1640. Check Array Formation Through Concatenation

Easy Array Hash Table

Leetcode Link

Problem Description

The problem presents a scenario where you have two arrays: `arr` is a single list of distinct integers, and `pieces` is a list of lists with each sublist also containing distinct integers. The goal is to determine if it is possible to form the `arr` array by concatenating the subarrays in `pieces` in any order. However, it is important to note that you cannot reorder the elements within the subarrays in `pieces`. You must use the subarrays in `pieces` as they are.

For example, if `arr` = [1,2,3,4] and `pieces` = [[2,3], [4], [1]], you could form `arr` by concatenating [1], [2,3], and [4] in order. However, if `pieces` were [[2,3], [1,4]], you could not form `arr` because that would require reordering the elements in a piece, which is not allowed.

The task is to return `true` if you can form `arr` by such concatenations or `false` otherwise.

Intuition

The intuition for solving this problem is to use a hash map to easily find the location of the subarrays in `pieces` that can potentially be concatenated to match `arr`. The first element of each subarray in `pieces` can be used as a unique key in the hash map because we are given that all integers are distinct. This way, you can quickly look up the starting element of each piece to see if you can continue building `arr` from the current position.

To arrive at the solution, we can follow these steps:

1. Create a hash map (`d`) where each key is the first integer of a subarray in `pieces`, and the value is the subarray itself.
2. Iterate over `arr`, and at each step:
   - Check if the current element exists in our hash map (`d`). If it doesn't, it means there's a number in `arr` that isn't the start of any piece, and thus we cannot form `arr` fully. Return `false`.
   - If the present element is in `d`, fetch the corresponding subarray and check if the subsequent elements in `arr` match this subarray exactly. If they don't, return `false` because the ordering within a piece cannot be altered.
   - If they do match, increment the index (`i`) by the length of the piece, effectively 'skipping' over these elements in `arr`.
3. If you reach the end of `arr` without returning `false`, it means you've been able to build all of `arr` using the `pieces` in the correct order, and hence return `true`.

The solution demonstrates a greedy approach, building `arr` incrementally from the start by mapping each element to a potential subarray from `pieces`, ensuring the order within both `arr` and `pieces` remains unchanged.

Solution Approach

The solution approach to this problem uses a hash table (in Python, a dictionary) and array slicing.

1. **Creating a hash table**: A hash table (`d`) is created where for every subarray (`p`) in `pieces`, there's an entry with the key being the first element of the subarray and the value being the subarray itself. This allows us to quickly look up the piece that could potentially match a segment of `arr` starting from a given position.

   ```
   d = {p[0]: p for p in pieces}
   ```

   This is very efficient because finding an item in a hash table has an average-case time complexity of O(1).

2. **Iterating over the target array (`arr`)**: We then start walking through `arr` from the first element, intending to find a match in `d`.

   ```
   i, n = 0, len(arr)
   while i < n:
   ```

3. **Checking existence in the hash table**: For each element in `arr`, we first check whether this element is a key in our hash table `d`.

   ```
   if arr[i] not in d:
       return False
   ```

   If it's not present, it means we cannot find a subarray in `pieces` to continue our concatenation, and the function should return `false`.

4. **Comparing subarrays**: If we find the element in `d`, we fetch the corresponding subarray (`p`). We then slice `arr` starting from the current index `i` up to the length of the `p` and compare it with the `p`.

   ```
   p = d[arr[i]]
   if arr[i: i + len(p)] != p:
       return False
   ```

   If the sliced portion of `arr` and the subarray `p` do not match, it means the current segment cannot be formed with the `pieces` provided, and we return `false`.

5. **Incrementing the index**: If a match is found, we increment our current index (`i`) by the length of the `p` to move past the part of `arr` we just confirmed.

   ```
   i += len(p)
   ```

6. **Returning the result**: Finally, after the loop, if we haven't returned `false`, it means `arr` has been successfully formed by concatenating the subarrays in `pieces`, we return `true`.

   ```
   return True
   ```

This algorithm's overall time complexity is O(n), where n is the number of elements in `arr`, as we are iterating through each element once and hash map operations are O(1) on average. It's an efficient and effective way to solve the problem provided.

