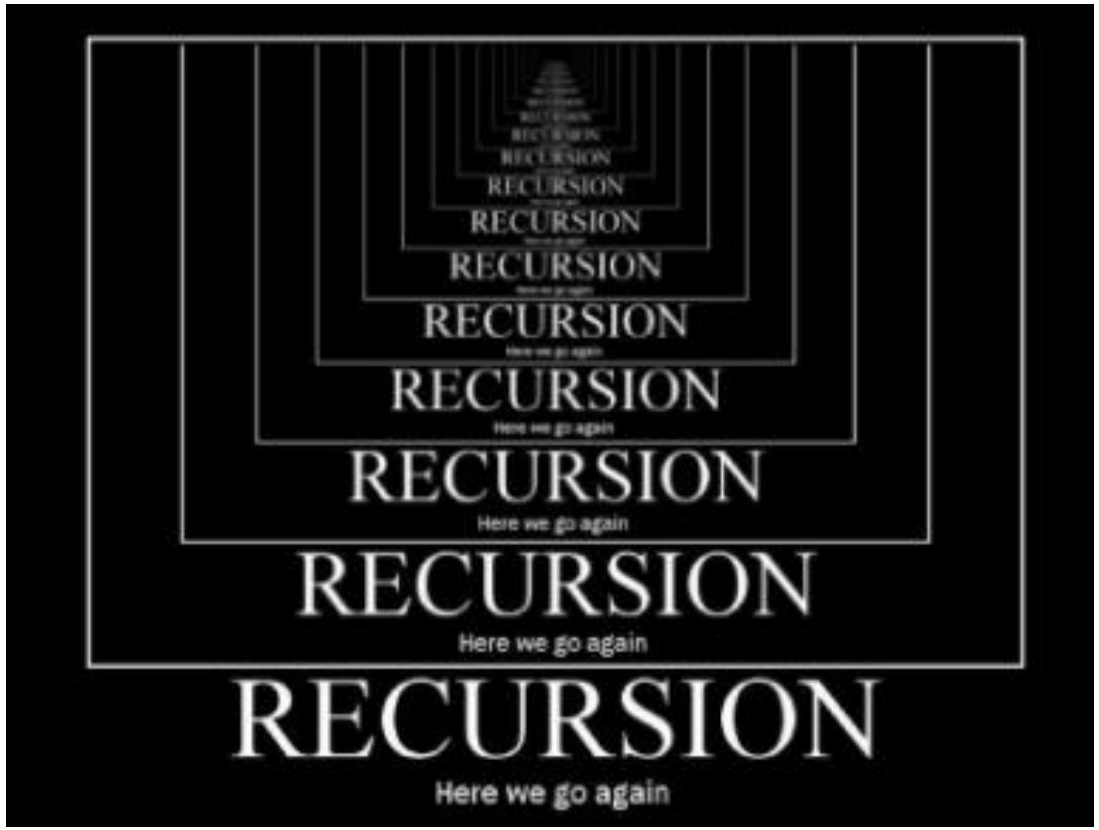


Chapter 5
Recursion

By: Dr. Aryaf A. Al-adwan
Faculty of Engineering Technology
Computer and Networks Engineering Dept.
Data Structures Course

Outline

- Repetition.
- What is recursion?
- Needs for recursion
- Examples of recursion

Repetition

- Iteration : using while, for , do while loops
- Recursion : function calls itself with different parameters

Iterative solution for factorial

```
public int fact ( int n)
{
n=5;
int f = 1;
for(int i=1;i<=n;i++)
    f=f*i;
return f;
}
```

