Recursive Algorithms

Recursion

Is when a function makes **a call to itself**.

We *keep defining* the problem as simpler versions of itself, until we get to the **base case**.

It works only if:

+ There is a **base case** defined.

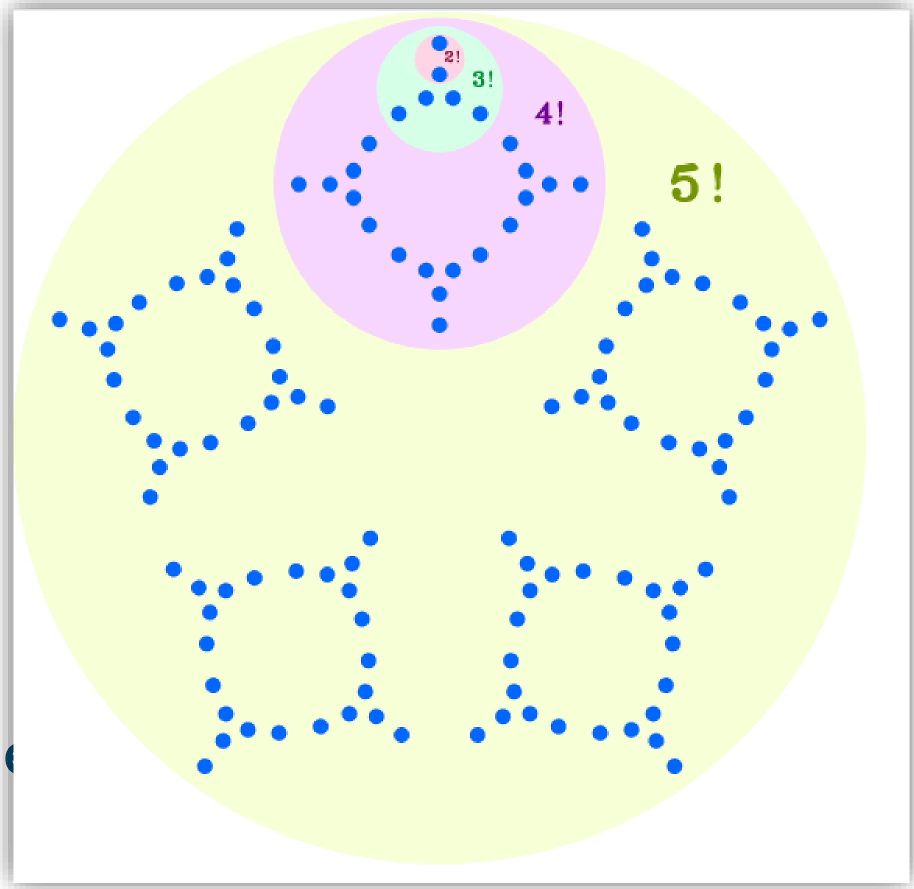
+ Parameters to the function on successive calls **change** (towards the base case).

Recursive algorithms: a classic example

- The **factorial** function

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$$

```
FACTORIAL(n)
  if n = 1
    return 1
  else
    return n * FACTORIAL(n-1)
```



https://freesvg.org/factorial

- The above is a **recursive algorithm** implementation of the factorial
  - *Repeated* calls to FACTORIAL( ), each time applied on a smaller number, are nested
  - Until a stopping case ($n = 1$) is reached

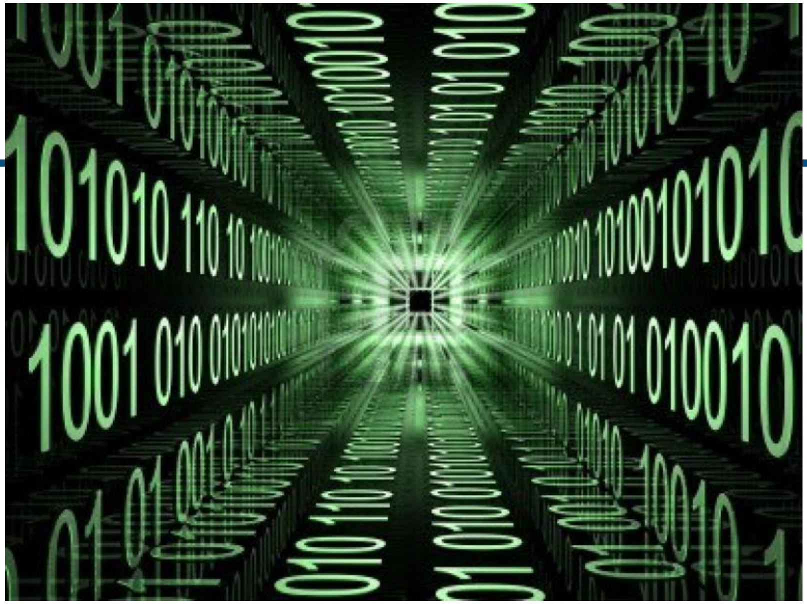
Recursive algorithms: Binary Search Trees

```
INORDER(node)
  if node != NIL
    INORDER(node.left)
    print node.key
    INORDER(node.right)
```

```
SEARCH(node,key)
  if node==NIL or key==node.key
    return node
  if key < node.key
    return SEARCH(node.left,key)
  else
    return SEARCH(node.right,key)
```

