2547. Minimum Cost to Split an Array

Hard · Array · Hash Table · Dynamic Programming · Counting · Leetcode Link

Problem Description

In this problem, we are given an array of integers, `nums`, and an integer `k`. The goal is to split `nums` into several non-empty subarrays so that the total cost of the split is minimal. The cost of a split is determined by the sum of the importance values of all the subarrays produced in the split.

Here's the interesting part: an importance value of a subarray is calculated differently than one might expect. First, you create a `trimmed(subarray)` by removing all numbers that appear only once within that subarray. The importance value is then `k + the length of the trimmed(subarray)`.

We are asked to find the minimum possible cost of a split for the array `nums`.

Intuition

The intuition behind solving this problem lies in understanding that we have to find the optimal point to split the array such that we minimize the cost each time. A brute-force approach would try every possible split, but that would be prohibitively expensive in terms of time complexity. Instead, we need an approach that efficiently evaluates the splits.

The solution uses dynamic programming (DP), which is a common technique to solve optimization problems like this one. In a DP-based approach, we try to break down the problem into smaller subproblems and then solve each subproblem only once, storing its solution so that we can reuse it later without recalculating it.

Here's the essence of the solution approach:

1. We define a recursive function `dfs(i)` which tries to find the minimum cost of splitting the subarray starting at index `i`.
2. For each position `j` starting from `i` to the end of the array, we include `nums[j]` in our current subarray and update the count of each number seen so far (using the `Counter` collection).
3. When the count of a number changes from 1 to 2, it means we have a duplicate and the `trimmed` subarray's length increases (by 1 for each duplicate pair found), decreasing the number of single occurrences by 1. The importance value of this subarray becomes `k + (j - i + 1 - number of single occurrences)`.
4. We recursively call `dfs(j + 1)` to calculate the cost of the split for the remainder of the array.
5. We keep track of the minimum cost (`ans`) and update it as we evaluate different splits starting from index `i`.
6. To avoid recalculating the cost of subproblems we've already solved, we use memoization (`@cache` decorator), which stores the result of `dfs(j + 1)` and reuses it when the same subproblem is encountered again.

The DP solution hence calculates the minimum split cost starting from the first index, using the recursive `dfs` function and memoization to efficiently find the minimum total cost.

Solution Approach

The Reference Solution Approach provided uses a recursive depth-first search (DFS) strategy with memoization. Let's walk through the implementation, highlighting the algorithms, data structures, and patterns used:

1. **Memoization with `@cache` decorator**: This function decorator, part of Python's functools module, is used on the `dfs(i)` function to memorize previously computed results for different starting indices `i`. Each time `dfs(i)` is called, the result is stored, and future calls with the same `i` quickly return the stored result instead of recomputing it. This reduces the overall time complexity significantly.
2. **Depth-First Search (DFS)**: The `dfs(i)` function models our recursive approach and represents the core of our algorithm. Starting from index `i`, it explores all possible splits by gradually including each subsequent element (`nums[j]`) into the current subarray and recursively finding the cost of the best split for the rest of the array.
3. **The use of `Counter`**: A `Counter` is a subclass of dictionary that is used to keep count of hashable objects. Here, it counts the occurrences of each number in the current subarray being considered. Since the trimmed array should only contain elements with more than one appearance, we use the counters to keep track of when elements appear for the first or second time.
4. **Trimming logic**: Inside the loop, when `cnt[nums[j]]` is incremented, it checks for the numbers appearing for the first time (`one += 1`) and those transitioning from a unique to a duplicate occurrence (`one -= 1`). The length of the trimmed subarray at any point is `j - i + 1 - one`, where `(j - i + 1)` is the initial subarray length, and `one` is the count of unique elements to be excluded.
5. **Minimization**: As we loop through the array and consider each element as a potential split point, the function calculates the cost of the current subarray plus the cost of the best split of the remainder and updates the answer `ans` to the minimum of this value and any previously found minimum.
6. **The base case of the recursion**: When the index `i` reaches or surpasses the length of `nums`, the recursion stops, and the function returns 0 (since there are no more elements to split).
7. **Function Call and Return Value**: The recursive process is initiated by calling `dfs(0)` from the main function body, which kicks off the recursive exploration starting at the first element of the array. The minimum cost of splitting the array `nums` in an optimal way is then returned.

To sum up, the solution intelligently combines DFS for exhaustive search with memoization to cut down on redundant work, effectively handling an otherwise exponential time complexity in a much more manageable way.

Example Walkthrough

Let's use a small example to illustrate the solution approach. Suppose we have the following array `nums` and `k`:

```
nums = [1, 2, 2, 3]
k = 5
```

Our aim is to split this array into subarrays with the minimal total cost. The importance value is given by `k + length of trimmed(subarray)`. A `trimmed(subarray)` only includes elements that appear more than once.

