120. Triangle

Medium Array Dynamic Programming

Problem Description

Given a `triangle` represented as a list of lists where each element represents a level in the triangle, we need to find the `minimum path sum` from the top to the bottom. On each level, you start on the element you arrived at from the level above and have two choices: move down to the next level's adjacent element that's either directly below or below to the right (index `i` or index `i + 1`). The goal is to determine the minimum sum achievable by following these rules from the top to the bottom of the triangle.

Intuition

The approach to solving this problem is based on `[dynamic programming](/problems/dynamic_programming_intro)`, which involves breaking down a complex problem into simpler subproblems and solving each only once, storing their solutions – usually in an array.

The intuition behind this solution comes from the realization that the path to reach an element in the triangle depends only on the elements from the previous level that are directly above and to the left, or directly above and to the right. Thus, instead of looking at the problem from top to bottom, we reverse the problem and look from bottom to top.

We use a bottom-up approach, starting from the second-to-last row and moving upward. For each element on the current row, we calculate the minimum path sum to reach it by taking the minimum of the two possible sums from the elements directly below it and adding the current element's value.

The `dp` array (short for dynamic programming array) keeps track of these minimum sums. By updating `dp` from the bottom row to the top, we ensure that `dp[0]` will eventually contain the minimum path sum to reach the top element of the triangle.

Solution Approach

The solution utilizes dynamic programming which significantly reduces the complexity by avoiding the repeated computation of subproblems. The key here is to realize that a bottom-up approach allows us to compute the minimum path sum for each level using the information from the level below it.

1. **Data Structures:**
   We use a single-dimensional array `dp` of size `n + 1`, where `n` is the number of rows in the triangle. The array is initialized with zeroes. This array will store the cumulative minimum path sums for each position of the current level, which will be updated as we iterate through the triangle from the bottom.
2. **Algorithm:**
   - We start iterating from the second-to-last row of the `triangle` because the last row's values are the base case for our `dp` array.
   - For each row `i`, we iterate through its elements, where `j` is the position of the element within the row.
   - We update the `dp[j]` with the minimum from the two adjacent "child" positions in the row below (`dp[j]` and `dp[j + 1]` since they are the potential ways to reach `dp[j]`) plus the current element's value `triangle[i][j]`. This update rule reflects the rule of the problem that you can only move to `i` or `i + 1` when proceeding to the next row.

   By continuously updating `dp` in this manner, by the time we reach the first row (`i = 0`), `dp[0]` will have the minimum path sum to reach the top of the triangle.
3. **Iteration Order:**
   - We iterate the rows in reverse (`range(n - 1, -1, -1)`), which means we start from the bottom of the triangle and move upwards.
   - For each row, we iterate through all its elements (`range(i + 1)`), where `i` represents the current row index.

The pattern used in the algorithm ensures that `dp[j]` always contains the minimum path ending at position `j` of the previous row. Therefore, after processing the top row of the triangle, `dp[0]` yields the overall minimum path sum from top to bottom.

Example Walkthrough

Let's consider a small triangle as an example to illustrate the solution approach:

```
   [2],
  [3, 4],
 [6, 5, 7],
[4, 1, 8, 3]
```

We want to find the minimum path sum from the top to the bottom by moving to adjacent numbers on the row below.

Applying the Solution Approach:

1. Initialize a `dp` array with zeros. In this case, since we have 4 rows, the size of `dp` will be `4 + 1 = 5`. Thus, `dp = [0, 0, 0, 0, 0]`.
2. Start from the second-to-last row and iterate up, updating the `dp` array:

- Begin with the bottom row `[4, 1, 8, 3]`, since it will serve as the base case. We don't make any changes to `dp` as the bottom row's values are the starting minimums.
- Move to the next row `[6, 5, 7]`. We examine each element to determine the minimum path sum from that position:
  - For `6` at index `0`, the minimum path is `6 + min(dp[0], dp[1]) => 6 + min(4, 1) = 7`.
  - For `5` at index `1`, the minimum path is `5 + min(dp[1], dp[2]) => 5 + min(1, 8) = 6`.
  - For `7` at index `2`, the minimum path is `7 + min(dp[2], dp[3]) => 7 + min(8, 3) = 10`.
  - Now `dp` is updated to `[7, 6, 10, 0, 0]`.
- Move to the next row `[3, 4]`:
  - For `3` at index `0`, the minimum path is `3 + min(dp[0], dp[1]) => 3 + min(7, 6) = 9`.
  - For `4` at index `1`, the minimum path is `4 + min(dp[1], dp[2]) => 4 + min(6, 10) = 10`.
  - Update `dp` to `[9, 10, 0, 0, 0]`.
- Finally, we move to the top row `[2]`:
  - For `2` at index `0`, the minimum path is `2 + min(dp[0], dp[1]) => 2 + min(9, 10) = 11`.
  - Update `dp` to `[11, 0, 0, 0, 0]`.

3. By the end of the iteration, `dp[0]` contains the minimum path sum which is `11`. This is the answer we were looking for – the minimum path sum from the top to the bottom of the triangle.