- **Recall:**
  - BSTs are essentially *recursive* data structures
- **So,**
  - The algorithms for operations on them can be defined recursively in a natural way
  - However, recall that alternative *iterative* implementations exist (and can be more efficient!)

Finally, some real code!



- Let's looks at factorial code in Python

(moodle chapter for this topic)

General principles for recursive algorithms

When calling itself, a recursive function makes a <u>clone</u> and calls the clone with appropriate parameters

A recursive algorithm must always

- **Rule 1**: reduce the size of the input, each time it is recursively called
- **Rule 2**: provide a stopping case (terminating condition)

When calling itself, a recursive function makes a clone and calls the clone with appropriate parameters

A recursive algorithm must always

- **Rule 1**: reduce size of data set, or the number its working on, each time it is recursively called
- **Rule 2**: provide a stopping case (terminating condition)

```
FACT(n)
  if n = 1
    return 1
  else
    return n * FACT(n-1)
```

```
SEARCH(node,key)
  if node==NIL or key==node.key
    return node
  if key < node.key
    return SEARCH(node.left,key)
  else
    return SEARCH(node.right,key)
```

```
INORDER(node)
  if node != NIL
    INORDER(node.left)
    print node.key
    INORDER(node.right)
```

Writing Recursive Functions – Tips

- Write the base case first

- For the rest of the cases, work out how to "reduce" them to simpler cases
  - Arguments/input should "converge" towards the base case

- Think/visualize in terms of small problem sizes (e.g. FACTORIAL(3))
  - Confirm that the base case works
  - Confirm that these specific, small non-base-case cases work
  - "Convince" yourself that generalizing to arbitrary input sizes will work!

```
def factorial(x):
    if x==1:
        return 1    # stop, it's 1
    else:
        return x * factorial(x-1)
```

Review this section from the book: http://openbookproject.net/thinkcs/python/english3e/recursion.html

Recursion Traces

Recursion trace

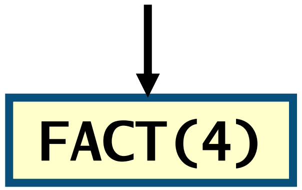
- Graphical method to visualise the execution of recursive algorithms
- Drawn as follows:
  - A box for each recursive call
  - An arrow from each caller to callee (in black)
  - An arrow from each callee to caller showing return value (in blue) (we will often omit this)
- Example with FACT(4)

```
FACT(n)
  if n = 1
    return 1
  else
    return n * FACT(n-1)
```

Recursion trace

- Graphical method to visualise the execution of recursive algorithms
- Drawn as follows:
  - A box for each recursive call
  - An arrow from each caller to callee (in black)
  - An arrow from each callee to caller showing return value (in blue) (we will often omit this)
- Example with FACT(4)

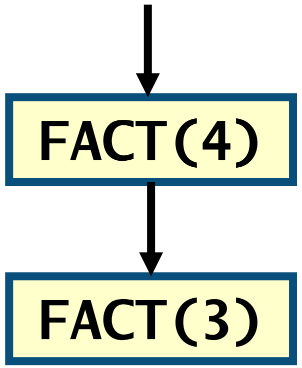
```
FACT(n)
  if n = 1
    return 1
  else
    return n * FACT(n-1)
```



Recursion trace

- Graphical method to visualise the execution of recursive algorithms
- Drawn as follows:
  - A box for each recursive call
  - An arrow from each caller to callee (in black)
  - An arrow from each callee to caller showing return value (in blue) (we will often omit this)
- Example with FACT(4)