Example Walkthrough

Let's illustrate the solution approach with a small example:

Say we have `arr` = [5, 6, 7] and `pieces` = [[7], [5, 6]]. According to the problem, we can only concatenate subarrays from `pieces` to form `arr` and cannot change the order within each subarray.

Step 1: Creating a hash table

Firstly, we create a hash table where each key is the first element of the subarray from `pieces`, and the value is the subarray itself.

```
d = {7: [7], 5: [5, 6]}
```

The hash table `d` is now `{5: [5, 6], 7: [7]}`.

Step 2: Iterating over the target array (`arr`)

We aim to match elements in `arr` to keys in `d`:

```
i, n = 0, len(arr)  # i - current index, n - length of arr
while i < n:
```

Step 3: Checking existence in the hash table

For each element `arr[i]`, check if it is a key in `d`.

- For `i = 0`, `arr[i]` is `5`. Since `5` is a key in `d`, we can continue to the next step.
- If `arr[i]` were not in `d`, for example, `8`, we would return `False`.

Step 4: Comparing subarrays

Now, we need to check if the segment of `arr` starting at index `i` matches the subarray in `d`.

- For `i = 0`, `d[arr[i]]` is `[5, 6]`. We compare `arr[0:2]` (`[5, 6]`) with `[5, 6]`, and they match.

If they didn't match or if `arr` had fewer elements than the subarray `p`, we would return `False`.

Step 5: Incrementing the index

After finding a match, we skip over the matched part of `arr`:

- Since the length of `[5, 6]` is `2`, we increment `i` by 2: `i += len([5, 6])`.

We then continue to the next iteration of the loop with `i = 2`.

Step 6: Iterating continues

- Now `i = 2` and `arr[i]` is `7`. We look up `7` in the hash table and get `[7]`.
- We compare `arr[2:3]` (`[7]`) with `[7]` and they match.

Step 7: Returning the result

Having reached the end of `arr` without returning `False`, we can conclude that `arr` can be formed by concatenating subarrays in `pieces`. Therefore, the final result is `True`.

By following these steps, we demonstrated that `arr` can be formed from `pieces`. This approach efficiently utilizes the hash table for quick lookups and array slicing for comparison, resulting in an efficient solution.

Python Solution

```python
class Solution:
    def can_form_array(self, arr: List[int], pieces: List[List[int]]) -> bool:
        # Create a dictionary that maps the first element of each piece to the piece itself
        index_dict = {piece[0]: piece for piece in pieces}

        i = 0  # Initialize a pointer to iterate through 'arr'
        n = len(arr)  # The length of 'arr'

        # Loop through 'arr' using the pointer 'i'
        while i < n:
            # If the current element in 'arr' does not start any piece, return False
            if arr[i] not in index_dict:
                return False

            # Retrieve the piece that starts with 'arr[i]'
            current_piece = index_dict[arr[i]]

            # Check if the next elements in 'arr' match this piece
            if arr[i: i + len(current_piece)] != current_piece:
                return False

            # Move the pointer 'i' forward by the length of the matched piece
            i += len(current_piece)

        # If all pieces are matched without any issues, return True
        return True

```

Java Solution

```java
class Solution {
    public boolean canFormArray(int[] arr, int[][] pieces) {
        // Create a hashmap to easily look up if a piece can be placed.
        Map<Integer, int[]> piecesMap = new HashMap<>();

        // Iterate over pieces and map the first element of each piece to the piece itself.
        for (int[] piece : pieces) {
            piecesMap.put(piece[0], piece);
        }

        // Use 'i' to traverse the arr array.
        for (int i = 0; i < arr.length;) {
            // Check if there is a starting piece for the current element in arr.
            if (!piecesMap.containsKey(arr[i])) {
                return false; // No piece starts with this element, return false.
            }

            // Get the piece that starts with arr[i].
            for (int val : piecesMap.get(arr[i])) {
                // Check if the element in arr matches the element in the piece.
                if (arr[i++] != val) {
                    return false; // Element doesn't match, array can't be formed.
                }
            }
        }

        // All elements matched correctly, array can be formed.
        return true;
    }
}

