Recursion

Data Structures
C++ for C Coders

한동대학교 김영섭교수
idebtor@gmail.com

Data Structures

Chapter 1

- ***algorithm specification***
  recursive algorithm
- *data abstraction*
- *performance analysis - time complexity*

Algorithm Specification

- Input
- Output
- Definiteness – clear and unambiguous
- Finiteness – it terminates after a finite number of steps
- Effectiveness – it is carried out and feasible

Algorithm Specification

- Input
- Output
- Definiteness – clear and unambiguous
- Finiteness – it terminates after a finite number of steps
- Effectiveness – it is carried out and feasible

- Ex. **program = algorithms + data structures**
  flowchart is not an algorithm.

Algorithm Specification

- Input
- Output
- Definiteness – clear and unambiguous
- Finiteness – it terminates after a finite number of steps
- Effectiveness – it is carried out and feasible

- Ex. **program = algorithms + data structures**
  flowchart is not an algorithm.

Recursion

- See recursion

Algorithm Specification

- Input
- Output
- Definiteness – clear and unambiguous
- Finiteness – it terminates after a finite number of steps
- Effectiveness – it is carried out and feasible

- Ex. **program = algorithms + data structures**
  flowchart is not an algorithm.

Recursion

- See recursion
- A process in which the result of each repetition is dependent upon the result of the next repetition
- Simplifies program structure at a cost of function calls

