Minimum path sum in Triangular Grid

Created by Chandra Prakash

Problem Statement

We are given a Triangular matrix. We need to find the minimum path sum from the first row to the last row. Ending point is not fixed.

At every cell we can move in only two directions: either to the bottom cell or to the bottom-right cell.

Editorial

- Greedy fails.
- Direction: Start from the beginning.

Recursion

Approach

Indices: `i` and `j`

Base Case: The destination, the last row.

Return the value of that particular column of the last row.

Note: Do not forget the fact that there is no requirement of out of boundary conditions here as we are stopping at the last row itself.

```
if(i == n-1) return mat[n-1][j];
```

Explore all paths:

```
down = mat[i][j] + f(i+1, j);
diag = mat[i][j] + f(i+1, j+1);
```

Return the minimum: `return min(down, diag);`

Code

```
int f(int i, int j, vector<vector<int>>& mat, int n) {
    if(i == n - 1) return mat[i][j];
    int down = mat[i][j] + f(i + 1, j, mat, n);
    int diag = mat[i][j] + f(i + 1, j + 1, mat, n);
    return min(down, diag);
}

int minimumPathSum(vector<vector<int>>& triangle) {
    int n = triangle.size();
    return f(0, 0, triangle, n);
```

Complexity Analysis

Time Complexity: `O(2^(n-1))` – From the top, each level branches into 2 choices and the depth is `n`.

Space Complexity: `O(n)` – Maximum recursion depth equals the height of the triangle.

Memoization

Approach

- Store the result.
- Return value if already calculated.

Code

```
int f(int i, int j, vector<vector<int>>& mat, int n, vector<vector<int>>& dp) {
    if(i == n - 1) return mat[i][j];
    if(dp[i][j] != -1) return dp[i][j];
    int down = mat[i][j] + f(i + 1, j, mat, n, dp);
    int diag = mat[i][j] + f(i + 1, j + 1, mat, n, dp);
    return dp[i][j] = min(down, diag);
}

int minimumPathSum(vector<vector<int>>& triangle) {
    int n = triangle.size();
    vector<vector<int>> dp(n, vector<int>(n, -1));
    return f(0, 0, triangle, n, dp);
}
```

Complexity Analysis

Time Complexity: `O(n^2)` – There are `n(n+1)/2` unique states and each is computed once.

Space Complexity: `O(n^2)+ O(n)` – For the `dp` table and stack depth.

Tabulation

Declare a `nxn` dp array.

Instead of starting from the top and branching recursively, we begin from the **last row** and move upward.

At each cell `(i, j)`, we calculate the minimum path sum by taking the current value and adding the minimum of the two cells directly below:

- `dp[i][j] = triangle[i][j] + min(dp[i+1][j], dp[i+1][j+1])`

The final answer will be stored at the top cell `dp[0][0]`.

Base case:

Fill all the last row values in the last row of the dp array.

Note: The tabulation will be from n-1 to 0 as we fill the table from n-1

Code

```
int minimumPathSum(vector<vector<int>>& triangle) {
    int n = triangle.size();
```

```
    vector<vector<int>> dp(n, vector<int>(n, 0));

    for(int j = 0; j < n; j++) {
        dp[n-1][j] = triangle[n-1][j];
    }

    for(int i = n - 2; i >= 0; i--) {
        for(int j = 0; j <= i; j++) {
            int down = dp[i+1][j];
            int diag = dp[i+1][j+1];
            dp[i][j] = triangle[i][j] + min(down, diag);
        }
    }

    return dp[0][0];
}
```

Complexity Analysis

Time Complexity: `O(n^2)` – Each cell in the triangle is visited once during the bottom-up computation.

Space Complexity: `O(n^2)` – 2D DP table.

Space Optimization

Approach

To compute values at row `i`, we only need values from row `i+1`.

Optimize space by using two 1D arrays:

- `curr` for the current row being processed.
- `next` for the row below (initially the last row of the triangle).

Update `curr` from bottom to top, and finally return `curr[0]`.

Code

```
int minimumPathSum(vector<vector<int>>& triangle) {
    int n = triangle.size();
    vector<int> next(triangle[n-1]);

    for(int i = n - 2; i >= 0; --i) {
        vector<int> curr(i + 1);
        for(int j = 0; j <= i; ++j) {
            int down = next[j];
            int diag = next[j + 1];
            curr[j] = triangle[i][j] + min(down, diag);
        }
        next = curr;
    }
```

```
    return next[0];
}
```

Complexity Analysis

Time Complexity: `O(n^2)` – Every cell in the triangle is still processed once.

Space Complexity: `O(n)` – Only two 1D arrays are used, each of size `n`.

THE END