Let's apply the solution approach to this example step-by-step:

1. **Initial state**: We start by calling `dfs(0)` which will attempt to find the minimum cost of splitting the array starting from index 0.
2. **DFS and memoization**: The `dfs` function is called recursively to explore all subarrays starting from index `i`. With the `@cache` decorator, results for each `dfs(i)` call are stored to avoid redundant calculations.
3. **Using a Counter to track occurrences**: We create a `Counter` to keep track of the number of occurrences of each number in our current subarray.
4. **Finding splits with DFS**: We try to find the best place to split the array such that the cost after the split is minimized. Starting from index `i = 0`, we explore each element subsequently to see if it should be part of the current subarray or start a new subarray.
   - For `i = 0`, we can split after each element and recursively check the cost:
     - Split `nums` into `[1]` and `[2, 2, 3]`: The trimmed part of `[1]` is empty, so its importance value is `k + 0 = 5`. Then we recursively calculate the cost for `[2, 2, 3]`.
     - Split `nums` into `[1, 2]` and `[2, 3]`: The trimmed part of `[1, 2]` is empty, so its importance value is `k + 0 = 5`. Then we recursively calculate the cost for `[2, 2, 3]`.
     - Split `nums` into `[1, 2, 2]` and `[3]`: The trimmed part of `[1, 2, 2]` contains the two `2`'s, so its importance value is `k + 2 = 7`. Then we recursively calculate the cost for `[3]`.
     - If `i` has reached the end of the array, the cost to split is `0` because there are no elements left to consider.
5. **Minimization**: As the `dfs` function explores these possibilities, it keeps track of the minimum cost found for each starting index `i` and updates the answer accordingly.
6. **Base case**: When our index `i` reaches the end of the array, we have no more elements to consider, and we return 0 as there's nothing left to split.
7. **Result:** The recursion happens in the background through our calls to `dfs`. Once all recursive calls are made, `dfs(0)` will return the minimum cost found, which gives us the answer we need.

In the given example, the minimum cost can be calculated by recursively considering each potential split. The final minimum cost to split the `nums` array is found by `dfs(0)`, which returns the accumulated minimum cost after considering all subarray possibilities.

Python Solution

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

class Solution:
    def minCost(self, nums: List[int], k: int) -> int:
        # Decorate the recursive function with lru_cache to memoize repetitive calls with the same arguments
        @lru_cache(None)
        def dfs(index):
            if index >= num_length:
                return 0
            # A counter for the occurrence of each number
            occurrences = Counter()
            # To keep track of the number of distinct elements
            unique_count = 0
            # Initialize the minimum cost to infinity
            min_cost = float('inf')
            for j in range(index, num_length):
                occurrences[nums[j]] += 1
                # If it's the first occurrence, increase the unique count
                if occurrences[nums[j]] == 1:
                    unique_count += 1
                # If we encounter a second occurrence, we decrease the unique count as it's not unique anymore
                elif occurrences[nums[j]] == 2:
                    unique_count -= 1
                # Calculate the cost combining the current segment cost and the cost from the remaining segments
                # k is the fixed cost, (j - index + 1) is the variable cost, unique_count is subtracted from the variable cost
                min_cost = min(min_cost, k + j - index + 1 - unique_count + dfs(j + 1))
            return min_cost

        # Calculate the overall length of the nums list
        num_length = len(nums)
        # Start the recursive function from the first index
        return dfs(0)

```

Java Solution

```java
class Solution {
    private Integer[] memoizationArray; // Used for memoization to store results of subproblems
    private int[] sequence; // The input sequence of numbers
    private int sequenceLength, maxIngredients; // Size of the input sequence and maximum unique ingredients

    // Helper method to calculate the minimum cost to make all dishes tasty
    public int minCost(int[] nums, int k) {
        sequenceLength = nums.length; // Store the length of the sequence
        maxIngredients = k; // Store the maximum allowed unique ingredients
        sequence = nums; // Assigning the input sequence to the class variable 'sequence'
        memoizationArray = new Integer[sequenceLength]; // Initialize memoization array
        return dfs(0); // Begin the depth-first search from index 0
    }

    // Depth-first search function to find minimum cost with memoization
    private int dfs(int currentIndex) {
        if (currentIndex >= sequenceLength) { // Base case: if we've reached beyond the last index
            return 0; // No cost if outside bound
        }
        if (memoizationArray[currentIndex] != null) { // If we've already computed this subproblem
            return memoizationArray[currentIndex]; // Return the stored result
        }
        int[] countUniqueIngredients = new int[sequenceLength]; // Array to count occurrences of ingredients
        int singleOccurrenceCount = 0; // Count of ingredients that appear exactly once
        int minCost = Integer.MAX_VALUE; // Initialize minimum cost with a large number
        for (int j = currentIndex; j < sequenceLength; ++j) {
            // Increment the count for the current ingredient
            int ingredientCount = ++countUniqueIngredients[sequence[j]];
            if (ingredientCount == 1) { // If the ingredient is unique (first occurrence)
                ++singleOccurrenceCount; // Increment count of unique ingredients
            } else if (ingredientCount == 2) { // If the ingredient occurred once before
                --singleOccurrenceCount; // Decrement count of unique ingredients
            }
            // Calculate cost for current segment and proceed to solve next subproblem recursively
            int currentCost = maxIngredients + (j - currentIndex + 1) - singleOccurrenceCount + dfs(j + 1);
            minCost = Math.min(minCost, currentCost); // Update minimum cost for this partition
        }
        // Store the result in the memoization array and return it
        return memoizationArray[currentIndex] = minCost;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <cstring>
#include <functional>

class Solution {
public:
    // Calculates the minimum cost to partition the vector such that each part contains unique numbers
    int minCost(vector<int>& nums, int k) {
        int size = nums.size();
        vector<int> memo(size, 0); // Used for memoization to store the results of subproblems

