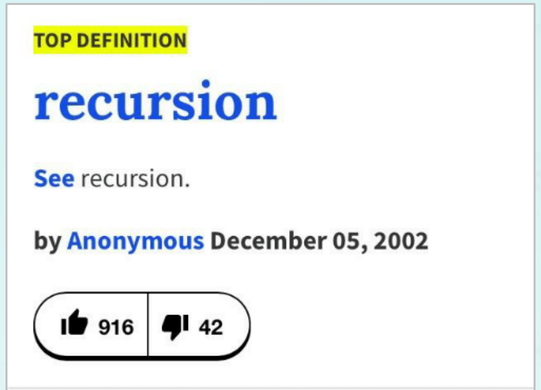
Data Structures
Chapter 1

1. **Recursion**
   - **Recursion**
   - Mergesort
2. Performance Analysis
3. Asymptotic Analysis

Prof. Youngsup Kim, idebtor@gmail.com, Data Structures, CSEE Dept., Handong Global University

Recursion

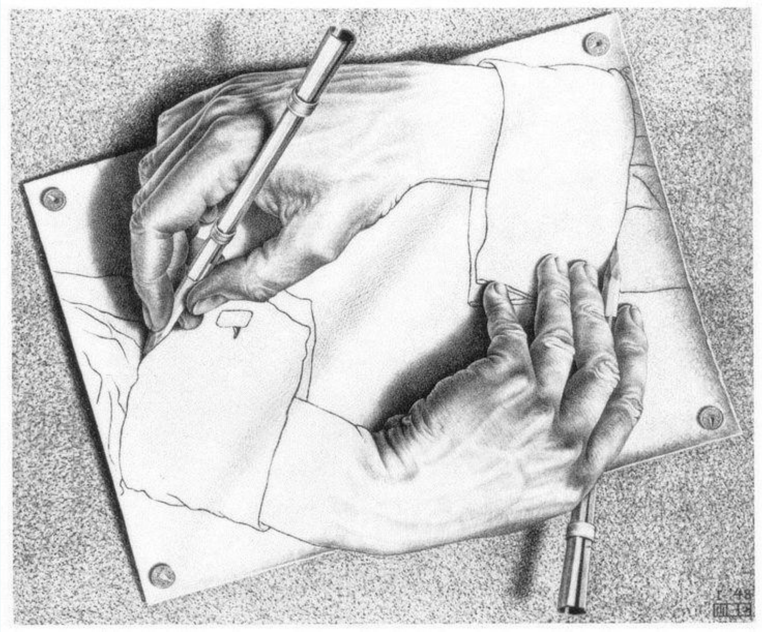
- See Recursion



Very descriptive definition

Recursion

- See Recursion
- Recursion is when a function calls itself
- Recursion simplifies program structure at a cost of function calls
- Recursion vs. Leap of faith



recursion **is**
when a function *calls itself*

Example 1: BunnyEars

- What is the output of the function `bunnyEars()`?
- What is the output of the function `main()`?
- How many times does two returns execute, respectively?

```
int bunnyEars(int n) {
    cout << n << endl;
    if (n > 0)
        return bunnyEars(n - 1) + 2;
    return 0;
}
```

```
int main() {
    cout << bunnyEars(4) << endl;
}
```

1	bunnyEars(4)
2	cout << 4
3	bunnyEars(3)
4	cout << 3
5	bunnyEars(2)
6	cout << 2
7	bunnyEars(1)
8	cout << 1
9	bunnyEars(0)
10	cout << 0 return 0

Example 1: BunnyEars

- What is the output of the function `bunnyEars()`?
- What is the output of the function `main()`?
- How many times does two returns execute, respectively?