Basic operation will performed N times

Recursion

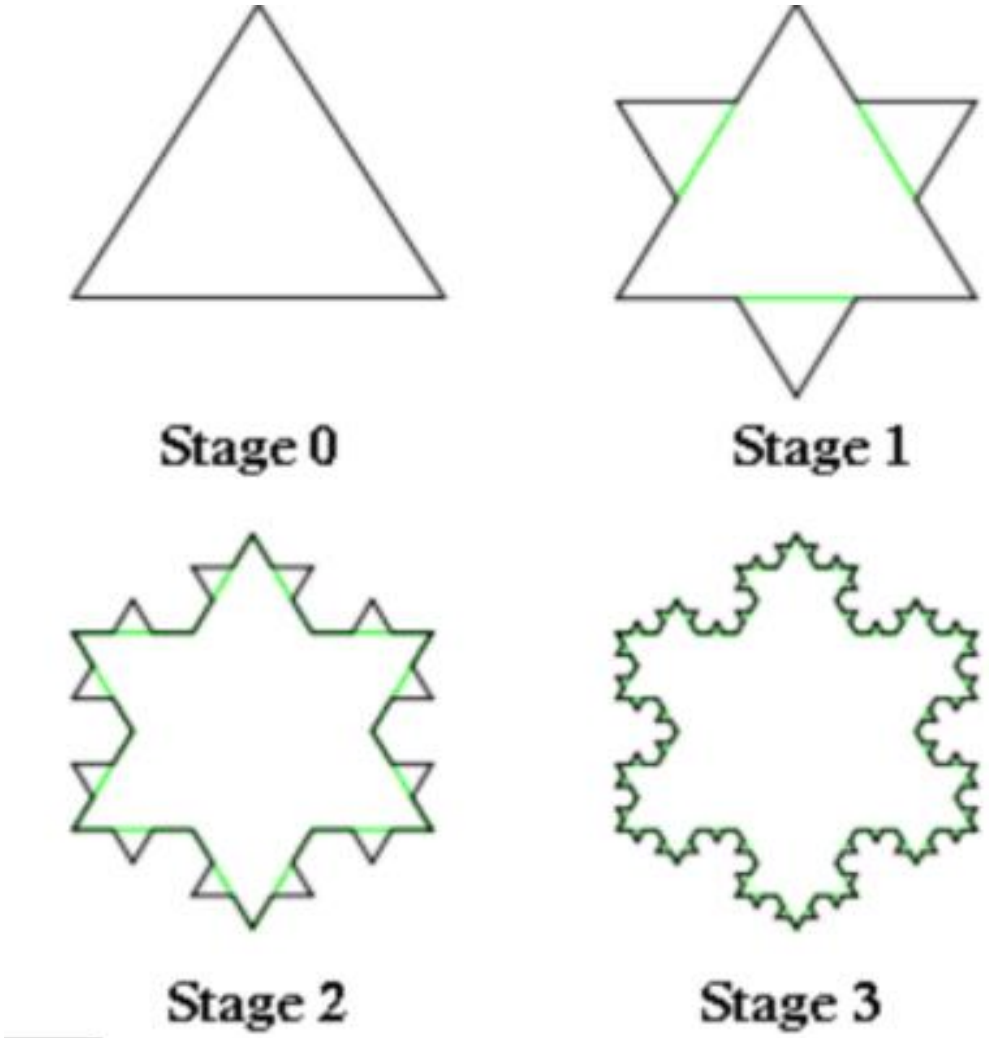
- In computer science, recursion is a method of solving a problem where the solution depends on solutions to smaller instances of the same problem.
- Such problems can generally be solved by iteration, but this needs to identify and index the smaller instances at programming time. Recursion solves such recursive problems by using functions that call themselves from within their own code.

Fractals

A fractal is a never-ending pattern. Fractals are infinitely complex patterns that are self-similar across different scales. They are created by repeating a simple process over and over in an ongoing feedback loop. Driven by recursion, fractals are images of dynamic systems .So essentially, a fractal is a perfect image. When you zoom in on any part of the fractal, a perfect copy of the larger image is in the smaller one.

Sierpinski gasket

Koch Snowflake



One of the first fractals ever discovered is called the **Koch Snowflake** (discovered in 1904) which was found by taking an equilateral triangle, dividing the sides into threes and using the middle segment as a base as another equilateral triangle on each side. (shown on the left of the gallery on the bottom) On the right of the Koch Snowflake, is the equation for the snowflake. Because the triangles expanding on each other, the area of this is infinite.

$$A_1 = A_0 + 3A_0 4^0 \left(\frac{1}{9}\right)^1$$

$$A_2 = A_1 + 3A_0 4^1 \left(\frac{1}{9}\right)^2$$

$$= A_0 + 3A_0 4^0 \left(\frac{1}{9}\right)^1 + 3A_0 4^1 \left(\frac{1}{9}\right)^2$$

$$A_3 = A_2 + 3A_0 4^2 \left(\frac{1}{9}\right)^3$$

$$= A_0 + 3A_0 4^0 \left(\frac{1}{9}\right)^1 + 3A_0 4^1 \left(\frac{1}{9}\right)^2 + 3A_0 4^2 \left(\frac{1}{9}\right)^3$$

