1764. Form Array by Concatenating Subarrays of Another Array

## **Problem Description**

**Greedy** Array

Medium

one-dimensional integer array named nums. The objective of the problem is to check whether it's possible to find n disjoint (nonoverlapping) subarrays within nums that match the subarrays described in groups in the exact same order. A few key points to understand the problem:

• If i > 0, the subarray groups [i - 1] must be found in nums before the subarray groups [i]. This enforces the order of the subarrays to be the

In this problem, we are provided with a two-dimensional integer array called groups, with a length of n. Also, we have another

• The subarray groups [i] should match exactly with a contiguous portion of nums.

same as their respective order in groups.

**String Matching** 

- Disjoint subarrays mean no element in nums can be part of more than one subarray that matches the groups. To summarize, the task is to verify if we can find matching sequences in nums for each of the subarrays in groups, following the
- sequence order and ensuring that the matching sequences in nums do not overlap.

ntuition When approaching the solution, it is crucial to sequentially search through the nums array and check for a subarray that matches

our position in nums past the matched sequence. Here's a step-by-step approach to the solution:

3. In each iteration, check if the subarray of nums starting from the current position j matches the groups [i] subarray. This can be done by

the current groups subarray. If we find a matching subsequence, we move on to the next subarray in groups while also advancing

1. Initialize two pointers - i to iterate over groups and j to iterate over nums. 2. Iterate over both arrays using the pointers until either of them reaches the end.

4. If a match is found, increment the pointer i to the next subarray in groups and increment the pointer j by the length of the matched group. This

comparing slices of nums.

ensures the subarrays are disjoint and follows the correct sequence. 5. If a match is not found, only increment the pointer j to check the next possible starting position in nums.

6. Continue this process until all groups are matched or we reach the end of nums.

- The key to the solution is realizing that all elements in each subarray in groups must appear in the same order and without interruption within nums. This means that the whole subarray must match a contiguous slice of nums. By updating pointers accordingly and ensuring we only progress in groups when a complete match is found, we can determine if there's a way to
- choose the disjoint subarrays that satisfy the conditions laid out in the problem description. Ultimately, the goal is to check if the pointer i equals the length of groups, which would indicate all subarrays from groups have
- been matched in nums. If this is the case, we return true; otherwise, false.

complex algorithms. Let's delve into the implementation details of the solution provided:

Initialize Pointers: We start by initializing two pointers i and j to 0. Pointer i will traverse the groups array to check each group one by one, whereas j will traverse the nums array to find the matching subarrays. Main Loop: We run a while loop where it continues as long as i is less than n (length of groups) and j is less than m (length of nums). This ensures we do not go out of bounds of either array.

Matching Subarrays: Inside the loop, we compare the subarray from nums starting at index j and having the same length as

The problem can be solved using a straightforward two-pointer approach without requiring any additional data structures or

## the current group we are checking (represented by groups[i]). If nums[j:j+len(groups[i])] equals groups[i], we have found a matching subarray.

we return false.

**Example Walkthrough** 

Let's start with the initial setup:

• j (pointer for nums) is at position 0.

Now, let's step through the algorithm:

j by 1 and keep i the same.

Current pointers: i = 1, j = 3.

starting point for the current group within nums.

space and O(m) time complexity, where m is the length of nums.

• i (pointer for groups) is at position 0, meaning we are looking for [1, 2].

increment i by 1 signaling we've found all group subarrays.

Current pointers: i = 2 (which equals the length of groups), j = 5 (end of nums).

def canChoose(self, groups: List[List[int]], nums: List[int]) -> bool:

# Pointers for the current group and current number in 'nums'

# Move to the next group after a successful match

# If no match, move to the next number in 'nums'

# Return True if all groups have been successfully matched, False otherwise

// If current group is found in 'nums' starting from currentIndexInNums

currentIndexInNums += groups[currentGroupIndex].length;

if (isGroupMatch(groups[currentGroupIndex], nums, currentIndexInNums)) {

// Move currentIndexInNums ahead by the length of the found group

Solution Approach

Advance Pointers: When a match is found, we increment i by 1 to move to the next group in groups and j by the length of the matched group (len(g)), ensuring that the next search in nums starts right after the end of the current matched group.