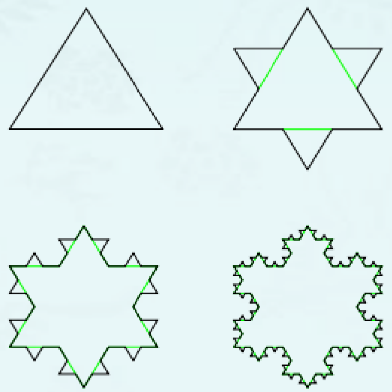
```
int bunnyEars(int n) {
    cout << n << endl;
    if (n > 0)
        return bunnyEars(n - 1) + 2;
    return 0;
}
```

```
int main() {
    cout << bunnyEars(4) << endl;
}
```

1	`bunnyEars(4)`	
2	`cout << 4`	`return f(3) + 2`
3	`bunnyEars(3)`	
4	`cout << 3`	`return f(2) + 2`
5	`bunnyEars(2)`	
6	`cout << 2`	`return f(1) + 2`
7	`bunnyEars(1)`	
8	`cout << 1`	`return f(0) + 2`
9	`bunnyEars(0)`	
10	`cout << 0`	

Recursion

- **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.

Example 3: Factorial

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

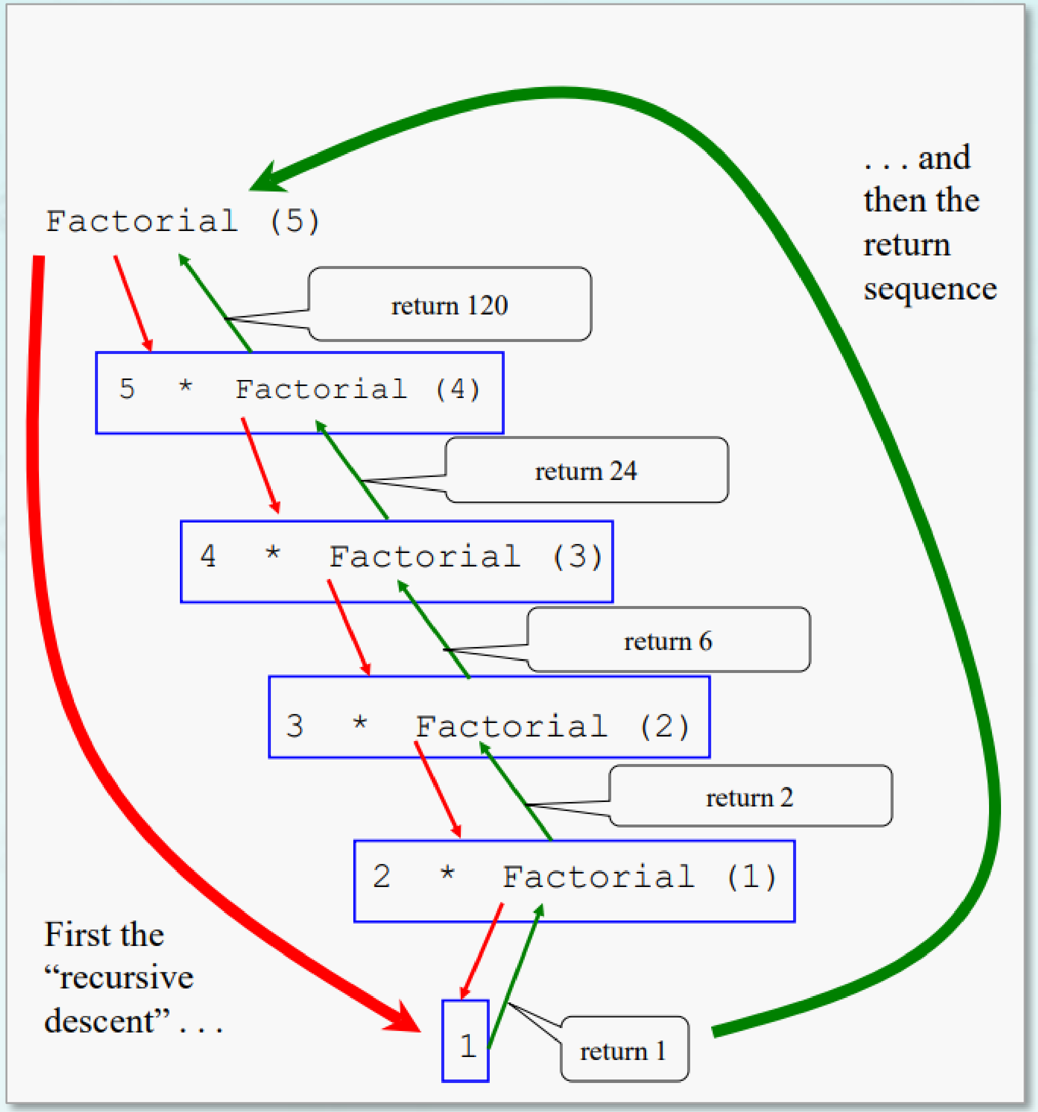
factorial(n)
function factorial
input: integer *n* such that *n* >= 0
output: [*n* × (*n*-1) × (*n*-2) × … × 1]
1. if *n* is 0, **return** 1
2. otherwise, **return** [*n* × factorial(*n*-1)]
end factorial

