1130. Minimum Cost Tree From Leaf Values

Medium | Stack | Greedy | Array | Dynamic Programming | Monotonic Stack

Problem Description

The problem given asks us to construct binary trees based on certain conditions and calculate the sum of the values of each non-leaf node. The special conditions provided for the binary trees are:

- Each node in the tree either has no children (making it a leaf) or has exactly two children (non-leaf nodes).
- The sequence of values found at the leaf nodes must match the sequence given by array `arr` in an in-order traversal.
- For each non-leaf node, its value is determined as the product of the largest leaf value from its left subtree and the largest leaf value from its right subtree.

The goal is to find the binary tree which results in the smallest sum of the values of all the non-leaf nodes combined, and return this minimum possible sum. It's guaranteed that the result will fit within a 32-bit integer.

Intuition

The intuition behind the solution comes from the fact that constructing a tree with a lower sum of non-leaf nodes heavily depends on how we group the elements to form subtrees, and consequently, the products at non-leaf nodes. As such, we want to ensure that smaller numbers participate in more products while larger numbers should be used less often to keep the sum as small as possible.

A dynamic programming approach allows us to solve this efficiently. We can define a two-dimensional array `f` where `f[i][j]` represents the minimum sum of the values of non-leaf nodes for the subarray `arr[i]` to `arr[j]`. A companion array `g` stores the maximum leaf values we can have between indices `i` and `j`. This is done because we know that each non-leaf node's value equals the product of the largest leaf value in each of its left and right subtrees. So, `g[i][j]` helps us quickly determine those products as we build our solution.

By starting with subarrays of length 1 and extending to the full length of `arr`, we can systematically find the minimum sum for any given segment of `arr`. We do so by iterating through every possible division within a subarray, creating potential left and right subtrees, computing their respective non-leaf node value, and maintaining the minimum result for each segment.

The final answer will be the value of `f[0][n - 1]` which represents the minimum sum for the entire array `arr` converted into a binary tree following the described criteria.

Solution Approach

The implementation of the solution utilizes dynamic programming, which is a method to solve complex problems by breaking them down into simpler subproblems and solving each of these subproblems just once, storing their solutions – usually in an array.

Let's dive into the step-by-step explanation of the code:

Step 1: Initialization

Two two-dimensional arrays, `f` and `g`, are initialized with dimensions `n x n`, where `n` is the length of the given array `arr`. Here, `n` represents the range of subarrays we will consider (from `arr[i]` to `arr[j]` inclusively):

- `f[i][j]` will store the minimum sum of the non-leaf nodes for the subarray `arr[i]` to `arr[j]`.
- `g[i][j]` will store the maximum leaf value in the subarray `arr[i]` to `arr[j]`.

```python
n = len(arr)
f = [[0] * n for _ in range(n)]
g = [[0] * n for _ in range(n)]
```

Step 2: Populate the g array

The `g` array is filled with the maximum leaf values. This step is crucial because it allows for quick access to the maximum leaf values later when calculating the non-leaf node values. To fill `g`, we iterate over all possible subarrays and populate `g[i][j]` with the maximum value between `arr[j]` and the current maximum `g[i][j - 1]`:

```python
for i in range(n - 1, -1, -1):
    g[i][i] = arr[i]
    for j in range(i + 1, n):
        g[i][j] = max(g[i][j - 1], arr[j])
```

Step 3: Calculate the minimum sum of non-leaf nodes

The main logic for solving the problem resides here, where `f[i][j]` is computed for different subarrays. The idea is to iterate through all possible splits `k` of the subarray `arr[i]` to `arr[j]` into two parts, `arr[i]` to `arr[k]` and `arr[k + 1]` to `arr[j]`. For each split, we calculate:

- The sum of non-leaf nodes for both parts (`f[i][k]` and `f[k + 1][j]`), which have already been calculated thanks to the dynamic programming approach.
- The product of the maximum leaf values for the left and right subtrees (`g[i][k] * g[k + 1][j]`), which gives the value of the root for this particular binary tree formed by this split.

We keep track of the minimum sum obtained among all possible splits, and that becomes the value of `f[i][j]`:

```python
for i in range(n - 1, -1, -1):
    for j in range(i + 1, n):
        f[i][j] = min(
            f[i][k] + f[k + 1][j] + g[i][k] * g[k + 1][j] for k in range(i, j)
        )
```

Step 4: Return the result

Finally, `f[0][n - 1]` gives us the minimum sum for the entire array `arr`, and thus, is returned as the final result:

```python
return f[0][n - 1]
```

This implementation effectively breaks down the complex task of finding the optimal binary tree into manageable subproblems, whose results are stored and reused – demonstrating the efficacy of dynamic programming.

Example Walkthrough

Let's take a small example to illustrate how the solution approach works. Suppose we have the array `arr` given as `[3, 11, 2, 5]`. We want to construct a binary tree following the conditions mentioned and determine the minimum sum of non-leaf nodes' values.