        // A recursive lambda function to compute the minimum cost via depth-first search
        function<int(int)> dfs = [&](int index) {
            // Base case: if the index is beyond the end of the vector
            if (index >= size) {
                return 0;
            }
            // If the subproblem is already solved, return the stored result
            if (memo[index]) {
                return memo[index];
            }
            vector<int> counts(size, 0); // Counts occurrences of numbers
            int uniqueCount = 0; // Counts unique numbers
            int cost = INT_MAX; // Initialize cost to a large value

            // Try splitting the vector from the current index to the end
            for (int j = index; j < size; ++j) {
                int currentValue = ++counts[nums[j]]; // Increment the count of the current number
                if (currentValue == 1) {
                    // If the number is unique
                    ++uniqueCount;
                } else if (currentValue == 2) {
                    // If this is the second occurrence of the number
                    --uniqueCount;
                }
                // Calculate minimum cost using the solution of the subproblem
                cost = min(cost, k + (j - index + 1 - uniqueCount) + dfs(j + 1));
            }
            // Store the computed result for the current index
            return memo[index] = cost;
        };

        // Start the recursive computation from the beginning of the vector
        return dfs(0);
    }
};

```

Typescript Solution

```typescript
function minCost(nums: number[], costFactor: number): number {
    const lengthOfNums = nums.length;
    const costMemo = new Array(lengthOfNums).fill(0);

    // Helper function to perform depth-first search and memoization.
    const calculateMinCost = (index: number): number => {
        // Base case: if index is out of bounds, no cost is incurred.
        if (index >= lengthOfNums) {
            return 0;
        }

        // If the cost at the current index is already computed, return it to avoid redundant calculations.
        if (costMemo[index]) {
            return costMemo[index];
        }

        // Keep a count of occurrences of each number.
        const occurrenceCount = new Array(lengthOfNums).fill(0);
        let uniqueNumbers = 0; // Count of unique numbers.
        let minCost = Number.MAX_SAFE_INTEGER; // Initialize to maximum safe integer.

        // Iterate through the array from the current position.
        for (let j = index; j < lengthOfNums; ++j) {
            const currentCount = ++occurrenceCount[nums[j]];

            // Update the count of unique numbers.
            if (currentCount === 1) {
                uniqueNumbers++;
            } else if (currentCount === 2) {
                uniqueNumbers--;
            }

            // Calculate the minimum cost considering the current partition.
            minCost = Math.min(
                minCost,
                costFactor + (j - index + 1 - uniqueNumbers) + calculateMinCost(j + 1)
            );
        }

        // Memoize the computed cost for the current index.
        costMemo[index] = minCost;
        return minCost;
    };

    // Start the recursive depth-first search calculation from the first index.
    return calculateMinCost(0);
}

```

Time and Space Complexity

The given Python code defines a recursive function `dfs` that is memoized using the `@cache` decorator, which is used to compute the minimum cost of splitting the list `nums` into groups of size at least `k`.

Time Complexity

The time complexity of the `dfs` function is determined by the number of unique subproblems times the complexity to solve each subproblem. The function `dfs` is called for each index `i` from `0` to `n-1`, where `n` is the length of `nums`. Within each call, there is a for-loop from `i` to `n-1` which involves updating the Counter `cnt` and calculating the minimum cost. The `dfs` function is called recursively for each possible next start index `j + 1`.

The number of unique subproblems is `O(n)`, which corresponds to the number of possible starting indices for the `dfs` call. For each subproblem, the for-loop can iterate up to `n - i` times. In the worst case, this can result in `O(n^2)` iterations for each subproblem.

The `Counter` operations inside the loop can be `O(1)` on average if we assume a reasonable number of unique elements, but this could degrade to `O(n)` in the worst case (if all the elements are unique).

Therefore, the overall time complexity would be `O(n^3)` in the worst case when we combine the number of subproblems with the work done per subproblem.

Space Complexity

The space complexity comprises the space used by the recursion call stack and the memoization cache.

1. The maximum depth of the recursion stack is `O(n)`, since `dfs` might be called for each element in `nums`.
2. The memoization cache will hold at most `O(n)` states of the computation, as it caches results for each unique call of `dfs`.
3. Additionally, there is the space used by the Counter `cnt` which, in the worst case, could store `n` unique elements amounting up to `O(n)` space.

Combining these factors, the space complexity of the algorithm is `O(n)`.