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

Example 3: 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



Example 4: GCD

- Compute **GCD** recursively with gcd (x=259, y=111) = ?

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

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

- Exercise: gcd(91, 52)

Reminder

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

Example 5: Recursive binary search

- Binary search is an efficient algorithm for finding an item from **a sorted list** of items.
  - It works by repeatedly **dividing in half the portion of the list** that could contain the item,
  - until you've **narrowed down the possible locations to just one.**

key=23

0	1	2	3	4	5	6	7	8	9
2	5	8	9	16	23	31	56	62	71

Example 5: Recursive binary search

- For instance, we want to search "23" from the array. If we find it, we return its array index; otherwise, -1 or something else.

	0	1	2	3	4	5	6	7	8	9
key=23	2	5	8	9	16	23	31	56	62	71

	lo=0	1	2	3	mi=4	5	6	7	8	hi=9
key>mi 23>16	2	5	8	9	16	23	31	56	62	71

	0	1	2	3	4	lo=5	6	mi=7	8	hi=9
key<mi 23<56	2	5	8	9	16	23	31	56	62	71

	0	1	2	3	4	mi=5 lo=5	hi=6	7	8	9
key=mi 23=23	2	5	8	9	16	23	31	56	62	71

```
int binarySearch(int list[], int key,
                              int lo, int hi) {
  if (lo > hi) return -1;

  mi = (lo + hi)/2;
  if (key == list[mi]) return mi;
  if (key <  list[mi])
    return binarySearch(list,                 );
  else
    return binarySearch(list,                 );
}
```

Example 5: Recursive binary search

- For instance, we want to search "23" from the array. If we find it, we return its array index; otherwise, -1 or something else.

key=23

0	1	2	3	4	5	6	7	8	9
2	5	8	9	16	23	31	56	62	71

key>mi
23>16

lo=0	1	2	3	mi=4	5	6	7	8	hi=9
2	5	8	9	16	23	31	56	62	71

key<mi
23<56

0	1	2	3	4	lo=5	6	mi=7	8	hi=9
2	5	8	9	16	23	31	56	62	71

key=mi
23=23

0	1	2	3	4	mi=5 lo=5	hi=6	7	8	9
2	5	8	9	16	23	31	56	62	71

```
int binarySearch(int list[], int key,
                               int lo, int hi) {
  if (lo > hi) return -1;

  mi = (lo + hi)/2;
  if (key == list[mi]) return mi;
  if (key <  list[mi])
    return binarySearch(list, key, lo, mi - 1);
  else
    return binarySearch(list, key, mi + 1, hi);
}
```