- This ensures the subarrays are disjoint, as the elements in nums used for the current match no longer participate in future matches. No Match, Increment j: If there is no match, we just increment j by 1. This is because we are trying to find the next possible
- we've reached the end of nums without matching all groups (failure case). **Return Result**: After exiting the loop, we check if i == n, which would mean all groups have been matched correctly. If this is true, we return true, indicating that it is possible to choose n disjoint subarrays from nums that match groups in order. If not,

The simplicity of the two-pointer approach lies in its linear traversal of the nums array and the incremental checking process

against groups. It allows us to verify the presence and correct sequencing of subarrays within an array using constant additional

**Termination**: The loop will terminate in one of two cases: We have either found all the groups within nums (success case), or

To illustrate the solution approach, let's work through a small example. Suppose groups = [[1, 2], [2, 3]] and nums = [1, 2, 3, 2, 3]. We need to check if we can find two disjoint subarrays within nums that match the subarrays described in groups in the exact same order.

At j = 0, we find that nums [0:2] is [1, 2], which matches groups [0]. So now we increment i by 1 (moving to the next group, which is [2, 3]) and j by 2 (the length of the matched group). Current pointers: i = 1, j = 2. Now looking at nums starting from j = 2, we have nums [2:4] is [3, 2], which does not match groups [1]. So we just increment

From j = 3, we see that nums [3:5] is [2, 3], which matches groups [1]. Now we've matched both subarrays in groups, so we

At the end of the steps, we've matched all subarrays in groups with disjoint subarrays in nums following the correct order. Since i equals the length of groups, we can return true.

Solution Implementation

from typing import List

class Solution:

exactly match the groups in order.

# Initialize variables

total\_nums = len(nums)

group\_idx = num\_idx = 0

group\_idx += 1

 $num_idx += 1$ 

return group\_idx == total\_groups

# Total number of groups

total\_groups = len(groups)

# Total number of numbers in 'nums'

# Loop through the groups and 'nums' list

num\_idx += len(current\_group)

public boolean canChoose(int[][] groups, int[] nums) {

// Move to the next group

if (group[i] != nums[startIdx]) {

// Iterate through 'nums' with index 'j'

**}**;

int totalGroups = groups.size(), numsSize = nums.size();

int currentGroupIndex = 0; // Index of the current group

return false; // Mismatch found, return false

for (int j = 0; currentGroupIndex < totalGroups && j < numsSize;) {</pre>

return i == groupSize; // All elements match if we have gone through the entire 'group'

++currentGroupIndex;

int groupCount = groups.length; // Number of groups

**Python** 

So, applying this solution approach to our example, it indicates we can choose disjoint subarrays from nums ([1, 2] and [2, 3]) that

while group\_idx < total\_groups and num\_idx < total\_nums:</pre> # Current group current\_group = groups[group\_idx] # If current group matches a subsequence in 'nums' starting from current num\_idx if current\_group == nums[num\_idx : num\_idx + len(current\_group)]:

# Move num\_idx ahead by the length of the current group since we found a match

// Method to determine if all groups can be found as subsequences in 'nums' array in the given order

```
int numLength = nums.length; // Length of 'nums' array
int currentGroupIndex = 0; // Current index of groups being searched for in 'nums'
// Loop through 'nums' array looking for each group
for (int currentIndexInNums = 0; currentGroupIndex < groupCount && currentIndexInNums < numLength;) {</pre>
```

else:

Java

class Solution {

```
} else {
                // If not found, increment the 'nums' index to check the next subsequence
                ++currentIndexInNums;
        // Return true if all groups have been found, false otherwise
        return currentGroupIndex == groupCount;
    // Helper method to check if a group is found at a specific starting index in 'nums'
    private boolean isGroupMatch(int[] group, int[] nums, int startIndex) {
        int groupLength = group.length; // Length of the current group
        int numLength = nums.length; // Length of 'nums' array
        int groupIndex = 0; // Index for iterating over the group elements
       // Loop over the group and 'nums' starting from startIndex
        for (; groupIndex < groupLength && startIndex < numLength; ++groupIndex, ++startIndex) {</pre>
           // If any element does not match, return false
            if (group[groupIndex] != nums[startIndex]) {
                return false;
        // Return true if all elements matched, meaning the whole group is found
        return groupIndex == groupLength;
C++
class Solution {
public:
    // Function to determine if all 'groups' can be chosen in the same order they appear, as subarrays from 'nums'
    bool canChoose(vector<vector<int>>& groups, vector<int>& nums) {
        // Lambda function to check if subarray starting from index 'startIdx' in 'nums' matches the 'group'
        auto isMatch = [&](vector<int>& group, vector<int>& nums, int startIdx) {
            int groupSize = group.size(), numsSize = nums.size();
            int i = 0;
            // Check each element of the 'group' against 'nums' starting from 'startIdx'
            for (; i < groupSize && startIdx < numsSize; ++i, ++startIdx) {</pre>
```

```
// Check if the current group matches a subsequence starting at 'j'th index of 'nums'
            if (isMatch(groups[currentGroupIndex], nums, j)) {
                j += groups[currentGroupIndex].size(); // Move 'j' past this group in 'nums'
                ++currentGroupIndex; // Move on to the next group
            } else {
                ++j; // Mismatch found, move 'j' to check for the current group starting at next index
       // If all groups have been matched, return true
       return currentGroupIndex == totalGroups;
TypeScript
// Represents a group of numbers and the main array of numbers
type NumGroup = number[];
type MainArray = number[];
// Function to determine if all 'groups' can be chosen in the same order they appear, as subarrays from 'nums'
function canChoose(groups: NumGroup[], nums: MainArray): boolean {
   // Helper function to check if subarray starting from index 'startIndex' in 'nums' matches the 'group'
   const isMatch = (group: NumGroup, nums: MainArray, startIndex: number): boolean => {
        const groupSize = group.length;
       const numsSize = nums.length;
        let i = 0;
       // Check each element of the 'group' against 'nums' starting from 'startIndex'
        for (; i < groupSize && startIndex < numsSize; i++, startIndex++) {</pre>
            if (group[i] !== nums[startIndex]) {
                return false; // Mismatch found, return false
       // All elements match if we have gone through the entire 'group'
       return i === groupSize;
   };
   const totalGroups = groups.length;
   const numsSize = nums.length;
    let currentGroupIndex = 0; // Index of the current group being matched
   // Iterate through 'nums' with index 'j'
    for (let j = 0; currentGroupIndex < totalGroups && j < numsSize;) {</pre>
       // Check if the current group matches a subarray starting at 'j'-th index of 'nums'
       if (isMatch(groups[currentGroupIndex], nums, j)) {
            j += groups[currentGroupIndex].length; // Move 'j' past this matched group in 'nums'
            currentGroupIndex++; // Move on to the next group
```

j++; // Mismatch found, move 'j' to check for the current group starting at the next index

```
return group_idx == total_groups
Time and Space Complexity
```

else:

} else {

from typing import List

# Initialize variables

total\_nums = len(nums)

group\_idx = num\_idx = 0

# Current group

group\_idx += 1

 $num_idx += 1$ 

# Total number of groups

total\_groups = len(groups)

# Total number of numbers in 'nums'

# Loop through the groups and 'nums' list

current\_group = groups[group\_idx]

num\_idx += len(current\_group)

class Solution:

// If all groups have been matched, return true

def canChoose(self, groups: List[List[int]], nums: List[int]) -> bool:

# Pointers for the current group and current number in 'nums'

# If current group matches a subsequence in 'nums' starting from current num\_idx

# Move num\_idx ahead by the length of the current group since we found a match

if current\_group == nums[num\_idx : num\_idx + len(current\_group)]:

# Return True if all groups have been successfully matched, False otherwise

# Move to the next group after a successful match

# If no match, move to the next number in 'nums'

while group\_idx < total\_groups and num\_idx < total\_nums:</pre>

return currentGroupIndex === totalGroups;

**Time Complexity:** 

The given Python code is designed to check whether all the groups of numbers in the groups list can be chosen sequentially from

```
    n as the length of groups

    m as the length of nums

    k as the average length of a group in groups
```

**Space Complexity:** 

Let's denote:

the nums list.

In the worst-case scenario, you may have to check each element of nums against each group in groups. The worst-case time complexity will be 0(n \* m \* k) since for each group we potentially check each element in nums, and for each comparison, we may compare up to k elements (the length of a group).

The time complexity of the code depends on the number of elements in both groups and nums.

- However, in practice, the pointer j advances by at least one with each outer loop iteration, so you should not always multiply n and m. The real time complexity is between 0(n \* k + m) and 0(n \* m \* k) depending on the structure of groups and nums.
- The space complexity of the code is 0(1), which is constant. This is because the code only uses a fixed amount of extra space (a few integer variables like i, j, and g). The slices in nums[j : j + len(g)] do not count towards extra space since Python utilizes an iterator over the sublist rather than copying it. Thus, the use of space does not scale with the size of the input.