COMP20010

Data Structures and Algorithms I
07 - Recursion I

Dr. Aonghus Lawlor
aonghus.lawlor@ucd.ie

Outline

- Recursion
- Linear and Binary Recursion
- Examples

Recursion

- Recursive Definitions
- Base and General Cases of Recursion
- What is a Recursive Algorithm
- Recursive Functions
- Using Recursive Functions

To understand *recursion*,

YOU MUST FIRST

understand *recursion*

Recursion

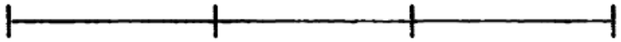
- Repetition can be achieved by writing loops, such as for loops and while loops.
- Another way to achieve repetition is through ***recursion***, which occurs when a function refers to itself in its own definition.
- Recursion provides an elegant and powerful alternative for performing repetitive tasks.

a function that calls itself
a way of solving problems using divide & conquer

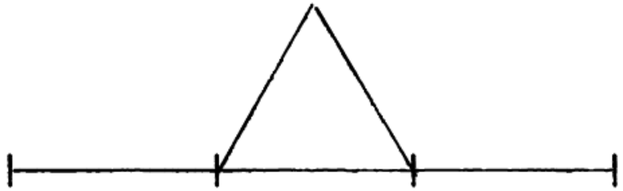
Koch Snowflake

1. Start with a line.

2. Divide the line into three equal parts.



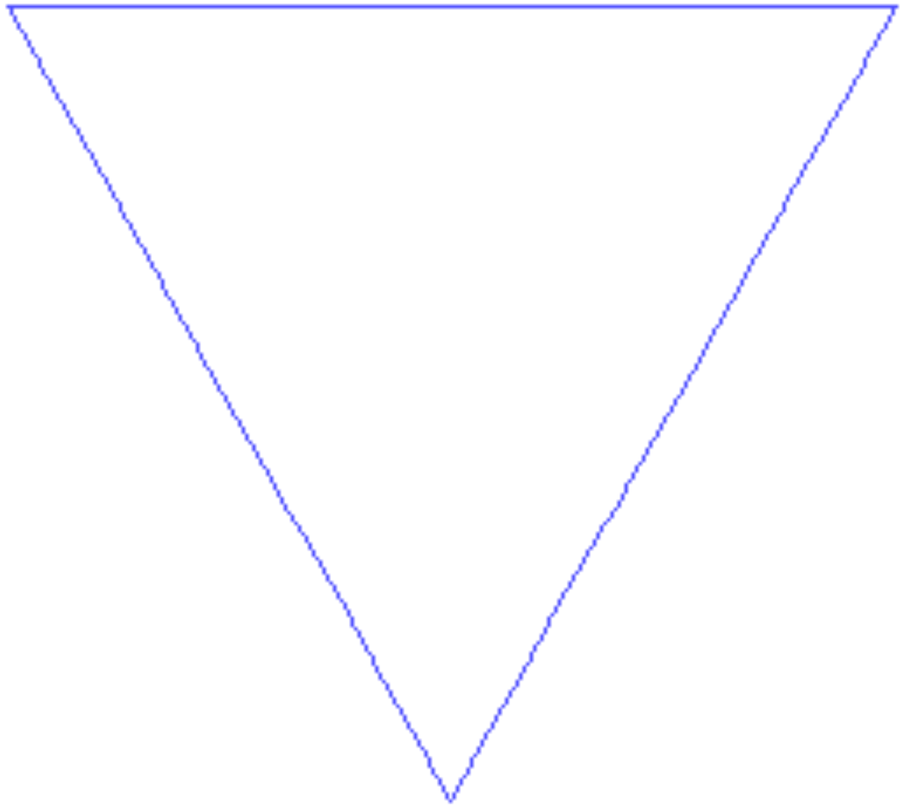
3. Draw an equilateral triangle (a triangle where all the sides are equal) using the middle segment as its base.



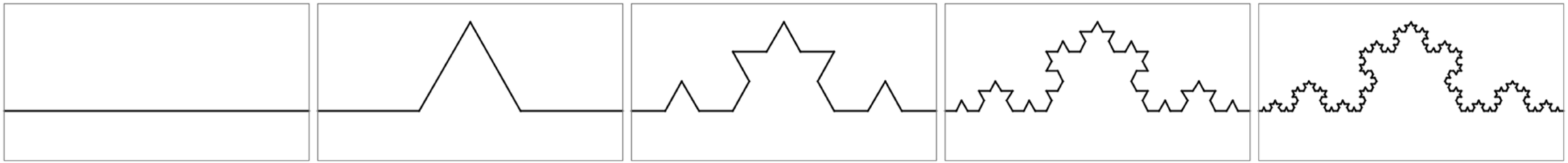
4. Erase the base of the equilateral triangle (the middle segment from step 2).

5. Repeat steps 2 through 4 for the remaining lines again and again and again.

Koch Snowflake

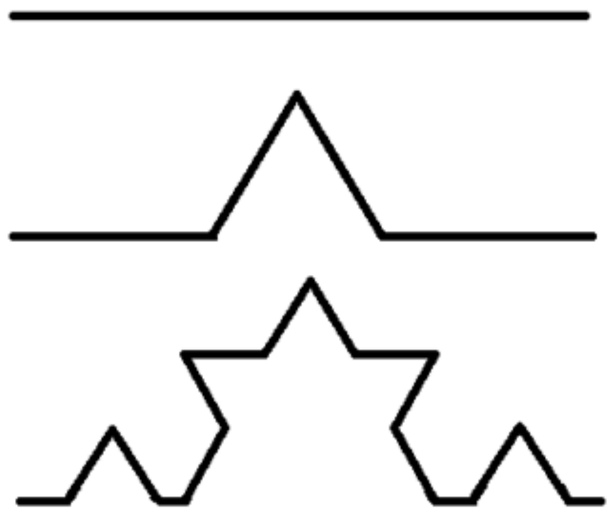


Koch Snowflake



Koch Snowflake

- a **fractal** is a drawing which also has *self-similar* structure, where it can be defined in terms of itself.
- An order 0 Koch fractal is simply a straight line of a given size.
- For an order 1 Koch fractal, instead of drawing just one line, draw four smaller segments
- if we repeat this pattern again on each of the order 1 segments, we get the order 2 Koch fractal