Example 5: Recursive binary search

- How many times is the `binarySearch()` called in terms of n?
- In one call to `binarySearch()`, we eliminate at least half the elements from consideration. Hence, it takes $\log_2 n$ (the base 2 logarithm of n) `binarySearch()` calls to compare down the possibilities to one. Therefore binarySearch takes time proportional to $\log_2 n$.

```
int binarySearch(int list[], int key, int lo, int hi) {
  if (lo > hi) return -1;

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

Example 6: Recursive binary search 100

- Given the numbers 1 to 100, what is the minimum number of guesses needed to find a specific number if you are given the hint 'higher' or 'lower' for each guess you make?
  - Since the numbers are sequential (or sorted), we can use **binary search**.
  - Look at the middle element: if it's after than the number we're looking for, search the first half. If it's before the number we're looking for, look at the second half.
  - Each check cuts the size of the list numbers in half; how many times can we do this?
  - If we think backwards, in terms of doubling the list, we'll need n doublings to generate a list of length 2^n = 100. What is the value of n?
  - Since $2^6 = 64$ and $2^7 = 128$ (or $log_2 64 = 6,\ \ log_2 128 = 7$), *n* = 6.x
    Therefore n = 7 guesses will be enough.

Example 6: Recursive binary search 1000

- Given the numbers 1 to 1000, what is the minimum number of guesses needed to find a specific number if you are given the hint ‘higher’ or ‘lower’ for each guess you make?
