1. **Recursion** is the general term for defining an object in terms of itself.
   e.g., An *inductive proof* establishes the truth of $P(k+1)$ *recursively* in terms of $P(k)$.
   e.g., A fractal is a recursive object (see the Koch snowflake).

2. A **recursive definition** of a function, predicate, set, or other structure, defines the larger elements in terms of the smaller elements.
   Example: Recall an arithmetic progression: $\{a_j\}$ where $a_j = b + jd$. We can define this series by recursion where $a_0 = b$ and $a_j = a_{j-1} + d$. (For instance $\{1, 3, 5, 7, 9, ...\}$.)

3. A function $f(n)$ where n is a nonnegative integer is **defined recursively** if

   (a) $f(0), f(1), \ldots, f(k)$ are given, and

   (b) for $n > k$, a rule is given for computing $f(n)$ from $f(0), f(1), \ldots, f(n-1)$.

   Example: $f(n) = n!$. Define as $f(0) = 1$ (base case) and $f(n) = nf(n-1)$ (recursion).

4. The Fibonacci numbers $f_0, f_1, f_2, \ldots$ are defined by the following:

$$f_n = \begin{cases} 0 & \text{if} \quad n = 0 \\ 1 & \text{if} \quad n = 1 \\ f_{n-1} + f_{n-2} & \text{if} \quad n \geq 2 \end{cases}$$

   $\lim_{n\to\infty}(f_{n+1}/f_n) = \phi$ where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.6180327868852$ ("the golden ratio").

5. A **recursive algorithm** solves a problem by reducing it to an instance of the same problem with a smaller input ("Top down"). A recursive algorithm calls itself.

   a) Algorithm 1 recursively computes the value of a^n.
   b) Algorithm 2 recursively computes the value of $n!$.

6. An **iterative algorithm** starts with the value of the function at one (or more) base case(s), and successively applies the recursive definition to build up the values of the function at successively larger integers ("Bottom up"). An iterative algorithm typically uses *for* loops.

   a) Algorithm 3 iteratively computes the value of $n!$.

7. *Binary search*: Algorithm 4 is a recursive, whereas Alg 5 is original iterative version.

8. Algorithm 6 is an algorithm for computing the nth Fibonacci number.
   Is it *recursive* or *iterative*?

Algorithm 1 A recursive algorithm for computing a^n.

```
POWER (a, n)
  if n = 0 then
    POWER(a,n)=1
  else
    POWER(a,n) = a· POWER(a, n − 1)
  end if
```

Algorithm 2 A recursive algorithm for computing $n!$.

```
FACTORIAL (n)
  if n = 0 then
    FACTORIAL(n) =1
  else
    FACTORIAL(n) = n· FACTORIAL(n − 1)
  end if
```

Algorithm 3 An iterative algorithm for computing $n!$.

```
FACTORIAL (n)
  x ← 1
  for i = 2, 3, . . . , n do
    x ← x · i
  end for
  return x
```

Algorithm 4 A recursive binary search algorithm. x, i, j are integers with $1 \geq i, j \geq n$.

```
BINARYSEARCH (x, i, j)
  m ← ⌊(i + j)/2⌋
  if x = a_m then
    return m
  else if x < a_m and i < m then
    return BINARYSEARCH(x, i, m − 1)
  else if x > a_m and j > m then
    return BINARYSEARCH(x, m + 1, j)
  else
    return 0
  end if
```

Algorithm 5 The original binary search algorithm from Handout 7.

```
BINARY-SEARCH (a, x)
  i ← 1 {i is the left endpoint of search interval}
  j ← n {j is the right endpoint of search interval}
  while i < j do
    m ← ⌊(i + j)/2⌋
    if x > a_m then
      i ← m + 1
    else
      j ← m
    end if
  end while
  if x = a_i then
    location ← i
  else
    location ← 0
  end if
  return location {location is the subscript of the term that equals x, or is 0 if x is not found}
```

Algorithm 6 An algorithm for computing the nth Fibonacci number.

```
FIBONACCI (n)
  if n = 0 then
    y ← 0
  else
    x ← 0, y ← 1
    for i = 1, 2, ..., n − 1 do
      z ← x + y
      x ← y
      y ← z
    end for
  end if
  return y
```

9. Note *recursion* is often less efficient than *iteration*! But it is easier to implement and elegant.

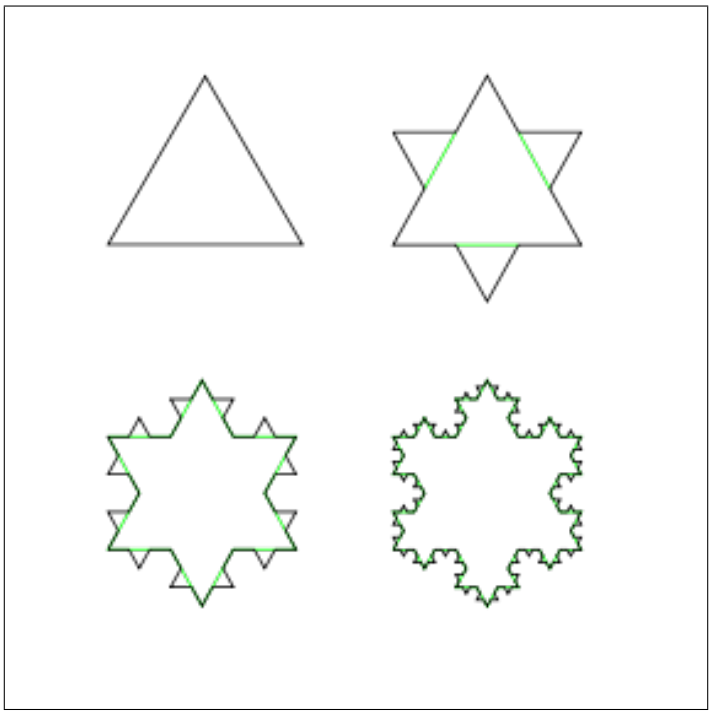

Fig 1. First four iterations in constructing the Koch snowflake.

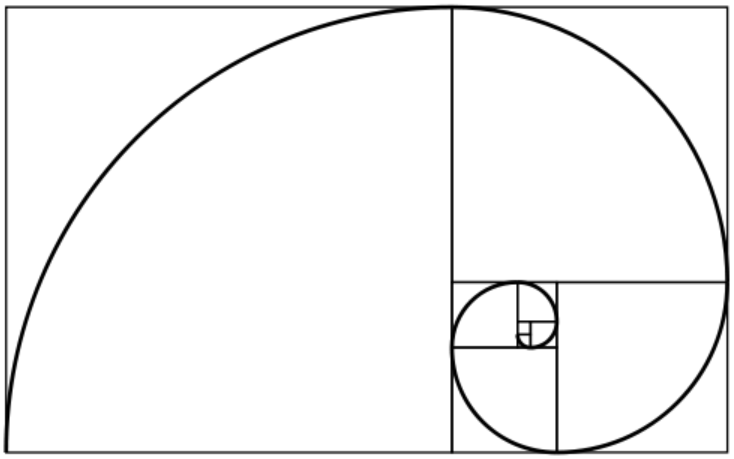

Fig 2. Fibonacci spiral out to squares of of size 34.