251. Flatten 2D Vector

Medium Design Array Two Pointers Iterator

Problem Description

The task is to design a special iterator for a 2D vector (i.e., an array of arrays or a list of lists in Python). The iterator should provide two operations: `next` and `hasNext`. The `next` operation should return the next element in the 2D vector, moving an internal pointer forward by one position. The `hasNext` operation should check if there are more elements to be traversed in the 2D vector.

The `Vector2D` class is initialized with a 2D vector. It maintains internal pointers (indices) to keep track of the current position within the 2D structure. For example, if the input 2D vector is `[[1,2], [3], [4,5,6]]`, the `next` method should return elements in the order 1, 2, 3, 4, 5, and 6.

It is important to notice that the 2D vector may contain empty sub-vectors, and the iterator should correctly handle this case by skipping them.

Intuition

The challenge in designing this iterator lies in dealing with the 2-dimensional structure of the input, as we need to iterate over elements in a row-by-row fashion. If we reach the end of a sub-vector (or if it's empty), we need to move to the next sub-vector to continue iterating.

A straightforward intuitive approach is to maintain two pointers (indices): one for the current sub-vector we're in (let's call it `i`) and one for the current element within that sub-vector (`j`). Using these pointers, the `next` operation retrieves the current element and advances `j`. If `j` exceeds the bounds of the current sub-vector, we increment `i` and reset `j` to zero, effectively moving to the next sub-vector.

For the `hasNext` operation, we need to check whether there are any more elements left. However, just checking if `i` and `j` are within bounds is not sufficient, as there may be empty sub-vectors ahead. We perform the same action as in `next` to skip empty sub-vectors and move `i` and `j` to the next available element, if any. Only then do we check if `i` is within bounds of the 2D vector. If it is, there are still elements left to iterate over, so it returns `true`. Otherwise, it returns `false`.

The `forward` helper function encapsulates this logic of skipping over empty sub-vectors and advancing the internal pointers to the next available element or the end of the vector. This helper is used in both `next` and `hasNext` to avoid code duplication and ensure that the traversal logic is consistent between those two operations.

Learn more about Two Pointers patterns.

Solution Approach

The `Vector2D` class is implemented with the following components:

1. **Initialization** - The constructor `__init__` initializes three instance variables: `self.i` and `self.j` are set to `0` to point at the start of the 2D vector, and `self.vec` is assigned the input 2D vector.

2. **`next` Method** - When this method is called, it first calls the `forward` method to ensure the `self.i` and `self.j` pointers are at a valid position. If the pointers point to an empty sub-vector or are out of bounds, `forward` will adjust them to the next available element. After this adjustment, `self.vec[self.i][self.j]` is guaranteed to reference a valid element, which is then returned. Afterwards, `self.j` is incremented to move to the next element, preparing for the next call to `next`.

3. **`hasNext` Method** - This method also starts by calling the `forward` method to adjust the pointers as necessary. Then, it does a simple check to see if `self.i` is still within the bounds of `self.vec`. If it is, this means there are still elements left to iterate over, so it returns `true`; otherwise, it returns `false`.

4. **`forward` Method** - This helper method is crucial for ensuring that the pointers skip over any empty sub-vectors and point to the next available element. The method uses a `while` loop that continues to run as long as `self.i` is less than the length of `self.vec` and `self.j` is greater than or equal to the length of the current sub-vector `self.vec[self.i]`. Within the loop, `self.i` is incremented to move to the next sub-vector, and `self.j` is reset to `0` to start at the beginning of the new sub-vector. This process repeats until a non-empty sub-vector is found or the end of the 2D vector is reached.

5. **Data Structures** - The primary data structure used in the solution is the input 2D vector itself, which is a list of lists. There are no additional data structures needed because the design intent is to iterate in-place without flattening the 2D vector into a 1D list, which would require extra space.

6. **Algorithm** - The overall algorithm is a linear traversal with a direct addressing scheme, facilitated by two pointers that act as coordinates (`i` for row and `j` for column). The solution's complexity is (O(1)) for both `next` and `hasNext` operations in the amortized sense, as each element and sub-vector are accessed a constant number of times across all calls.

By combining these components, the `Vector2D` class delivers an efficient and intuitive way to iterate over a 2D vector with varying-length sub-vectors and potentially empty sub-vectors, avoiding unnecessary space complexity and adhering to the iterator design pattern.

Example Walkthrough

Let's illustrate the solution approach using a 2D vector example: `[[1,2], [], [3], [4,5]]`.

