334. Increasing Triplet Subsequence

Medium Greedy Array

Problem Description

The problem gives us an array of integers, `nums`, and asks us to determine if it is possible to find a sequence of three elements, where each element comes after the previous one in the order they appear in the array (i.e., they have increasing indices), and each element is greater than the one before it in this sequence. This sequence should satisfy the condition nums[i] < nums[j] < nums[k], where i, j, and k are the indices of these elements in the array, such that `i < j < k`. If such a sequence exists in `nums`, the function should return `true`, otherwise, it should return `false`.

This can be visualized as checking if there's an ascending subsequence of at least three numbers within the original array. It's essentially a question about identifying a pattern within a sequence without reordering it or changing the values.

Intuition

To arrive at the solution efficiently, instead of looking for the whole triplet right away, we can keep track of the smallest number we have encountered so far (`mi`) and a middle number that is greater than `mi` but smaller than any potential third number (`mid`). As we iterate through the array, we can update these two numbers whenever possible. The idea here is to maintain the lowest possible values for `mi` and `mid` as we move forward, giving us the best chance to find a number that would be greater than both, thus forming our triplet.

- If the current number (`num`) is less than or equal to `mi`, it becomes the new `mi` because we're always interested in the smallest starting point of the potential triplet.
- If `num` is greater than `mi` but less than `mid`, we have found a possible middle part of our triplet, so we set `mid` to this new number.
- If `num` is greater than `mid`, this means we have successfully found a triplet (because `num` is greater than both `mi` and `mid`, which also implies that `mi` is less than `mid`), and we can return `true`.

This efficient approach uses a greedy-like method to continuously look for the most optimal `mi` and `mid` with the hope of finding a third number that could fit the sequence. It does so using a linear scan and constant extra space, without the need for sorting or extra arrays.

Solution Approach

The Reference Solution Approach begins with the creation of two variables: `mi` and `mid`, both initialized to `inf` (infinity). Here's the reasoning for each line of code:

- `mi` and `mid` serve as placeholders to store the smallest and middle elements of the potential increasing triplet subsequence. By initializing them to infinity, it ensures that any number in the array will be smaller, allowing for proper updating of these variables.

The main algorithm unfolds within a single pass through the array of numbers (`nums`):

- A `for` loop iterates through the `nums` array.
- For each `num` in `nums`, there is a series of checks and updates:
  - `if num > mid`: This is the condition that tells us we have found a valid triplet. If the current `num` is greater than our `mid` value, then we already have a `mi` which is less than `mid`, and hence, we have found a sequence where `mi < mid < num`. We return `True` immediately.
  - `if num <= mi`: If the current `num` is smaller than or equal to the current smallest value `mi`, it means that we can potentially start a new triplet sequence with this `num` as the smallest element, thus we update `mi` with the value of `num`.
  - `else`: If the current `num` is greater than `mi` and not greater than `mid`, it fits between the `mi` and `mid`, so we update the `mid` to be `num` since it could potentially be the middle element of a valid triplet.

As the for loop continues, `mi` and `mid` are updated accordingly until either a valid triplet is found—causing the function to return `True`—or the algorithm completes the array iteration without finding a triplet, resulting in a return value of `False`.

It's important to note that the code uses a greedy approach to always maintain the smallest possible values for `mi` and `mid` as it iterates over `nums`. By consistently updating these with the smallest possible values at each step, it optimizes the chance of finding a valid triplet later in the array. No additional data structures are necessary, making this solution notably efficient with O(n) time complexity (due to single iteration through the array) and O(1) space complexity (using only two extra variables).

Example Walkthrough

Let's illustrate the solution approach using a small example where our input `nums` array is `[1, 2, 3, 4, 5]`.

- We initialize `mi` and `mid` to infinity. Current state: `mi = inf, mid = inf`.
- We iterate over the array:
  - We compare the first element, `1`, with `mid`. Since `1 < inf`, we cannot form a triplet yet, but `1` becomes our new `mi`. Updated state: `mi = 1, mid = inf`.
  - We move to the next element, `2`. Now, `2` is greater than `mi` but still less than `mid`. So, `2` becomes our new `mid`. Updated state: `mi = 1, mid = 2`.
  - Next element is `3`. It is greater than both `mi` and `mid`. We now have a valid triplet: `1 < 2 < 3`. Hence, according to our solution approach, we return `True`. There is no need to check the remaining elements (`4` and `5`) because we have successfully found an increasing triplet.

In this example, the approach quickly identifies the increasing sequence with the satisfaction of the conditions outlined in the problem description. The algorithm efficiently updates `mi` and `mid` and only stops when it confirms the existence of the increasing triplet.