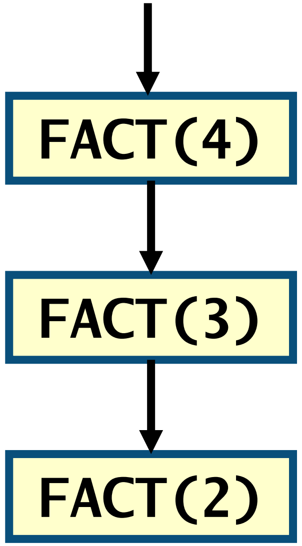
```
FACT(n)
  if n = 1
    return 1
  else
    return n * FACT(n-1)
```



Recursion trace

- Graphical method to visualise the execution of recursive algorithms
- Drawn as follows:
  - A box for each recursive call
  - An arrow from each caller to callee (in black)
  - An arrow from each callee to caller showing return value (in blue) (we will often omit this)
- Example with FACT(4)

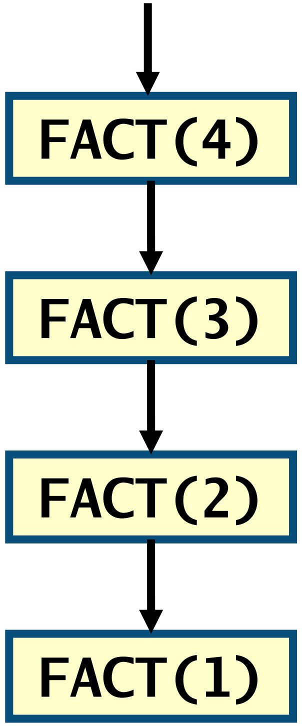
```
FACT(n)
  if n = 1
    return 1
  else
    return n * FACT(n-1)
```



Recursion trace

- Graphical method to visualise the execution of recursive algorithms
- Drawn as follows:
  - A box for each recursive call
  - An arrow from each caller to callee (in black)
  - An arrow from each callee to caller showing return value (in blue) (we will often omit this)
- Example with FACT(4)

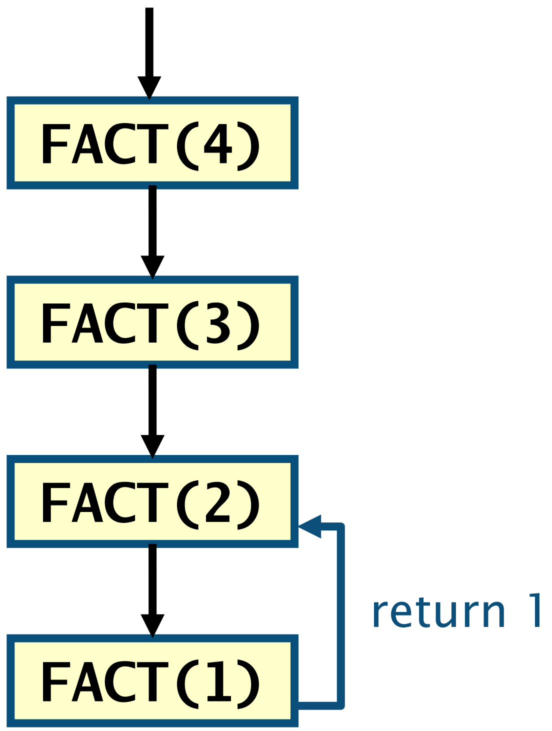
```
FACT(n)
  if n = 1
    return 1
  else
    return n * FACT(n-1)
```



Recursion trace

- Graphical method to visualise the execution of recursive algorithms
- Drawn as follows:
  - A box for each recursive call
  - An arrow from each caller to callee (in black)
  - An arrow from each callee to caller showing return value (in blue) (we will often omit this)
- Example with FACT(4)

