1437. Check If All 1's Are at Least Length K Places Away

Easy Array Leetcode Link

Problem Description

The problem provides us with a binary array called `nums`, which means it only contains `0`s and `1`s. We are also given an integer `k`. The task is to determine if all the `1`s in the array are separated by at least `k` places from each other. Put differently, after any `1` in the array, the next `1` should not appear before we have seen at least `k` number of `0`s. If this condition is met for all `1`s in the array, we should return `true`. Otherwise, the function should return `false`.

For instance, if we have `nums = [1,0,0,0,1]` and `k = 2`, the function should return `true` since the two `1`s are separated by three `0`s which is at least `k` places apart. Conversely, if `k` were `3`, the function should return `false` since the `1`s are not separated by at least `3` places.

Intuition

The intuition behind the solution is to scan through the given array while keeping track of the position of the last `1` we encountered. As we iterate over the array, whenever we find a `1`, we check if there was any previous `1`. If there was, we calculate the distance between the current `1` and the last `1`. This distance must be greater than or equal to `k+1` places (since we start counting from 0), indicating at least `k` `0`s are in between them. If this condition is not met, then we return `false` immediately, as we have found two `1`s that are too close to each other. If no such pair of `1`s violating our separation condition is found by the end of the array, we can safely return `true`.

To implement this, we initialize a variable `j` with a very small number (`-inf`, negative infinity) to represent the position of the last `1` we've seen. We use this extreme initial number to handle the case where the first `1` appears at the start of the array; since there is no other `1` before it, the algorithm should not incorrectly flag it as being too close to a previous `1`. As we iterate through the array with index `i`, if we find a `1`, we check if the distance `i - j - 1` is less than `k`. If it is, it means that `1`s are not `k` places apart, and we should return `false`. If the condition is not violated, we update `j` to the current index `i`. If we reach the end of the array without finding poorly spaced `1`s, we return `true`.

Solution Approach

The solution uses a simple linear scan approach to traverse the array, which is a common algorithm pattern when we need to check each element in a sequence. There are no complex data structures used in this solution; it relies on a single integer `j` to keep track of the index of the previous `1`. The choice of `j` being initialized to `-inf` is a deliberate one to ensure that the first `1` encountered does not falsely trigger our condition for being too close to another `1`.

Following is a step-by-step explanation of the algorithm as implemented in the provided code:

1. We initialize `j` to `-inf` to represent the index of the last `1` seen. This is purposely a very small number to ensure that the first `1` in the array does not compare against a non-existent previous `1`.

2. We iterate through the array using a `for` loop, utilizing `enumerate` to get both the index `i` and the value `x` at that index.

3. Within the loop, we check if the current value `x` is `1`. If it's not, we do nothing and continue to the next iteration.

4. If `x` is `1`, we enter our if-statement where the condition is `i - j - 1 < k`. This condition checks the distance between the current `1` and the last `1`. Since distance is index-based and starts from 0, we subtract an additional `1` from `i - j`, effectively ensuring that there are at least `k` zeros in-between. If the condition is `True`, it means the `1`s are too close, and we immediately return `False`.

5. If the condition in step 4 is not met, meaning the `1`s are sufficiently spaced apart, we update `j` to be the current index `i`, marking it as the new position of the last `1`.

6. After the loop, if we have not returned `False`, it means all `1`s were appropriately spaced apart according to the given `k` value, and we can safely return `True` to indicate the array satisfies the condition.

The algorithm runs in O(n) time since it only needs to traverse the array once, where n is the number of elements in `nums`. The space complexity is O(1) as we only use a fixed number of variables regardless of the size of the input.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Suppose we have the binary array `nums` and integer `k` given as follows:

```
nums = [1, 0, 0, 1, 0, 1]
k = 2
```

We need to determine if all `1`s in this array are separated by at least `k` places. Here's a step-by-step walkthrough following the solution approach:

1. Initialize `j` to `-inf`. For the sake of the example, let's consider `-inf` to be `-1` since array indices are zero-based. This is to handle the case where the first `1` is correctly positioned regardless of `k`.

2. Start iterating over `nums`. Our iteration will go through indices `0` to `5`.

3. At `i = 0`, `x` is `1`. Since this is the first `1`, there is no previous `1` to compare against, so no action is necessary apart from updating `j` to `0`.

4. At `i = 1` and `i = 2`, `x` is `0`. Nothing is done.

5. At `i = 3`, `x` is `1` again. We now check if `i - j - 1 < k`, which is `3 - 0 - 1 < 2`. The actual comparison is `2 < 2`, which is `false`. Since there are `2` zeroes between the `1`s, they are at least `k` places apart, which is our requirement. So, we update `j` to `3`.