Visualizing the Problem:

For `arr` = `[3, 11, 2, 5]`, the in-order traversal of the leaf nodes in the binary tree must be `[3, 11, 2, 5]`.

Step 1: Fill in g for maximum leaf values

We compute `g[i][j]` for all subarrays:

- For subarrays of length 1 (i.e., single elements), `g[i][i]` = `arr[i]`.
- For longer subarrays, we would compute the maximum as follows:

```
g[0][1] = max(arr[0], arr[1]) = max(3, 11) = 11
g[1][2] = max(arr[1], arr[2]) = max(11, 2) = 11
g[2][3] = max(arr[2], arr[3]) = max(2, 5) = 5
```

- And so on, for all subarray combinations.

Step 2: Populate g using dynamic programming

Here's the filled `g` matrix for our example array:

```
g = [
    [3, 11, 11, 11],
    [0, 11, 11, 11],
    [0,  0,  2,  5],
    [0,  0,  0,  5]
]
```

Step 3: Calculate f for minimum sums of non-leaf nodes

Now we need to compute `f[i][j]`. Being a dynamic programming solution, we start with the smallest subarrays and move to larger ones.

- For subarrays of length 1 (single elements), `f[i][i]` = 0, as single elements are leaf nodes and don't contribute to the sum of non-leaf nodes.
- Then we consider all subarrays of length 2 and above. For our example, let's compute `f[0][2]` which represents the subarray `[3, 11, 2]`.

We would need to consider splitting this subarray into `[3]` and `[11, 2]` or `[3, 11]` and `[2]`. We'll compute the sum of non-leaf nodes for both scenarios:

For split `[3]` and `[11, 2]`, the sum would be `f[0][0] + f[1][2] + g[0][0] * g[1][2]`. Here, `f[0][0]` and `f[1][2]` are 0 (since `f` is initialized with zeros and gets updated gradually), and `g[0][0] * g[1][2]` is `3 * 11`.

For split `[3, 11]` and `[2]`, the sum would be `f[0][1] + f[2][2] + g[0][1] * g[2][2]`. In this case `f[0][1]` and `f[2][2]` are again 0, and `g[0][1] * g[2][2]` is `11 * 2`.

So, `f[0][2] = min((0 + 0 + 3 * 11), (0 + 0 + 11 * 2)) = min(33, 22) = 22`.

- We compute `f[i][j]` for all possible `i` and `j` in a similar fashion.

Step 4: Construct f matrix and determine the result

We proceed computing `f` values for increasingly larger subarrays of `arr`, keeping track of the minimum sum of non-leaf nodes using dynamic programming.

After filling out the `f` matrix, we'll get:

```
f = [
    [0, 33, 22, min_sum],
    [0,  0, 22, min_sum],
    [0,  0, 0, 10],
    [0,  0, 0,  0]
]
```

The `min_sum` in `f[0][3]` will be the final answer, which is computed by considering all the possibilities, as we just did for `f[0][2]`.

By following these steps and populating `f` and `g`, we find that `f[0][3]` equals the minimum sum of the binary tree non-leaf nodes constructed from array `arr`, following the given conditions.

In this way, the dynamic programming solution helps us reduce the complexity of the problem by avoiding redundant calculations and focusing on subproblems, storing and reusing the solutions as the algorithm progresses.

Python Solution

```python
from typing import List

class Solution:
    def mctFromLeafValues(self, arr: List[int]) -> int:
        # n is the length of the array
        n = len(arr)

        # 'dp' will store minimum total sum needed to merge the trees
        dp = [[0] * n for _ in range(n)]

        # 'max_matrix' will keep track of the maximum leaf value from arr[i] to arr[j]
        max_matrix = [[0] * n for _ in range(n)]

        # Fill out 'max_matrix' and 'dp' diagonally
        for start in range(n - 1, -1, -1):
            # On the diagonal, the max value is the value itself since it's a single leaf
            max_matrix[start][start] = arr[start]
            for end in range(start + 1, n):
                # For each interval from 'start' to 'end', find the max leaf in this range
                max_matrix[start][end] = max(max_matrix[start][end - 1], arr[end])

                # Set an initial high value to compare against in the first iteration
                dp[start][end] = float('inf')

                # Now calculate the minimum sum by breaking the interval into two parts
                # and find the pair that provides minimum sum when merged
                for k in range(start, end):
                    dp[start][end] = min(
                        dp[start][end],
                        dp[start][k] + dp[k + 1][end] + max_matrix[start][k] * max_matrix[k + 1][end]
                    )

        # Return the minimum sum for the interval from 0 to n-1, which represents the whole array
        return dp[0][n - 1]

# Example usage:
# sol = Solution()
# result = sol.mctFromLeafValues([6, 2, 4])
# print(result)  # Output would be the minimum possible sum
```

