659. Split Array into Consecutive Subsequences

Medium Greedy Array Hash Table Heap (Priority Queue) Leetcode Link

Problem Description

You're given a sorted integer array `nums` with non-decreasing elements. Your task is to check whether you can split the array into one or more subsequences where each subsequence is a strictly increasing consecutive sequence (each number is exactly one more than the previous one), and each subsequence must contain at least three numbers. If it's possible to do such a split, you should return `true`, otherwise return `false`.

Consider the array `[1,2,3,3,4,5]`, it can be split into two subsequences `[1,2,3]` and `[3,4,5]` which satisfy the conditions above, so the output is `true`.

Intuition

The intuition behind the solution is to use a greedy approach. We try to build all possible subsequences by iterating through the `nums` array, and for each element, we want to extend a subsequence that ends at the element before the current one (this would make it a consecutive sequence). If such a subsequence does not exist, we'll start a new subsequence.

To make this efficient, we use a hashmap to store the ends of the subsequences, mapping the last element of a subsequence to a min-heap of lengths of all subsequences ending with that element. Min-heaps are used because we want to extend the shortest subsequence available, which minimizes the chance of not being able to create valid subsequences as we add more elements.

If at any point we can extend a subsequence, we'll pop the length of the shortest subsequence from the min-heap of the predecessor of the current element, increment the length (since we're adding the current element to it), and push the new length to the current element's min-heap. If there is no subsequence to extend, we start a new one by pushing a length of `1` to the current element's min-heap.

In the end, we go through the hashmap and check if any subsequence lengths are less than `3`. If we find any, that means we have an invalid subsequence, and we return `false`. If all lengths are `3` or greater, we return `true` as we've successfully created valid subsequences.

Thus, the key to this approach is maintaining and updating the min-heaps efficiently as we move through the array.

Solution Approach

The implementation uses a hashmap (`defaultdict` from the `collections` module in Python) paired with a min-heap (from the `heapq` module) for each unique key. These data structures are crucial as they store subsequences' lengths efficiently and allow for quick retrieval and update operations.

Here's the approach broken down into steps:

1. Iterate over the sorted `nums` array:
   - For each number `v` in `nums`, check if there is a subsequence that ended just before `v` (i.e., at `v - 1`).
2. If such a subsequence exists (`if h := d[v - 1]:`):
   - Pop the length of the shortest such subsequence from the min-heap at `v - 1` (using `heappop(h)`).
   - Increment this length by `1`, since `v` will extend this subsequence (`heappop(h) + 1`).
   - Push the new length to the min-heap for `v` to keep track of this extended subsequence (`heappush(d[v], heappop(h) + 1)`).
3. If no such subsequence exists (`else:`):
   - Initialize a new subsequence with a single element `v` by pushing `1` to the min-heap for `v` (`heappush(d[v], 1)`).

At each step, subsequences are either extended or started, and the `d` hashmap tracks these actions efficiently using min-heaps.

After processing all the elements in `nums`, check if any subsequence is invalid (less than 3 elements). The final check (`return all(not v or v and v[0] > 2 for v in d.values())`) iterates over all the min-heaps stored in the hashmap:

- `not v` checks if the min-heap is empty, which would mean no invalid subsequence was found for that key, so a `True` is implied.
- `v and v[0] > 2` checks if the min-heap is not empty and the length of the shortest subsequence is greater than `2` for that particular element, implying a valid subsequence.

If all min-heaps validate the condition (meaning all subsequences are of length `3` or more), `True` is returned, otherwise `False`.

Example Walkthrough

Let's walk through a smaller example to illustrate the solution approach using the array `[1, 2, 3, 4, 4, 5]`.

Following the steps outlined in the solution approach:

1. We start by iterating through the sorted array `nums = [1, 2, 3, 4, 4, 5]`.
2. For the first element `1`, there is no previous subsequence, so we initiate a subsequence by pushing `1` into min-heap for `1`: `d[1] = [1]`.
3. Next element `2`, there is a subsequence ending at `1` so we take its length `1`, increment by one, and push `2` into min-heap for `2`: `d[2] = [2]`.
4. For the next element `3`, there is a subsequence ending at `2`, so again we take its length `2`, increment it by one, and push `3` into min-heap for `3`: `d[3] = [3]`.
5. Now we encounter the first `4`. A subsequence ends at `3`, so we take the length `3`, increment it, and push `4` to the min-heap for `4`: `d[4] = [4]`.
6. The next element is `4` again. Since we cannot extend the subsequence ending in `3` anymore (as it's already been extended), we start a new subsequence with `4`: `d[4] = [4, 1]`.
7. The final element `5` has the potential to extend the subsequence ending with the second `4`. We pop `1`, increment it to `2`, and push it to min-heap for `5`: `d[5] = [2]`.