Solution Implementation

Python

```python
class Solution:
    def increasingTriplet(self, nums: List[int]) -> bool:
        # Initialize two variables with infinity which will
        # reapresent the smallest and middle numbers of the triplet.
        smallest = float('inf')
        middle = float('inf')

        # Iterate over the list of numbers.
        for num in nums:
            # If current number is greater than the middle number,
            # an increasing triplet exists.
            if num > middle:
                return True

            # If current number is less than or equal to the smallest,
            # update the smallest number to be the current number.
            if num <= smallest:
                smallest = num
            # Otherwise, if the current number is between the smallest
            # and the middle, update the middle number.
            else:
                middle = num

        # Return False if no increasing triplet is found.
        return False
```

Java

```java
class Solution {

    // Method to check if there exists an increasing triplet in the array.
    public boolean increasingTriplet(int[] nums) {
        // Initialize two variables to hold the smallest and the middle value found so far.
        int smallest = Integer.MAX_VALUE;
        int middle = Integer.MAX_VALUE;

        // Iterate over each number in the array.
        for (int num : nums) {
            // If the current number is greater than the middle value found,
            // an increasing triplet sequence exists.
            if (num > middle) {
                return true;
            }

            // If the current number is the smallest we've seen so far,
            // we update the smallest value.
            if (num <= smallest) {
                smallest = num;
            } else {
                // Otherwise, if it's between the smallest and the middle value,
                // we update the middle value.
                middle = num;
            }
        }

        // If we did not return true within the loop, then no increasing
        // triplet sequence was found.
        return false;
    }
}
```

C++

```cpp
#include <vector>
#include <climits> // Include for INT_MAX

class Solution {
public:
    // This function checks if there exists an increasing triplet subsequence
    // The sequence is increasing if nums[i] < nums[j] < nums[k] where i < j < k
    // Parameters:
    //     nums - a vector of integers
    bool increasingTriplet(std::vector<int>& nums) {
        int firstMin = INT_MAX; // Store the smallest number encountered
        int secondMin = INT_MAX; // Store the second smallest number encountered

        // Iterate over the input vector
        for (int num : nums) {
            // If we find a number greater than second smallest,
            // this means we found a triplet; return true
            if (num > secondMin) {
                return true;
            }

            // If current number is less than or equal to firstMin,
            // update firstMin to the current number
            if (num <= firstMin) {
                firstMin = num;
            } else {
                // If current number is between firstMin and secondMin
                // update secondMin to the current number
                // This is because we want the smallest possible value for secondMin
                // that is greater than firstMin
                secondMin = num;
            }
        }

        // If we have reached this point, it means we did not find
        // an increasing triplet subsequence
        return false;
    }
};
```

TypeScript

```typescript
/**
 * Checks whether there exists an increasing triplet subsequence within the array
 * @param nums - Array of numbers to check for the increasing triplet
 * @returns boolean - True if there is an increasing triplet, False otherwise
 */
function increasingTriplet(nums: number[]): boolean {
    let length = nums.length;

    // If the array has fewer than 3 items, it can't have an increasing triplet
    if (length < 3) return false;

    // Initialize the smallest and middle values in the triplet
    let smallest = nums[0];
    let middle = Number.MAX_SAFE_INTEGER;

    // Iterate through the array to find the increasing triplet
    for (let number of nums) {
        if (number <= smallest) {
            // Current number becomes the new smallest if it's smaller or equal to the current smallest
            smallest = number;
        } else if (number <= middle) {
            // Current number is greater than smallest but smaller or equal to middle,
            // so it becomes the new middle
            middle = number;
        } else {
            // If we found a number greater than both smallest and middle, we found an increasing triplet
            return true;
        }
    }

    // If no increasing triplet is found, return false
    return false;
}
```

```python
class Solution:
    def increasingTriplet(self, nums: List[int]) -> bool:
        # Initialize two variables with infinity which will
        # reapresent the smallest and middle numbers of the triplet.
        smallest = float('inf')
        middle = float('inf')

        # Iterate over the list of numbers.
        for num in nums:
            # If current number is greater than the middle number,
            # an increasing triplet exists.
            if num > middle:
                return True

            # If current number is less than or equal to the smallest,
            # update the smallest number to be the current number.
            if num <= smallest:
                smallest = num
            # Otherwise, if the current number is between the smallest
            # and the middle, update the middle number.
            else:
                middle = num

        # Return False if no increasing triplet is found.
        return False
```

Time and Space Complexity

The time complexity of the given code is `O(n)` where `n` is the number of elements in the `nums` list. This is because the code iterates through the entire `nums` list once with a single loop, and within each iteration, it performs a constant number of operations.

The space complexity of the given code is `O(1)` regardless of the size of the input list. It uses only two extra variables, `mi` and `mid`, which consume a constant amount of space.