6. At `i = 4`, `x` is `0`. Nothing is done.

7. At `i = 5`, `x` is `1`. We check `i - j - 1 < k`, which is `5 - 3 - 1 < 2`. This simplifies to `1 < 2`, which is `true`. The `1`s are not at least `k` places apart because there is only `1` zero between the `1` at index `3` and the `1` at index `5`.

8. Since the condition is `true`, the algorithm returns `false` as the `1`s are too close to each other based on the `k` value provided.

Throughout this example, we can see how the algorithm determines the correct spacing between `1`s. By the end of the array, we have successfully identified an instance where `1`s were not separated by at least `k` places, determining the correct output of the function which, in this case, is `false`.

Python Solution

```python
from math import inf

class Solution:
    def kLengthApart(self, nums: List[int], k: int) -> bool:
        # Initialize the index of the previous '1' found to negative infinity
        # to handle the case where the first '1' appears at index 0.
        previous_one_index = -inf

        # Iterate over the indices and values in the nums array.
        for index, value in enumerate(nums):
            # Check if the current value is '1'
            if value == 1:
                # If the gap between the current and previous '1' is
                # less than k, return False.
                if index - previous_one_index - 1 < k:
                    return False
                # Update the position of the last found '1' to the current index.
                previous_one_index = index

        # If all '1's are at least k positions apart, return True.
        return True

```

Java Solution

```java
class Solution {

    // Method to check if all '1's in the array are at least k length apart
    public boolean kLengthApart(int[] nums, int k) {
        // Initialize previous index of '1' found to a position that is
        // k positions before the start of the array
        int lastOneIndex = -(k + 1);

        // Iterate over all elements in the array
        for (int currentIndex = 0; currentIndex < nums.length; ++currentIndex) {
            // Check if we have found a '1'
            if (nums[currentIndex] == 1) {
                // If the distance between this '1' and the previous '1'
                // is less than k, return false, as the requirement is not met
                if (currentIndex - lastOneIndex - 1 < k) {
                    return false;
                }
                // Update the index of the last found '1'
                lastOneIndex = currentIndex;
            }
        }

        // If we finish looping through the array without returning false,
        // it means all '1's are at least k length apart, so return true
        return true;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to check if all 1's are at least 'k' distance apart
    bool kLengthApart(vector<int>& nums, int k) {
        // Initialize the variable to store the index of the last seen 1.
        // We start with an index that is smaller by more than k. So, the first comparison is guaranteed to succeed.
        int lastOneIndex = -(k + 1);

        // Iterate through the array of nums
        for (int i = 0; i < nums.size(); ++i) {
            // Check if we have found a 1
            if (nums[i] == 1) {
                // Check if the distance from the last seen 1 is less than k
                if (i - lastOneIndex - 1 < k) {
                    // If 'k' constraint is not satisfied, return false
                    return false;
                }
                // Update the index of the last seen 1
                lastOneIndex = i;
            }
        }
        // If all 1's are at least 'k' distance apart, return true
        return true;
    }
};

```

Typescript Solution

```typescript
// This function checks whether all 1's in the array are at least 'k' distance apart.
// @param nums - array of numbers, consisting of 0's and 1's
// @param k - minimum distance required between two 1's
// @returns true if all 1's are 'k' or more apart, otherwise false
function kLengthApart(nums: number[], k: number): boolean {
    // Initialize the previous index of 1 to a value such that the first comparison succeeds
    let previousOneIndex = -(k + 1);

    // Loop through the array to find the 1's and check their distances.
    for (let currentIndex = 0; currentIndex < nums.length; ++currentIndex) {

        // Check if the current element is 1
        if (nums[currentIndex] === 1) {

            // Check if the distance from the current 1 to the previous 1 is less than k.
            // If it is, return false.
            if (currentIndex - previousOneIndex - 1 < k) {
                return false;
            }

            // Update the index of the most recently found 1.
            previousOneIndex = currentIndex;
        }
    }

    // Return true if no pairs of 1's are found that are less than 'k' distance apart.
    return true;
}

```

Time and Space Complexity

- **Time Complexity:** The function iterates over each element in the `nums` list once. The number of iterations is therefore linearly proportional to the length of `nums`, denoted as `N`. There are no nested loops, and operations within the loop are constant time operations. Therefore, the time complexity of the code is `O(N)`.

- **Space Complexity:** The space complexity of the code is constant, `O(1)`. This is because only a fixed number of variables (`j`, `i`, `x`) are used, regardless of the input size. The space used by the input `nums` and the integer `k` are not counted towards the space complexity since they are part of the input.