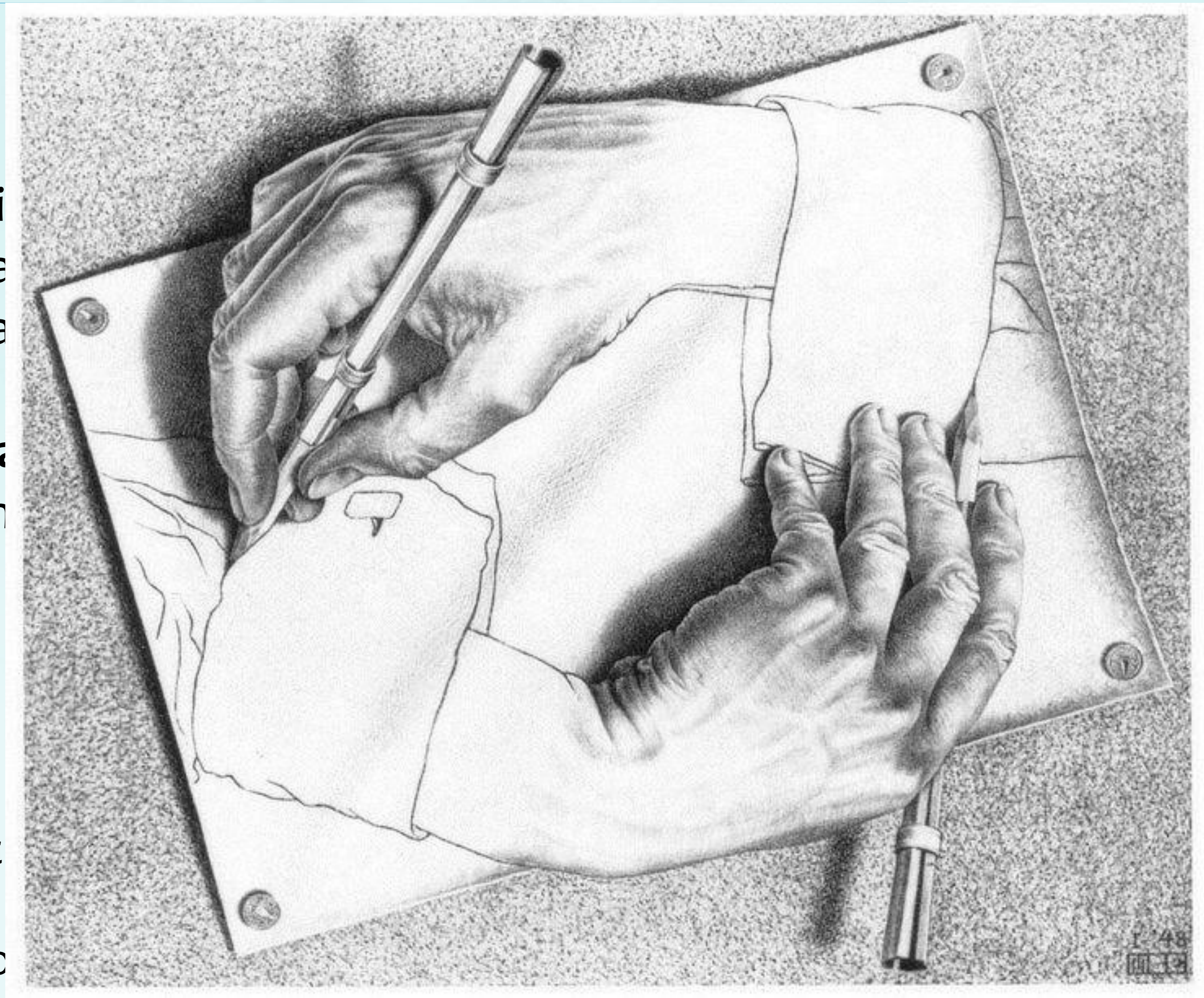
Algorithm Specification

- Input
- Output
- Definiteness – clear and unambi
- Finiteness – it terminates after a
- Effectiveness – it is carried out a

- Ex. **program = algorithms + data**
  flowchart is not an algorithm

Recursion

- See recursion
- A process in which the result of eac of the next repetition
- Simplifies program structure at a co



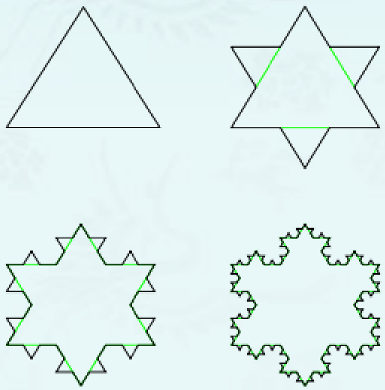
recursion is when a function *calls itself*

Recursive algorithms

Recursion is a method where the solution to a problem depends on solutions to smaller instances of the same problem (as opposed to iteration).

Recursive algorithm is expressed in terms of

1. **base case(s)** for which the solution can be stated **non-recursively**,



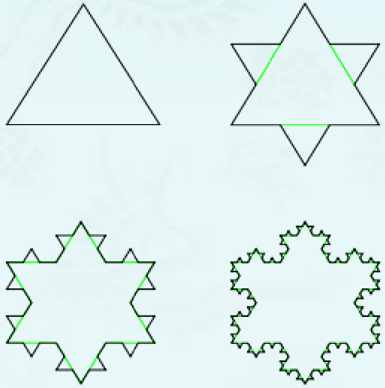
Four stages in the construction of a **Koch snowflake**. The stages are obtained via a recursive definition.

Recursive algorithms

Recursion is a method where the solution to a problem depends on solutions to smaller instances of the same problem (as opposed to iteration).

Recursive algorithm is expressed in terms of

1. **base case(s)** for which the solution can be stated **non-recursively**,
2. **recursive case(s)** for which the solution can be expressed in terms of a smaller version of itself.



Four stages in the construction of a **Koch snowflake**. The stages are obtained via a recursive definition.

Recursive algorithms

Example: Factorial

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

Recursive algorithms

Example: Factorial

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

factorial(n)
function factorial **input**: integer *n* such that *n* >= 0 **output**: [*n* × (*n*-1) × (*n*-2) × … × 1] 1. if *n* is 0, **return** 1 2. otherwise, **return** [*n* × factorial(*n*-1)] **end** factorial

Exercise: With four students, compute 4! using recursion.

