219. Contains Duplicate II

Easy | Array | Hash Table | Sliding Window

Problem Description

The goal is to find out whether there is a pair of indices `i` and `j` in the given integer array `nums` such that `nums[i]` is equal to `nums[j]` and the absolute difference between `i` and `j` is less than or equal to `k`. This problem is checking for duplicates within a specified range of indices in an array. The crucial point here is that `i` and `j` must be distinct, which means you cannot compare an element with itself. If such a pair exists, then the correct output is `true`; otherwise, `false`.

Intuition

The intuitive approach for this problem uses a hash table to track the indices of elements we've already seen. As we iterate through the `nums` array, we keep updating the hash table with the elements as keys and their indices as values. At each step, we look at the current element `x` and check if it's already in the hash table. If it is, and the stored index is close enough to the current index (the difference is less than or equal to `k`), then we've found the required pair, and we can immediately return `true`. If there's no such element or the distance is too large, we simply keep the hash table updated by overwriting the index of the current element. After checking all elements, if no such pair is found, the function returns `false`.

This approach is efficient as it only requires a single pass through the array, and the lookups and updates in the hash table are performed in constant time. Hence, the total time complexity is O(n), where n is the number of elements in the `nums` array.

Solution Approach

The algorithm utilizes a hash table to implement the solution efficiently. The hash table stores elements from the array `nums` as keys, and the most recent index at which each element occurs as the corresponding value.

We initiate the iteration through the array with a `for` loop. At each step, we consider the current element and its index (`i, x` fetched from `enumerate(nums)`).

Here's the step-by-step implementation:

1. If the current element `x` already exists in the hash table `d` and the difference between the current index `i` and the previously stored index for `x` (`d[x]`) is less than or equal to `k`, then we have found two indices `i` and `j` (where `j` is `d[x]`) that satisfy both conditions: `nums[i] == nums[j]` and `abs(i - j) <= k`. In this case, the function immediately returns `true`.
2. If the current element `x` does not satisfy the condition, or if it is not present in the hash table, it means we have not yet found a duplicate within the specified range. Therefore, we update the hash table with the current index `i` for element `x`: `d[x] = i`. This step is crucial because it keeps the hash table updated with the latest index where each number is found, ensuring that when checking for the range condition, we are always using the smallest possible index difference.
3. After iterating through all the elements in the array, if we have not returned `true`, it means no two indices satisfy the given conditions, and the function concludes by returning `false`.

In summary, the hash table is essential to keep track of the indices of the elements we've seen, enabling us to check the existence of nearby duplicates in constant time. This approach leads to a linear time complexity, which is O(n), with n being the length of the array `nums`.

Example Walkthrough

Let's take a small example to better understand the solution approach. Consider the array `nums = [1, 2, 3, 1, 2, 3]` and let's say `k = 2`. We want to find if there are any duplicates within an index range of 2.

1. We start with an empty hash table `d = {}`.
2. We iterate through the nums array with index `i` and element `x`.
   a. At `i = 0`, `x = 1`. We add `1` to the hash table with its index: `d = {1: 0}`.
   b. At `i = 1`, `x = 2`. The hash table is now: `d = {1: 0, 2: 1}`.
   c. At `i = 2`, `x = 3`. The hash table updates to: `d = {1: 0, 2: 1, 3: 2}`.
   d. At `i = 3`, `x = 1`. The current element `1` is already in the hash table. We check the index of the last occurrence of `1` in `d`, which is `0`. The difference between the current index `3` and the last index `0` is 3 which is not less than or equal to `k`. We update `d` with the new index for `1`: `d = {1: 3, 2: 1, 3: 2}`.
   e. At `i = 4`, `x = 2`. Again, `2` is already in the hash table. The last occurrence was at index `1`. The difference between `i = 4` and `1` is 3, which again, is not less than or equal to `k`. We update `d`: `d = {1: 3, 2: 4, 3: 2}`.
   f. At `i = 5`, `x = 3`. The `3` is found in the hash table at index `2`. The difference between `i = 5` and `2` is 3, which is also not less than or equal to `k`. Update `d`: `d = {1: 3, 2: 4, 3: 5}`.
3. After completing the iteration, we have not found any elements where the absolute difference in their indices was less than or equal to `k`. So, the function returns `false`.

Through this example, it's clear how the hash table `d` is used to keep track of the indices of elements. Despite encountering duplicates, the conditions for `k` were not satisfied, which the algorithm correctly identified.