Java Solution

```java
public class Solution {

    public int mctFromLeafValues(int[] arr) {
        int size = arr.length; // Get the size of the array

        int[][] dp = new int[size][size]; // Create a DP array to store minimum cost
        int[][] maxLeafValue = new int[size][size]; // Create an array to store max leaf value in the subarray [i...j]

        // Fill the maxLeafValue array by iterating from the end to the beginning
        // The value maxLeafValue[i][j] represents the maximum leaf value in the subarray arr[i...j]
        for (int i = size - 1; i >= 0; --i) {
            maxLeafValue[i][i] = arr[i]; // A single leaf's max value is itself
            for (int j = i + 1; j < size; ++j) {
                maxLeafValue[i][j] = Math.max(maxLeafValue[i][j - 1], arr[j]);
                dp[i][j] = Integer.MAX_VALUE; // Initialize the DP array with max values

                // Compute the minimum cost for subarray [i...j] by trying out all possible partitions
                for (int k = i; k < j; ++k) {
                    dp[i][j] = Math.min(dp[i][j],
                                        dp[i][k] + dp[k + 1][j] + maxLeafValue[i][k] * maxLeafValue[k + 1][j]);
                }
            }
        }

        // Return the final answer, which is the minimum cost for the whole array
        return dp[0][size - 1];
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <cstring>
using namespace std;

class Solution {
public:
    int mctFromLeafValues(vector<int>& arr) {
        int size = arr.size(); // 'n' is renamed to 'size' for clarity.
        int dp[size][size]; // 'f' is renamed to 'dp' to indicate it is used for dynamic programming.
        int maxInRange[size][size]; // 'g' is renamed to 'maxInRange' indicating it stores the maximum values in ranges.

        // Initialize the dp matrix with zeroes.
        memset(dp, 0, sizeof(dp));

        // Iterate over the array in reverse to build the maximum values in ranges.
        for (int i = size - 1; i >= 0; --i) {
            maxInRange[i][i] = arr[i]; // Initialize the diagonal with the same array values.
            for (int j = i + 1; j < size; ++j) {
                // Fill maxInRange[i][j] with the maximum value found between i and j.
                maxInRange[i][j] = max(maxInRange[i][j - 1], arr[j]);

                dp[i][j] = INT_MAX; // Initialize the cell with the maximum possible value.
                for (int k = i; k < j; ++k) {
                    // Calculate the minimum cost for the range i to j by partitioning the range at k.
                    dp[i][j] = min(dp[i][j],
                                   dp[i][k] + dp[k + 1][j] + maxInRange[i][k] * maxInRange[k + 1][j]);
                }
            }
        }

        // Return the minimum cost to build the tree from the root (0, size-1).
        return dp[0][size - 1];
    }
};
```

Typescript Solution

```typescript
// Function to calculate the minimum cost to merge the leaf values.
function mctFromLeafValues(arr: number[]): number {
  // Get the length of the input array.
  const length = arr.length;

  // Initialize the dp arrays with default values.
  const dp: number[][] = new Array(length).fill(0).map(() => new Array(length).fill(0));
  const maxLeafValue: number[][] = new Array(length).fill(0).map(() => new Array(length).fill(0));

  // Process the array in reverse.
  for (let start = length - 1; start >= 0; --start) {
    // Base case where the interval only contains a single element.
    maxLeafValue[start][start] = arr[start];

    // Process all possible pairs of intervals (start, end).
    for (let end = start + 1; end < length; ++end) {
      // Determine the maximum leaf value in the current interval.
      maxLeafValue[start][end] = Math.max(maxLeafValue[start][end - 1], arr[end]);
      // Initialize the minimum cost to be a large number.
      dp[start][end] = Number.MAX_SAFE_INTEGER;

      // Try to divide the current interval into two intervals ([start, mid], [mid + 1, end])
      // and find the minimum cost for this division approach.
      for (let mid = start; mid < end; ++mid) {
        dp[start][end] = Math.min(dp[start][end], dp[start][mid] + dp[mid + 1][end] + maxLeafValue[start][mid] * maxLeafValue[mid + 1
      }
    }
  }

  // Return the minimum cost to merge the entire range of leaves.
  return dp[0][length - 1];
}
```

Time and Space Complexity

The given code is a dynamic programming solution designed to minimize the sum of the products of non-leaf nodes' values in a binary tree built from an array, where each leaf is a value from the array. We'll analyze both time complexity and space complexity.

Time Complexity:

We have a nested loop where `i` goes from `n-1` to `0` and for each `i`, `j` goes from `i+1` to `n`, and within each pair of `i` and `j`, there is an inner loop where `k` goes from `i` to `j-1`. The time complexity can be evaluated as the sum of the series in the form of a cubic polynomial.

The number of times the innermost operation (the `min` calculation) is executed can be roughly approximated by the cubic sum `Σ (i=1 to n) Σ (j=i+1 to n) Σ (k=i to j-1) 1`, which simplifies to `O(n^3)` where `n` is the length of the array.

Thus, the time complexity of this solution is `O(n^3)`.

Space Complexity:

The space complexity is determined by the two 2D arrays `f` and `g` which are each of size `n*n`, where `n` is the length of the input array `arr`. There are no other data structures that scale with the input size. Hence, the space complexity is `O(n^2)` due to these two 2D arrays.