  - For an array whose length is 1000, the closest lower power of 2 is 512, which is 2^9.
  - We can thus estimate that $log_2 1000$ is a number greater than 9 and less than 10, or use a calculator to see that its about 9.97. Adding one to that yields about 10.97.
  - In the case of a decimal number, we round down to find the actual number of guesses.
  - Therefore, for a 1000-element array, binary search would require at most **10 guesses.**

Reference: https://www.geeksforgeeks.org/complexity-analysis-of-binary-search/

Example 5: Recursive binary search

0	1	2	3	4	5	6	7	8	9
2	5	8	9	16	23	31	56	62	71

lo=0	1	2	3	mi=4	5	6	7	8	hi=9
2	5	8	9	16	23	31	56	62	71

0	1	2	3	4	lo=5	6	mi=7	8	hi=9
2	5	8	9	16	23	31	56	62	71

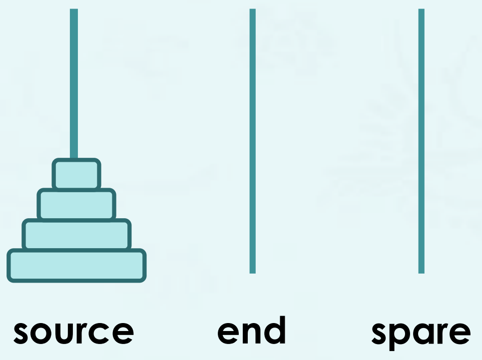
0	1	2	3	4	mi=5 lo=5	hi=6	7	8	9
2	5	8	9	16	23	31	56	62	71

	Stack	Stack	Heap
search()	lo=5 hi=6 mi=5	key=23 list[.]	
search()	lo=5 hi=9 mi=7	key=23 list[.]	
search()	lo=0 hi=9 mi=4	key=23 list[.]	
search()	key=23	list[.]	[2 5 8 9 16 23 31 56 62 71]
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.

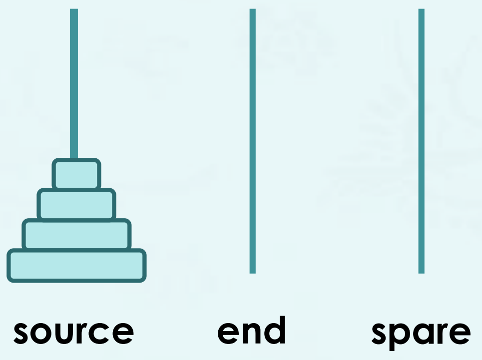
Example 6: Tower of Hanoi

- 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.*



Example 6: Tower of Hanoi

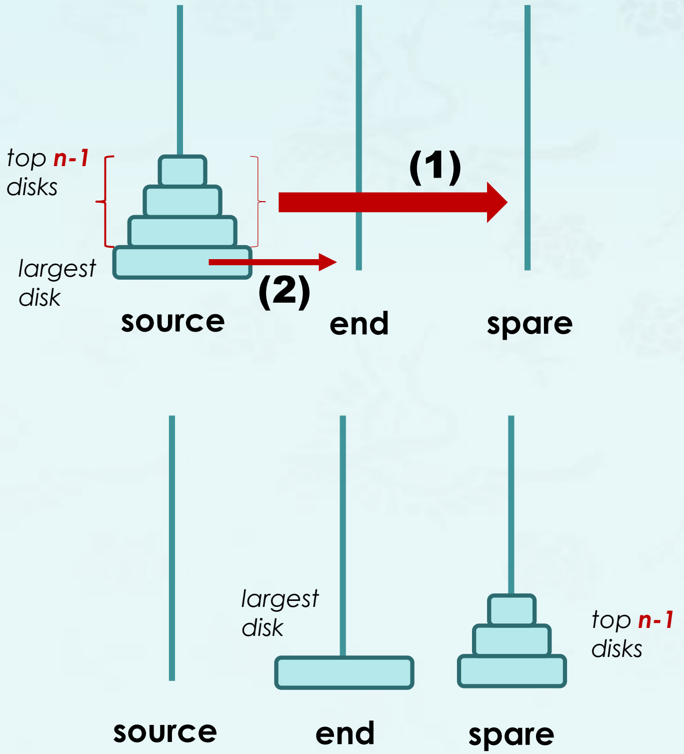
- Given three pegs, one with a set of N disks of increasing size, determine the minimum (optimal) number of steps it takes to move all the disks from their initial position to a single stack on another peg *without placing a larger disk on top of a smaller one. Only one disk can be moved at any time.*
