COMP1021
Introduction to Computer Science

Recursion

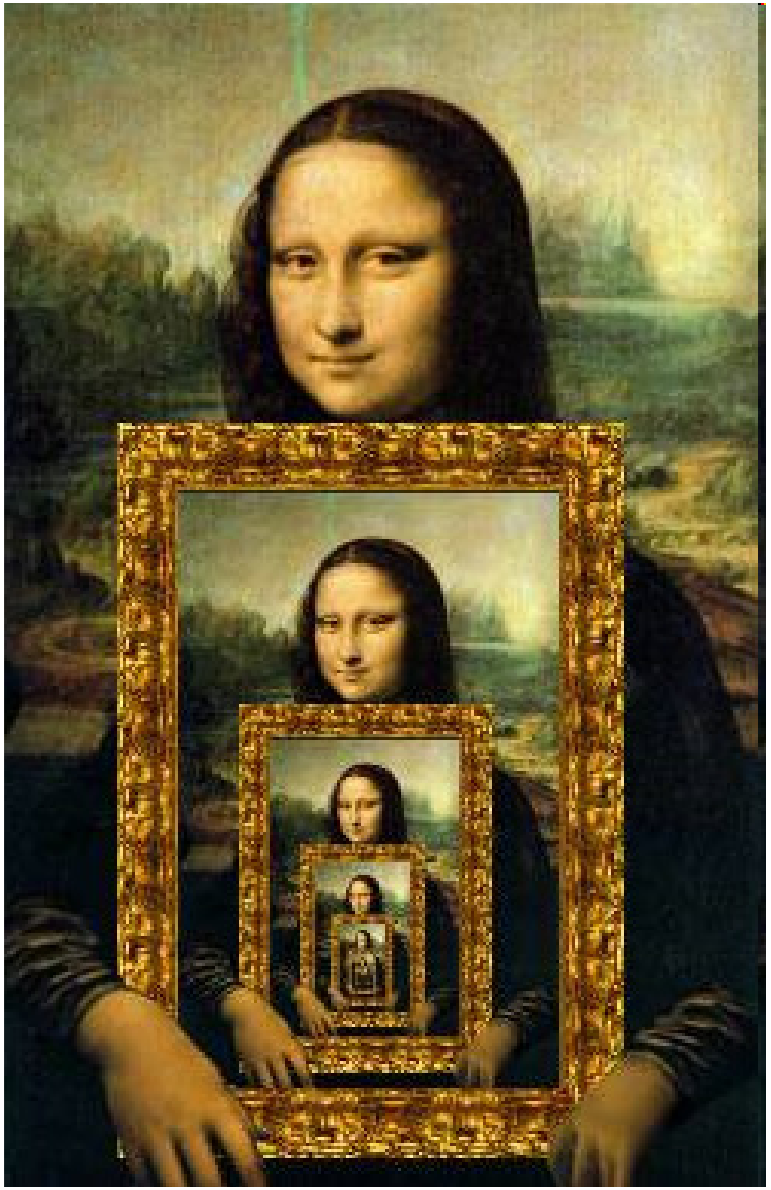
David Rossiter

Outcomes

- After completing this presentation, you are expected to be able to:
  1. Explain what recursion is
  2. Construct the recursion depth diagram of a recursive function

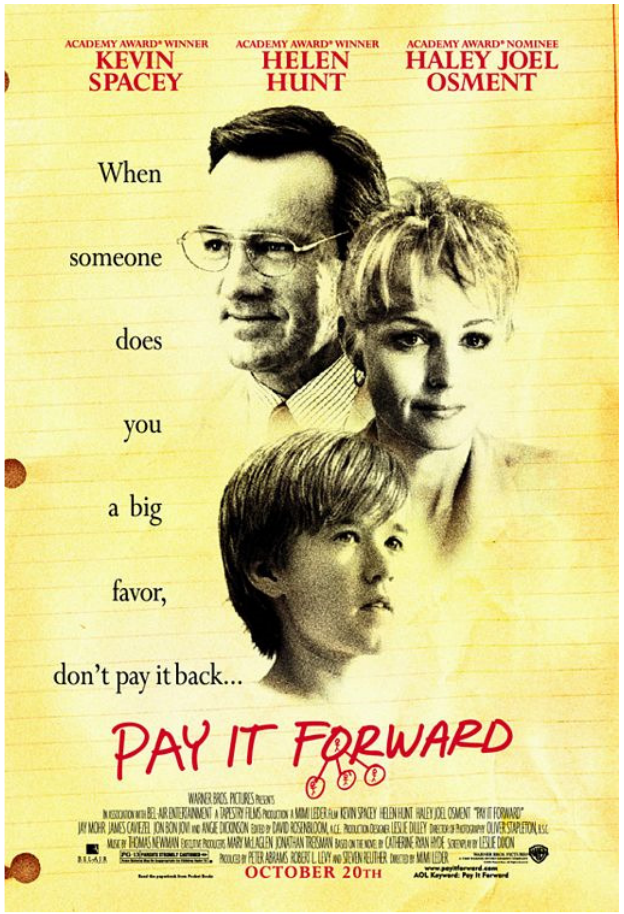
What is Recursion?