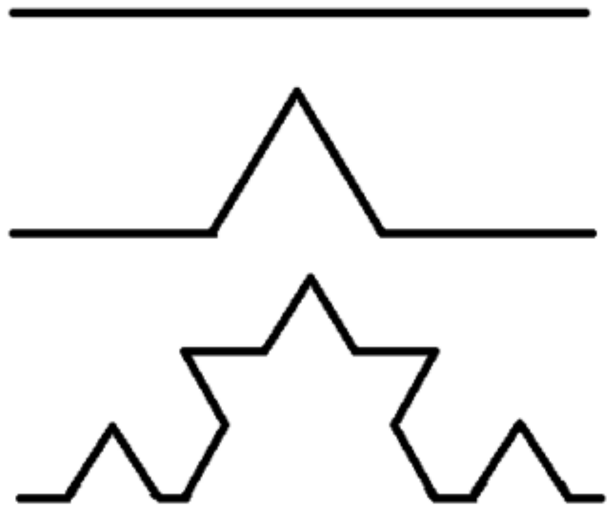
Stage 0

Stage 1

Stage 2

Koch Snowflake

- Now let us think about it the other way around.
- To draw a Koch fractal of order 2, we can simply draw four order 1 Koch fractals. But each of these in turn needs four order 0 Koch fractals
- Ultimately, the only drawing that will take place is at order 0.



Stage 0

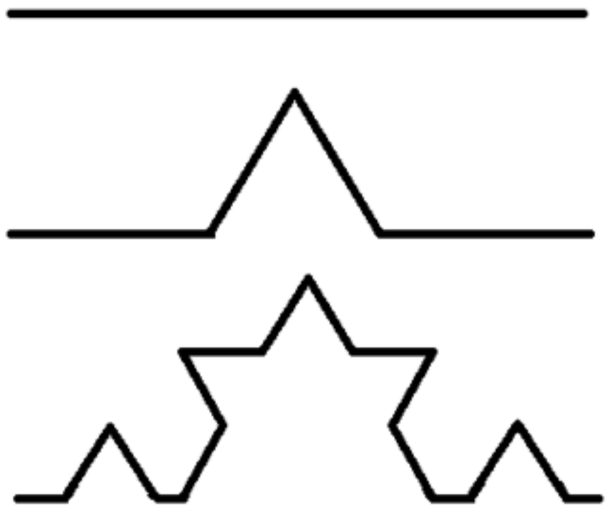
Stage 1

Stage 2

Koch Snowflake

```
def koch(t, order, size):
    """
       Make turtle t draw a Koch fractal of 'order' and 'size'.
       Leave the turtle facing the same direction.
    """

    if order == 0:          # The base case is just a straight line
        t.forward(size)
    else:
        koch(t, order-1, size/3)   # Go 1/3 of the way
        t.left(60)
        koch(t, order-1, size/3)
        t.right(120)
        koch(t, order-1, size/3)
        t.left(60)
        koch(t, order-1, size/3)
```



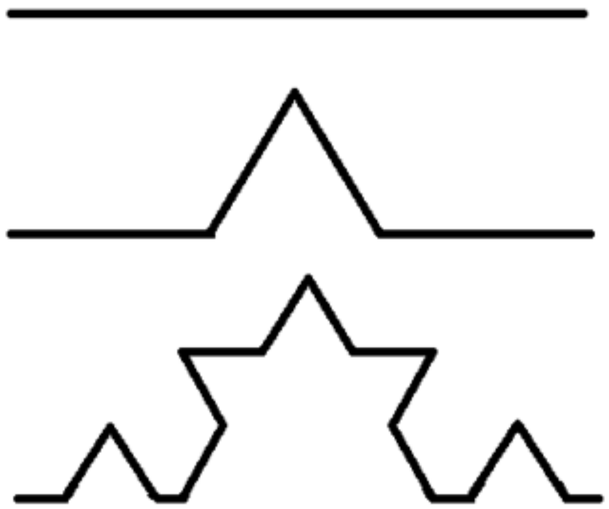
Stage 0

Stage 1

Stage 2

Koch Snowflake

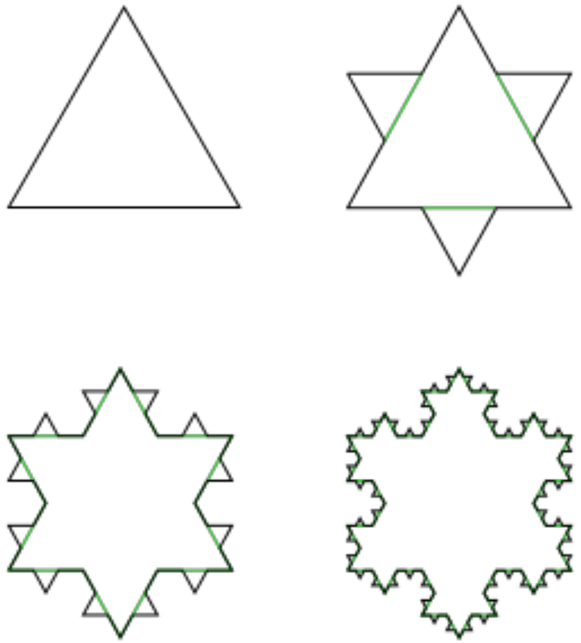
```
def koch(t, order, size):
    if order == 0:
        t.forward(size)
    else:
        for angle in [60, -120, 60, 0]:
            koch(t, order-1, size/3)
            t.left(angle)
```

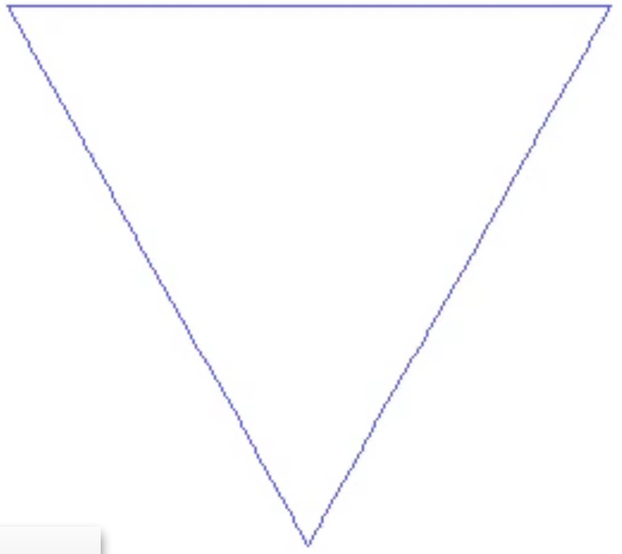


The key thing that is new here is that if order is not zero, `koch` calls itself recursively to get its job done.

