1566. Detect Pattern of Length M Repeated K or More Times

Easy Array Enumeration

Problem Description

In this problem, we are given an array of positive integers `arr`. Our task is to determine if there exists a subarray (a consecutive sequence of elements) of a certain length `m` that appears at least `k` times in the array, one immediately after the other (consecutively and without overlapping). The repeated subarrays represent the 'pattern' we are looking for.

For example, if the array is `[1, 2, 1, 2, 1, 2, 1, 3]` and `m = 2`, `k = 3`, the pattern `[1, 2]` appears three times consecutively without overlapping, so the answer would be `true`.

The problem asks us to return `true` if such a pattern exists and `false` if it does not.

Intuition

The straightforward way to solve this problem is to check each possible subarray of length `m` to see if it is followed by itself `k-1` more times. This approach requires us to iterate through the given array and attempt to match every sequence of length `m` followed by `k-1` identical sequences.

Specifically, we:

1. Iterate over the array from the start up to the point where there is still room for `m*k` elements (inclusive), since we need at least that many elements for a valid pattern to exist.
2. For each start position `i`, we check the following `m*k` elements to see if the sequence repeats `k` times. We compare each element in this window to the corresponding element in the first `m` elements. If all these elements match as required, it means we have found our pattern, and we can return `true`.
3. If, after checking all possible starting points, we haven't returned `true`, it means no such pattern exists, and we return `false`.

Solution Approach

The implementation of the solution follows a simple but effective algorithm, utilizing basic iteration and comparison without requiring sophisticated data structures or patterns beyond array manipulation and the concept of a sliding window.

Here are the steps the algorithm uses:

1. Calculate the length of the array `n`.
2. Start iterating through the array with the variable `i`, which indicates the starting index of the current window. The loop's ending condition ensures that we don't check patterns starting in places where there wouldn't be enough room left in the array for `m * k` elements, hence `i` stops at `n - m * k + 1`.
3. Initialize the inner loop with the variable `j` to zero. This inner loop will step through the elements in the current window, checking if the pattern is repeated `k` times.

The following pseudocode outlines the iteration process:

```
for i from 0 to (n - m * k + 1):
  set j to 0
  while j < m * k:
    if arr[i + j] != arr[i + (j % m)]:
      break out of the while-loop
    increment j
  if j == m * k:
    return True
```

Explanation of crucial parts of the above pseudocode:

- `for i from 0 to (n - m * k + 1)`: It ensures that we do not start a pattern check where the remaining elements in the array are fewer than needed to make `k` repetitions of length `m`.
- `while j < m * k`: It's essential to check `m * k` elements for consecutive repetition.
- `if arr[i + j] != arr[i + (j % m)]`: This comparison is vital for the algorithm. The index `i + j` represents the current element we are checking, while `i + (j % m)` gives us the corresponding index in the original `m` length pattern with which we are comparing. If a mismatch is found, we break out of the inner loop, as the current starting index `i` cannot be the start of a valid pattern.
- `if j == m * k`: After the inner loop terminates, if `j`, the count of consecutive matches, equals `m * k`, it means we have found a pattern repeated `k` times. Hence, we return `True`.

If the main loop terminates without returning `True`, no pattern of length `m` repeated `k` times has been found, so the solution returns `False`.

This approach effectively employs a brute-force mechanism to check for the pattern in all possible places by using a nested loop where the inner loop validates the repetitiveness of the pattern while the outer loop shifts the starting position of the check.

Example Walkthrough

Let's illustrate the solution approach using a small example. Suppose we have the array `arr = [4, 5, 4, 5, 4, 5, 6]`, and we want to check if there's a subarray of length `m = 2` that repeats `k = 3` times.

Step-by-step:

1. We start by calculating the length of the array `n`, which is `7` in this case.
2. We begin iterating through the array, starting at index `i = 0`. The outer loop will only go up to index `i = 7 - (2 * 3) + 1 = 2` to ensure there's enough room for a subarray of length `m` repeated `k` times.
3. With each `i`, we have:
   - At `i = 0`: We check if `[4, 5]` repeats `3` times. The inner loop will check the elements following the index `i` to see if they match the initial subarray `[4, 5]`. We check `arr[0]` with `arr[0 + 0 % 2]` and `arr[1]` with `arr[0 + 1 % 2]`, then `arr[2]` with `arr[0 + 2 % 2]` and `arr[3]` with `arr[0 + 3 % 2]`, and so on until we have checked `2 * 3 = 6` elements for consecutiveness. Since the pattern `[4, 5]` repeats for the required `3` times, `j` will equal `6` at the end of the while-loop, and we will return `True`.
4. Since we found a valid pattern starting at `i = 0`, there is no need to continue with further iterations.

Thus, for the given array `arr`, the function will return `True`.