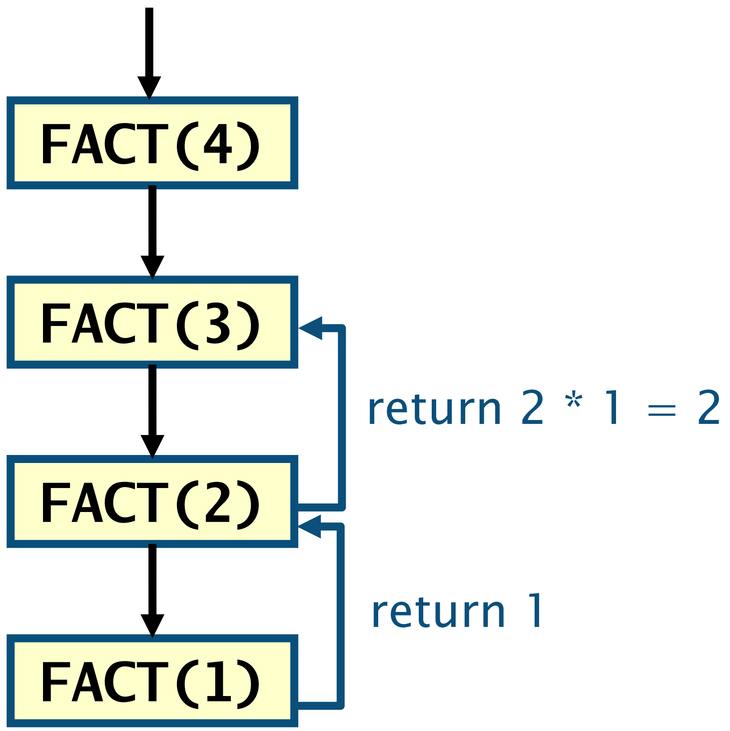
```
FACT(n)
  if n = 1
    return 1
  else
    return n * FACT(n-1)
```



Recursion trace

- Graphical method to visualise the execution of recursive algorithms
- Drawn as follows:
  - A box for each recursive call
  - An arrow from each caller to callee (in black)
  - An arrow from each callee to caller showing return value (in blue) (we will often omit this)
- Example with FACT(4)

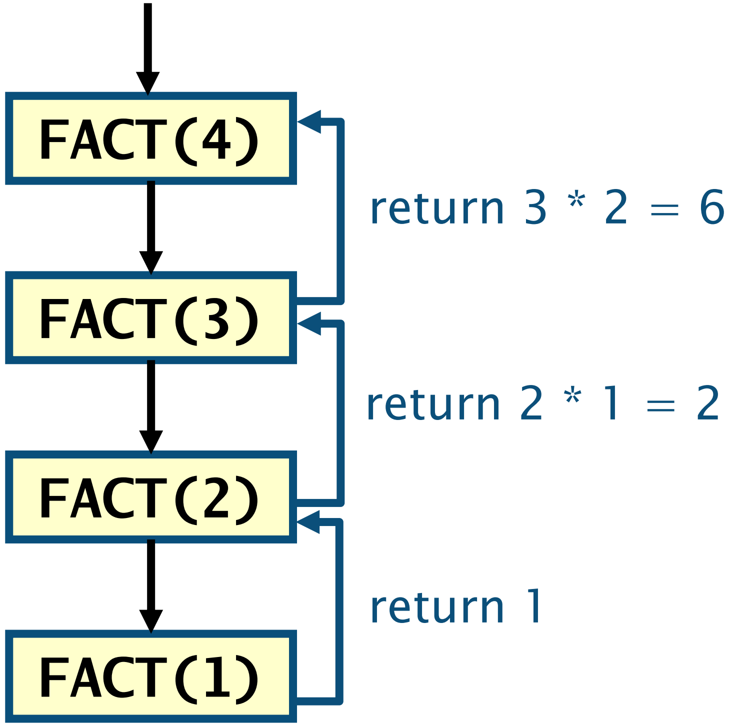
```
FACT(n)
  if n = 1
    return 1
  else
    return n * FACT(n-1)
```



Recursion trace

- Graphical method to visualise the execution of recursive algorithms
- Drawn as follows:
  - A box for each recursive call
  - An arrow from each caller to callee (in black)
  - An arrow from each callee to caller showing return value (in blue) (we will often omit this)
- Example with FACT(4)