Let's make a simple observation and tighten up this code. Remember that turning right by 120 is the same as turning left by -120.

Koch Snowflake

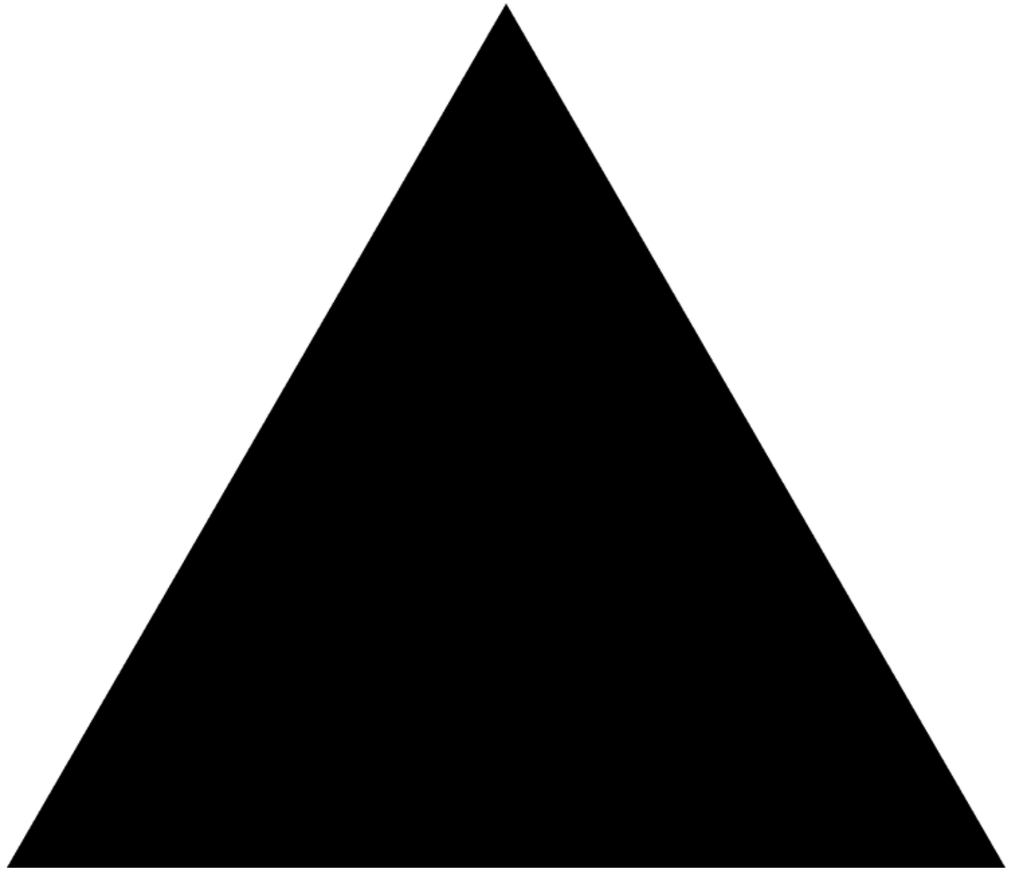




- ignore the subproblem while you solve the big problem.
- think about how it will work when called for order 2 *under the assumption that it is already working for level 1*.
- And, in general, if I can assume the order n-1 case works, can I just solve the level n problem?

Sierpiński Triangle/Sieve

Sierpiński Triangle/Sieve



Applications of the Sierpiński Triangle to MusicalComposition

Recursion

Often it is difficult to express the members of an object or numerical sequence explicitly.

example: factorial

$$8! = 1 * 2 * 3 * 4 * 5 * 6 * 7 * 8$$

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \ldots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1 \end{cases}$$

Recursion

Often it is difficult to express the members of an object or numerical sequence explicitly.

example: factorial

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \ldots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1 \end{cases}$$

```
1: function FACTORIAL (n)
2:     Input: n non-negative integer
3:     Output: factorial of n: n!
4:     s ← 1
5:     for i = n to 2 do
6:         s ← s * i
7:     end for
8:     return s
9: end function
```

Recursion

There may, however, be some “local” connections that can give rise to a **recursive definition** –a formula that expresses higher terms in the sequence, in terms of lower terms

Definition:
The process of solving a problem by reducing it to smaller versions of itself is called **recursion.**

Recursion

$$8! = 1 * 2 * 3 * 4 * 5 * 6 * 7 * 8$$

$$factorial(n) = \begin{cases} 1 & \text{if } n = 0 \\ nfactorial(n-1) & \text{if } n \geq 1 \end{cases}$$

initialisation/base

recursion

Recursion

$$8! = 1 * 2 * 3 * 4 * 5 * 6 * 7 * 8$$

$$factorial(n) = \begin{cases} 1 & \text{if } n = 0 \\ n\,factorial(n-1) & \text{if } n \geq 1 \end{cases}$$

initialisation/base

recursion

First, it contains one or more ***base cases***, which are defined non-recursively in terms of fixed quantities. In this case, *n* = 0 is the base case.

It also contains one or more ***recursive cases***, which are defined by appealing to the definition of the function being defined. There is no circularity in this definition because each time the function is invoked, its argument is smaller by one.

