Triangle

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.

For example, given the following triangle

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

The minimum path sum from top to bottom is `11` (i.e., 2 + 3 + 5 + 1 = 11).

Note:
Bonus point if you are able to do this using only $O(n)$ extra space, where n is the total number of rows in the triangle.

Solution 1

This problem is quite well-formed in my opinion. The triangle has a tree-like structure, which would lead people to think about traversal algorithms such as DFS. However, if you look closely, you would notice that the adjacent nodes always share a 'branch'. In other word, there are **overlapping subproblems**. Also, suppose x and y are 'children' of k. Once minimum paths from x and y to the bottom are known, the minimum path starting from k can be decided in O(1), that is **optimal substructure**. Therefore, dynamic programming would be the best solution to this problem in terms of time complexity.

What I like about this problem even more is that the difference between 'top-down' and 'bottom-up' DP can be 'literally' pictured in the input triangle. For 'top-down' DP, starting from the node on the very top, we recursively find the minimum path sum of each node. When a path sum is calculated, we store it in an array (memoization); the next time we need to calculate the path sum of the same node, just retrieve it from the array. However, you will need a cache that is at least the same size as the input triangle itself to store the pathsum, which takes O(N^2) space. With some clever thinking, it might be possible to release some of the memory that will never be used after a particular point, but the order of the nodes being processed is not straightforwardly seen in a recursive solution, so deciding which part of the cache to discard can be a hard job.

'Bottom-up' DP, on the other hand, is very straightforward: we start from the nodes on the bottom row; the min pathsums for these nodes are the values of the nodes themselves. From there, the min pathsum at the ith node on the kth row would be the lesser of the pathsums of its two children plus the value of itself, i.e.:

```
minpath[k][i] = min( minpath[k+1][i], minpath[k+1][i+1]) + triangle[k][i];
```

Or even better, since the row minpath[k+1] would be useless after minpath[k] is computed, we can simply set minpath as a 1D array, and iteratively update itself:

```
For the kth level:
minpath[i] = min( minpath[i], minpath[i+1]) + triangle[k][i];
```

Thus, we have the following solution

```
int minimumTotal(vector<vector<int> > &triangle) {
    int n = triangle.size();
    vector<int> minlen(triangle.back());
    for (int layer = n-2; layer >= 0; layer--) // For each layer
    {
        for (int i = 0; i <= layer; i++) // Check its every 'node'
        {
            // Find the lesser of its two children, and sum the current value in the triangle with it.
            minlen[i] = min(minlen[i], minlen[i+1]) + triangle[layer][i];
        }
    }
    return minlen[0];
}
```

written by stellari original link here

Solution 2

```
public class Solution {
    public int minimumTotal(List<List<Integer>> triangle) {
        for(int i = triangle.size() - 2; i >= 0; i--)
            for(int j = 0; j <= i; j++)
                triangle.get(i).set(j, triangle.get(i).get(j) + Math.min(triangle.
get(i + 1).get(j), triangle.get(i + 1).get(j + 1)));
        return triangle.get(0).get(0);
    }
}
```

The idea is simple.

1) Go from bottom to top.

2) We start form the row above the bottom row [size()-2].

3) Each number add the smaller number of two numbers that below it.

4) And finally we get to the top we the smallest sum.

written by SteveLee1989 original link here

Solution 3

```
class Solution {
public:
    int minimumTotal(vector<vector<int> > &triangle)
    {
        vector<int> mini = triangle[triangle.size()-1];
        for ( int i = triangle.size() - 2; i>= 0 ; --i )
            for ( int j = 0; j < triangle[i].size() ; ++ j )
                mini[j] = triangle[i][j] + min(mini[j],mini[j+1]);
        return mini[0];
    }
};
```

written by rishav2 original link here