Recursive algorithms

Example: Factorial

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

factorial(n)
function factorial **input**: integer *n* such that *n* >= 0 **output:** [*n* × (*n*-1) × (*n*-2) × … × 1] 1. if *n* is 0, **return** 1 2. otherwise, **return** [*n* × factorial(*n*-1)] **end** factorial

factorial (n = 4)
$f_4 = 4 * f_3$ $= 4 * (3 * f_2)$ $= 4 * (3 * (2 * f_1))$ $= 4 * (3 * (2 * (1 * f_0)))$ $= 4 * (3 * (2 * (1 * 1)))$ $= 4 * (3 * (2 * 1))$ $= 4 * (3 * 2)$ $= 4 * 6$ $= 24$

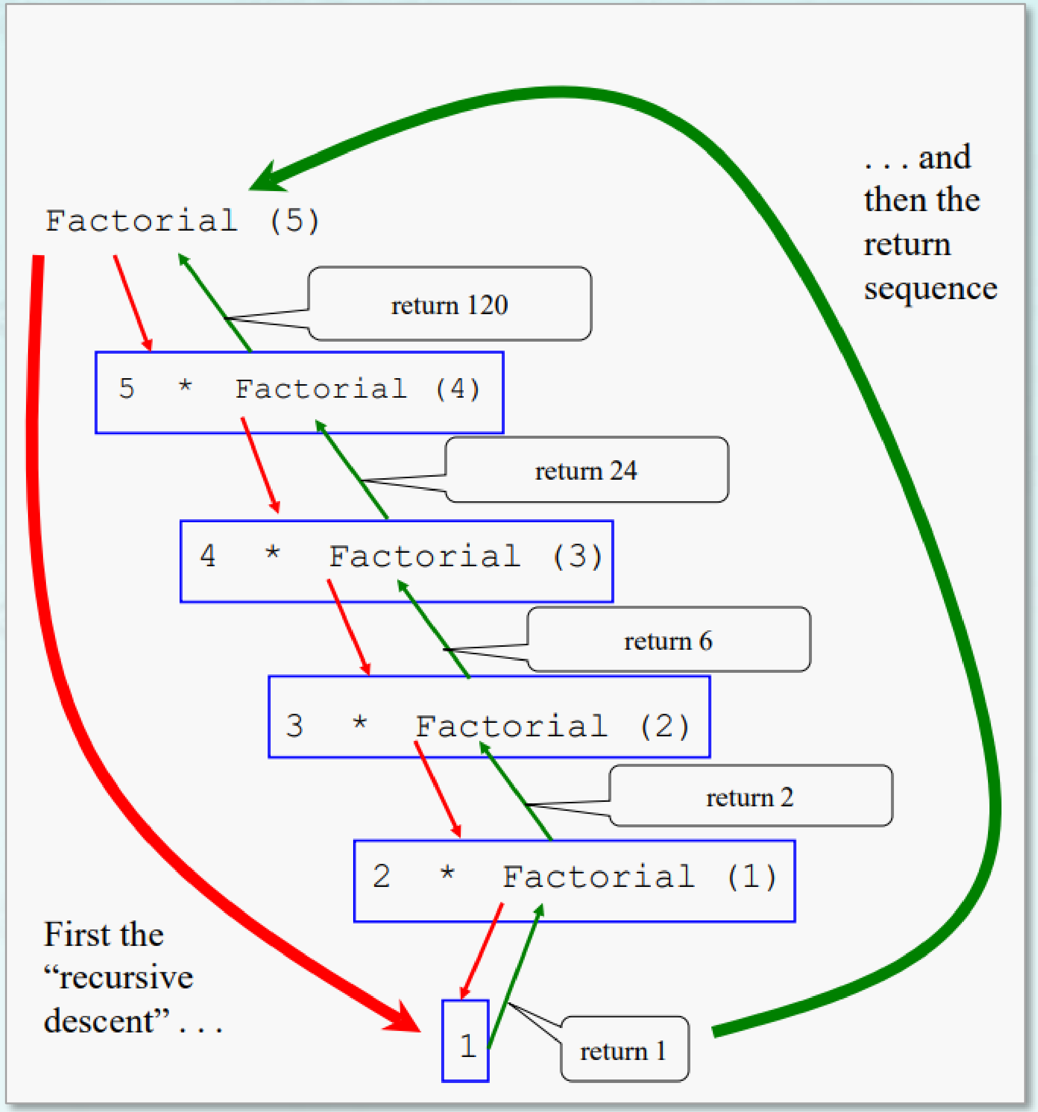
Exercise: With four students, compute 4! using recursion.