1. **Initialization**: We instantiate our `Vector2D` class with our example 2D vector.
   - `self.vec` will be initialized to `[[1,2], [], [3], [4,5]]`.
   - `self.i` and `self.j` will both be initialized to `0`.
2. **`next` Method**: We call the `next` method.
   - Since `self.i` and `self.j` are both `0`, `self.vec[0][0]` equals `1`, which is returned.
   - `self.j` increments by `1`, so `next` is prepared to return `self.vec[0][1]` on the next call.
3. **`next` Method**: We call the `next` method again.
   - Now we're at `self.vec[0][1]` with `self.i=0` and `self.j=1`. That position has the value `2`, so it returns `2`.
   - `self.j` increments by `1` but now points beyond the current sub-vector, so `self.i` will need to increment to point to the next sub-vector on the next call to `next` or `hasNext`.
4. **`hasNext` Method**: Let's call `hasNext` now to see if more elements are available.
   - The `forward` method will be triggered and notice that `self.j` is beyond the current sub-vector at `self.i=0`.
   - `self.i` increments to `1`, but since `self.vec[1]` is empty, it skips that and increments to `2`. `self.j` resets to `0`.
   - Now `self.i=2` and `self.j=0`, which is a valid position (`self.vec[2][0]` contains `3`), so `hasNext` returns `true`.
5. **`next` Method**: Now, if we call `next` again, it returns `3`.
   - The internal state is now `self.i=2` and `self.j=1`, but since the sub-vector at `self.i=2` only has one element, `forward` will be called on the next `next` or `hasNext` to move the pointers forward.
6. **`hasNext` and `next`**: If we call `hasNext` again, it would return `true`, as there are still elements in `self.vec[3]`.
   - Calling `next` would then return `4`, and after that, the subsequent call to `next` would return `5`. At this point, with `self.i=3` and `self.j=2`, all elements are exhausted, so `hasNext` would return `false`.

Throughout this process, no additional space was used, and at no point was the 2D vector converted into a 1D array. All operations are done in-place, and while the internal pointers (`self.i` and `self.j`) constantly move forward to ensure correct iteration across both sub-vectors and individual elements within them, bypassing any empty sub-vectors.

Python Solution

```python
# A class to implement a 2D vector iterator.
class Vector2D:
    def __init__(self, vec):
        """
        Initializes a new instance of the Vector2D class.

        :param vec: A list of lists of integers to iterate over.
        """
        self.current_row = 0  # Initialize the row index
        self.current_col = 0  # Initialize the column index
        self.vector = vec  # Store the 2D vector

    def next(self) -> int:
        """
        Get the next element in the 2D vector.

        :return: The next integer in the 2D vector.
        """
        # Ensure the indices are pointing to an existing value
        self.advance_to_next()
        # Retrieve the next value
        value = self.vector[self.current_row][self.current_col]
        # Move the column index forward
        self.current_col += 1
        return value

    def hasNext(self) -> bool:
        """
        Check whether the 2D vector has more elements.

        :return: True if there are more elements, False otherwise.
        """
        # Adjust the current indices to ensure they are pointing to a value
        self.advance_to_next()
        # Check if current row is within the vector bounds
        return self.current_row < len(self.vector)

    def advance_to_next(self):
        """
        Advance the indices to the next available element, if needed.
        This moves to the next row if the current one is done.
        """
        # Loop while current row is within bounds and current column is beyond the current row bounds
        while self.current_row < len(self.vector) and self.current_col >= len(self.vector[self.current_row]):
            self.current_row += 1  # Go to the next row
            self.current_col = 0  # Reset column index to start of the new row


# Example of how to use the Vector2D class:
# obj = Vector2D([[1, 2], [3], [4, 5, 6]])
# while obj.hasNext():
#     param_1 = obj.next()
#     print(param_1)

```

Java Solution

```java
class Vector2D {
    private int rowIndex; // keeps track of the current row in the 2D vector
    private int colIndex; // keeps track of the current column in the current row
    private int[][] vector; // stores the reference to the 2D vector

    // Constructor initializes the 2D vector and the indices
    public Vector2D(int[][] vec) {
        this.vector = vec;
        rowIndex = 0;   // start at the first row
        colIndex = 0;   // start at the first column
    }

    // Returns the next element in the 2D vector and moves the pointer
    public int next() {
        // Move to a valid position if necessary
        moveToNextValid();
        // Return the current element and move the column index forward
        return vector[rowIndex][colIndex++];
    }

    // Checks if there are more elements to iterate over in the 2D vector
    public boolean hasNext() {
        // Move to a valid position if necessary
        moveToNextValid();
        // Determine if we have a valid next element by comparing the current row index with the vector length
        return rowIndex < vector.length;
    }

    // Moves the indices to the next valid position if the current one is not
    private void moveToNextValid() {
        // If the current row is exhausted (colIndex >= the row length), move to the next row until a valid element is found
        while (rowIndex < vector.length && colIndex >= vector[rowIndex].length) {
            rowIndex++;  // move to the next row
            colIndex = 0; // start at the beginning of the new row
        }
    }
}

/**
 * The Vector2D class provides a way to iterate through a 2D vector (array of arrays) as if it were a flat array.
 * The 'next' and 'hasNext' methods function similarly to those in an iterator, allowing for an iteration of elements in a 2D vector.
 * The code snippet shows how to instantiate the Vector2D object and call its methods:
 * Vector2D obj = new Vector2D(vec); // create a new Vector2D object
 * int element = obj.next(); // retrieve the next element in the 2D vector
 * boolean hasMore = obj.hasNext(); // check if more elements are available
 */