- A recursive function is one which calls itself
- Recursive functions are sometimes very useful for some computing tasks
- For example, you can use one cleverly written small recursive function instead of lots of lines of code



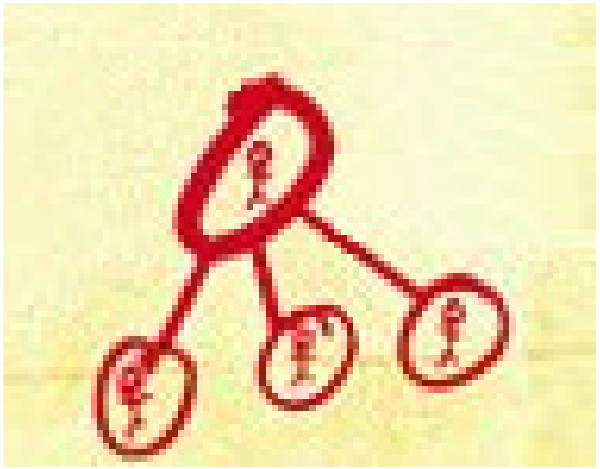
‘Pay It Forward’

- A movie about a boy who has been asked to come up with a plan that will change the world
- He comes up with the plan that when someone receives a good deed, he/she helps 3 different other people



‘Pay It Forward’ Pseudo-Code

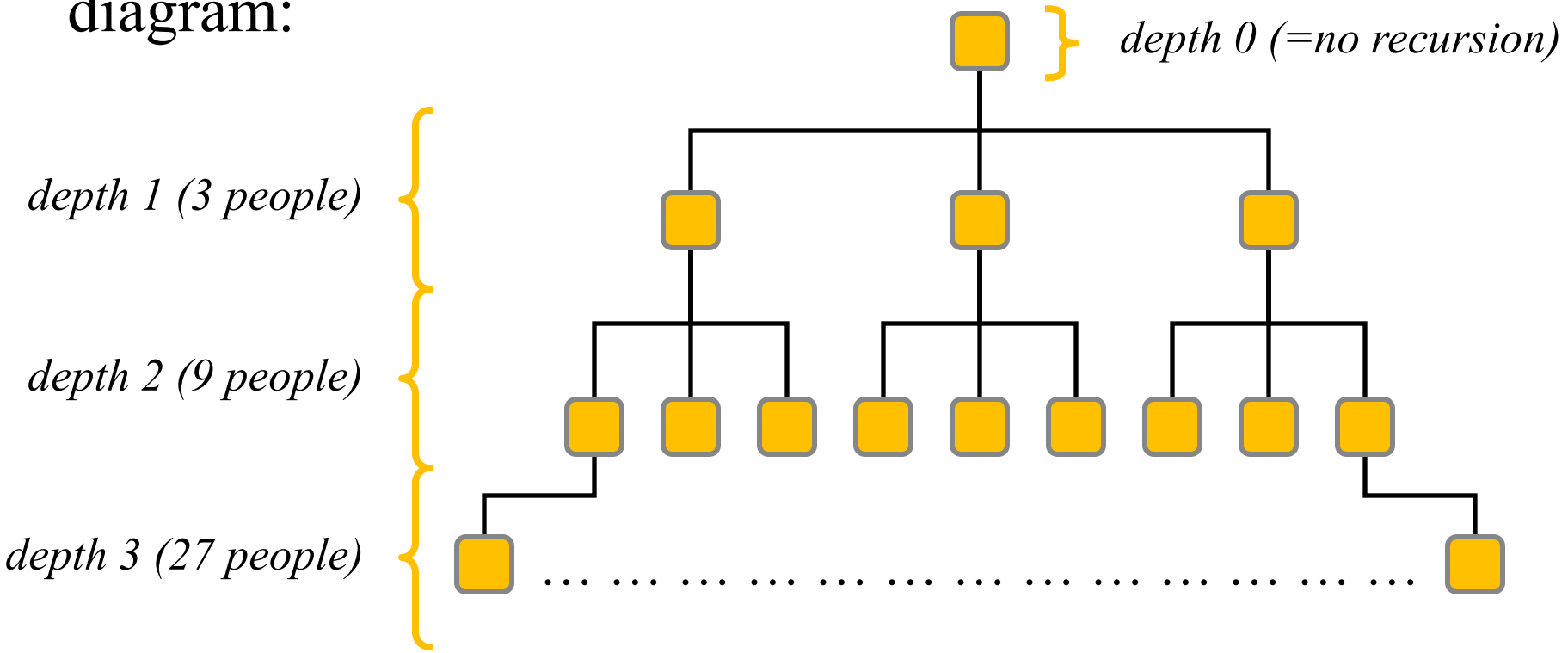
```
def help(benefactor, person):
    person receives help from benefactor
    help(person, random_person1)
    help(person, random_person2)
    help(person, random_person3)
```