This example demonstrates the simplicity and effectiveness of the brute force solution approach in finding whether the array contains a subarray that repeats consecutively `k` times.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def contains_pattern(self, arr: List[int], pattern_length: int, repetitions: int) -> bool:
        # Calculate the size of the array
        array_length = len(arr)

        # Loop over the array up to the point where the pattern can possibly fit
        for start_index in range(arr_length - pattern_length * repetitions + 1):
            # Initialize a pointer to traverse the pattern
            pattern_index = 0

            # Keep traversing while the pattern matches the subsequent blocks of the same size
            while pattern_index < pattern_length * repetitions:
                # If an element does not match its corresponding element in the first block, break
                if arr[start_index + pattern_index] != arr[start_index + (pattern_index % pattern_length)]:
                    break
                # Move to the next element in the pattern
                pattern_index += 1

            # If we have traversed the entire pattern without a break, the pattern is present
            if pattern_index == pattern_length * repetitions:
                return True

        # If we exit the loop without returning True, the pattern is not present
        return False
```

Java

```java
class Solution {

    // Function to check if the arrav contains a repeated pattern of length m, repeated k times.
    public boolean containsPattern(int[] arr, int m, int k) {
        // Find the length of the array.
        int n = arr.length;

        // Loop through the array up to the point where the pattern could fit.
        for (int i = 0; i <= n - m * k; ++i) {

            // Initialize 'j' which will iterate over the length of the pattern times 'k'.
            int j = 0;
            for (; j < m * k; ++j) {

                // Check if the current element doesn't match with the correspondina element in the pattern.
                // The modulo operation finds the corresponding position in the pattern.
                if (arr[i + j] != arr[i + (j % m)]) {
                    break; // If any element doesn't match, break the loop.
                }
            }

            // If 'j' runs through the full pattern without breaking, the pattern exists in the array.
            if (j == m * k) {
                return true;
            }
        }

        // If we traverse the entire array without returning true, the pattern does not exist.
        return false;
    }
}
```

C++

```cpp
#include <vector>

class Solution {
public:
    bool containsPattern(std::vector<int>& arr, int m, int k) {
        int size = arr.size();
        // Loop through the array, but only up to the point where we can fit m * k elements
        for (int i = 0; i <= size - m * k; ++i) {
            int patternLength = 0;
            // Try to match a pattern of length m, repeated k times
            for (; patternLength < m * k; ++patternLength) {
                // The index for pattern comparison is the current index i
                // plus the current patternLength, but wrapped by m
                // to restart comparison every m elements
                if (arr[i + patternLength] != arr[i + (patternLength % m)]) {
                    break; // Pattern does not match, break and move to next starting index
                }
            }
            // If we matched a full pattern (mxk elements)
            if (patternLength == m * k) {
                return true; // A repeat pattern of length m, repeated k times is found
            }
        }
        // After checking the entire array, no pattern was found
        return false;
    }
};
```

TypeScript

```typescript
function containsPattern(arr: number[], m: number, k: number): boolean {
    // Get the length of the array.
    const arrayLength = arr.length;

    // Loop through the array, but only up to the point where a pattern of length m repeated k times could fit.
    for (let startIndex = 0; startIndex <= arrayLength - m * k; ++startIndex) {
        let patternIndex;

        // Check if the pattern repeats k times from the current starting index.
        for (patternIndex = 0; patternIndex < m * k; ++patternIndex) {
            // The pattern breaks if the current element does not match the corresponding element in the pattern.
            if (arr[startIndex + patternIndex] !== arr[startIndex + (patternIndex % m)]) {
                break;
            }
        }

        // If the loop completed, the pattern was found repeated k times.
        if (patternIndex === m * k) {
            return true;
        }
    }

    // If no matching pattern repetition was found, return false.
    return false;
}
```

```python
from typing import List

class Solution:
    def contains_pattern(self, arr: List[int], pattern_length: int, repetitions: int) -> bool:
        # Calculate the size of the array
        array_length = len(arr)

        # Loop over the array up to the point where the pattern can possibly fit
        for start_index in range(arr_length - pattern_length * repetitions + 1):
            # Initialize a pointer to traverse the pattern
            pattern_index = 0

            # Keep traversing while the pattern matches the subsequent blocks of the same size
            while pattern_index < pattern_length * repetitions:
                # If an element does not match its corresponding element in the first block, break
                if arr[start_index + pattern_index] != arr[start_index + (pattern_index % pattern_length)]:
                    break
                # Move to the next element in the pattern
                pattern_index += 1

            # If we have traversed the entire pattern without a break, the pattern is present
            if pattern_index == pattern_length * repetitions:
                return True

        # If we exit the loop without returning True, the pattern is not present
        return False
```

Time and Space Complexity

Time Complexity

The main operation of the algorithm is a nested loop where the outer loop runs `n - m * k + 1` times. The inner loop runs up to `m * k` times, but often breaks earlier if the pattern condition is not met.

The worst-case scenario occurs when `arr[i + j] == arr[i + (j % m)]` for each `i` and `j` until the last iteration, which means that even though we do not have a complete pattern, each partial comparison is true. If we assume the worst-case, the inner loop would run `m * k` for each of the `n - m * k + 1` iterations.

As a result, the time complexity in the worst case is $O((n - m * k + 1) * m * k)$, which simplifies to $O(n * m * k)$.

Space Complexity

The algorithm uses a fixed amount of space, with only simple variables defined and no use of any data structures that grow with the input size.

Therefore, the space complexity is $O(1)$.