Quan Yuan
Nov1, 2022
Project 7 report

This project generate the Hosoya's triangle.

There are 2 functions in the program, one function takes in an int as levels, and compute the triangle with the levels, then return it as a list of lists. Another function takes in the triangle as a list, and the levels, then print the triangle as a left-leaned triangle.

For the function computeTriangle(levels), the followings are the logic, flowchart and pseudocode.

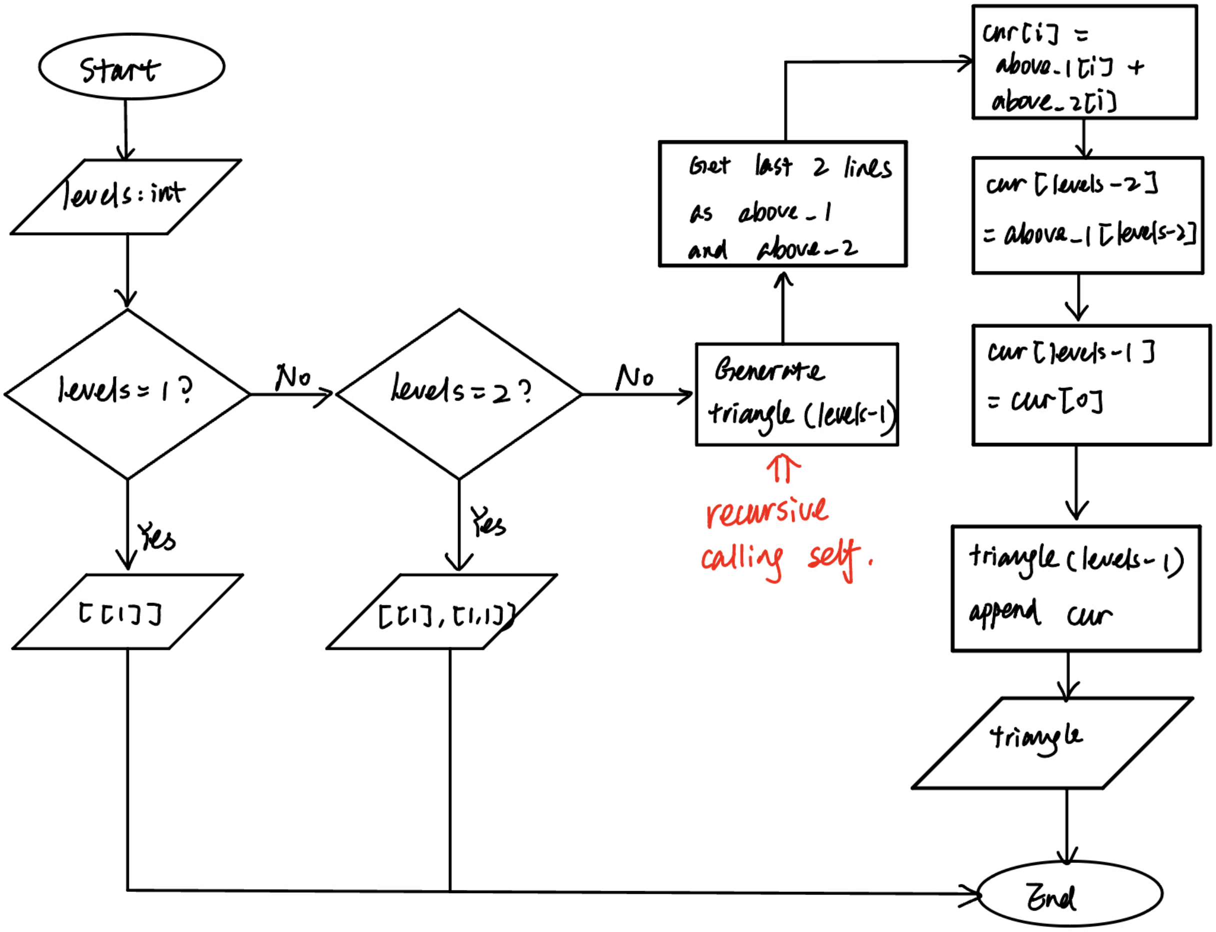
We can easily notice that the nth line has exactly n elements. And Each element is the sum of the 2 elements in 2 previous row with the same index. For example, in 3rd line, the first element is 2, which is 1 + 1, the first element in line 2 and line 1 respectively.

Because the line 2 line above has n-2 elements, which is 2 less than current line. So the (n-1)th element in current line will be the sum of the (n-1)th element in previous line and 0. It is exactly the same as the last element in previous line.

The nth element in current line, however, is not equal to 0, but equal to the sum of the last elements in previous 2 lines. We can also find out that it is the same as the first element in current line.

So to generate the current line, we need to have the 2 above lines. It will be easier to have the triangle with n-1 levels, as it is the current function itself. And then we store the 2 last elements as 2 last lines. Then we based on this 2 lines to calculate the current line, and append the current line to the previous triangle.

Flowchart:


Start
levels: int
levels = 1 ?
No
levels = 2 ?
No
Generate triangle (levels-1)
recursive calling self.
Get last 2 lines as above_1 and above_2
cur[i] = above_1[i] + above_2[i]
cur[levels-2] = above_1[levels-2]
cur[levels-1] = cur[0]
triangle(levels-1) append cur
triangle
Yes
[[1]]
Yes
[[1],[1,1]]
End

Pseudocode:

There is a corner case, which is the input is negative or 0, we return an empty list.

```
if levels<0 or levels==0:
    return []
```

There are 2 base cases, when there are not enough previous lines to generate the current line. Which are, when levels equals 1 or 2, the amount of total previous lines are 0 and 1 respectively. So we have to define these 2 cases explicitly:

```
if  levels == 1, return [ [1] ];
if  levels == 2, return [ [1], [1, 1] ].
```

Starting from 3, there are enough previous lines to generate the current line. So we compute the triangle with n-1 levels first, and then get the last 2 lines.

```
prev_triangle = computeTriangle(n-1)
above_1_line = prev_triangle[-1]
above_2_line = prev_triangle[-2]
```

Then we can calculate the current line based on the 2 above lines.

```
cur_line = [] (n elements)
the first n-2 elements: cur_line[i] = above_1_line[i] + above_2_line[i]
the (n-1)th element: cur_line[n-2] = above_1_line[n-2]
the nth element: cur_line[n-1] = above_1_line[-1] + above_2_line[-1], or
                cur_line[n-1] = cur_line[0]
```

Finally, we append current line to previous triangle and return the triangle.

```
prev_triangle.append(cur_line)
return prev_triangle
```

Reflection:

By using recursive, it is easy to realize some complicated programs. What we need to find is just the relations between the current outcome and the previous outcome(s).

Acknowledgement:

None