Recursive algorithms

Example: Factorial

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

factorial(n)
function factorial **input**: integer *n* such that *n* >= 0 **output:** [*n* × (*n*-1) × (*n*-2) × … × 1] 1. if *n* is 0, **return** 1 2. otherwise, **return** [*n* × factorial(*n*-1)] **end** factorial



Exercise: With four students, compute 4! using recursion.

Recursive algorithms

Example: Factorial

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

factorial(n)
function factorial **input**: integer *n* such that *n* >= 0 **output:** [*n* × (*n*-1) × (*n*-2) × … × 1] 1. if *n* is 0, **return** 1 2. otherwise, **return** [*n* × factorial(*n*-1)] **end** factorial

factorial (n = 4)
$f_4 = 4 * f_3$ $= 4 * (3 * f_2)$ $= 4 * (3 * (2 * f_1))$ $= 4 * (3 * (2 * (1 * f_0)))$ $= 4 * (3 * (2 * (1 * 1)))$ $= 4 * (3 * (2 * 1))$ $= 4 * (3 * 2)$ $= 4 * 6$ $= 24$

Exercise: **GCD** recursively with gcd (x=259, y=111) = ?

Recursive algorithms

Example: GCD (Great common divisor)

$$\gcd(x, y) = \begin{cases} x & \text{if } y = 0 \\ \gcd(y, \text{remainder}(x, y)) & \text{if } y > 0 \end{cases}$$

Recursive algorithms

Example: GCD (Great common divisor)

$$\gcd(x, y) = \begin{cases} x & \text{if } y = 0 \\ \gcd(y, \text{remainder}(x, y)) & \text{if } y > 0 \end{cases}$$

gcd(x, y)
function gcd **input**: integer *x*, *y* such that x >= y, y > 0 **output:** gcd of x and y 1. if y is 0, **return** x 2. otherwise, **return** [*gcd* (y, *x%y*)] **end** gcd

Recursive algorithms

Example: GCD (Great common divisor)

$$\gcd(x, y) = \begin{cases} x & \text{if } y = 0 \\ \gcd(y, \text{remainder}(x, y)) & \text{if } y > 0 \end{cases}$$

gcd(x, y)
function gcd **input**: integer *x*, *y* such that x >= y, y > 0 **output:** gcd of x and y 1. if y is 0, **return** x 2. otherwise, **return** [*gcd* (y, *x%y*)] **end** gcd

gcd (x=259, y=111)
gcd(259, 111) = gcd(111, 259% 111) = **gcd(111, 37)** = gcd(37, 111%37) **= gcd(37, 0)** = 37

Recursive algorithms

Example: GCD (Great common divisor)

$$\gcd(x,y) = \begin{cases} x & \text{if } y = 0 \\ \gcd(y, \text{remainder}(x,y)) & \text{if } y > 0 \end{cases}$$

gcd(x, y)
function gcd **input**: integer *x*, *y* such that x >= y, y > 0 **output:** gcd of x and y 1. if y is 0, **return** x 2. otherwise, **return** [*gcd* (y, *x%y*)] **end** gcd

gcd (x=259, y=111)
gcd(259, 111) = gcd(111, 259% 111) = **gcd(111, 37)** = gcd(37, 111%37) **= gcd(37, 0)** = 37

Exercises: gcd(91, 52)

Exercises: Fibonacci, Binomial coefficients(p.14), Akerman's function(p.17)

Recursive algorithms

Execution sequence of recursive functions:

Exercise: What is the output of the function (num=0)?

execution sequence

```
void  recursiveFunction(int num) {
      printf("%d\n", num);
      if (num < 4)
            recursiveFunction(num + 1);
}
```

Execution sequence of recursive functions:

Exercise: What is the output of the function (num=0)?

execution sequence

```
void  recursiveFunction(int num) {
      printf("%d\n", num);
      if (num < 4)
            recursiveFunction(num + 1);
}
```

1	recursiveFunction (0)				
2	printf (0)				
3		recursiveFunction (0+1)			
4		printf (1)			
5			recursiveFunction (1+1)		
6			printf (2)		
7				recursiveFunction (2+1)	
8				printf (3)	
9					recursiveFunction (3+1)
10					printf (4)

Recursive algorithms

Execution sequence of recursive functions:

Exercise: What is the output of the function (num=0)?