$$\vdots$$

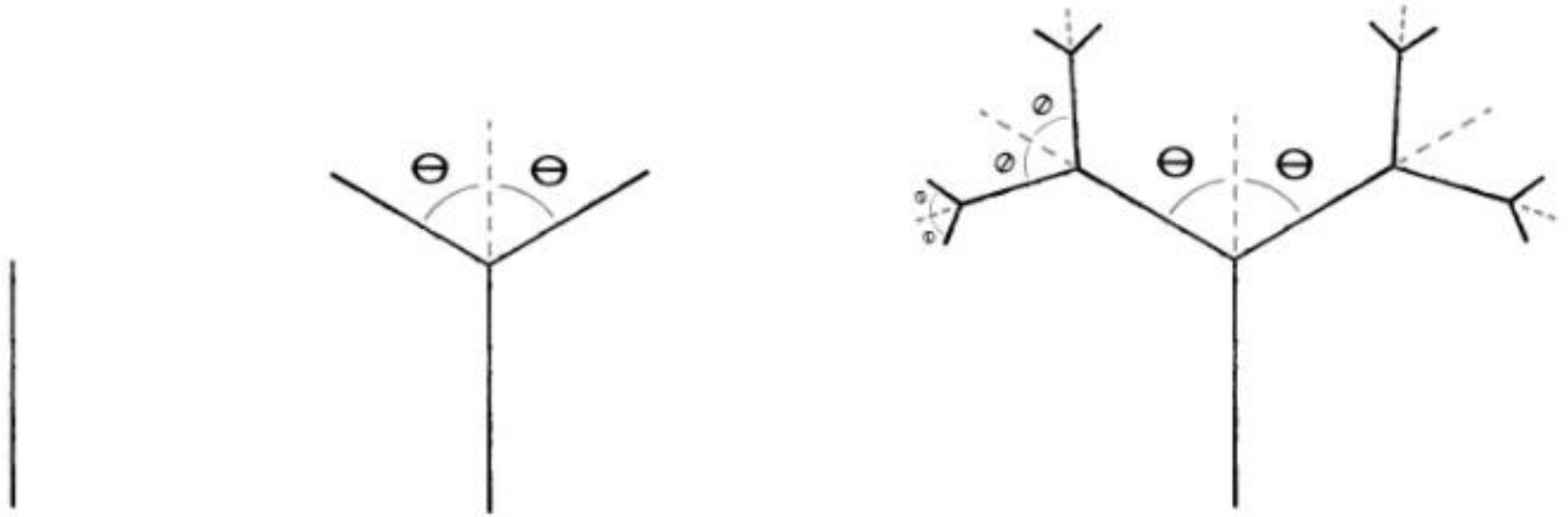
$$A_n = A_0 + 3A_0 \sum_{i=1}^{n} 4^{i-1} \left(\frac{1}{9}\right)^i$$

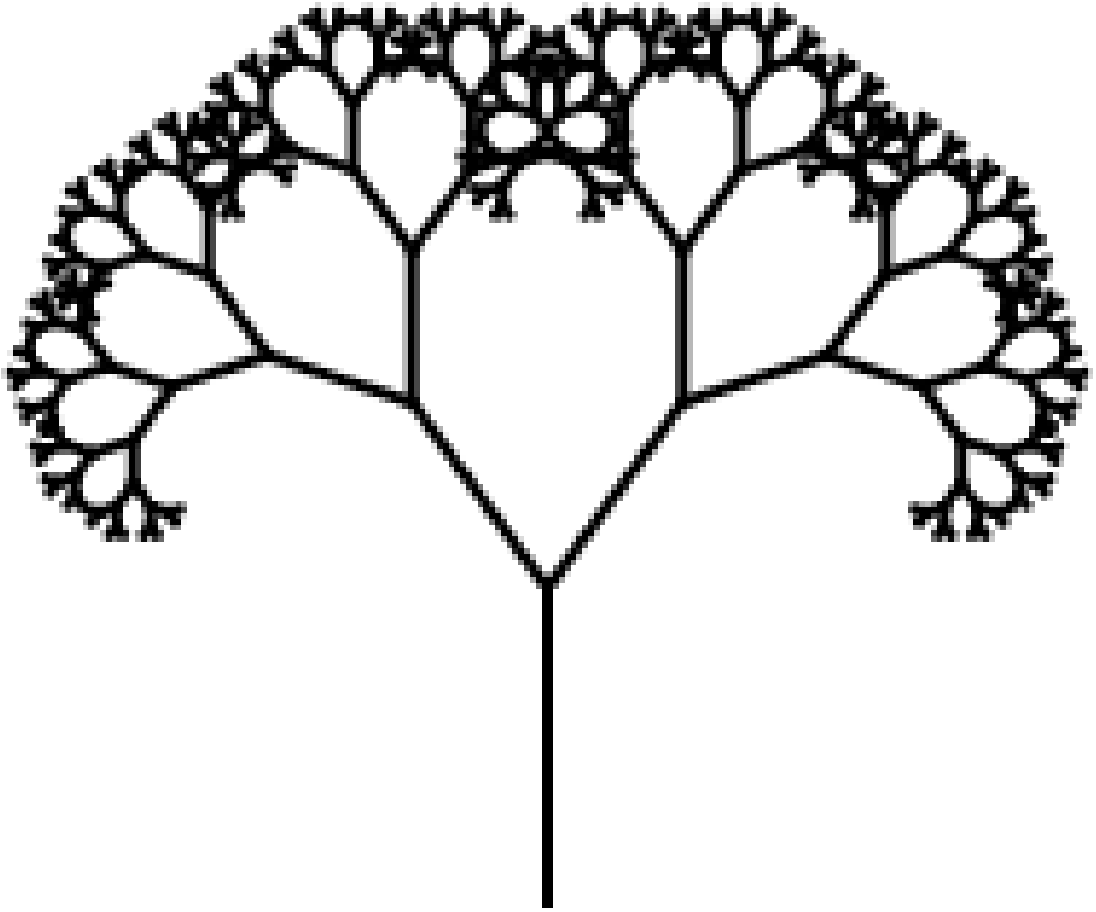
$$= A_0 + \frac{3}{4} A_0 \sum_{i=1}^{n} \left(\frac{4}{9}\right)^i$$

