813. Largest Sum of Averages

Medium Array Dynamic Programming Prefix Sum Leetcode Link

Problem Description

The problem deals with an integer array `nums` and an integer `k`. We're tasked with partitioning the array into at most `k` non-empty adjacent subarrays. Our goal is to maximize the 'score' of the partition, where the score is defined as the sum of the averages of each subarray. Important to note is that we must use every integer in `nums` and that the partitions we create will affect the score. We need to find the best possible way to split the array to achieve the maximum score.

Intuition

The core of this problem lies in dynamic programming, particularly in finding the best partition at each stage to maximize the score. A naïve approach might try every possible partition, but this would be prohibitively slow. Instead, we need an efficient way to solve smaller subproblems and combine their solutions to solve the larger problem.

The intuition behind the dynamic programming solution is that if we know the best score we can get from the first `i` elements with `k` subarrays, we can use this to compute the best score for the first `i+1` elements with `k` subarrays, and so on. The recursive function `dfs` represents this notion, where `dfs(i, k)` returns the best score we can get starting from index `i` with `k` subarrays left to form.

The solution uses a top-down approach with memoization (caching) to avoid re-computing the scores for the same `i` and `k` values more than once. It also uses a prefix sum array `s` to quickly compute the sum of elements for any given subarray, which is used to calculate the averages needed to compute the scores for the partitions.

By caching results and avoiding repetition of work, we arrive at the best solution in a much faster and more efficient way than brute-force methods.

Solution Approach

The solution uses several algorithms and concepts:

Dynamic Programming:

Dynamic programming is utilized to break the problem down into smaller, manageable subproblems. A function `dfs` recursively computes the maximum score obtainable from a given starting index `i` and a certain number of partitions `k`. The essence of dynamic programming is apparent as `dfs` computes the scores based on previously solved smaller problems.

Memoization:

To optimize the dynamic programming solution, memoization is used. The `@cache` decorator in Python automatically memorizes the result of the `dfs` function calls with particular arguments, so when the function is called again with the same arguments, the result is retrieved from the cache instead of re-computing it.

Prefix Sums:

To efficiently calculate the average of any subarray within `nums`, a prefix sum array `s` is created using the `accumulate` function from Python's `itertools` module, with `initial=0` to include the starting point. This data structure allows constant-time retrieval of the sum of elements between any two indices.

Implementation Details:

The main function `largestSumOfAverages` first computes the cumulative sums of `nums`. Then it defines the `dfs` function with parameters `i` (the starting index for considering partitions) and `k` (the number of remaining partitions).

The `dfs` function works as follows:

- If `i == n`, where `n` is the length of `nums`, it means we've considered all elements and there is no score to be added, so it returns `0`.
- If `k == 1`, we can only make one more partition, so the best score is the average of all remaining elements, calculated by `(s[-1] - s[i]) / (n - i)`.
- For other cases, the function iterates through the elements starting from `i` up to the last element, creating a subarray from `i` to `j` and calculating its average. It then recursively calls `dfs(j + 1, k - 1)` to compute the score of the remaining array with one less partition.
- The `max` function keeps track of the highest score found during the iteration.

At the end of the `dfs` calls, `dfs(0, k)` provides the maximum score for the entire array with `k` partitions.

The implementation takes advantage of Python's concise syntax and powerful standard library functions like `itertools.accumulate` and `functools.cache` to create an elegant and efficient solution.

Example Walkthrough

Let's assume we have `nums = [9, 1, 2, 3, 9]` and `k = 3`. We want to find the maximum score by partitioning the array into at most `k` adjacent subarrays.

First, let's calculate the prefix sums to efficiently compute the sums of subarrays:

`nums = [9, 1, 2, 3, 9]` `prefix_sums = [0, 9, 10, 12, 15, 24]` (include a 0 at the beginning for easy calculation)

Here's the Python function `dfs(i, k)` that will compute the maximum score starting from index `i` with `k` partitions left.

Now, let's walk through the solution:

Step 1: Call `dfs(0, 3)` for the full array with 3 partitions allowed.

