Daily Coding Problem #201

Problem

This problem was asked by Google.

You are given an array of arrays of integers, where each array corresponds to a row in a triangle of numbers. For example, `[[1], [2, 3], [1, 5, 1]]` represents the triangle:

```
  1
 2 3
1 5 1
```

We define a path in the triangle to start at the top and go down one row at a time to an adjacent value, eventually ending with an entry on the bottom row. For example, 1 -> 3 -> 5. The weight of the path is the sum of the entries.

Write a program that returns the weight of the maximum weight path.

Solution

A brute force solution would be to generate all possible paths and take the maximum of their sums. Since there are 2^N ways of getting from the top to the bottom of our triangle, this would be prohibitively expensive.

Let's look at the example above. If we start at the top, we must either go down to 2 or 3. Our maximum weight, then, is either 1 + (the maximum weight of a path starting with 2) or 1 + (the maximum weight of a path starting with 3). So to find a solution we can figure out which of the left and right subpaths has a greater maximum weight, and add 1 to that value. This points the way to a recursive solution:

- If we are at the bottom level of our triangle, the maximum weight of a path is simply the value at a given index.
- Otherwise, it will be the value at that index, plus the larger of the maximum weights of its left and right subpaths.

```
def max_path_weight(arrays, level=0, index=0):
    if level == len(arrays) - 1:
        return arrays[level][index]
    else:
        return arrays[level][index] + max(
            max_path_weight(arrays, level + 1, index), max_path_weight(arrays,
level + 1, index + 1)
        )
```

Although this is neater than a brute force solution, it isn't more efficient, because we are still traversing every path. Instead, we can use dynamic programming to speed things up by storing precomputed values for subpaths.

We already know that the maximum weight for any index on the bottom row is the value at that index. So starting with the second-to-last row, we move up one row at a time, incrementing the value at each of its indices with the the maximum weight of its left or right subpath. Let's see this in action:

```
  1
 2 3
1 5 1
```

We start with the second row, looking at 2. `max(1, 5) = 5`, so we increment 2 by 5 to get 7. Similarly when we look at 3, `max(5, 1) = 5`, so we increment 3 by 5 to get 8. This results in the following triangle.

```
   1
  7 8
 1 5 1
```

Finally, we replace the 1 in the first row with `1 + max(7, 8)`, giving us 9, our solution. Since we examine each element once, this algorithm is O(N), where N is the total number of elements.

```
def max_path_weight(arrays):
    for level in range(len(arrays) - 2, -1, -1):
        for index in range(level + 1):
            arrays[level][index] += max(arrays[level + 1][index], arrays[level +
1][index + 1])

    return arrays[0][0]
```