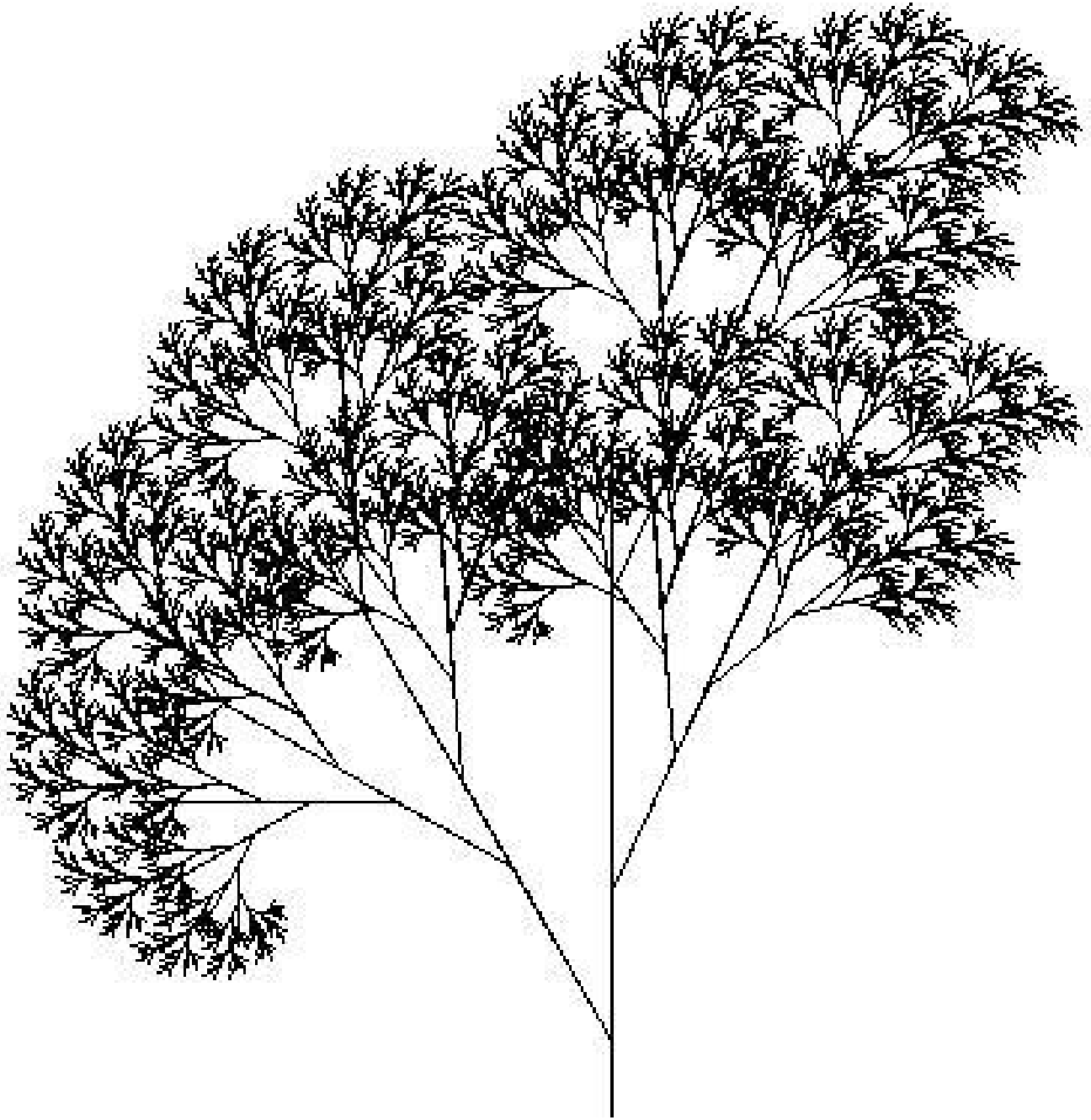
1. Draw a line.