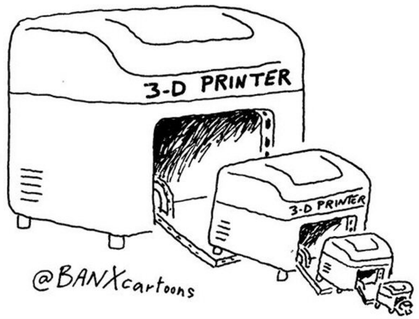
Recursion

$$8! = 1 * 2 * 3 * 4 * 5 * 6 * 7 * 8$$

initialisation/base

$$factorial(n) = \begin{cases} 1 & \text{if } n = 0 \\ n factorial(n-1) & \text{if } n \geq 1 \end{cases}$$

recursion



1. Every recursive definition must have one (or more) base cases.
2. The general case must eventually reduce to a base case.
3. The base case stops the recursion

Induction analogy

we want to show that P(n) holds for all natural numbers n

$$P(n) = 0 + 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

Start with a **base**. Show the statement is true for n=0

$$P(0) = 0 = \frac{0(0+1)}{2}$$

Inductive step, if P(k) is true then P(k+1) must be true.

$$P(k) + (k+1) = \frac{(k+1)((k+1)+1)}{2}$$

Then it must be true for all n.

Recursion

a recursive definition usually consists of two parts:

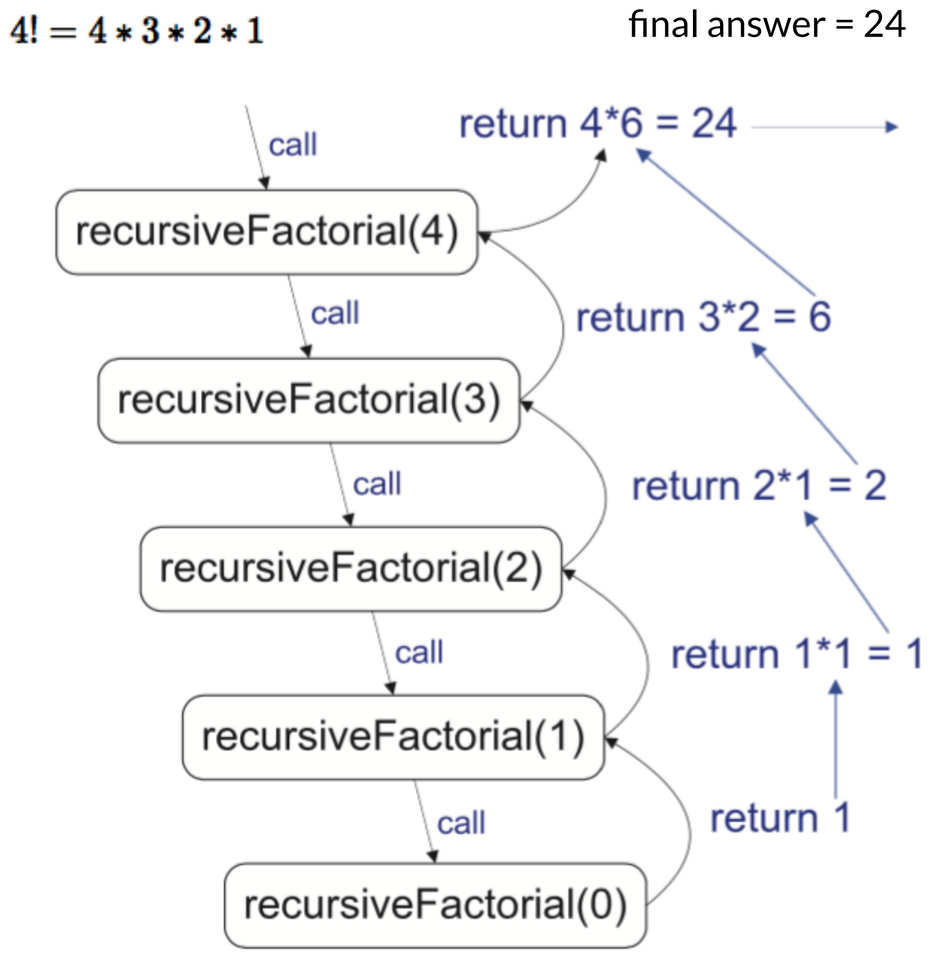
Initialisation –analogous to induction **base cases**

Recursion –analogous to **induction** step

Recursion

We can illustrate the execution of a recursive function definition by means of a ***recursion trace***. Each entry of the trace corresponds to a recursive call. Each new recursive function call is indicated by an arrow to the newly called function.

```
1: function FACTORIAL (n)
2:     Input: n non-negative integer
3:     Output: factorial of n: n!
4:     s ← 0
5:     if n = 0 then
6:         return 1
7:     end if
8:     return n × Factorial(n − 1)
9: end function
```