The hash map `d` now looks like this: `{1: [1], 2: [2], 3: [3], 4: [4, 1], 5: [2]}`.

8. Looking into each min-heap, we see the following lengths:
   - For `1`, the subsequence length is `1`, which is less than `3`. This is a problem, as we need all subsequences to be at least length `3`.
   - For `4`, one of the subsequences has a length `1`, which is also less than `3`.
   - For `5`, the subsequence length is `2`, which is less than `3`.

These insights infer that there are subsequences that do not meet the requirement. Therefore, the output for the array `[1, 2, 3, 4, 4, 5]` would be `False` since not all subsequences contain at least three numbers.

This example demonstrates the process of iterating through the numbers, trying to extend existing subsequences using a min-heap and a hashmap, and finally verifying that all subsequences have met the minimum length requirement.

Python Solution

```python
from collections import defaultdict
import heapq

class Solution:
    def isPossible(self, nums: List[int]) -> bool:
        # Create a dictionary to map numbers to a list of sequence lengths
        sequence_lengths = defaultdict(list)

        # Iterate through all numbers in the provided list
        for num in nums:
            # If there exists a sequence ending with a number one less than the current number
            if sequence_lengths[num - 1]:
                # Pop the shortest sequence that ends with num - 1
                shortest_sequence = heapq.heappop(sequence_lengths[num - 1])
                # Add the current number to this sequence (incrementing its length)
                heapq.heappush(sequence_lengths[num], shortest_sequence + 1)
            else:
                # Start a new sequence with the current number (with length 1)
                heapq.heappush(sequence_lengths[num], 1)

        # Check all the (non-empty) sequences in the dictionary
        return all(len(sequence) == 0 or (len(sequence) > 0 and sequence[0] > 2) for sequence in sequence_lengths.values())
```

Java Solution

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

class Solution {
    public boolean isPossible(int[] nums) {
        // HashMap to store the end element of each subsequence and a priority queue of their lengths
        Map<Integer, PriorityQueue<Integer>> seqEndMap = new HashMap<>();

        // Iterate over each number in the input array
        for (int value : nums) {
            // If there's a sequence to append to, which ends with the current value - 1
            if (seqEndMap.containsKey(value - 1)) {
                // Get the priority queue of subsequences that end with value - 1
                PriorityQueue<Integer> lengths = seqEndMap.get(value - 1);
                // Remove the shortest subsequence from the queue and increment length
                int length = lengths.poll() + 1;
                // If the current priority queue becomes empty after removal, remove the mapping
                if (lengths.isEmpty()) {
                    seqEndMap.remove(value - 1);
                }
                // Append the incremented length to the current value's priority queue
                seqEndMap.computeIfAbsent(value, k -> new PriorityQueue<>()).offer(length);
            } else {
                // No preceding subsequences, start a new one with length 1
                seqEndMap.computeIfAbsent(value, k -> new PriorityQueue<>()).offer(1);
            }
        }

        // Validate the subsequences' lengths
        for (PriorityQueue<Integer> lengths : seqEndMap.values()) {
            // If the shortest subsequence is less than 3, fail the check
            if (lengths.peek() < 3) {
                return false;
            }
        }

        // All subsequences have a length >= 3, the split is possible
        return true;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <unordered_map>
#include <queue>

class Solution {
public:
    // Function to assess if it is possible to split the given sequence
    bool isPossible(vector<int>& nums) {
        // A map that associates each number with a min heap that represents the different possible sequence lengths ending with th
        unordered_map<int, priority_queue<int, vector<int>, greater<int>>> lengthsMap;

