2369. Check if There is a Valid Partition For The Array

Medium Array Dynamic Programming Leetcode Link

Problem Description

In this problem, we're given an integer array `nums`, and our goal is to partition this array into one or more contiguous subarrays. A partition is considered valid if each resulting subarray meets one of three specific conditions:

1. The subarray consists of exactly 2 equal elements, such as `[2, 2]`.
2. The subarray consists of exactly 3 equal elements, like `[4, 4, 4]`.
3. The subarray consists of exactly 3 consecutive increasing elements, which means the difference between adjacent elements is 1. An example of such a subarray is `[3, 4, 5]`.

We need to determine if there's at least one such valid partition for the given array, and we should return `true` if there is, or `false` otherwise.

Intuition

The intuition behind the solution is to use Depth First Search (DFS) to explore all possible ways to partition the array. Starting from the beginning of the array, we recursively check if we can form a valid subarray according to the rules given. If we can, we proceed to explore partitions of the remaining part of the array.

Here's the thought process for DFS:

1. If we reach the end of the array (index `i` equals the length of the array `n`), it means we've successfully partitioned the array into valid subarrays, so we return `true`.
2. Starting from the current index `i`, we check if we can form a valid subarray with the next one or two elements based on the given conditions:
   - If the current and the next element are the same, we check the possibility of forming a valid partition with the subsequent elements by calling the DFS for the index `i + 2`.
   - If the current and the next two elements are the same, we again call the DFS for index `i + 3`.
   - If the current element and the next two elements form a sequence of three consecutive increasing numbers, we call the DFS for the index `i + 3`.
3. We consider a partition valid if any of these subarray conditions lead to a successful partition of the rest of the array.
4. To optimize the solution, we use memoization (`@cache`) to avoid re-computing the same state of the array multiple times.

By exploring all possible partitions in a DFS manner and using memoization to reduce duplicate work, we can determine if the array has at least one valid partition in an efficient way.

Solution Approach

The solution uses a recursive Depth First Search (DFS) approach with memoization to avoid redundant computations. In the code, we define a recursive function `dfs` that attempts to find a valid partition starting from index `i`. The base case for our recursive function is when `i` is equal to `n`, the length of the array, which means we have successfully reached the end of the array with a valid partition and thus return `true`.

Let's walk through the implementation details:

- **Recursion and DFS**: The core of the solution is the recursive function `dfs(i)` which is called with the starting index `i` to explore all possible valid subarrays from that index onwards.
- **Memoization**: This function is decorated with `@cache`, which is a feature in Python used for memoization. It stores the results of the function calls with particular arguments so that future calls with the same arguments can return the stored result immediately, without re-executing the whole function.
- **Base Case**: When `i` equals `n`, we've reached the end and hence return `true`, implying a successful partition.
- **Recursive Calls**:
  - If `nums[i]` is equal to `nums[i + 1]`, we attempt to create a subarray of length 2 by making a recursive call to `dfs(i + 2)`.
  - If `nums[i]`, `nums[i + 1]`, and `nums[i + 2]` are equal, we attempt to create a subarray of length 3 by making a recursive call to `dfs(i + 3)`.
  - If `nums[i]`, `nums[i + 1]`, and `nums[i + 2]` form a sequence of three consecutive increasing numbers, we attempt to create a subarray of length 3 by making a recursive call to `dfs(i + 3)`.
- **Early Termination**: The `res` variable is used to keep track of whether a valid partition has been found. As soon as we find a valid partition, `res` becomes `true` and subsequent calls won't be made since the `or` operator short-circuits.
- **Efficient Checking**: To evaluate the conditions, the implementation uses simple if checks and arithmetic comparisons. No additional data structures are required, which keeps the space complexity low.

Finally, `dfs(0)` is called to initiate the partitioning process starting from the first element of the array, and the result of this call determines whether at least one valid partition exists. If it returns `true`, the entire array can be partitioned according to the given rules. If not, the array does not have a valid partition configuration.

Example Walkthrough

Let's consider an example `nums` array to illustrate the solution approach: `[3, 3, 2, 2, 1, 1, 2, 3, 4]`.