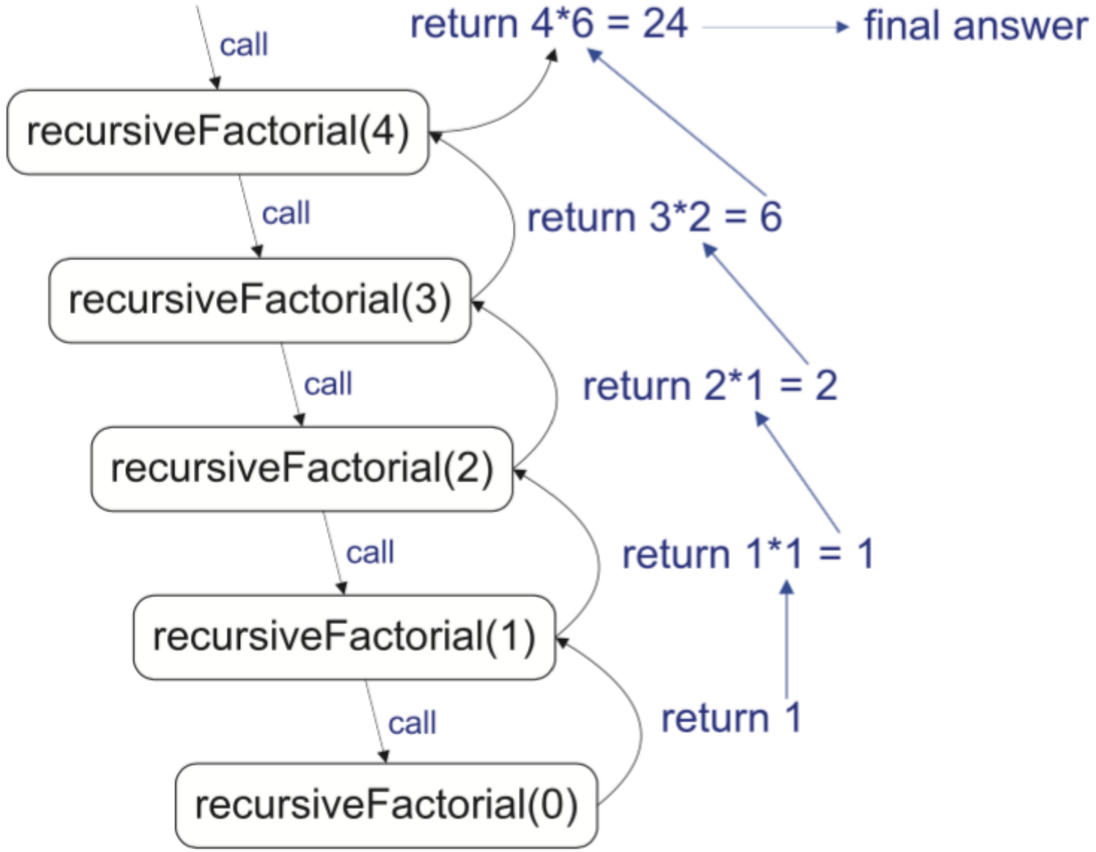


Recursion

Think of a recursive function as having infinitely many copies of itself.

Every call to a recursive function has its own code and its own set of parameters and its own local variables.

After completing a particular recursive call, control goes back to the calling environment, ie: the previous call.



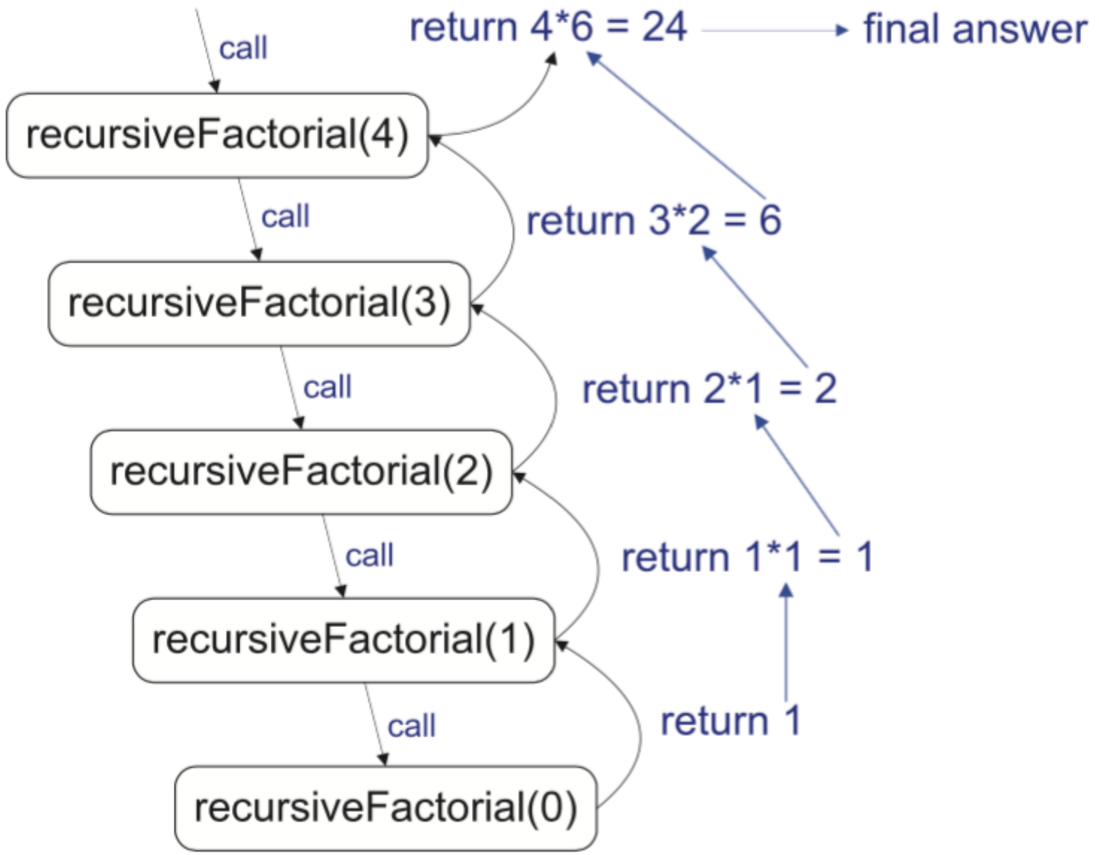
Recursion

The current (recursive) call must execute completely before the control goes back to the previous call.

The execution in the previous call begins from the point immediately following the recursive call.

A recursive function in which the last statement executed is the recursive call is called a **tail recursive function**.

The factorial function is an example of a tail recursive function.



Tail recursion

Using recursion can often be a useful tool for designing algorithms that have elegant, short definitions. But this usefulness does come at a modest cost.

An algorithm uses tail recursion if it uses linear recursion and the algorithm makes a recursive call as its very last operation.

It is not enough that the last statement in the function definition includes a recursive call, however. In order for a function to use tail recursion, the recursive call must be absolutely the last thing the function does (except the base case!)

When an algorithm uses tail recursion, we can convert the recursive algorithm into a non-recursive one, by iterating through the recursive calls rather than calling them explicitly.

Recursion

linear recursion: a function is defined so that it makes at most one recursive call each time it is invoked.

simplest form of recursion

useful when we view an algorithmic problem in terms of a first or last element plus a remaining set that has the same structure as the original set.