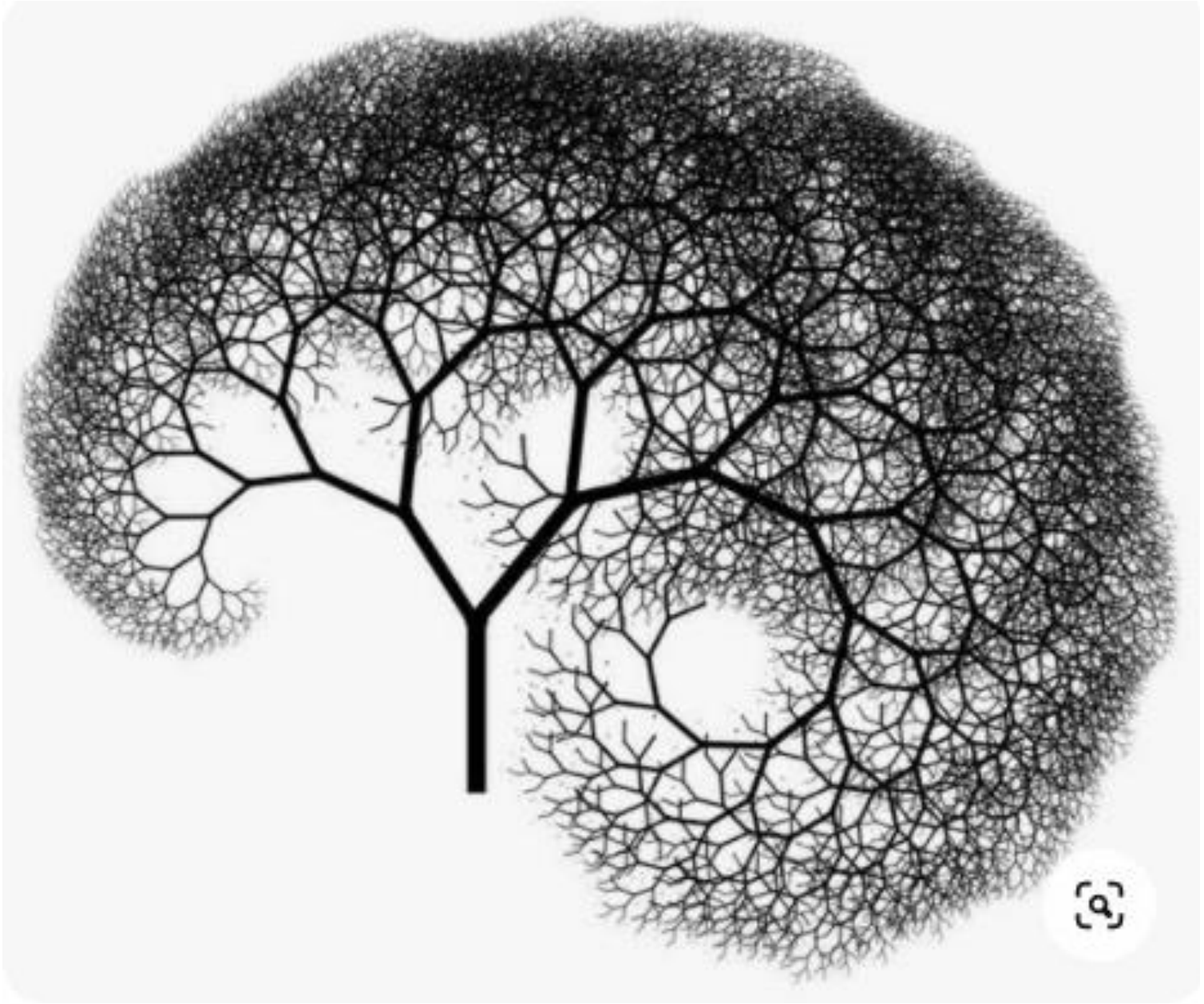
2. At the end of the line, (a) rotate to the left and draw a shorter line and (b) rotate to the right and draw a shorter line.

3. Repeat step 2 for the new lines, again and again and again.









Recursive function

- In recursive algorithm we must:
  1. Determine base case
  2. Determine recursive case
- Two types of recursion
  1. Linear recursion
  2. Binary recursion

Recursion

- In some problems, it may be natural to define the problem in terms of the problem itself.
- Recursion is useful for problems that can be represented by a simpler version of the same problem.
- Example: the factorial function

```
6! = 6 * 5 * 4 * 3 * 2 * 1
```

We could write:

```
6! = 6 * 5!
```

How to write recursively?

```
int recur_fn(parameters)
{
  if(stopping condition)
       return stopping value;
  // other stopping conditions if needed
  return function of recur_fn(revised parameters);
}
```

Example 1: factorial function

In general, we can express the factorial function as follows:

```
n! = n * (n-1)!
```

Is this correct? Well… almost.