- The whole process starts with one person helping another, for example: help(me, you)
- The above example uses pseudo-code, but the rest of this presentation uses real Python code

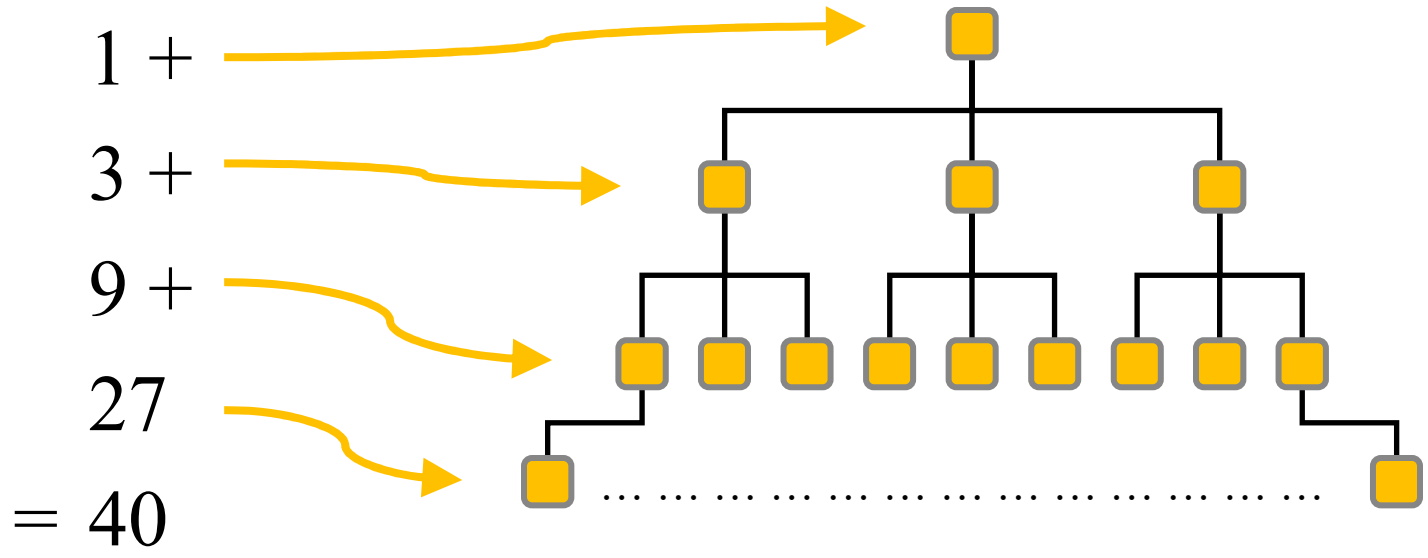
Recursive Depths 1/2

- How many good deeds are done in total after 3 depths?
- You can see what we mean by *depth* in the following diagram:



Recursive Depths 2/2

- The answer is that when the maximum depth is 3, the total number of good deeds is:



A Recursive Function in Python

- Here is an example recursive function:

```
def printsomenumbers(num):
    print(num)
    if num < 4:
        printsomenumbers(num + 1)

printsomenumbers(0)
```

The recursive function is started by running `printsomenumbers(0)`

- This is the execution of the code `printsomenumbers(0)`

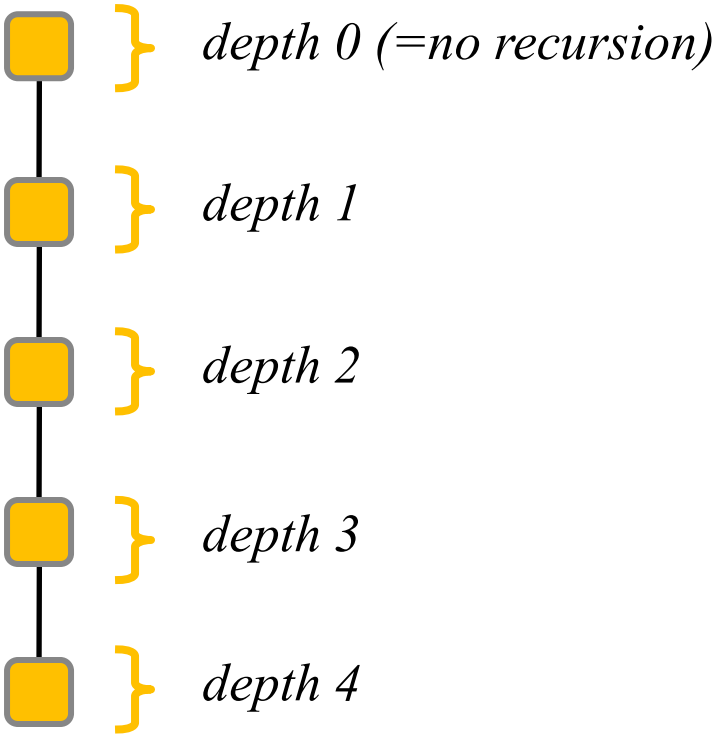
```
printsomenumbers(0)
    def printsomenumbers( 0 ):
        print(0)
        printsomenumbers(0 + 1)
            def printsomenumbers( 1 ):
                print(1)
                printsomenumbers(1 + 1)
                    def printsomenumbers( 2 ):
                        print(2)
                        printsomenumbers(2 + 1)
                            def printsomenumbers( 3 ):
                                print(3)
                                printsomenumbers(3 + 1)
                                    def printsomenumbers( 4 ):
                                        print(4)
```

*There are no more **function calls** when this value becomes 4, because of the **if** statement*

- The result is **0 1 2 3 4**

Recursive Depth Diagram

- So for this example, the pattern of depth looks like this:



Recursive Functions and Iterative Code

- The recursive example discussed in the last few slides generates a result of:

```
0
1
2
3
4
```

- On the next slide we will show two *iterative* code examples which produce the same result
- 'Iterative' means 'looping without recursion'

Iterative Code Examples

- Iterative code example 1:

```
for num in range(0, 5):
    print(num)
```

- Iterative code example 2:

```
num = 0
while num < 5:
    print(num)
    num = num + 1
```

- You can write recursive code and iterative code which do the same thing
- However, sometimes it is easier to write things using recursion, as you will see later

Changing the Order

- Let's change the example recursive function by swapping two parts of the code:

```
def printsomenumbers(num):
    if num < 4:
        printsomenumbers(num + 1)
    print(num)

printsomenumbers(0)
```