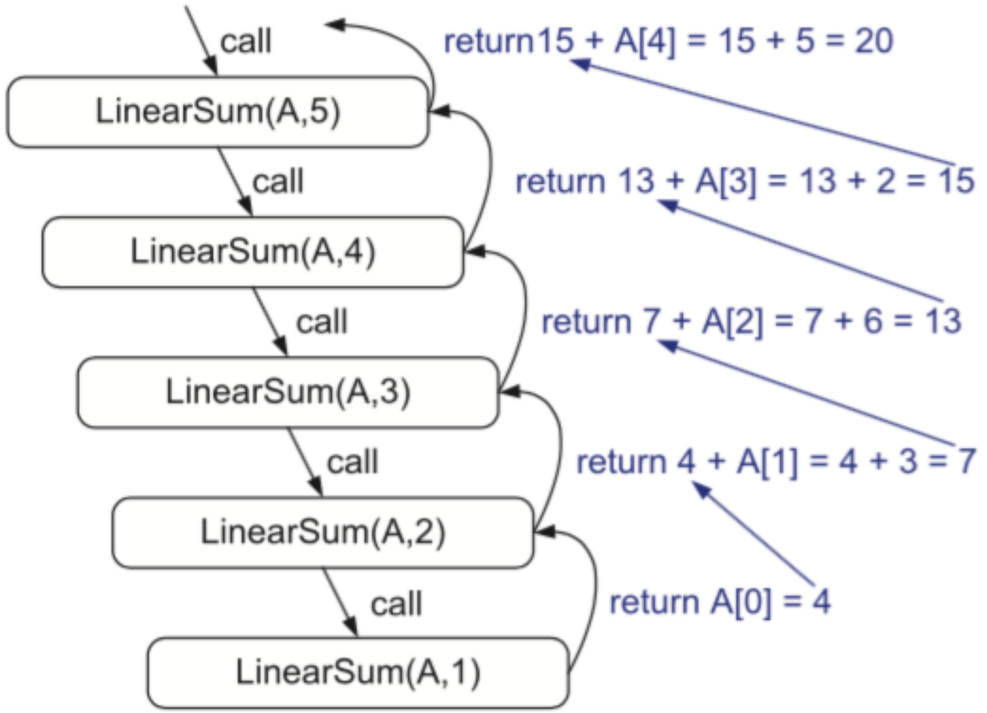
```
Algorithm LinearSum(A,n):
Input: A integer array A and an integer n ≥ 1, such that A has at least n elements
Output: The sum of the first n integers in A

if n = 1 then
    return A[0]
else
    return LinearSum(A, n - 1) + A[n - 1]
```

Recursion

O(n) growth factor

also O(n) in memory since we need space for each function call




```
Algorithm LinearSum(A,n):
Input: A integer array A and an integer n ≥ 1,
such that A has at least n elements
Output: The sum of the first n integers in A

if n = 1 then
    return A[0]
else
    return LinearSum(A, n - 1) + A[n - 1]
```

Reversing an Array

- How to reversing the *n* elements of an array, *A*, so that the first element becomes the last, the second element becomes second to the last, and so on.
- We can solve this problem using linear recursion, by observing that the reversal of an array can be achieved by swapping the first and last elements and then recursively reversing the remaining elements in the array

Reverse Array

A simple recursive algorithm to reverse the elements of an array

We start by calling ReverseArray(*A*,0,*n*–1)

```
1: function ReverseArray (A, i, j)
2:     Input: An array A and non-negative integer indices i and j
3:     Output: Reversal of elements in A starting at i and ending at j
4:     if i < j then
5:         swap A[i] and A[j]
6:         ReverseArray(A, i + 1, j − 1)
7:     end if
8:     return
9: end function
```

There two base cases: *i* = *j* and *i* > *j* (in either case, we simply terminate)

In the recursive step we are guaranteed to make progress towards one of these two base cases.

If *n* is odd, we eventually reach the *i* = *j* case, and if *n* is even, we eventually reach the *i* > *j* case.

The recursive algorithm is guaranteed to terminate.

Reverse Array

When we use a recursive algorithm to solve a problem, we have to use some memory for the state of each active recursive call.

It is often useful to be able to derive non-recursive algorithms from recursive ones.

We can easily convert algorithms that use ***tail recursion***.

An algorithm uses tail recursion if it uses linear recursion and the algorithm makes a recursive call as its very last operation.

The recursive ReverseArray would be converted to something like this (iteration):

```
1:  function IterativeReverseArray (A, i, j)
2:      Input: An array A and non-negative integer indices i and j
3:      Output: Reversal of elements in A starting at i and ending at j
4:      while i < j do
5:          swap A[i] and A[j]
6:          i ← i + 1
7:          j ← j − 1
8:      end while
9:      return
10: end function
```

Recursion

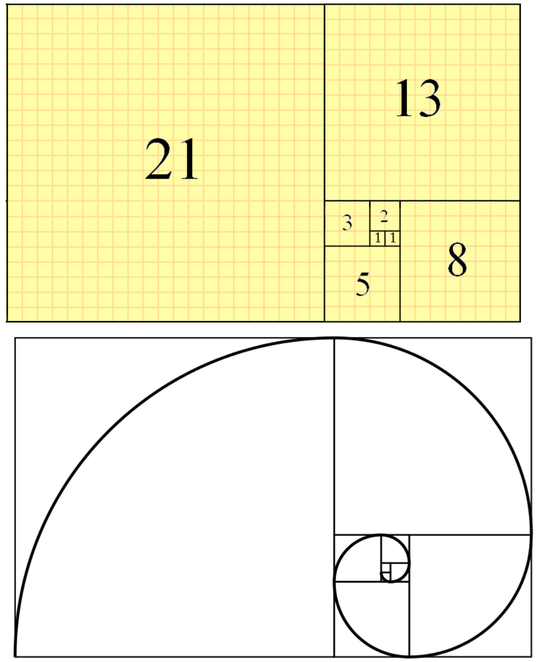
- ***recursion*** is the concept of defining a function that makes a call to itself.
- When a function calls itself, this is a ***recursive*** call.
- A function *M* is recursive if it calls another function that ultimately leads to a call back to *M*.
- Benefit: recursion allows us to take advantage of the repetitive structure present in the problem
- We can sometimes use recursion to avoid complex case analyses and nested loops
- more readable algorithms
- reasonable efficiency (with care!)

