1426. Counting Elements

Easy Array Hash Table

Leetcode Link

Problem Description

The problem provides an integer array called `arr`. We are asked to count the number of elements, `x`, in this array such that there exists another element in the array which is exactly one more than `x` (that is, `x + 1`). If there are multiple instances of the same value in `arr`, each occurrence should be considered separately for our count.

An example to illustrate the problem could be if we have the array `[1, 2, 3]`. Here, the element `1` has a companion `2`, and `2` has a companion `3`. Thus, we have two elements (`1` and `2`) that meet the condition, so our result would be `2`.

Intuition

To find an efficient solution to the problem, we realize that checking for the presence of `x + 1` for each element `x` can be made faster by using a set. A set is a data structure that allows for O(1) look-up times on average, meaning we can quickly determine if `x + 1` exists in our array.

Here's how we can break down the solution:

1. **Create a Set**: Convert the list `arr` into a set `s`. This allows for rapid look-ups and also removes any duplicate values, which we don't need to consider since we're counting duplicates separately anyway.
2. **Count with a Condition**: We go through each element `x` in the original array `arr` and check if `x + 1` exists in our set `s`.
3. **Sum the Counts**: By summing the boolean results of the check (since `True` equates to `1` and `False` to `0` in Python), we get a count of how many times the condition is met across our array.

The solution's beauty lies in its simplicity and efficiency, transforming the problem into a series of O(1) look-up operations that result in the final count.

Solution Approach

The Reference Solution Approach uses simple yet effective programming techniques and takes advantage of Python's built-in data structures.

Here's a step-by-step explanation of how the code works:

1. **Convert to Set**: The first step in the `countElements` method involves creating a set `s` from the input list `arr`.

   ```
   s = set(arr)
   ```

   Sets in Python are implemented as hash tables, which is why we get excellent average-case time complexity for look-up operations.

2. **Iteration and Element Check**: Next, we iterate over the elements in the original array `arr`. For each element `x`, we check if `x + 1` is present in the set `s`.

   ```
   sum(x + 1 in s for x in arr)
   ```

   The expression `x + 1 in s` is a boolean check that returns `True` if the element `x + 1` exists in the set `s`, and `False` otherwise.

3. **Summing the True Counts**: The `sum` function in Python adds up all the items in an iterable. Since `True` is counted as `1` and `False` as `0`, this line effectively counts all instances where `x + 1` is found in the set:

   ```
   return sum(x + 1 in s for x in arr)
   ```

   Each time the check is `True`, it adds `1` to our running total. When it's `False`, it adds nothing. The final result is the number of elements that meet the condition, which the function returns.

The algorithm could be classified as a counting algorithm, which is commonly used to solve problems where we are asked to count occurrences of certain conditions or items that meet specific criteria.

In summary, the solution hinges on the constant-time look-up properties of sets in Python and the use of a generator expression within the `sum` function to iterate and tally our count efficiently.

Example Walkthrough

Let us consider a small example to illustrate the solution approach. Suppose we have an array `arr = [3, 1, 2, 3]`. We want to count the number of elements in this array for which the element plus one also exists.

1. First, we convert the input list `arr` into a set `s`.

   ```
   s = set(arr)  # Now s is {1, 2, 3}
   ```

   By converting the list into a set, we can check for the existence of `x + 1` in constant time.

2. We now iterate over each element `x` in the original array `arr` and use a boolean check to see if `x + 1` exists in the set `s`. For `x = 3`: `x + 1` is `4`, which does not exist in set `s`. For `x = 1`: `x + 1` is `2`, which exists in set `s`. For `x = 2`: `x + 1` is `3`, which exists in set `s`. For the second occurrence of `x = 3`, the result is the same as the first time: `x + 1` is `4`, which does not exist in set `s`.

3. We sum the boolean results:

   ```
   return sum(x + 1 in s for x in arr)  # Evaluates to sum([False, True, True, False]) which is 0 + 1 + 1 + 0 = 2
   ```

   Each `True` result adds `1` to the count. In this case, there are two instances where `x + 1` exists in our set; hence, the sum is `2`.