In this example, the minimum path from top to bottom is 2 → 3 → 5 → 1, which sums to 11.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def minimumTotal(self, triangle: List[List[int]]) -> int:
        # Get the number of rows in the triangle
        num_rows = len(triangle)

        # Initialize a DP array with an extra space to avoid index out of range
        # This DP array will hold the minimum paths sum from the bottom to the top
        min_path_sum = [0] * (num_rows + 1)

        # Start from the second to last row of the triangle and move upwards
        for row in range(num_rows - 1, -1, -1):
            # For each cell in the row, calculate the minimum path sum to reach that cell
            for col in range(row + 1):
                # The minimum path sum for the current cell is the minimum of the path sums
                # of the two cells directly below it in the triangle plus the cell's value
                min_path_sum[col] = min(min_path_sum[col], min_path_sum[col + 1]) + triangle[row][col]

        # After updating the whole DP array, the minimum path sum starting from the top
        # will be in the first cell of the DP array
        return min_path_sum[0]
```

Java

```java
class Solution {
    public int minimumTotal(List<List<Integer>> triangle) {
        // Get the size of the triangle, which is also the height of the triangle
        int height = triangle.size();
        // Create a DP array of size 'height + 1' for bottom-up calculation
        // This extra space is used to avoid index out of bounds and simplifies calculations
        int[] dp = new int[height + 1];

        // Start from the second last row of the triangle and move upwards
        for (int layer = height - 1; layer >= 0; --layer) {
            // Iterate through all the elements in the current layer
            for (int index = 0; index <= layer; ++index) {
                // Calculate the minimum path sum for position 'index' on the current layer
                // The sum consists of the current element at (layer, index) and the minimum sum of the two adjacent numbers in the l
                dp[index] = Math.min(dp[index], dp[index + 1]) + triangle.get(layer).get(index);
            }
        }
        // The top element of the DP array now contains the minimum path sum for the whole triangle
        return dp[0];
    }
}
```

C++

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

class Solution {
public:
    int minimumTotal(vector<vector<int>>& triangle) {
        // Get the depth of the triangle
        int depth = triangle.size();

        // 'minPathSums' will store the minimum path sum from bottom to top
        vector<int> minPathSums(depth + 1, 0);

        // Start from the bottom of the triangle and move upwards
        for (int row = depth - 1; row >= 0; --row) {
            // Traverse each element of the current row
            for (int col = 0; col <= row; ++col) {
                // For each position, calculate the minimum path sum by taking the lesser of the two
                // adjacent values in the row directly below, and add the current triangle value.
                // This updates 'minPathSums' in-place for the current row.
                minPathSums[col] = min(minPathSums[col], minPathSums[col + 1]) + triangle[row][col];
            }
        }

        // The top of 'minPathSums' now contains the minimum path sum for the entire triangle
        return minPathSums[0];
    }
};
```

TypeScript

```typescript
// Function to find the minimum path sum from top to bottom of a triangle
function minimumTotal(triangle: number[][]): number {
    // Get the size of the triangle which is the length of the outer array
    const triangleSize = triangle.length;

    // Start from the second-to-last row of the triangle and move upwards
    for (let row = triangleSize - 2; row >= 0; row--) {
        // Iterate over each element of the current row
        for (let col = 0; col <= row; col++) {
            // Update the current element by adding the smaller of the two adjacent numbers from the row below
            triangle[row][col] += Math.min(triangle[row + 1][col], triangle[row + 1][col + 1]);
        }
    }

    // After completion, the top element of the triangle contains the minimum path sum
    return triangle[0][0];
}

// The function modifies the input triangle array by storing the sum of the minimum paths at each element,
// so that the final result is accumulated at the triangle's apex.
```


```python
from typing import List

class Solution:
    def minimumTotal(self, triangle: List[List[int]]) -> int:
        # Get the number of rows in the triangle
        num_rows = len(triangle)

        # Initialize a DP array with an extra space to avoid index out of range
        # This DP array will hold the minimum paths sum from the bottom to the top
        min_path_sum = [0] * (num_rows + 1)

        # Start from the second to last row of the triangle and move upwards
        for row in range(num_rows - 1, -1, -1):
            # For each cell in the row, calculate the minimum path sum to reach that cell
            for col in range(row + 1):
                # The minimum path sum for the current cell is the minimum of the path sums
                # of the two cells directly below it in the triangle plus the cell's value
                min_path_sum[col] = min(min_path_sum[col], min_path_sum[col + 1]) + triangle[row][col]

        # After updating the whole DP array, the minimum path sum starting from the top
        # will be in the first cell of the DP array
        return min_path_sum[0]
```


Time and Space Complexity

The given code implements a dynamic programming approach to solve the triangle problem.

Time complexity:

The time complexity of the algorithm is `O(n^2)`, where `n` is the number of rows in the triangle. This is because the algorithm consists of two nested loops. The outer loop runs `n` times, where `n` decreases from the last row to the first row of the triangle. The inner loop runs up to `i + 1` times for the `i-th` iteration of the outer loop (since each row of the triangle has one more element than the row above it). As we sum up `1 + 2 + ... + n`, which is the series of natural numbers, it leads to the formula `n * (n + 1) / 2`. This series is simplified to `O(n^2)`.

Space complexity:

The space complexity of the algorithm is `O(n)`, where `n` is the number of rows in the triangle. This is because the algorithm uses an auxiliary array `dp` of size `n + 1`. This array is used for storing the minimum path sum from the bottom row to each position `(i, j)` in the triangle. As the size of this array does not change with the input size (other than the number of rows `n`), the space complexity is linear in the number of rows.