Fibonacci Numbers

- Simple integer sequence recursively defined as the sum of the previous two numbers in the sequences

$$Fib(n) = 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

$$Fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ Fib(n-1) + Fib(n-2) & \text{if } n > 1 \end{cases}$$



Recursion

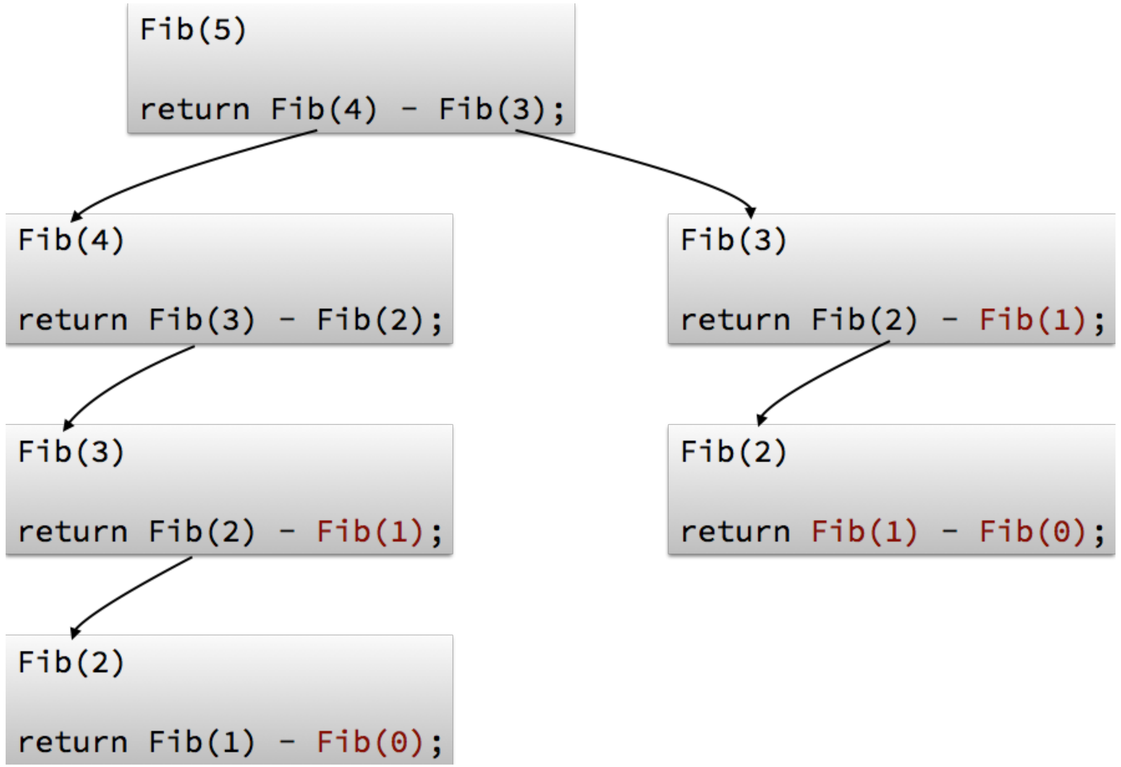
- recursive fibonacci:

```
int fibonacci(int n) {
    if(n <= 1) {
        return n;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

2 recursive calls per iteration

Recursion

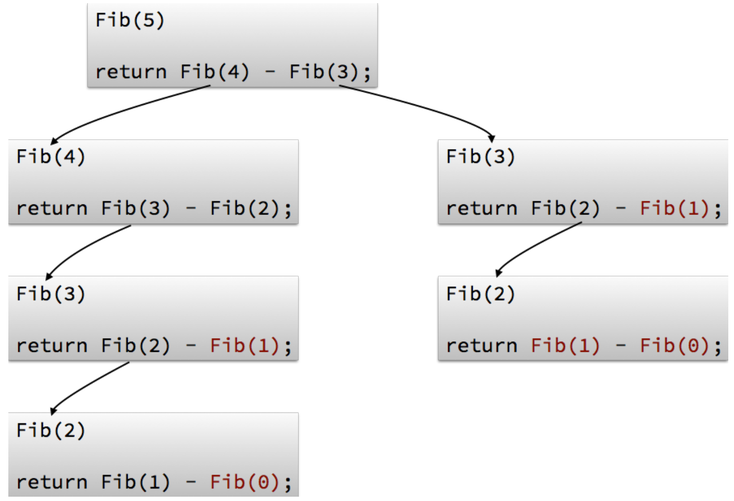
the recursive algorithm for fibonacci is very inefficient



Recursion

the recursive algorithm for fibonacci is very inefficient.
It is exponential in the argument.

The number of calls more than doubles every second k



k			nk
n0	1		
n1	1		
n2	n1 + n0 + 1	1+1+1	3
n3	n2 + n1 + 1	3 + 1 + 1	5
n4	n3 + n2 + 1	5 + 3 + 1	9
n5	n4 + n3 + 1	9 + 5 + 1	15

$$n_k \sim 2^{k/2}$$

Recursion

- recursive fibonacci:

- $O(2^n)$ because going from n to n-1 spawns 2 method calls, and each of these in turn spawns 2, ...

```
int fibonacci(int n) {
    if(n <= 1) {
        return n;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

List functions

- print a linked list
- currently use an iterative method

```
public String toString() {
  StringBuilder sb = new StringBuilder("(");
  Node<E> walk = head;
  while (walk != null) {
    sb.append(walk.getElement());
    if (walk != tail)
      sb.append(", ");
    walk = walk.getNext();
  }
  sb.append(")");
  return sb.toString();
}
```

List functions

- we would like a recursive print function:

```
printNode(Node node)
if node is null
    return
else
    System.out.prinlnt(node)
    printNode(l.next)
```

base

recursion

List functions

```
public void toString() {
  printNode(this.head);
}

private void printNode(Node n) {
  if (n == null) {
    return;
  }
  else {
    System.out.println(n.getElement());
    printNode(n.getNext());
  }
}
```

we start here: toString()

the linked list calls this internal printNode

List functions

Is this any easier than the iterative method?