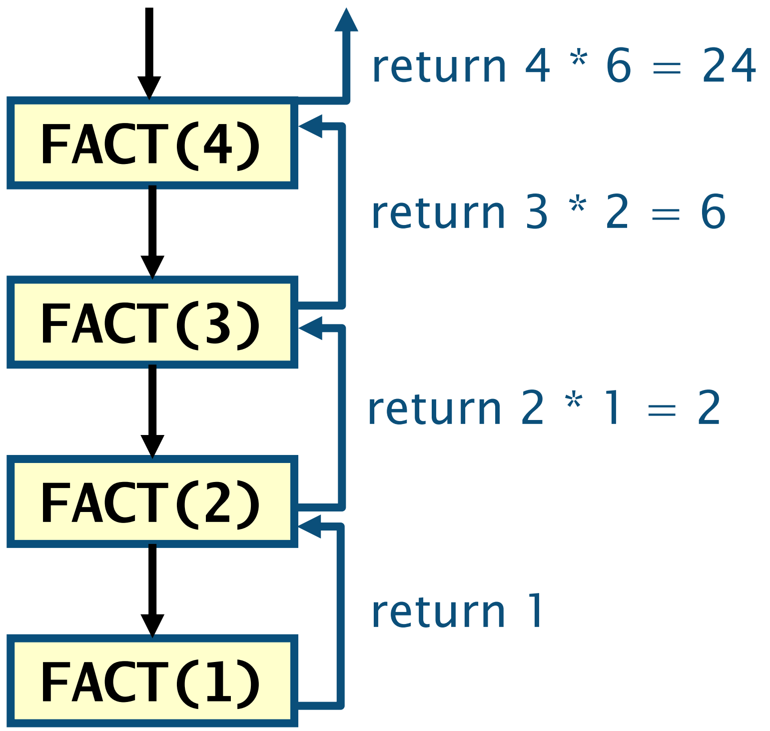
```
FACT(n)
  if n = 1
    return 1
  else
    return n * FACT(n-1)
```



Recursion trace

- Graphical method to visualise the execution of recursive algorithms
- Drawn as follows:
  - A box for each recursive call
  - An arrow from each caller to callee (in black)
  - An arrow from each callee to caller showing return value (in blue) (we will often omit this)
- Example with FACT(4)

```
FACT(n)
  if n = 1
    return 1
  else
    return n * FACT(n-1)
```



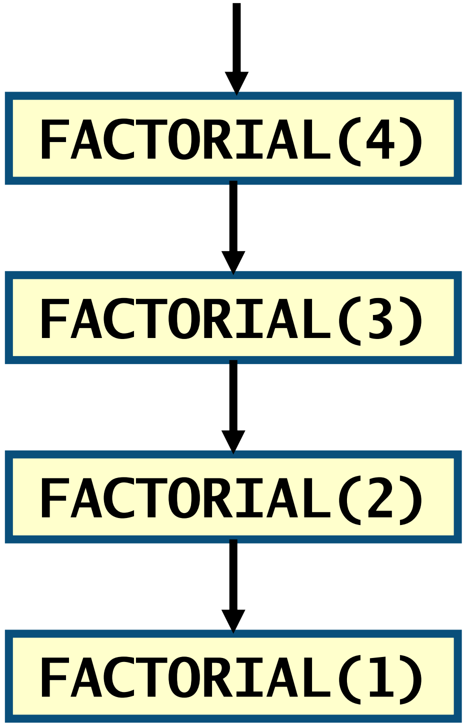
LINEAR AND BINARY RECURSION

Linear recursion

- At most **one** recursive call at each invocation

- The amount of space needed, to keep track of all nested calls, **grows linearly** (wrt the size of the input)
  - The recursion trace is a visualization of *space requirements*

- Example: FACTORIAL(n)
  – A <u>single</u> recursive call to FACTORIAL(n-1)

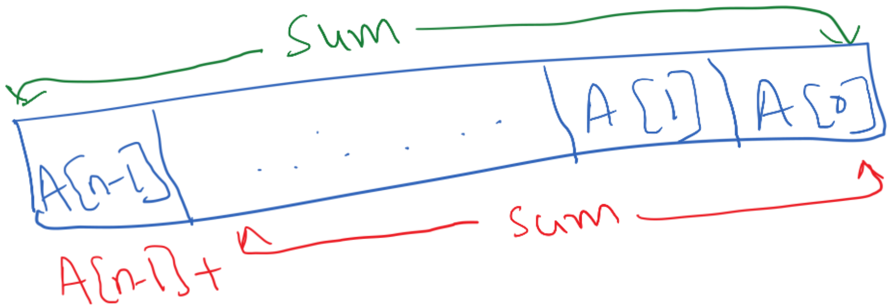



```
FACTORIAL(n)
  if n = 1
    return 1
  else
    return n * FACTORIAL(n-1)
```