These two parts of code have been swapped

- This is the execution of the code `printsomenumbers(0)`

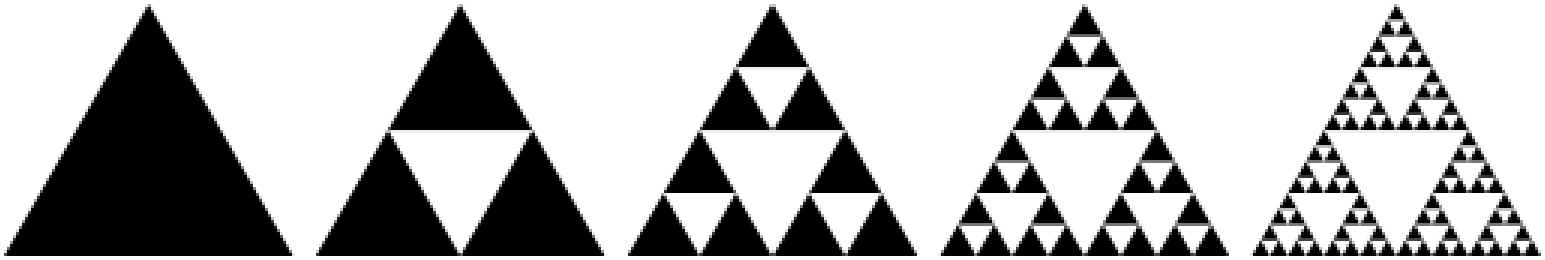
```
printsomenumbers(0)
    def printsomenumbers( 0 ):
        printsomenumbers(0 + 1)
            def printsomenumbers( 1 ):
                printsomenumbers(1 + 1)
                    def printsomenumbers( 2 ):
                        printsomenumbers(2 + 1)
                            def printsomenumbers( 3 ):
                                printsomenumbers(3 + 1)
                                    def printsomenumbers( 4 ):
                                    print(4)
                                print(3)
                        print(2)
                print(1)
        print(0)
```

*There are no more **function calls** when this value becomes 4, because of the **if** statement*

- The result is **4 3 2 1 0** , which is the opposite order compared to the previous program's result

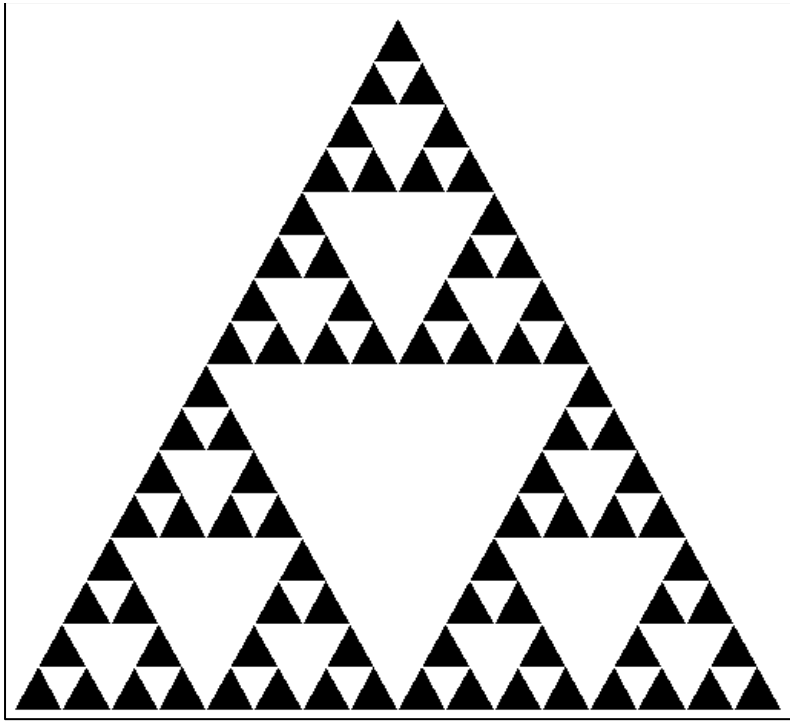
Making Pictures with Recursion

- Recursive functions are used for lots of purposes
- One of them is to make computer graphics containing a lot of repetitions, like this:



- In the following slides, we will discuss using recursion to draw the above triangles, and to build a tree

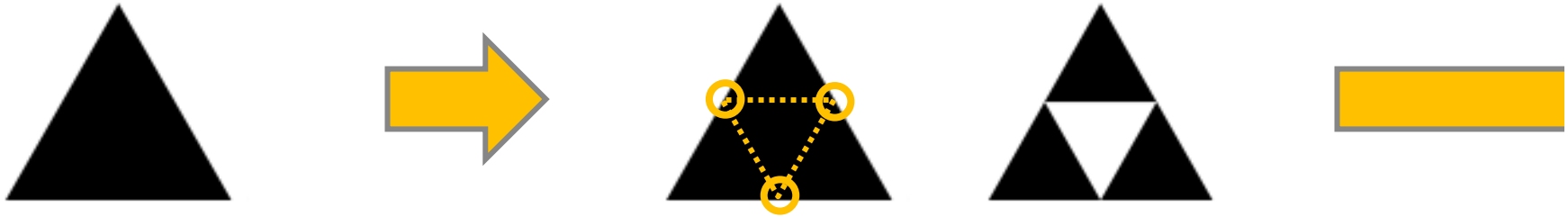
The Sierpinski Triangle



- The computer graphics example shown on the previous slide is called the Sierpinski triangle
- Basically, we start with a black triangle
- We draw a white triangle in the middle area
- Then the process repeats itself for the three ‘corner’ black triangles