- **Recursive algorithm:**
  (1) *Move the top **n-1** disks from **source** to **spare**.*
  (2) *Move the remaining (**largest**) disk from **source** to **end.***
  (3) *Move the **n-1** disks from **spare** to **end.***



Example 6: Tower of Hanoi

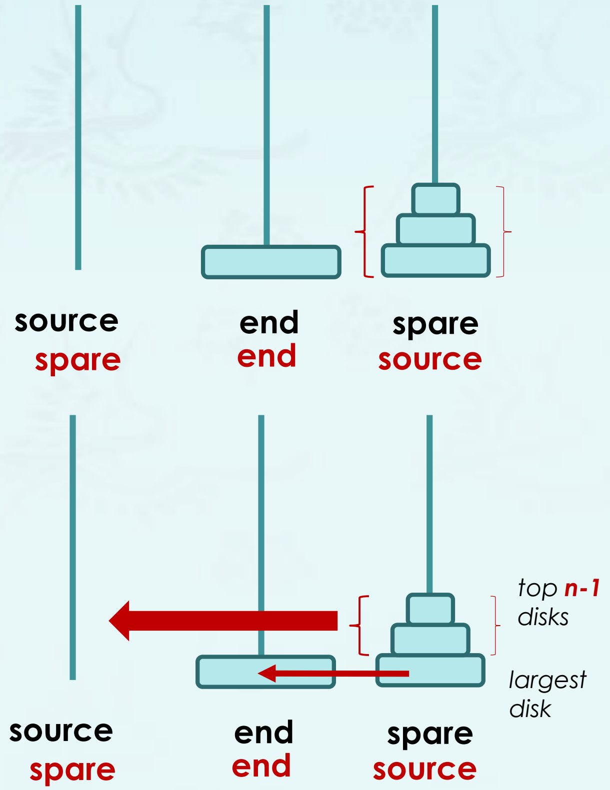
- **Recursive algorithm:**
  - *(1) Move the top **n-1** disks from **source** to **spare**.*
  - *(2) Move the remaining (**largest**) disk from **source** to **end.***
  - *(3) Move the **n-1** disks from **spare** to **end.***

4 disk problem



(3) It becomes a **3 disk problem.**
Go back to step 1.
Treat the **spare as source** and
the **source as spare**.

3 disk problem



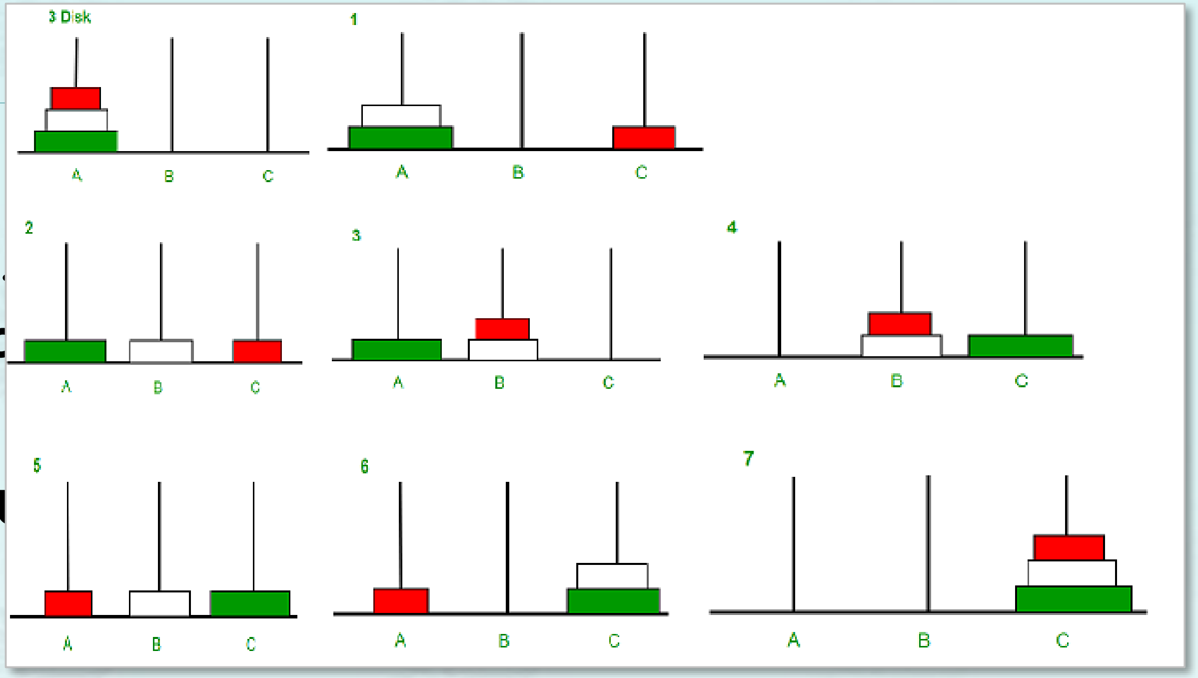
Example 6: Tower of Hanoi

- **Recursive algorithm:**
  (1) *Move the top **n-1** disks from **source** to **spare**.*
  (2) *Move the remaining (**largest**) disk from **source** to **end.***
  (3) *Move the **n-1** disks from **spare** to **end.***
- **How do you program this to have the output as shown below?**
  (1) Disk 1 from A to C
  (2) Disk 2 from A to B
  (3) Disk 1 from C to B
  (4) Disk 3 from A to C
  (5) Disk 1 from B to A
  (6) Disk 2 from B to C
  (7) Disk 1 from A to C

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

Example 6: Tower of Hanoi

- **Recursive algorithm:**
  - (1) *Move the top **n-1** disks from **source** to **spare**.*