```

C++ Solution

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

class Solution {
public:
    // Returns true if the arr can be formed by concatenating subarrays from pieces
    bool canFormArray(std::vector<int>& arr, std::vector<std::vector<int>>& pieces) {
        // Creating a hashmap to map the first element of each piece to its corresponding vector
        std::unordered_map<int, std::vector<int>> piece_map;
        for (auto& piece : pieces) {
            piece_map[piece[0]] = piece;
        }

        // Iterate over the elements of arr
        for (int i = 0; i < arr.size();) {
            // If the current element of arr is not the first element of any piece, return false
            if (piece_map.count(arr[i]) == 0) {
                return false;
            }
            // Retrieve the piece that starts with the current element of arr
            for (int& value : piece_map[arr[i]]) {
                // Check if the current subsequence of arr matches the piece
                // Increment index i for each match
                if (arr[i++] != value) {
                    return false; // If there's a mismatch, return false
                }
            }
        }

        // If all elements of arr are matched with a piece correctly, return true
        return true;
    }
};

```

Typescript Solution

```typescript
/**
 * Checks if it's possible to form an array by concatenating subarrays in 'pieces' to match 'arr'.
 * Each subarray in 'pieces' will appear at most once in 'arr'.
 * The concatenation of all subarrays in 'pieces' is allowed to be in any order.
 *
 * @param {number[]} arr - The target array that we want to form.
 * @param {number[][]} pieces - The subarrays that can be concatenated to form 'arr'.
 * @returns {boolean} true if the 'arr' can be formed from 'pieces', otherwise false.
 */
function canFormArray(arr: number[], pieces: number[][]): boolean {
    // The length of the target array.
    const targetLength = arr.length;

    let currentIndex = 0; // A pointer to track the current index in 'arr'

    // While there are unprocessed elements in 'arr'
    while (currentIndex < targetLength) {
        // The current element we want to find in 'pieces'
        const currentTargetValue = arr[currentIndex];

        // Find the subarray in 'pieces' that starts with 'currentTargetValue'
        const currentPiece = pieces.find(piece => piece[0] === currentTargetValue);

        // If no such subarray exists, we cannot form 'arr'
        if (!currentPiece) {
            return false;
        }

        // Iterate through the elements of the found subarray
        for (const item of currentPiece) {
            // If any element doesn't match the corresponding element in 'arr', we cannot form 'arr'
            if (item !== arr[currentIndex]) {
                return false;
            }
            // Move the index in 'arr' forward
            currentIndex++;
        }
    }

    // If we processed all elements without issues, we can form 'arr'
    return true;
}

```

Time and Space Complexity

The given Python code defines a method `canFormArray` which determines whether an array can be formed by concatenating the arrays in a given list `pieces`. Let's analyze the time and space complexity of the code:

Time Complexity

- The construction of the dictionary `d` has a time complexity of `O(m)`, where `m` is the total number of elements in `pieces`, since each of the `k` sublists in `pieces` is iterated over once.
- The `while` loop iterates over each element of `arr`, which has `n` elements. Inside the loop, the check if `arr[i]` is in `d` is `O(1)` due to the hash map lookup, and the slice comparison `arr[i : i + len(p)] != p` is `O(l)` where `l` is the length of the current `p` in `pieces`.
- In the worst case scenario, all `n` elements need to be checked, and we might have to compare against each of the `k` pieces with an average length of `l` (assuming the pieces are approximately the same size; if they vary significantly, we'd consider an average size for a more accurate analysis).

Putting it all together, the overall time complexity is `O(m + n * l)`, where `m` is the total number of elements in `pieces`, `n` is the number of elements in `arr`, and `l` is the average length of the subarrays in `pieces`.

Space Complexity

- The main extra space usage comes from the dictionary `d`, which contains at most `k` entries where `k` is the number of sublists in `pieces`. The space complexity for storing `d` is `O(k)`.
- There is also the space used by variable `p` which at maximum can hold a list of length `l`. However, since `l` is at most the length of `arr`, this does not exceed `O(n)` space.
- Therefore, the space complexity of the code is `O(k + n)`.

Considering that the space required for `d` is dependent on the number `k` of sublists and each list is stored entirely, whereas the space required for `p` and other local variables is negligible compared to the space for `d`, we could simplify the space complexity to `O(m)` because `m` includes both the number of sublists and their individual lengths.