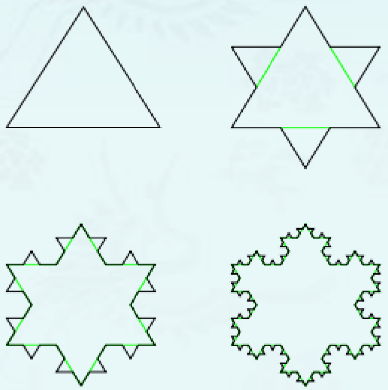
execution sequence

```
void  recursiveFunction(int num) {
      if (num < 4)
            recursiveFunction(num + 1);
      printf("%d\n", num);
}
```

1	recursiveFunction (0)				
2		recursiveFunction (0+1)			
3			recursiveFunction (1+1)		
4				recursiveFunction (2+1)	
5					recursiveFunction (3+1)
6					printf (4)
7				printf (3)	
8			printf (2)		
9		printf (1)			
10	printf (0)				

Recursive algorithms

Recursion is a method where the solution to a problem depends on solutions to smaller instances of the same problem (as opposed to iteration).



Four stages in the construction of a **Koch snowflake**. The stages are obtained via a recursive definition.

Recursive algorithms

Example: Recursive binary search

It searches a *sorted* array of **int**s for a particular **int**. Let **i** be an array of **int**s sorted from least to greatest. For instance, {-3, -2, 0, 0, 1, 5, 5}. We want to search **the array for the value** "wallly". If we find "wally", we return its array *index*; otherwise, we return FAILURE(-1).
Let's suppose "wally" is 1.

Recursive algorithms

Example: Recursive binary search

It searches a *sorted* array of **int**s for a particular **int**. Let **i** be an array of **int**s sorted from least to greatest. For instance, {-3, -2, 0, 0, 1, 5, 5}. We want to search **the array for the value** "wallly". If we find "wally", we return its array *index*; otherwise, we return FAILURE(-1).
Let's suppose "wally" is 1.

-3	-2	0	0	1	5	5

compare with 0

compare with 5

1	5	5

compare with 1

1

Exercise: Base case(s) & recursive case(s):?

int binarySearch(int list[], int wally, int **lo**, int **hi**)

Recursive algorithms

Example: Recursive binary search

Exercise: Base case(s) & recursive case(s):?

```
int binarySearch(int list[], int wally, int lo, int hi)

if (lo > hi) return -1;                        // base case

mid = (lo + hi)/2;
if (wally == list[mid]) return mid;                    // base case
if (wally < list[mid])
    return binarySearch(list, wally, lo, mid-1   ); // recursive
else
    return binarySearch(list, wally, mid+1, hi    ); // recursive
```

Recursive algorithms

Example: Recursive binary search

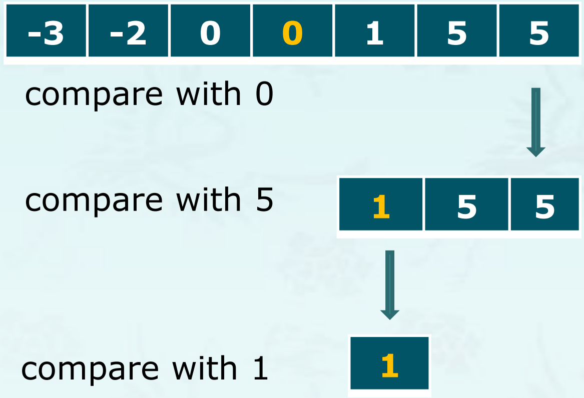
Exercise: Base case(s) & recursive case(s):?

```
int binarySearch(int list[], int wally, int lo, int hi)

if (lo > hi) return -1;                          // base case

mid = (lo + hi)/2;
if (wally == list[mid]) return mid;                    // base case
if (wally < list[mid])
    return binarySearch(list, wally, lo, mid-1   ); // recursive
else
    return binarySearch(list, wally, mid+1, hi   ); // recursive
```