How about reverse printing?

we start here: toString()

```
public void toString() {
  printNode(this.head);
}

private void printNode(Node n) {
  if (n == null) {
    return;
  }
  else {
    printNode(n.getNext());
    System.out.println(n.getElement());
  }
}
```

the linked list calls this internal printNode

printing come *after* the recursive call

List functions

- size of a list?

```
public int size() {
  Node n = head;
  int count = 0;
  while(n != null) {
    n = n.getNext();
    count += 1;
  }
  return count;
}
```

pointer hopping + counter

List functions

- size of a list?

```
public int size() {
  return size(head);
}
private int size(Node n) {
  if(n == null) {
    return 0;
  }
  return 1 + size(n.getNext());
}
```

counter is incremented on each recursive call

Recursion

Recursion is an important problem solving approach that is an alternative to iteration.

These are questions to answer when using recursive solution:

1. How can you define the problem in terms of a smaller problem of the same type?

2. How does each recursive call diminish the size of the problem?

3. What instance of the problem can serve as the base case?

4. As the problem size diminishes, will you reach this base case?

Recursion

In general, starting a recursive function of this form:

$$f(n) = \begin{cases} \text{output}_1 & \text{if } n \text{ in } \text{range}_1 \\ \dots & \\ \text{output}_2 & \text{if } n \text{ in } \text{range}_2 \end{cases}$$

gives rise to a recursive algorithm of this form:

```
output-type f(input-type n) {
    if (n in range1) {
        return output1;
    }
    //…
    if (n in rangek) {
        return outputk;
    }
}
```

Recursion or Iteration

Iterative control structures use a looping structure to repeat a set of statements.

while,

for,

or do...while,

There are usually two ways to solve a particular problem

iteration and recursion.

which is better?

Recursion or Iteration

- Another key factor in determining the best solution method is efficiency.
- Recall that whenever a function is called, memory space for its formal parameters and local variables is allocated.
- When the function terminates, that memory space is then deallocated.
- We also know that every (recursive) call has its own set of parameters and (automatic) local variables.

Recursion or Iteration

- Overhead involved with recursion:
  - Memory space
  - Time of execution
- Today's computers
  - fast
  - inexpensive memory
- The execution overhead of a recursion function is not (usually) noticeable.

Recursion or Iteration

Rule of thumb:

If an iterative solution is more obvious or easier to understand than a recursive solution: use the iterative solution, which would be more efficient.

When the recursive solution is more obvious or easier to construct … use recursion

Linear Recursion

- Linear recursion is by far the most common form of recursion.
- In this style of recursion, the function calls itself repeatedly until it hits the termination condition.
- After hitting the termination condition, it simply returns the result to the caller

```
public static int Factorial(int n)
{
    if (n <= 1)
        return 1;
    else
        return n * Factorial(n - 1);
}
```

Tail Recursion

- Tail recursion is a specialised form of the linear recursion where the last operation of the function happens to be a recursive call.
- Often, the value of the recursive call is returned.
- tail recursive functions can often be easily implemented in an iterative manner; by taking out the recursive call and replacing it with a loop

One of the earliest known numerical algorithms is that developed by Euclid in about 300 B.C. for computing the greatest common divisor (GCD) of two positive integers.

Let `GCD(x,y)` be the GCD of positive integers `x` and `y`. If `x = y`, then obviously

```
GCD(x,y) = GCD(x,x) = x
```

Euclid's insight was to observe that, if `x > y`, then

```
GCD(x,y) = GCD(x-y,y)
```

Tail Recursion

- Tail recursion is a specialised form of the linear recursion where the last operation of the function happens to be a recursive call.
- Often, the value of the recursive call is returned.
- tail recursive functions can often be easily implemented in an iterative manner; by taking out the recursive call and replacing it with a loop

```
// iterative GCD
  int gcd(int K, int M) {
    int k = K; // In order to state a simple, elegant loop invariant,
    int m = M; // we keep the formal arguments constant and use
               // local variables to do the calculations.
               // loop invariant: GCD(K,M) = GCD(k,m)
    while (k != 0 && m != 0) {
      if (k > m) {
        k = k % m;
      } else {
        m = m % k;
      }
    }
    // At this point, GCD(K,M) = GCD(k,m) = max(k,m)
    return Math.max(k, m);
  }
```

Tail Recursion

- Tail recursion is a specialised form of the linear recursion where the last operation of the function happens to be a recursive call.
- Often, the value of the recursive call is returned.
- tail recursive functions can often be easily implemented in an iterative manner; by taking out the recursive call and replacing it with a loop

```
1:  function GCD (n, m)
2:      Output: Greatest common divisor of integers n and m
3:      r ← 0
4:      if m < n then
5:          return GCD(m, n)
6:      end if
7:      r = n%m
8:      if r = 0 then
9:          return m
10:     end if
11:     return GCD(m, r)
12: end function
```

tail recursive GCD

Binary Recursion

- When an algorithm makes two recursive calls, we say that it uses ***binary recursion***.
- These calls can, for example, be used to solve two similar halves of some problem
- Fibonacci algorithm is an example of binary recursion
- (assignment will be to write a better version)

```
int fibonacci(int n) {
    if(n <= 1) {
        return n;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Multiple Recursion

- When an algorithm makes more than two recursive calls, we say that it uses ***multiple recursion***.

Nested Recursion

- different flavour to the other styles of recursion.
- All recursion can be converted to iterative (loop) except nested recursion.
- Take a look at the Ackermann function:

```
int Ackermann(int x, int y) {
    // Base or Termination Condition
    if (0 == x) {
      return y + 1;
    }
    // Error Handling condition
    if (x < 0 || y < 0) {
      return -1;
    }
    // Recursive call by Linear method
    else if (x > 0 && 0 == y) {
      return Ackermann(x - 1, 1);
    }
    // Recursive call by Nested method
    else {
      return Ackermann(x - 1, Ackermann(x, y - 1));
    }
}
```

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1,A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

$$A(4,3) = 2^{2^{65536}} - 3$$