Step 2: `dfs(0, 3)` explores partitioning the array from index 0. It tries partitioning after every index to find the maximum score:

- Partition after index 0: average of first part `[9]` is `9`, now call `dfs(1, 2)`.
- Partition after index 1: average of first part `[9, 1]` is `5`, now call `dfs(2, 2)`.
- Partition after index 2: average of first part `[9, 1, 2]` is `4`, now call `dfs(3, 2)`.
- Partition after index 3: average of first part `[9, 1, 2, 3]` is `3.75`, now call `dfs(4, 2)`.

Step 3: For each of the above calls to `dfs(i, 2)`, it again divides the remaining part of the array and calls `dfs(j, 1)`. When `k == 1`, it simply takes the average of the remaining elements since it has to be one partition.

Step 4: Each time the score is computed, we take the average of the current partition plus the result of `dfs` for the remaining partitions. The maximum value from these recursive calls is the answer for the current `dfs`.

Step 5: Following the steps recursively will lead us to find the maximum score. For instance, one of the optimal solutions is to partition the array into `[9], [1, 2, 3], [9]` with scores `9 + 2 + 9 = 20`.

Step 6: Once all possibilities for `dfs(0, 3)` are evaluated, the maximum score that can be returned is cached to ensure that the same state is not recomputed.

Step 7: Finally, `dfs(0, k)` returns the maximum score for the entire array with `k` partitions.

The recursive and memoizing nature of the `dfs` function allows us to efficiently explore all possibilities, while the prefix sums provide a quick way to calculate averages as needed without repeated summation. By combining these techniques, the algorithm efficiently finds the solution.

Python Solution

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

class Solution:
    def largestSumOfAverages(self, nums: List[int], k: int) -> float:
        # Total number of elements in nums
        num_elements = len(nums)

        # Prefix sum array including an initial 0 for convenience
        prefix_sums = list(accumulate(nums, initial=0))

        # Memoization function for our depth-first search
        @lru_cache(maxsize=None)
        def dfs(index, remaining_groups):
            # Base case: when we have considered all elements
            if index == num_elements:
                return 0
            # When only one group is left, return the average of the remaining elements
            if remaining_groups == 1:
                return (prefix_sums[-1] - prefix_sums[index]) / (num_elements - index)

            max_average_sum = 0
            # Try to form a group ending at each element from the current index
            for j in range(index, num_elements):
                # Current group average sum
                current_average = (prefix_sums[j + 1] - prefix_sums[index]) / (j - index + 1)
                # Recursively calculate the sum of averages for the remaining groups
                total_average_sum = current_average + dfs(j + 1, remaining_groups - 1)
                # Update the maximum average sum encountered
                max_average_sum = max(max_average_sum, total_average_sum)

            return max_average_sum

        # Launch depth-first search with initial position and group count
        return dfs(0, k)

```

Java Solution

```java
class Solution {
    private Double[][] memoization; // 2D array for memoization to store intermediate results
    private int[] prefixSums; // 1D array to store the prefix sums of the input array
    private int length; // Length of the input array

    // Calculates the largest sum of averages
    public double largestSumOfAverages(int[] nums, int k) {
        length = nums.length;
        prefixSums = new int[length + 1]; // Array size is length+1 because we start from 1 for easy calculations
        memoization = new Double[length + 1][k + 1]; // using Double wrapper class to store null initially
        for (int i = 0; i < length; ++i) {
            prefixSums[i + 1] = prefixSums[i] + nums[i]; // Compute prefix sums
        }
        return dfs(0, k); // Begin depth-first search
    }

