120. Triangle

Medium 6406 419 Add to List Share

Given a `triangle` array, return *the minimum path sum from top to bottom*.

For each step, you may move to an adjacent number of the row below. More formally, if you are on index `i` on the current row, you may move to either index `i` or index `i + 1` on the next row.

Example 1:

```
Input: triangle = [[2],[3,4],[6,5,7],[4,1,8,3]]
Output: 11
Explanation: The triangle looks like:
   2
  3 4
 6 5 7
4 1 8 3
The minimum path sum from top to bottom is 2 + 3 + 5 + 1 = 11
(underlined above).
```

Example 2:

```
Input: triangle = [[-10]]
Output: -10
```

Constraints:

- `1 <= triangle.length <= 200`
- `triangle[0].length == 1`

```
class Solution {
public:
    int minimumTotal(vector<vector<int>>& triangle) {
        vector<vector<int>> dp;
        int height = triangle.size();
        for (int i = 0; i < height; i ++) {
            dp.push_back(vector<int> ());
            for (int j = 0; j < triangle[i].size(); j ++) {
                dp[i].push_back(INT_MAX);
            }
        }
        dp[0][0] = triangle[0][0];
        for (int i = 1; i < height; i ++) {
            for (int j = 0; j < triangle[i].size(); j ++) {
                if (j == 0) dp[i][j] = dp[i - 1][j] + triangle[i][j];
                else if (j == triangle[i].size() - 1) dp[i][j] = dp[i - 1][j - 1]
+ triangle[i][j];
                else dp[i][j] = min(dp[i - 1][j], dp[i - 1][j - 1]) + triangle[i]
[j];
            }
        }
        int ans = INT_MAX;
        for (int i = 0; i < triangle[height - 1].size(); i ++) {
            ans = min(dp[height - 1][i], ans);
        }
        return ans;
    }
};
```

Proof by Loop invariant

Loop invariant: `dp[i][j]`always the minimum path sum from top to (i, j) cell in the triangle where i is the row number count from the top, and j is the column number count from left to right.

If the given Loop invariant is proof correct, we could just output the minimum result for `dp[i][j]` in the last row of the triangle

Initialization:

When i = 0, j = 0, dp[0][0] = triangle[0][0], it is obvious.

Maintenance:

When the loop goes to calculate dp[i][j], it has 3 cases:

1. case 1: `j == 0`, which means cell `(i, 0)` is at the left edge of the triangle, so there is only one way to get this cell: from cell`(i - 1, 0)`. And because `dp[i - 1][0]` is the minimum path sum to cell`(i - 1, 0)` so `dp[i][0] = dp[i - 1][0] + triangle[i][0]` would be the minimum path sum to cell `(i, 0)`. The variant maintains
2. case2: `j == triangle[i].size() - 1` , which means `(i, trangle[i].size() - 1)` is at the right edge of the triangle. It is same to the case1, `dp[i][triangle[i].size() - 1] = dp[i - 1][triangle[i].size() - 2] + triangle[i][triangle[i].size() - 1]` is the minimum path sum to cell `(i, trangle[i].size() - 1)`.The variant maintains
3. case3: the other cases, cell `(i, j)` is not locate at edges of the triangle. So the path to cell `(i, j)` can either from cell `(i - 1, j)` or cell `(i - 1, j - 1)` , which are the two cells immediate above the cell `(i, j)`. And we decide the path by choosing the smaller minimum path sum from these two cells. i.e., `dp[i][j] = min(dp[i - 1][j], dp[i - 1][j - 1]) + triangle[i][j];` It is the minimum path sum to cell `(i, j)`. Therefore, the invariant maintains.

Termination:

When `i = triangle.size() - 1, j = triangle[i].size() - 1`, the algorithem ends. And the invariant still holds.