Given this example, the function `countElements` would return `2`, as `1` and `2` have companions `2` and `3`, respectively. Each occurrence is treated separately, even though `3` appears twice, since `3 + 1` is not in the array, those occurrences do not contribute to our count.

Python Solution

```python
class Solution:
    def countElements(self, nums: List[int]) -> int:
        # Create a set from the list for O(1) lookups
        unique_elements = set(nums)

        # Count the elements that have an immediate consecutive successor in the set
        count = sum(1 for num in nums if num + 1 in unique_elements)

        # Return the total count of such elements
        return count
```

Java Solution

```java
class Solution {
    public int countElements(int[] arr) {
        // A set to store the unique elements from the array
        Set<Integer> uniqueElementsSet = new HashSet<>();

        // Adding each element from the array to the set
        // Duplicate elements will not be added to a set, ensuring uniqueness
        for (int num : arr) {
            uniqueElementsSet.add(num);
        }

        // Counter to keep track of the number of elements that satisfy the condition
        int count = 0;

        // Iterating through the array to check if the element's consecutive number is in the set
        for (int num : arr) {
            if (uniqueElementsSet.contains(num + 1)) {
                // Increment the counter if the set contains the consecutive number
                ++count;
            }
        }

        // Returning the count of elements that satisfy the condition
        return count;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <unordered_set>
using namespace std;

class Solution {
public:
    int countElements(vector<int>& arr) {
        // Create an unordered_set to hold unique elements for fast lookup.
        unordered_set<int> uniqueElements(arr.begin(), arr.end());

        int count = 0; // Initialize the counter for the elements that match the criteria.

        // Iterate over the array to count elements such that the element + 1 is also in the array.
        for (int element : arr) {
            // If the set contains (element+1), increment the counter.
            count += uniqueElements.count(element + 1);
        }

        return count; // Return the total count of elements meeting the criteria.
    }
};
```

Typescript Solution

```typescript
/**
 * Function to count the elements in an array where each element `x` has another element `x+1`.
 * @param arr - An array of number elements.
 * @return The count of elements that satisfy the condition.
 */
function countElements(arr: number[]): number {
    // Initialize a new Set to store unique elements.
    const uniqueElements = new Set<number>();

    // Add each element of the array to the Set to ensure uniqueness.
    for (const element of arr) {
        uniqueElements.add(element);
    }

    // Initialize the answer variable to store the final count.
    let count: number = 0;

    // Iterate over the array to find elements where `element + 1` exists in the Set.
    for (const element of arr) {
        if (uniqueElements.has(element + 1)) {
            // Increment the count if such an element is found.
            count++;
        }
    }

    // Return the final count.
    return count;
}

// Example usage in TypeScript:
// const result = countElements([1, 2, 3]); // result should be 2
```

Time and Space Complexity

The provided code snippet defines a function `countElements` that counts the number of elements in an array where the element plus one is also in the array.

Time Complexity

The time complexity of the function is `O(n)`. This efficiency is due to two separate operations that each run in linear time relative to the number of elements, `n`, in the input array `arr`:

1. `s = set(arr)`: Transforming the list to a set has a time complexity of `O(n)` because each of the `n` elements must be processed and inserted into the set.
2. `sum(x + 1 in s for x in arr)`: The sum function iterates over all elements `x` in `arr` once, performing a constant-time check `x + 1 in s` to see if `x + 1` exists in the set. Since set lookups are `O(1)`, and there are `n` elements, this operation is also `O(n)`.

Therefore, the overall time complexity is the sum of the two operations, which remains `O(n)`.

Space Complexity

The space complexity of the function is `O(n)`. This is determined by the size of the set `s` that is created from the input array. In the worst case, if all elements in `arr` are unique, the set will contain `n` elements, leading to a space complexity of `O(n)`.