```

C++ Solution

```cpp
#include <vector>

using std::vector;

class Vector2D {
public:
    // Constructor which takes a nested vector as an input
    // and moves it to the member variable 'nestedVector'.
    Vector2D(vector<vector<int>>& vec) {
        nestedVector = std::move(vec);
    }

    // Returns the next element in the 2D vector.
    int next() {
        moveToNextValid(); // Ensure that the current position is valid
        return nestedVector[rowIndex][colIndex++]; // Return the element and move to the next
    }

    // Checks if there are any more elements left in the 2D vector.
    bool hasNext() {
        moveToNextValid(); // Ensure that the current position is valid
        return rowIndex < nestedVector.size(); // Check if rows are still left
    }

private:
    int rowIndex = 0; // Row index for the current element
    int colIndex = 0; // Column index for the current element
    vector<vector<int>> nestedVector; // 2D vector to be flattened

    // Adjusts the row and column indices to point to the next valid element.
    void moveToNextValid() {
        // Continue moving to the next row if the current row is empty or
        // if the column index is equal to the size of the current row (invalid).
        while (rowIndex < nestedVector.size() && colIndex >= nestedVector[rowIndex].size()) {
            ++rowIndex; // Move to the next row
            colIndex = 0; // Reset column to the start
        }
    }
};

/**
 * Your Vector2D object will be instantiated and called as such:
 * Vector2D* obj = new Vector2D(vec);
 * int param_1 = obj->next();
 * bool param_2 = obj->hasNext();
 */

```

Typescript Solution

```typescript
// Global index variables for keeping track of the current position in the 2D vector
let currentIndexI: number = 0;
let currentIndexJ: number = 0;
// The 2D vector to be iterated over
let vector: number[][];

/**
 * Initializes the global variables with a new 2D vector.
 * @param vec The 2D vector to be used for iteration.
 */
function vector2DConstructor(vec: number[][]): void {
    currentIndexI = 0;
    currentIndexJ = 0;
    vector = vec;
}

/**
 * Advances to the next element in the 2D vector and returns it.
 * @return The next element in the 2D vector.
 */
function next(): number {
    forward(); // Ensure the indices are at the correct position for 'next' operation.
    return vector[currentIndexI][currentIndexJ++]; // Return current element and increment the inner index.
}

/**
 * Checks if there is a next element in the 2D vector.
 * @return `true` if there is a next element, `false` otherwise.
 */
function hasNext(): boolean {
    forward(); // Adjust the indices to point to the next available element, if any.
    return currentIndexI < vector.length; // Return true if there are more elements to iterate over.
}

/**
 * Adjusts the indices to skip empty inner arrays and point to the next available element.
 */
function forward(): void {
    // Loop until a non-empty row is found or end of the vector is reached
    while (currentIndexI < vector.length && currentIndexJ >= vector[currentIndexI].length) {
        ++currentIndexI; // Move to the next row in the 2D vector.
        currentIndexJ = 0; // Reset the inner index.
    }
}

// Example usage:
// vector2DConstructor([[1, 2], [3], [], [4, 5, 6]]);
// while (hasNext()) {
//     console.log(next());
// }

```

Time and Space Complexity

Time Complexity

1. **`__init__`**: It takes constant time, `O(1)`, since it only involves assigning the input list to an instance variable and initializing a couple of integers for iteration.

2. **`next`**: This method invokes `self.forward()`, which loops until it finds a non-empty list or reaches the end of the outer list. In the worst case, where there are `m` empty inner lists before we find a non-empty one, and `n` is the total number of inner lists, this can take `O(m)`. Afterward, returning the next element takes `O(1)`. However, since each inner list element is accessed exactly once, the amortized time complexity for multiple `next` operations is `O(1)` per operation.

3. **`hasNext`**: Similar to `next`, it calls `self.forward()`, but only needs to determine if there is another element, which is `O(1)` after `self.forward()` completes. For multiple calls, just like with `next`, the amortized time complexity for `hasNext` is `O(1)` per operation due to the way elements are accessed sequentially.

4. **`forward`**: While `forward` is called by the other methods and may in the worst case iterate through all the inner lists, it does so only as far as necessary to skip empty inner lists. Thus, while the worst-case complexity for a single call is `O(n)` with `n` as the number of inner lists, the total number of operations that `forward` performs across all calls is bounded by `O(n)` because every inner list is visited at most once. Consequently, the amortized complexity of `forward` per call of `next` or `hasNext` becomes `O(1)`.

Conclusively, `next` and `hasNext` have an amortized time complexity of `O(1)` per operation.

Space Complexity

1. **`__init__`**: Except the space taken by the input, the constructor only uses constant additional space, `O(1)`, for the indices.

2. **`next`**, **`hasNext`**, and **`forward`**: These methods do not use additional space that scales with the size of the input. They only use a constant amount of additional space for locally-scoped variables and pointers, leading to `O(1)` additional space complexity.

Conclusively, the overall additional space complexity of the methods in the `Vector2D` class is `O(1)`.