Drawing the Sierpinski Triangle

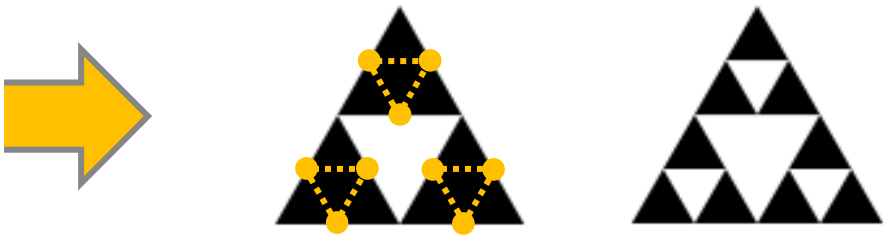
Depth = 0



1. Start with a black triangle

2. Find the mid-point for each of the 3 sides, fill the middle triangle with white

Depth = 1

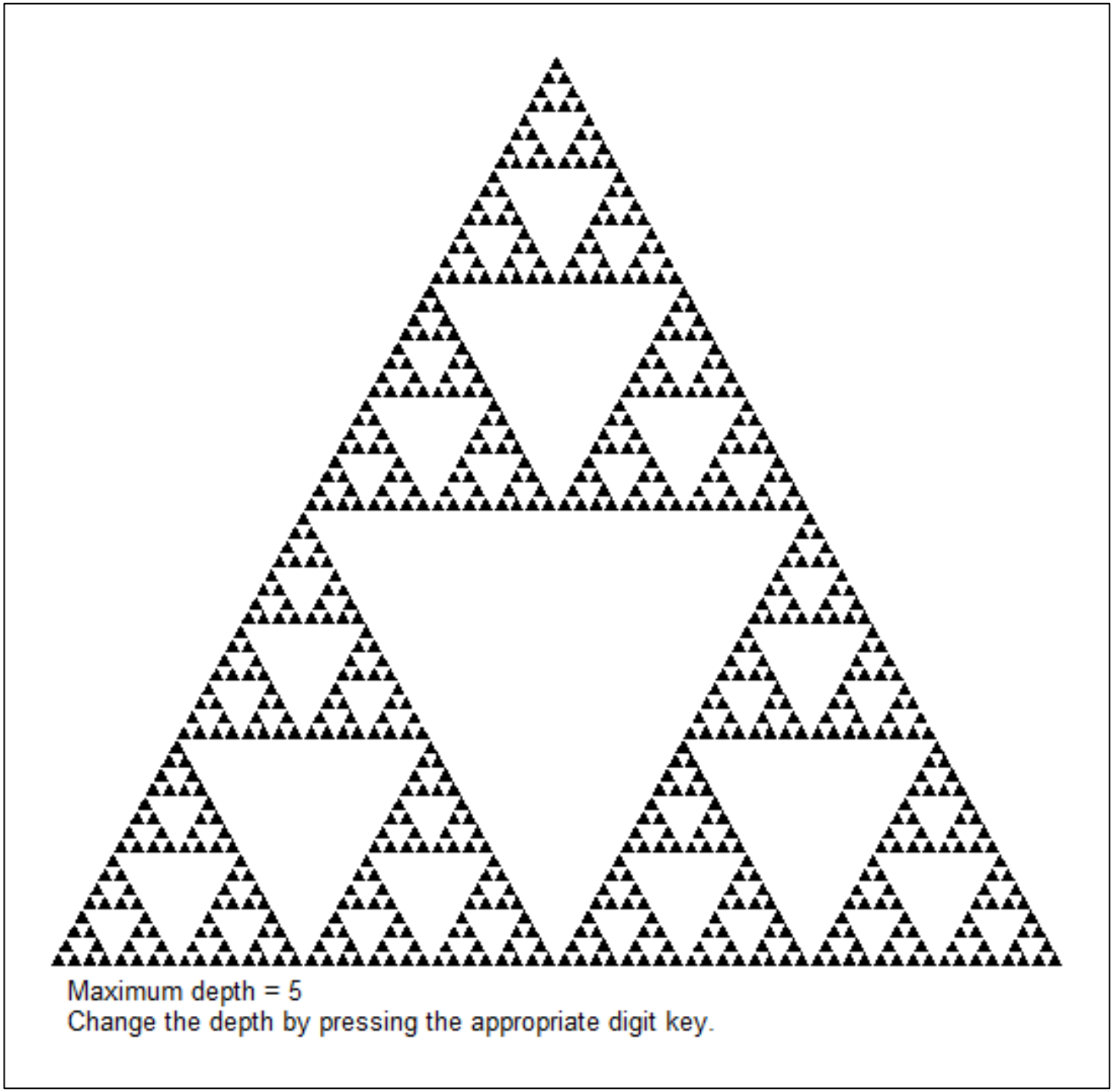