How long does the binarySearch() take?
In one call to binarySearch(), we eliminate at least half the elements from consideration. Hence, it takes $\boldsymbol{log_2 n}$ (the base 2 logarithm of n) binarySearch() calls to pare down the possibilities to one. Therefore binarySearch takes time proportional to $\boldsymbol{log_2 n}$**.**

Recursive algorithms

Example: Recursive binary search – revisited



	Stack	Stack	Heap
binsearch()	lo[4] hi[4] mid[4]	wally[1] list[.]	
binsearch()	lo[4] hi[6] mid[5]	wally[1] list[.]	
binsearch()	lo[0] hi[6] mid[3]	wally[1] list[.]	
binsearch()	wally[1]	list[.]	[-3 -2 0 0 1 5 5]
main()		args[.]	args[]

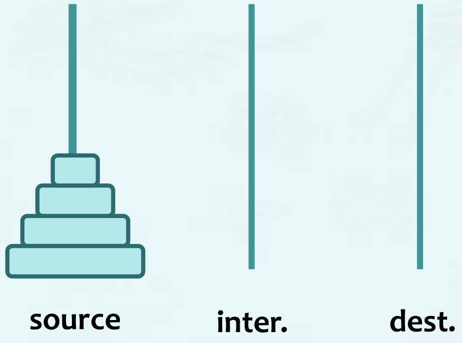
Most operating systems give a program enough stack space for a few thousand stack frames. If you use a recursive procedure to walk through a million-node list, the program will try to create a million stack frames, and **the stack will run out of space**. The result is a run-time error.

Example: Tower of Hanoi (Refer to p.17, Ex11)

Given three pegs, one with a set of N disks of increasing size, determine the minimum (optimal) number of steps it takes to move all the disks from their initial position to a single **stack** on another peg *without placing a larger disk on top of a smaller one. Only one disk can be moved at any time.*

Recursive algorithm:

(1) Move the top ***n-1*** *disks from* ***source*** *to* ***intermediate****.*
*(2) Move the remaining (****largest****) disk from* ***source*** *to* ***destination****.*
(3) Move the ***n-1*** *disks from* ***intermediate*** *to* ***destination****.*



Recursive algorithms

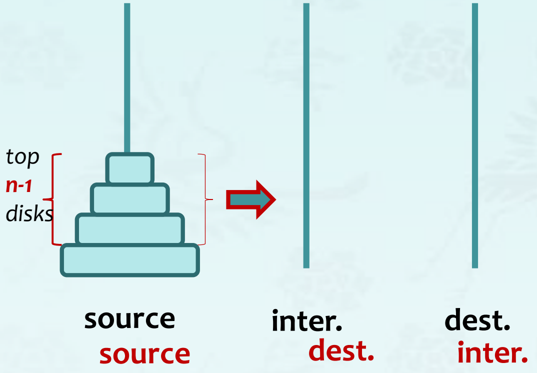
Example: Tower of Hanoi

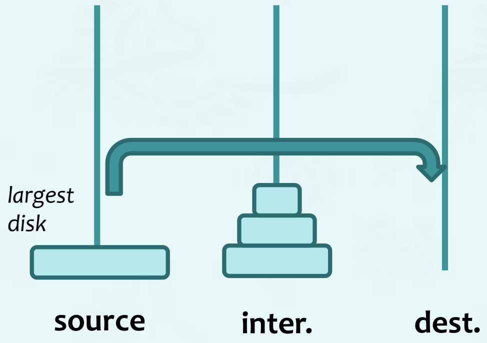
Recursive algorithm:

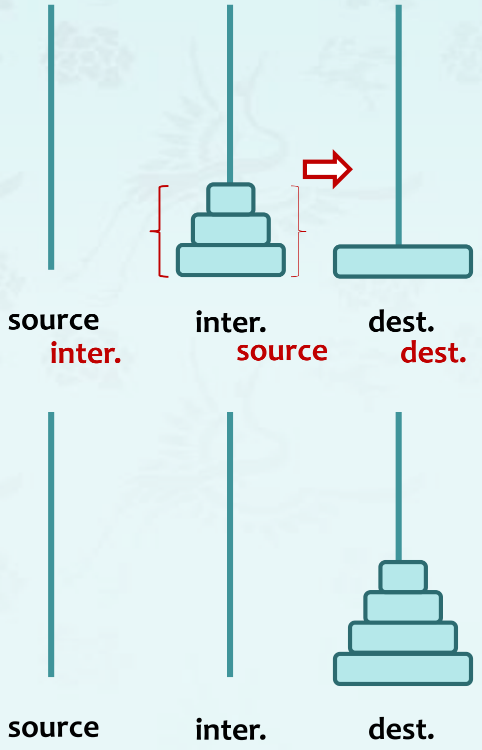
*(1) Move the top **n-1** disks from **source** to **intermediate**.*

*(2) Move the remaining (**largest**) disk from **source** to **destination**.*

*(3) Move the **n-1** disks from **intermediate** to **destination**.*







Recursive algorithms

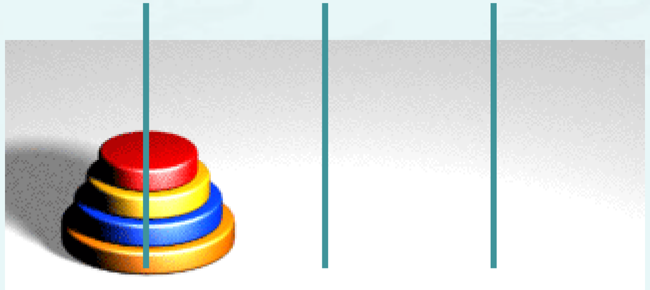
Example: Tower of Hanoi

Recursive algorithm:

*(1) Move the top **n-1** disks from **source** to **intermediate**.*

*(2) Move the remaining (**largest**) disk from **source** to **destination**.*

*(3) Move the **n-1** disks from **intermediate** to **destination**.*



source inter. dest.

Exercise: Tower of Hanoi – revisited

Recursive algorithm:

*(1) Move the top **n-1** disks from **source** to **intermediate**.*
*(2) Move the remaining (**largest**) disk from **source** to **destination**.*
*(3) Move the **n-1** disks from **intermediate** to **destination**.*

How do you program this to have the output as shown below?

Disk 1 from A to C
Disk 2 from A to B
Disk 1 from C to B
Disk 3 from A to C
Disk 1 from B to A
Disk 2 from B to C
Disk 1 from A to C

hanoi()

```
void hanoi(int n, char from, char inter, char to) {
    if (n == 1)
        printf ("Disk 1 from %c to %c\n", from, to);
    else {
        hanoi(n – 1  from, to, inter ) ;
        printf("Disk %d from %c to %c\n", n, from, to);
        hanoi(n – 1, inter, from, to );
    }
}
```

Recursive algorithms

Exercise: How many moves for n disks in Tower of Hanoi, hanoi(n)?

Recursive algorithm:

*(1) Move the top **n-1** disks from **source** to **intermediate**.*
*(2) Move the remaining (**largest**) disk from **source** to **destination**.*
*(3) Move the **n-1** disks from **intermediate** to **destination**.*

hanoi(*n-1*) move
hanoi(*1*) move
hanoi(*n-1*) move

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

Exercise:

hanoi(2) = 3
hanoi(4) = 15
hanoi(32) = 4,294,967,295
hanoi(64) = 18,446,744,073,709,600,000

hanoi(n = 4)
hanoi(4) = 2*hanoi(3) + 1 = 2*(2*hanoi(2) + 1) + 1 = 2*(2*(2*hanoi(1) + 1) + 1) + 1 = 2*(2*(2*1 + 1) + 1) + 1 = 2*(2*(3) + 1) + 1 = 2*(7) + 1 = 15

~ 584,942,417,355 years

How many years will take to move 64 disks? https://hanoi.aimary.com/index_en.php

Recursive algorithms

Q: Is the recursive version usually faster?

A: No -- it's usually slower (due to the overhead of maintaining the stack)

Q: Does the recursive version usually use less memory?

