Problem : Triangle

Description :

Given a traingle in the form of 2D array. We need to find a path from top to bottom such that the sum of elements on the path is minimized with additional constraint that we are only allowed to go to adjancent elements when we move down.

Eg.[

```
    [2],

   [3,4],

  [6,5,7],

 [4,1,8,3]

]
```

Answer = 2 + 3 + 5 + 1 = 11

Intution :

In the question, it is given that we can move to adjancent elements present downwards. Each elements (except elements of the last row) has two paths that leads to the bottom array. Our goal is to find a path from row 0 to row (n-1) such that cost is minimized. We can model the problem as a binary tree, where element triangle[0][0] is the root. Now each element has two children.
For example triangle[0][0] has triangle[1][0] and triangle[1][1] as its children.

Greedy Doesn’t Work.

Greedy Approach : Start from root. Take the minimum of the two children and go forward.
This approach might come to mind at first, but this doesn’t work. There is because greedy approach always try to choose the local minimum hence, will try to minimize the local sum. Our objective is to minimize the global sum, so to do that, we might sometimes, have to choose a bigger numbers.
Eg.[

```
    [2],

   [3,4],

  [6,5,100],

 [100 , 100 , 100 , 3]

]
```

Greedy Approach will give ans = 110. Optimal Answer : 2 + 4 + 100 + 3 = 109.

Brute Force Solution :
Lets try to build a brute force solution and improve it. Lets say we assume the initial sum to be 0 and try to traverse all possible paths from root to the leaves of the tree using recursion and take minimum all over them. This approach is correct, however this is too slow.

Recursive Brute Force :

```
int ans = 1000000000;
void solve(vector<vector<int>>& arr, int i, int j, int n, int sum = 0) {
      if(i == n-1) {
         ans = min(ans, sum + min(arr[i][j], arr[i][j+1]));
         return ;
      }
      else {
         solve(arr, i+1, j, n, sum + arr[i][j]);
         solve(arr, i+1, j+1, n, sum + arr[i][j+1]);
      }
   }
```

Here arr is a copy of array triangle.

Dynamic Programming Approach :

Now we look at how dynamic programming can be used to reduce the time complexity.

Overlapping Subproblem :
Lets look at the traversal of the binary tree of the brute force solution. We start from the root and we have two choices. We explore both the choices recursively and then we are trying to find the minimum sum.
To solve the problem for the root node we are trying to solve the problem for its children, so if we precompute the answer for the children, we just need to take the minimum sum from root and we will have the answer. The brute force solution has many overlaps to the same child node as a child node is shared between two parent.
Eg.[

```
    [2],

   [3,4],

  [6,5,7],

 [4,1,8,3]

]
```

In the above example, possible paths are :

2 –> 3 -> 5 -> 1 and 2 -> 4 -> 5 -> 1

We can clearly see that the brute force solution has overlapping subproblems. The problem to solve for the node with value 5 is repeated although the path they arrive is different. Hence we can use tabulation and memoization to solve our required problem.

Algorithm :

Bottom Up Approach :
Lets maintain a new array arr.

Initial Step :
We copy all the elements of triangle[n-1] to arr[n-1]. This serves as the base case.

Transition Step :
Lets say we know the solution of the children node. Then to find the solution of the root node, we just need to take the minimum of the two children and add it to the root node. This is true because the path from the children node to the leaf node is optimal. To create a path from root node to the child node, we have the direct edge, hence we can simply optimize the path from root to child node, and that itself will be the optimal path from root to the leaves.

For the root node arr[i][j] we know the solution of arr[i+1][j] and arr[i+1][j+1] hence,

arr[i][j] = triangle[i][j] + min(arr[i+1][j], arr[i+1][j+1])

This is the required transition step.

Implementation Details :

The implementation of the above approach is simple. We need to iterate in reverse from leaves to the root and fill the 2 D table arr and then return the root node arr[0][0].
Implementation needs to be careful as there is a base case that when n = 1, that is, we only have one node in the array.

Final Implementation in C++ :

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

Time Complexity :
We are iterating through the array elements only once, hence the time taken by each iteration = 1 + 2 + 3 + ... + n = $O(n^2)$

Space Complexity :
We are building a dp table which is the same size as that of the original array. Hence space complexity = $O(n^2)$

Memory Optimization :

In this question, we can optimize memory so that we can get a more efficient solution.
The key idea in doing this is to observe that once we compute the states i for all j, we have no use for the states i + 1 .... n-1.
We can delete those states as, we no longer need them.

Implementation :

To implement this idea, we basically create a 1D array with size n. Then we fill it with the base cases.
Now we know that to calculate the answer for j^{th} element of the i^{th} array, we need states : (i+1, j) and (i+1, j+1). After calculating the answer for this state, observe that we do not need the state (i+1, j). Hence we can use the location j to store our new state.

Implementation With Code

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

Space Complexity :

Now we are using very less memory. The total size of dp table is now equal to N.
Hence Space Complexity = O(n)
n is the number of rows in the triangle array.