entire `nums` again.

Space Complexity

The space complexity of the algorithm comes from the list `ans` which stores the indices. In the worst case, where the condition `abs(i - j) <= k and nums[j] == key` is true for every element, `ans` could have the same length as the number of elements in `nums`. This means that it could potentially store `n` indices. Therefore, the space complexity is `O(n)`.