3. Repeat step 2 with EACH of the three smaller black triangles

Depth = 2

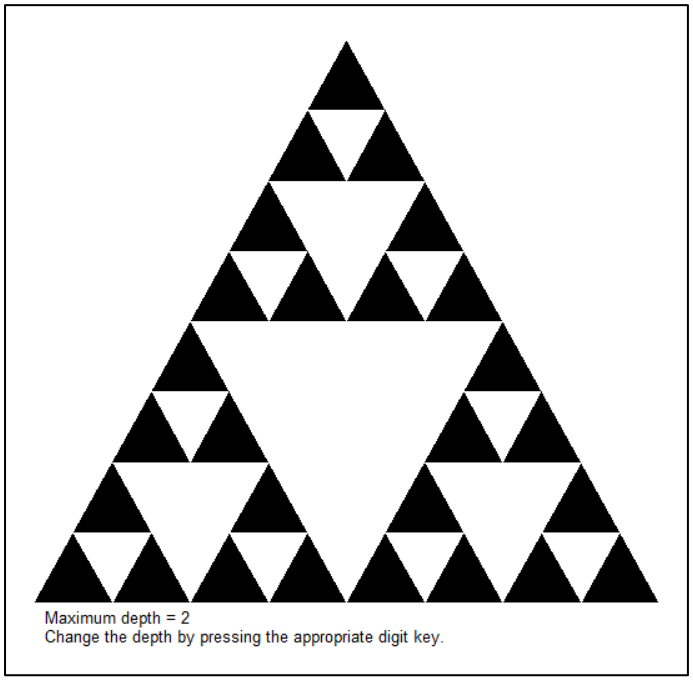
4. Keep on repeating step 2 with the smaller triangles

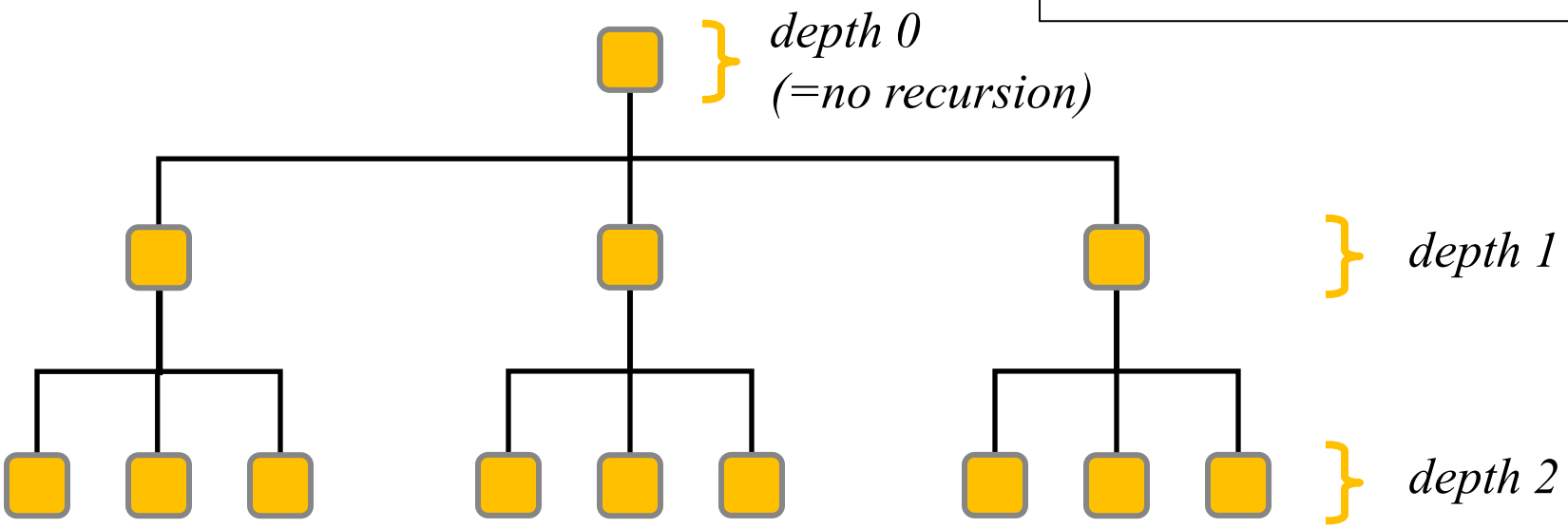
The Sierpinski triangle, with a maximum depth of 5