    // Performs depth-first search to find the maximum sum of averages
    private double dfs(int startIndex, int groups) {
        if (startIndex == length) {
            return 0; // Base case: when we've considered all elements
        }
        if (groups == 1) {
            // If only one group left, return the average of the remaining elements
            return (double)(prefixSums[length] - prefixSums[startIndex]) / (length - startIndex);
        }
        if (memoization[startIndex][groups] != null) {
            return memoization[startIndex][groups]; // Return cached result if available
        }
        double maxAverage = 0;
        for (int i = startIndex; i < length; ++i) {
            // Choose different points to split the array
            double current = (double)(prefixSums[i + 1] - prefixSums[startIndex]) / (i - startIndex + 1) + dfs(i + 1, groups - 1);
            maxAverage = Math.max(maxAverage, current); // Keep the maximum average found
        }
        memoization[startIndex][groups] = maxAverage; // Store the result in memoization array
        return maxAverage;
    }
}

```

C++ Solution

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

using namespace std;

class Solution {
public:
    double largestSumOfAverages(vector<int>& nums, int k) {
        int n = nums.size();
        vector<int> prefix_sum(n + 1, 0); // Create a vector to store the prefix sums
        vector<vector<double>> memo(n, vector<double>(k + 1, 0)); // Create a 2D vector for memoization

        // Calculate the prefix sums
        for (int i = 0; i < n; ++i) {
            prefix_sum[i + 1] = prefix_sum[i] + nums[i];
        }

        // Recursive lambda function for depth-first search
        function<double(int, int)> dfs = [&](int start, int partitions) -> double {
            if (start == n) return 0; // Base case: no more elements to partition
            if (partitions == 1) // Base case: only one partition is left
                return static_cast<double>(prefix_sum[n] - prefix_sum[start]) / (n - start);

            if (memo[start][partitions] > 0) return memo[start][partitions]; // If value already computed return it

            double max_average = 0;
            for (int end = start; end < n; ++end) {
                double current_average = static_cast<double>(prefix_sum[end + 1] - prefix_sum[start]) / (end - start + 1);
                double rest_average = dfs(end + 1, partitions - 1);
                max_average = max(max_average, current_average + rest_average); // Update max_average if the sum of averages is lar
            }

            return memo[start][partitions] = max_average; // Memoize and return the result
        };

        return dfs(0, k); // Initiate the recursive search with the entire array and k partitions
    }
};

```

Typescript Solution

```typescript
// Import required packages for utility functions if needed (TypeScript doesn't use imports in a similar way, but you may need exte

// Define the type alias for 2D array of numbers
type NumberMatrix = number[][];

// Define the function to calculate the largest sum of averages
function largestSumOfAverages(nums: number[], k: number): number {
    const n: number = nums.length;
    const prefixSum: number[] = new Array(n + 1).fill(0); // Create an array to store the prefix sums
    const memo: NumberMatrix = Array.from({length: n}, () => new Array(k + 1).fill(0)); // Create a 2D array for memoization

    // Calculate the prefix sums
    for (let i = 0; i < n; ++i) {
        prefixSum[i + 1] = prefixSum[i] + nums[i];
    }

    // Recursive function for depth-first search
    const dfs: (start: number, partitions: number) => number = (start, partitions) => {
        if (start === n) return 0; // Base case: no more elements to partition
        if (partitions === 1) // Base case: only one partition is left
            return (prefixSum[n] - prefixSum[start]) / (n - start);

        if (memo[start][partitions] > 0) return memo[start][partitions]; // If value already computed, return it

        let maxAverage: number = 0;
        for (let end = start; end < n; ++end) {
            const currentAverage: number = (prefixSum[end + 1] - prefixSum[start]) / (end - start + 1);
            const restAverage: number = dfs(end + 1, partitions - 1);
            maxAverage = Math.max(maxAverage, currentAverage + restAverage); // Update maxAverage if the sum of averages is larger
        }

        return memo[start][partitions] = maxAverage; // Memoize and return the result
    };

    return dfs(0, k); // Initiate the recursive search with the entire array and k partitions
}

// Example usage:
let nums = [9, 1, 2, 3, 9];
let k = 3;
console.log(largestSumOfAverages(nums, k)); // Output will be the result of the function

```

Time and Space Complexity

Time Complexity

The time complexity of the given recursive algorithm involves analyzing the number of subproblems solved and the time it takes to solve each subproblem. Here, `n` represents the length of the `nums` array, and `k` represents the number of partitions.

- There are at most `n * k` subproblems because for each starting index `i` in `nums`, there are at most `k` partitions possible.
- For each subproblem defined by a starting index `i` and a remaining partition count `k`, the algorithm iterates from `i` to `n-1` to consider all possible partition points.

The time taken for each subproblem is `O(n)` because of the for-loop from `i` to `n-1`. Therefore, the overall time complexity is `O(n^2 * k)`.

Space Complexity

The space complexity consists of the space required by the recursion stack and the caching of subproblem results.

- The maximum depth of the recursion stack is `k`, because the algorithm makes a recursive call with `k-1` whenever it makes a partition.
- The cache stores results for the `n * k` subproblems.

Therefore, the space complexity is `O(n * k)` due to caching, plus `O(k)` for the recursion stack, which simplifies to `O(n * k)`.