The factorial function is only defined for *positive* integers. So we should be a bit more precise:

```
n! = 1 (if n is equal to 1)
n! = n * (n-1)!   (if n is larger than 1)
```

Base and recursive cases

$$\text{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{fact}(n-1) & \text{if } n > 0 \end{cases}$$

factorial function

```
int fac(int numb)
{
    if(numb<=1)
        return 1;
    else
        return numb * fac(numb-1);
}
```

recursion means that a function calls itself

factorial function

- Assume the number typed is 3, that is, numb=3.
- **fac(3) :**

```
3 <= 1 ?                    No.
fac(3) = 3 * fac(2)
  fac(2) :
     2 <= 1 ?               No.
     fac(2) = 2 * fac(1)
        fac(1) :
           1 <= 1 ?         Yes.
           return 1
     fac(2) = 2 * 1 = 2
     return fac(2)
fac(3) = 3 * 2 = 6
return fac(3)
fac(3)  has the value 6
```

```
int fac(int numb){
   if(numb<=1)
       return 1;
   else
       return numb * fac(numb-1);
}
```

Example 2: Fibonacci numbers

- Fibonacci numbers:

  `0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...`

  where each number is the sum of the preceding two.

- Recursive definition:
  - `F(0) = 0;`
  - `F(1) = 1;`
  - `F(number) = F(number-1)+ F(number-2);`