Here is a step-by-step walk-through:

1. Start with index `i = 0`, attempt to find valid subarrays beginning at that index.
2. Since `nums[0]` and `nums[1]` are equal (`3`, `3`), try to make a subarray `[3, 3]` and call `dfs(2)` for the next index.
3. At index `i = 2`, `nums[2]` and `nums[3]` are also equal (`2`, `2`), so take subarray `[2, 2]` and call `dfs(4)` for the next index.
4. At index `i = 4`, `nums[4]` and `nums[5]` are equal (`1`, `1`), form subarray `[1, 1]` and call `dfs(6)` for the next index.
5. At index `i = 6`, `nums[6]`, `nums[7]`, and `nums[8]` form a sequence of three consecutive increasing numbers (`2`, `3`, `4`), so take subarray `[2, 3, 4]` and call `dfs(9)` for the next index.
6. Now `i = 9` which equals `n` (the length of `nums`). According to our base case, we've successfully found a partition for the entire array and return `true`.

With this walk-through, you can see that the original array can be partitioned into valid subarrays `[3, 3]`, `[2, 2]`, `[1, 1]`, and `[2, 3, 4]` conforming to the given rules. The DFS approach systematically checks for each subarray possibility at every index, and the use of memoization ensures that we do not perform redundant calculations, leading to an efficient solution.

Python Solution

```python
from typing import List
from functools import lru_cache

class Solution:
    def validPartition(self, nums: List[int]) -> bool:
        # Calculate the length of the nums list.
        length_of_nums = len(nums)

        # Define the depth-first search function with memoization.
        @lru_cache(maxsize=None)
        def dfs(index):
            # Base case: if the whole array has been checked, return True
            if index == length_of_nums:
                return True

            # Initialize the result as False.
            res = False

            # Check if the current and next items are the same and recurse for the remaining array.
            if index + 1 < length_of_nums and nums[index] == nums[index + 1]:
                res = res or dfs(index + 2)

            # Check for three consecutive elements with the same value and recurse.
            if index + 2 < length_of_nums and nums[index] == nums[index + 1] == nums[index + 2]:
                res = res or dfs(index + 3)

            # Check for a sequence of three consecutive increasing numbers and recurse.
            if index + 2 < length_of_nums and nums[index + 1] - nums[index] == 1 and nums[index + 2] - nums[index + 1] == 1:
                res = res or dfs(index + 3)

            # Return the result.
            return res

        # Call the dfs function starting from index 0.
        return dfs(0)
```

Java Solution

```java
import java.util.Arrays; // Import necessary for Arrays.fill

class Solution {
    // Class-wide variables to hold the state of the problem.
    private int[] arrayLength;
    private int[] memo;
    private int[] numbers;

    // The function to be called to check if a valid partition exists.
    public boolean validPartition(int[] nums) {
        this.numbers = nums;
        arrayLength = nums.length;
        memo = new int[arrayLength]; // memo array for memoization to avoid re-computation.
        Arrays.fill(memo, -1); // Initialize all elements of memo to -1.
        return dfs(0); // Start the depth-first search from the first element.
    }

    // Helper method to perform depth-first search and check for a valid partition.
    private boolean dfs(int index) {
        if (index == arrayLength) { // Base case: if we've reached the end of the array, return true.
            return true;
        }
        if (memo[index] != -1) { // If we have a memoized result, return it.
            return memo[index] == 1;
        }

        boolean result = false; // Initialize the result as false initially.

        // Check if the current and next elements are the same, which forms a valid partition of two elements.
        if (index < arrayLength - 1 && numbers[index] == numbers[index + 1]) {
            result = result || dfs(index + 2); // Recursively check the partition from the next index.
        }

        // Check if three consecutive elements are identical, which forms a valid partition.
        if (index < arrayLength - 2 && numbers[index] == numbers[index + 1] && numbers[index + 1] == numbers[index + 2]) {
            result = result || dfs(index + 3); // Recursively check the partition from the next index.
        }

        // Check if three consecutive elements form a contiguous sequence, which is also a valid partition.
        if (index < arrayLength - 2 && numbers[index + 1] - numbers[index] == 1 && numbers[index + 2] - numbers[index + 1] == 1) {
            result = result || dfs(index + 3); // Recursively check the partition from the next index.
        }

        memo[index] = result ? 1 : 0; // Memoize the result for the current index.
        return result; // Return the result of the current recursion.
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    vector<int> memo; // memoization table to store intermediate results
    vector<int> numbers; // reference to the input array
    int size; // size of the input array

    // Main function to check if the given vector can be partitioned according to the rules
    bool validPartition(vector<int>& nums) {
        size = nums.size();
        numbers = nums;
        memo.assign(size, -1); // initialize memoization table with -1, indicating uncomputed states
        return dfs(0); // start the recursion from the first element
    }

    // Helper function using Depth-First Search and memoization to find if valid partition exists
    bool dfs(int index) {
        // Base case: if we have reached the end, the partition is valid
        if (index == size) return true;

        // If this state has been computed before, return its result
        if (memo[index] != -1) return memo[index] == 1;

        // Initialize the validity of the current position's partition as false
        bool isValidPartition = false;

        // Check for two consecutive numbers with the same value
        if (index < size - 1 && numbers[index] == numbers[index + 1])
            isValidPartition = isValidPartition || dfs(index + 2);

        // Check for three consecutive numbers with the same value
        if (index < size - 2 && numbers[index] == numbers[index + 1] && numbers[index + 1] == numbers[index + 2])
            isValidPartition = isValidPartition || dfs(index + 3);

        // Check for three consecutive numbers forming a strict increasing sequence by 1
        if (index < size - 2 && numbers[index + 1] - numbers[index] == 1 && numbers[index + 2] - numbers[index + 1] == 1)
            isValidPartition = isValidPartition || dfs(index + 3);

        // Store the result in memoization table before returning
        memo[index] = isValidPartition ? 1 : 0;

        return isValidPartition;
    }
};
```