  - (2) *Move the remaining (**largest**) disk from **sourc***
  - (3) *Move the **n-1** disks from **spare** to **end**.*
- **How do you program this to have the ou**
  - *(1) Disk 1 from A to C*
  - *(2) Disk 2 from A to B*
  - *(3) Disk 1 from C to B*
  - *(4) Disk 3 from A to C*
  - *(5) Disk 1 from B to A*
  - *(6) Disk 2 from B to C*
  - *(7) Disk 1 from A to C*




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

Example 6: Time complexity of Tower of Hanoi - $O(2^n)$

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

Exercise: How many years will it take to move 64 disks while assuming one move per pico second at the clock speed of a super fast machine?

(1) hanoi(1) = 1
(2) hanoi(2) = 3
(3) hanoi(3) = 7
(4) hanoi(4) = 15
(5) hanoi(5) = 31
(6) hanoi(32) = 4,294,967,295
(7) 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

Recursion

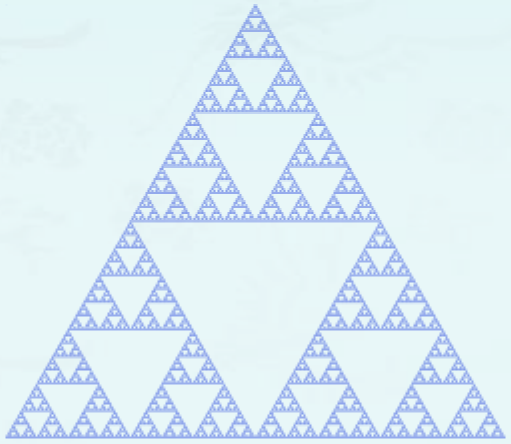
Q: Is the recursive version usually **faster?**

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.

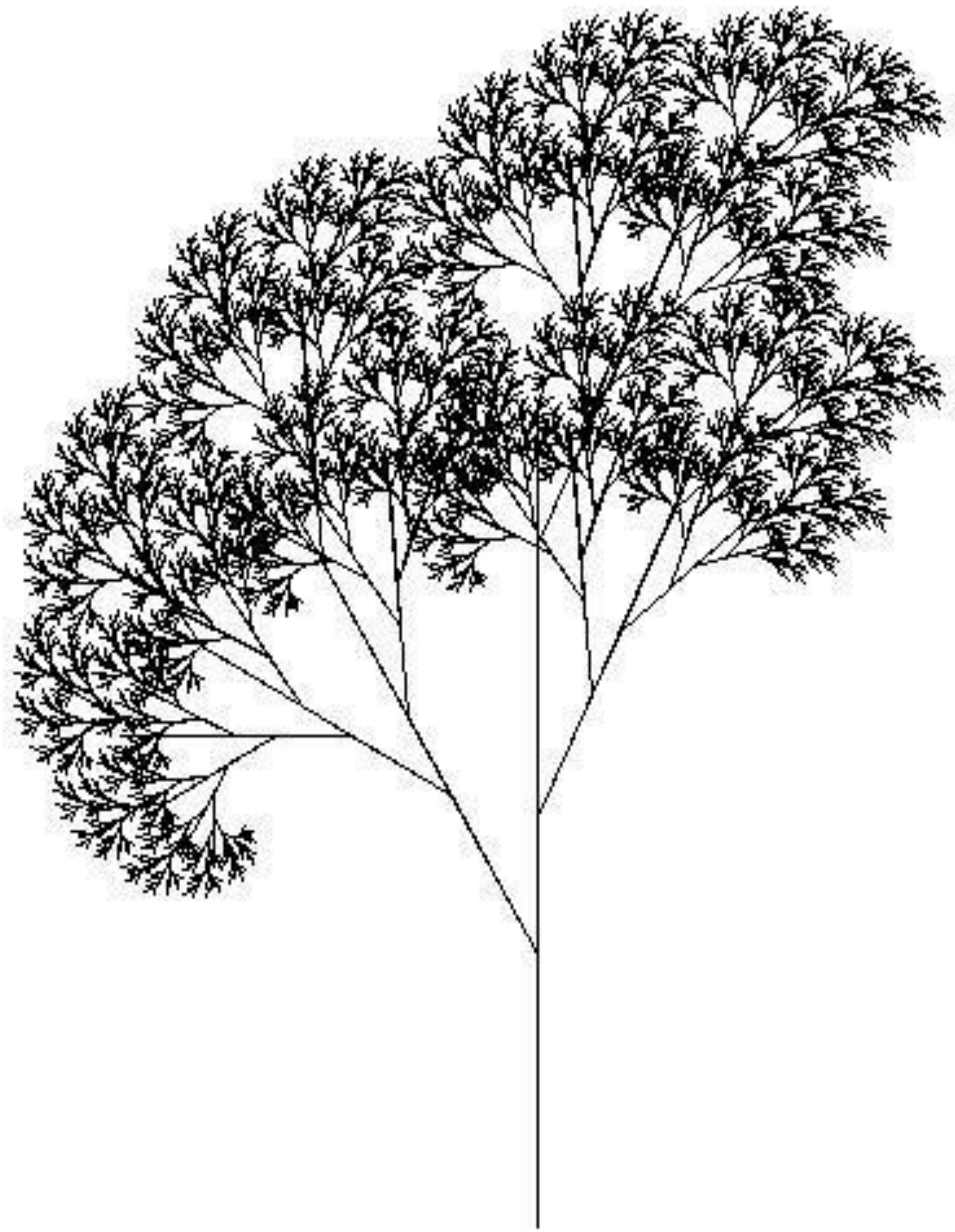
Because the recursive version causes an ***activation record*** *to be pushed onto the system stack for every call, it is also more limited than the iterative version (it will fail, with a "stack overflow" error), for large values of N.*



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

Recursion

Recursion *see Recursion*



Data Structures
Chapter 1

1. **Recursion**
   - **Recursion**
   - Mergesort
2. Performance Analysis
3. Asymptotic Analysis