Solution Implementation

Python

```
class Solution:
    def containsNearbyDuplicate(self, nums: List[int], k: int) -> bool:
        # Initialize a dictionary to store the value as key and its index as value
        index_map = {}

        # Enumerate over the list to have both index and value
        for index, value in enumerate(nums):
            # If the value is in index_map, and the current index minus the
            # stored index is less than or equal to k, a nearby duplicate exists
            if value in index_map and index - index_map[value] <= k:
                return True
            # Update the index value in the index map for each value
            # It ensures that if the same value comes up again, the index_map stores
            # the latest index, which is useful for distance calculation
            index_map[value] = index

        # Return False if no nearby duplicates found within the given k distance
        return False
```

To ensure Pvthon 3 svntax. especiallv for tvpina:
- Make sure `List` is imported from `typing` module, which will be used for the `type` hint of the `nums` argument.

Here is how you would include that import:

```python
from typing import List

class Solution:
    # the rest of the code remains the same as above
```

Java

```
import java.util.HashMap;
import java.util.Map;

class Solution {

    public boolean containsNearbyDuplicate(int[] nums, int k) {
        // Initialize a HashMap to store the value and its most recent index
        Map<Integer, Integer> indexMap = new HashMap<>();

        for (int currentIndex = 0; currentIndex < nums.length; ++currentIndex) {
            // Use getOrDefault to find the last index of the current number.
            // If it's not found, use a default value that is far away from any possible index.
            int lastIndex = indexMap.getOrDefault(nums[currentIndex], -1000000);

            // Check if the current index and the last index of the same value are within k steps
            if (currentIndex - lastIndex <= k) {
                // If so, we found a nearby duplicate and return true.
                return true;
            }

            // Update the map with the current value and its index for the next iteration checks
            indexMap.put(nums[currentIndex], currentIndex);
        }

        // If no nearby duplicates are found, return false.
        return false;
    }
}
```

C++

```
#include <vector>
#include <unordered_map>
using namespace std;

class Solution {
public:
    bool containsNearbyDuplicate(vector<int>& nums, int k) {
        // Create a hashmap to store the most recent index of each value observed
        unordered_map<int, int> valueToIndexMap;

        for (int i = 0; i < nums.size(); ++i) {
            // Check if the current value exists in the map, i.e., it has been encountered before
            if (valueToIndexMap.count(nums[i])) {
                // If the previous occurrence is within k indices from the current index, return true
                if (i - valueToIndexMap[nums[i]] <= k) {
                    return true;
                }
            }
            // Update the hashmap with the current index for this value
            valueToIndexMap[nums[i]] = i;
        }

        // If no duplicate elements are within the given k indices, return false
        return false;
    }
};
```

TypeScript

```
// This function checks if there are any duplicates within 'k' indices apart in the array 'nums'
function containsNearbyDuplicate(nums: number[], k: number): boolean {
    // Create a map to keep track of the numbers and their indices
    const indexMap: Map<number, number> = new Map();

    // Iterate through the 'nums' array
    for (let index = 0; index < nums.length; ++index) {
        // Check if the current number is in the map and if the difference between
        // the indices is less than or equal to 'k'
        if (indexMap.has(nums[index]) && index - indexMap.get(nums[index])! <= k) {
            return true; // Duplicate found within 'k' distance
        }

        // Update the index of the current number in the map
        indexMap.set(nums[index], index);
    }

    // No duplicate found within 'k' distance
    return false;
}
```

```
class Solution:
    def containsNearbyDuplicate(self, nums: List[int], k: int) -> bool:
        # Initialize a dictionary to store the value as key and its index as value
        index_map = {}

        # Enumerate over the list to have both index and value
        for index, value in enumerate(nums):
            # If the value is in index_map, and the current index minus the
            # stored index is less than or equal to k, a nearby duplicate exists
            if value in index_map and index - index_map[value] <= k:
                return True
            # Update the index value in the index map for each value
            # It ensures that if the same value comes up again, the index_map stores
            # the latest index, which is useful for distance calculation
            index_map[value] = index

        # Return False if no nearby duplicates found within the given k distance
        return False
```

To ensure Pvthon 3 svntax. especiallv for tvpina:
- Make sure `List` is imported from `typing` module, which will be used for the `type` hint of the `nums` argument.

Here is how you would include that import:

```python
from typing import List

class Solution:
    # the rest of the code remains the same as above
```

Time and Space Complexity

The time complexity of the provided code is `O(n)` because the code uses a single loop that iterates over the elements in 'nums' once. Inside the loop, the code performs constant time operations including checks in a dictionary and assignment.

The space complexity is also `O(n)` as the code allocates a dictionary 'd' that could potentially store all the unique elements of 'nums' if there are no nearby duplicates. In the worst case, the dictionary will have as many entries as there are elements in 'nums'.