Typescript Solution

```typescript
function validPartition(nums: number[]): boolean {
    const length = nums.length;
    const visited = new Array(length).fill(false); // This keeps track of visited indices to prevent reprocessing.
    const queue: number[] = [0]; // Queue for breadth-first search, starting with the first index.

    // Process the queue until it's empty.
    while (queue.length > 0) {
        const currentIndex = queue.shift()!; // Safely extract an element since queue is non-empty.

        // If we've reached the end of the array, return true.
        if (currentIndex === length) {
            return true;
        }

        // Condition to check if a partition of two identical numbers is possible
        if (!visited[currentIndex + 2] && currentIndex + 1 < length && nums[currentIndex] === nums[currentIndex + 1]) {
            queue.push(currentIndex + 2); // Add next index to the queue.
            visited[currentIndex + 2] = true; // Mark this index as visited.
        }

        // Condition to check if a partition of three consecutive numbers (either identical or increasing) is possible
        if (
            !visited[currentIndex + 3] &&
            currentIndex + 3 <= length &&
            ((nums[currentIndex] === nums[currentIndex + 1] && nums[currentIndex + 1] === nums[currentIndex + 2]) ||
                (nums[currentIndex] === nums[currentIndex + 1] - 1 && nums[currentIndex + 1] === nums[currentIndex + 2] - 1))
        ) {
            queue.push(currentIndex + 3); // Add next index to the queue.
            visited[currentIndex + 3] = true; // Mark this index as visited.
        }
    }

    // If the loop completes without returning true, there is no valid partition.
    return false;
}
```

Time and Space Complexity

The given code defines a recursive function `dfs` that uses memoization to return whether a valid partition exists starting from the `i`-th index of the array `nums`. The function recursively tries to partition the array into groups of two or three adjacent elements, either with the same value or forming a consecutive increasing sequence.

Time Complexity:

The time complexity of the `dfs` function primarily depends on the number of subproblems it needs to solve, which is directly related to the length of `nums` (`n`). Due to memoization (through the use of the `@cache` decorator), each subproblem (each starting index of the array) is solved only once. There are `n` possible starting indices.

For each index, there are at most three recursive calls to check for three different partitioning conditions. However, these recursive calls don't multiply the complexity because memoization ensures that we do not recompute results for the same inputs.

Therefore, the time complexity of the algorithm is $O(n)$, as each index is processed a constant number of times.

Space Complexity:

The space complexity consists of the memory used for the memoization cache and the stack space used by the recursive calls.

1. Memoization cache: The cache will store results for each index of the array, leading to a space complexity of $O(n)$ for the memoization cache.
2. Recursive stack: In the worst case, the function might end up calling itself recursively $n/2$ times if the partition consists entirely of pairs of identical numbers (`[1, 1, 2, 2, ..., n/2, n/2]`). This would lead to a stack depth of $n/2$. Hence, the worst-case space complexity for the call stack is $O(n)$.

Considering both the memoization cache and the recursive call stack, the overall space complexity is also $O(n)$.