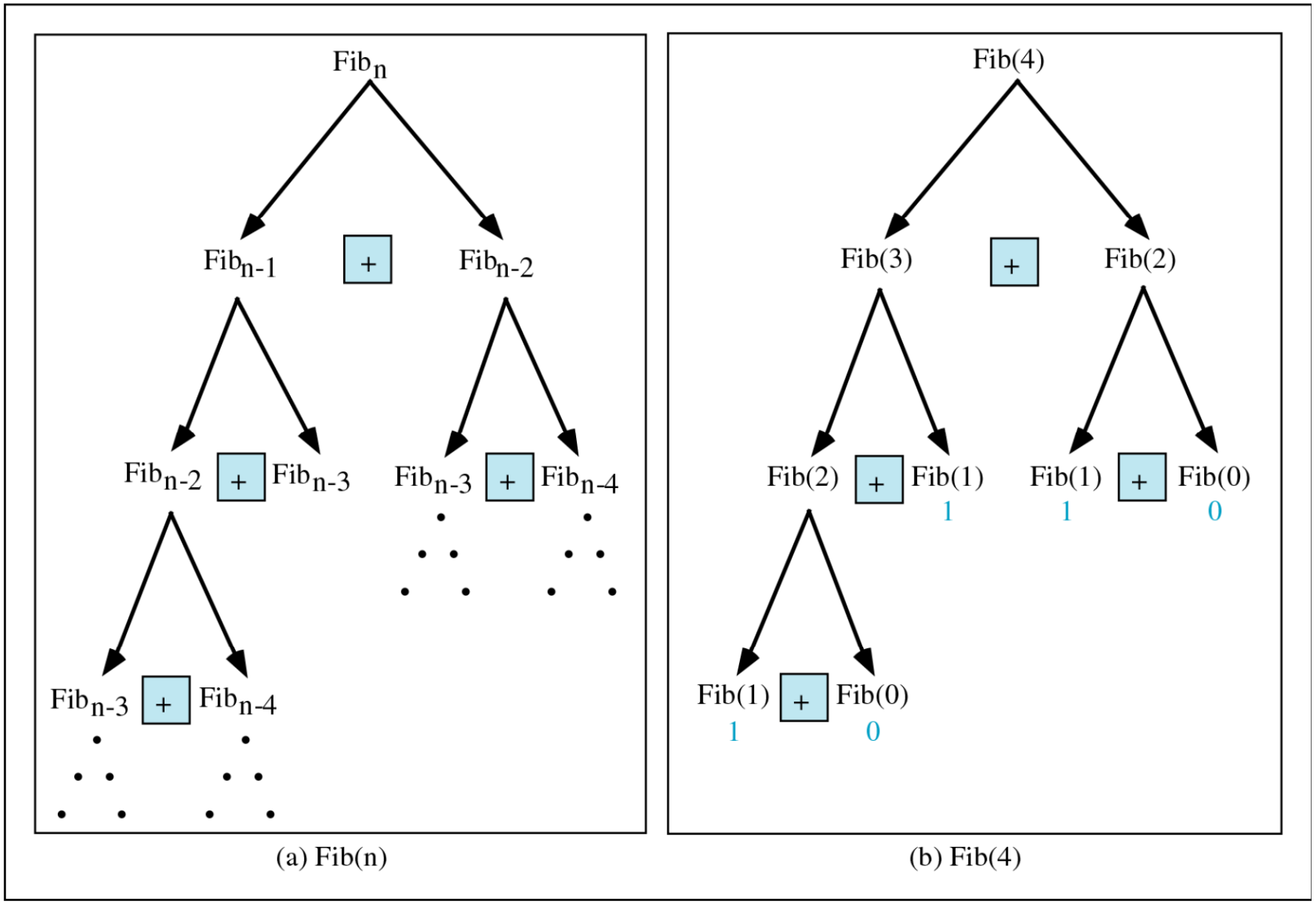
Base and recursive cases

$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1\,. \\ F(n-1) + F(n-2) & n > 1 \end{cases}$$

Example 3: Fibonacci numbers

```
//Calculate Fibonacci numbers using recursive function.
//A very inefficient way, but illustrates recursion well

int fib(int number)
{
  if (number == 0) return 0;
  if (number == 1) return 1;
  return (fib(number-1) + fib(number-2));
}
```

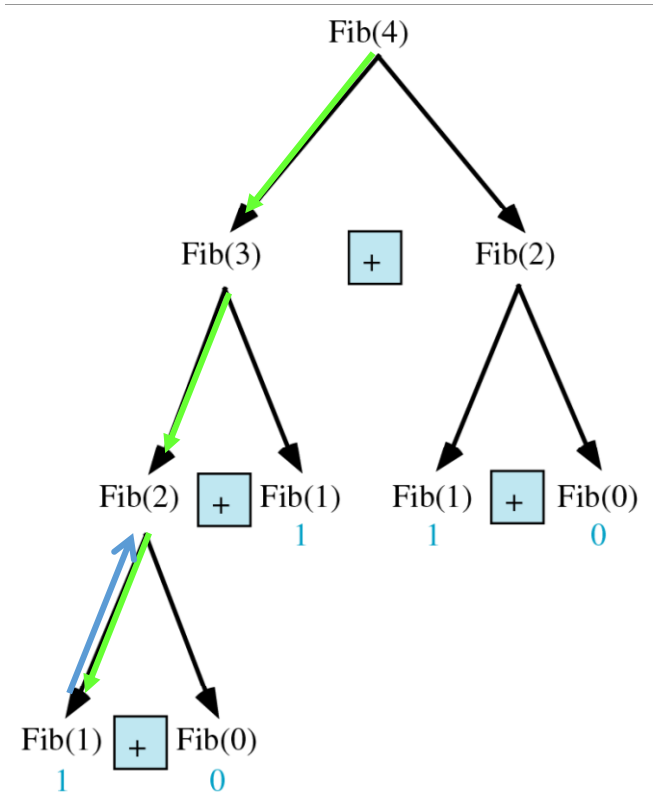

Fib$_n$
Fib$_{n-1}$
+
Fib$_{n-2}$
Fib$_{n-2}$
+
Fib$_{n-3}$
Fib$_{n-3}$
+
Fib$_{n-4}$
Fib$_{n-3}$
+
Fib$_{n-4}$
(a) Fib(n)
Fib(4)
Fib(3)
+
Fib(2)
Fib(2)
+
Fib(1)
1
Fib(1)
+
Fib(0)
0
Fib(1)
1
+
Fib(0)
0
(b) Fib(4)