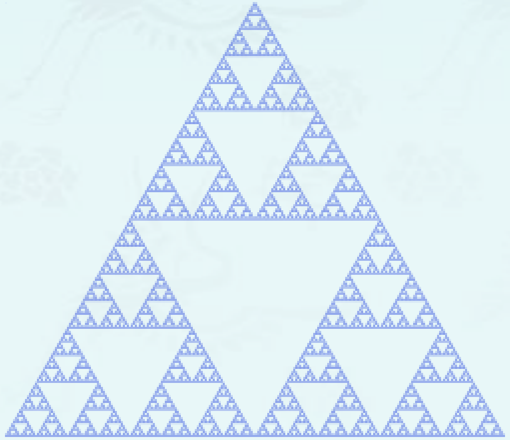
A: No -- it usually uses ***more*** memory (for the stack).

Q: Then ***why*** use recursion?

A: Sometimes it is much simpler to write the recursive version.

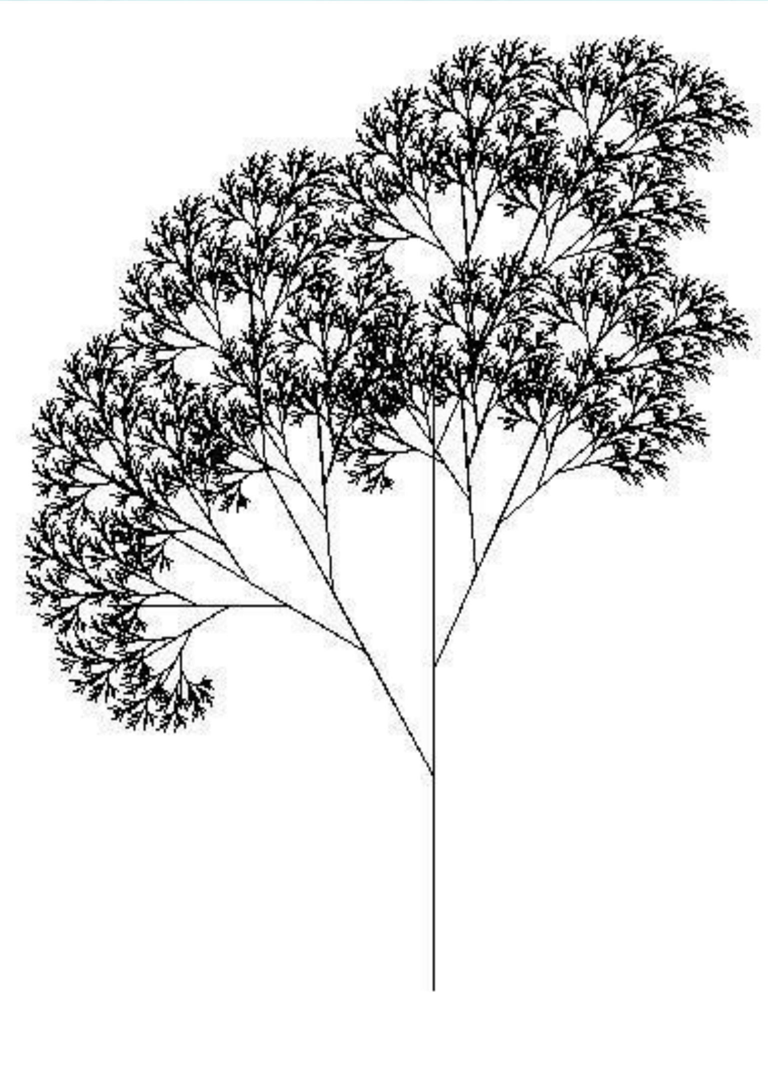
How the function call work? See[System Stack] in p.108.

*Because the recursive version causes an **activation record** to be pushed onto the system stack for every call, it is also more limited than the iterative version (it will fail, with a "stack overflow" error), for large values of N.*



Sierpinski Triangle:
a confined recursion of triangles to form a geometric lattice

Recursive algorithms



Recursion	*see Recursion*
GNU	GNU's not Unix.

Data Structures

Chapter 1

- *algorithm specification*
  recursive algorithm
- ***data abstraction***
- *performance analysis - time complexity*