        // Iterate over each number in the provided vector
        for (int value : nums) {
            // If there's a sequence ending with the previous number (value - 1)
            if (lengthsMap.count(value - 1)) {
                // Get the min heap (priority queue) for sequences ending with value - 1
                auto& prevQueue = lengthsMap[value - 1];
                // Add the current value to extend a sequence, increasing its length by 1
                lengthsMap[value].push(prevQueue.top() + 1);
                // Pop the extended sequence length from the min heap of the previous value
                prevQueue.pop();
                // If there are no more sequences ending with this previous value, remove it from the map
                if (prevQueue.empty()) {
                    lengthsMap.erase(value - 1);
                }
            } else {
                // If there is no previous sequence to extend, start a new sequence with length 1
                lengthsMap[value].push(1);
            }
        }

        // Check all sequences to ensure they all have a length of at least 3
        for (auto& entry : lengthsMap) {
            auto& sequenceLengths = entry.second;
            // If the smallest length is less than 3, return false
            if (sequenceLengths.top() < 3) {
                return false;
            }
        }

        // All sequences have the required minimum length, so return true
        return true;
    }
};
```

Typescript Solution

```typescript
// TypeScript has no built-in priority queue, but we can use an array and sort it.
type PriorityQueue = number[];

// Inserts an element into the priority queue (min-heap)
function insert(q: PriorityQueue, num: number) {
    q.push(num);
    q.sort((a, b) => a - b); // Sort to ensure the smallest element is at the front
}

// Pops the smallest element from the priority queue
function pop(q: PriorityQueue): number {
    return q.shift(); // Remove and return the first element which is the smallest
}

// Function to determine if it is possible to split the sequence into consecutive subsequences of length at least 3
function isPossible(nums: number[]): boolean {
    // Map that links each number to a priority queue representing possible sequence lengths ending with that number
    const lengthsMap: Record<number, PriorityQueue> = {};

    // Iterate over each number in the array
    for (const value of nums) {
        const prevValue = value - 1;

        // If there's a sequence ending with the previous number (value - 1)
        if (lengthsMap[prevValue]) {
            const prevQueue = lengthsMap[prevValue]; // Get the priority queue for sequences ending with prevValue
            const smallestLength = pop(prevQueue); // Get and remove the smallest sequence length
            insert(lengthsMap[value] || (lengthsMap[value] = []), smallestLength + 1); // Extend the sequence length by 1 and add t

            // If the priority queue for prevValue is empty, remove it from the map
            if (prevQueue.length === 0) {
                delete lengthsMap[prevValue];
            }
        } else {
            // If there's no previous sequence to extend, start a new sequence with length 1
            insert(lengthsMap[value] || (lengthsMap[value] = []), 1);
        }
    }

    // Check that all sequences have a length of at least 3
    for (const sequences of Object.values(lengthsMap)) {
        if (sequences[0] < 3) { // Since the array is sorted, the smallest length is at the front
            return false; // A sequence is shorter than 3 so it's not possible to split
        }
    }

    // All sequences have the required minimum length of 3
    return true;
}
```

Time and Space Complexity

The given Python code performs an operation on a list of integers `nums` to determine if it is possible to split `nums` into consecutive subsequences of length at least 3.

Time Complexity:

The time complexity of the code is determined by:

1. The iteration through each number in the list (`for v in nums`), which takes `O(n)` time where `n` is the length of `nums`.
2. The heap operations (heappush and heappop) which are `O(log(k))` where `k` is the number of entries in the heap at any time.

For each number in `nums`, the code might do a heappush and a heappop, each of which can take up to `O(log(k))` time. Note that `k` can potentially be as large as `n` in the worst case when there are many different sequences being formed. However, since it's observed that each number in the array can only belong to one subsequence and hence `k` is related to the number of possible subsequences that can be formed which is actually smaller than `n`.

The final iteration to check if each subsequence has a length of more than 2 takes `O(m)` where `m` is the number of distinct elements in `nums`.

Putting this all together, the overall time complexity is `O(n log(k))` with the note that `k` tends to be smaller than `n` and is related to the distribution of the input numbers.

Space Complexity:

The space complexity of the code is given by:

1. The extra space used by the dictionary `d` which holds a heap for each possible terminal value of a subsequence, leading to `O(m)` where `m` is the number of distinct elements in `nums`.
2. The heaps themselves, which together contain all `n` numbers, giving `O(n)`.

Since both the dictionary and the heaps need to be stored, their space complexities are additive, leading to a total space complexity of `O(n + m)`. In the worst case, each element is unique so `m` could be as large as `n`, making the space complexity `O(n)`.