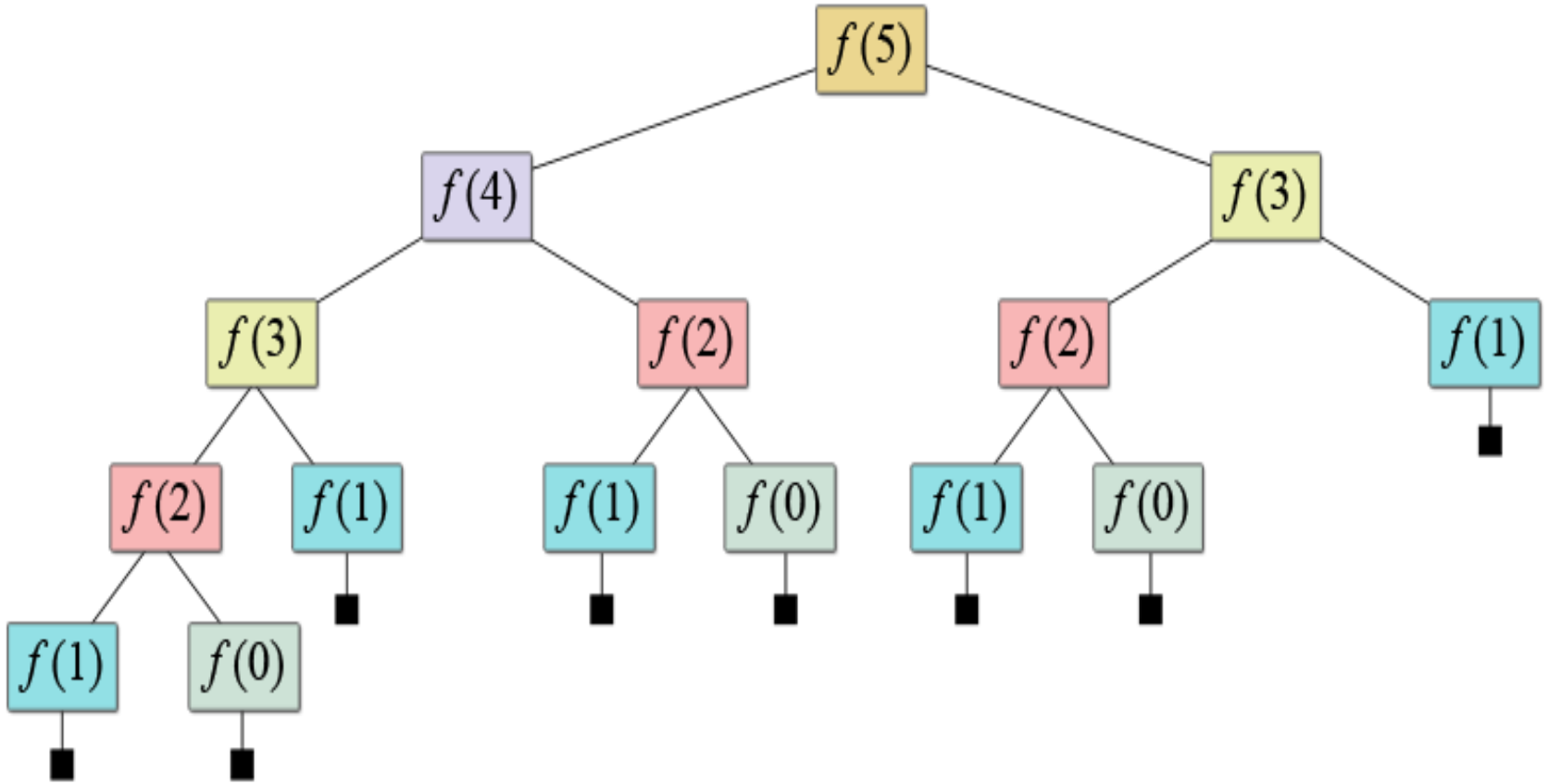
Trace a Fibonacci Number

```
int fib(int num)
{
    if (num == 0) return 0;
    if (num == 1) return 1;
    return
      (fib(num-1)+fib(num-2));
}
```

- Assume the input number is 4, that is, num=4:

```
fib(4):
    4 == 0 ? No;   4 == 1?   No.
    fib(4) = fib(3) + fib(2)
    fib(3):
        3 == 0 ? No; 3 == 1? No.
        fib(3) = fib(2) + fib(1)
        fib(2):
            2 == 0? No; 2==1? No.
            fib(2) = fib(1)+fib(0)
            fib(1):
                1== 0 ? No; 1 == 1? Yes.
                fib(1) = 1;
                return fib(1);
```





This tree shows how inefficient this algorithm

Inefficient algorithm

n	Number of Terms Computed
0	1
1	1
2	3
3	5
4	9
5	15
6	25

Fibonacci number w/o recursion

```
//Calculate Fibonacci numbers iteratively
//much more efficient than recursive solution

int fib(int n)
{
  int f[n+1];
  f[0] = 0;
   f[1] = 1;
  for (int i=2; i<= n; i++)
      f[i] = f[i-1] + f[i-2];
  return f[n];
}
```

Iterative solution vs. recursive

n	$n+1$	$2^{n/2}$	Execution Time Using Algorithm 1.7	Lower Bound on Execution Time Using Algorithm 1.6
40	41	1,048,576	41 ns[*]	1048 μs[†]
60	61	1.1×10^9	61 ns	1 s
80	81	1.1×10^{12}	81 ns	18 min
100	101	1.1×10^{15}	101 ns	13 days
120	121	1.2×10^{18}	121 ns	36 years
160	161	1.2×10^{24}	161 ns	3.8×10^7 years
200	201	1.3×10^{30}	201 ns	4×10^{13} years

[*] 1 ns = 10^{-9} second.

[†] 1 μs = 10^{-6} second.

Algorithm 1.7 : iterative fibonacci

Algorithm 1.6 : recursive fibonacci

End