Recursive Depths

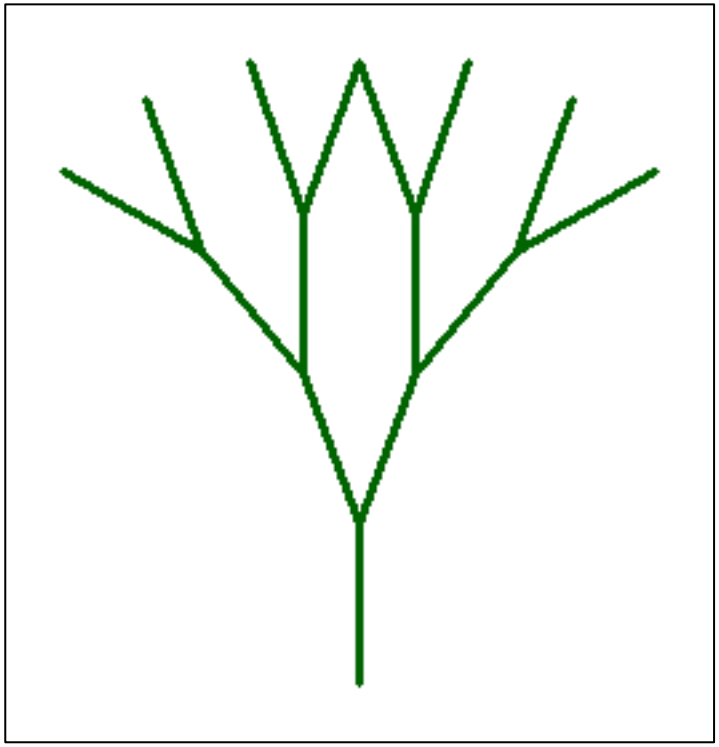
- For this example, when the maximum depth is 2, this is what the depth diagram looks like:





The Recursive Tree

- We can also use recursion to draw a simple tree
- First, a main trunk is drawn
- Then the process repeats itself twice, to draw two branches



Drawing the Recursive Tree

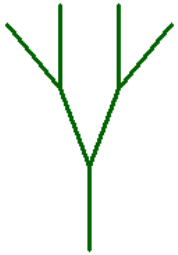
Depth = 0

1. Start with a main trunk

Depth = 1

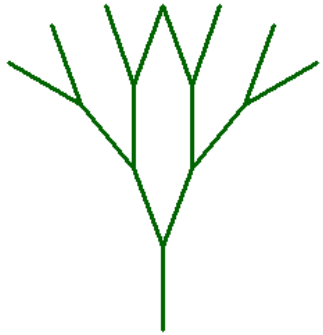
2. Branch to the left and right of trunk

Depth = 2



3. Repeat step 2 for each branch

Depth = 3



4. Keep on repeating step 2 the branches

The Recursive Function

- Here is the recursive function for building the tree:

```
def buildtree(current_depth):
    if current_depth <= max_depth:
        turtle.forward(branch_length)
        turtle.right(angle_between_branches / 2)
        buildtree(current_depth + 1)
        turtle.left(angle_between_branches)
        buildtree(current_depth + 1)
        turtle.right(angle_between_branches / 2)
        turtle.backward(branch_length)
```

Control how 'deep' the tree is

Build the right child branch of the current branch

Build the left child branch of the current branch