Another example: summing the elements of an array

- Input: An array A of integers and integer $n \geq 1$, such that A has at least n elements
- Output: The sum of the first n integers in A

```
LINEAR-SUM(A,n)
```



Example: sum of array elements

- Input: An array **A** of integers and integer **n** ≥ 1, such that **A** has at least **n** elements
- Output: The sum of the first **n** integers in **A**

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

- Does LINEAR-SUM satisfy our "recursion rules"?
  - Rule 1: input reduced at each recursive call
  - Rule 2: valid terminating condition

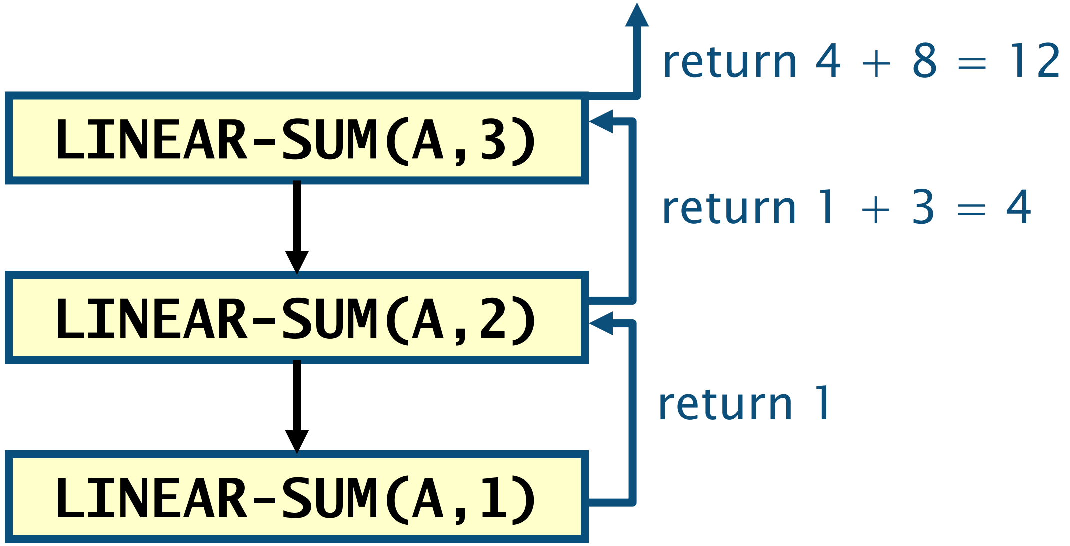
Yes: see 2nd return statement

Yes: if statement

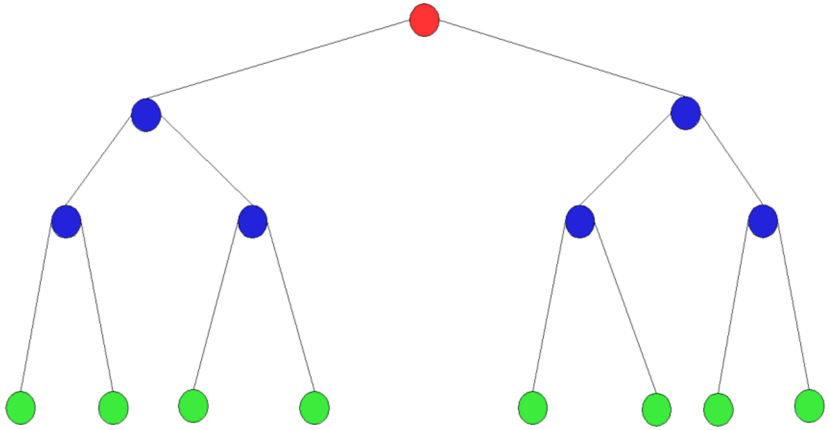
Sum of array elements: recursion trace

- Call LINEAR-SUM(A,3) on input A = [1,3,8,6,4,3]

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



Binary recursion



Binary recursion

- When an algorithm makes **two** recursive calls, we say that it uses **binary recursion**
  - To solve **two halves** of some problem
- Classic example: **Fibonacci** numbers are a sequence of numbers defined by
  - Every number in the sequency is the sum of previous two numbers
  - $F_n = F_{n-1} + F_{n-2}$ for $n > 1$ with $F_0 = 0$ and $F_1 = 1$
- In pseudocode

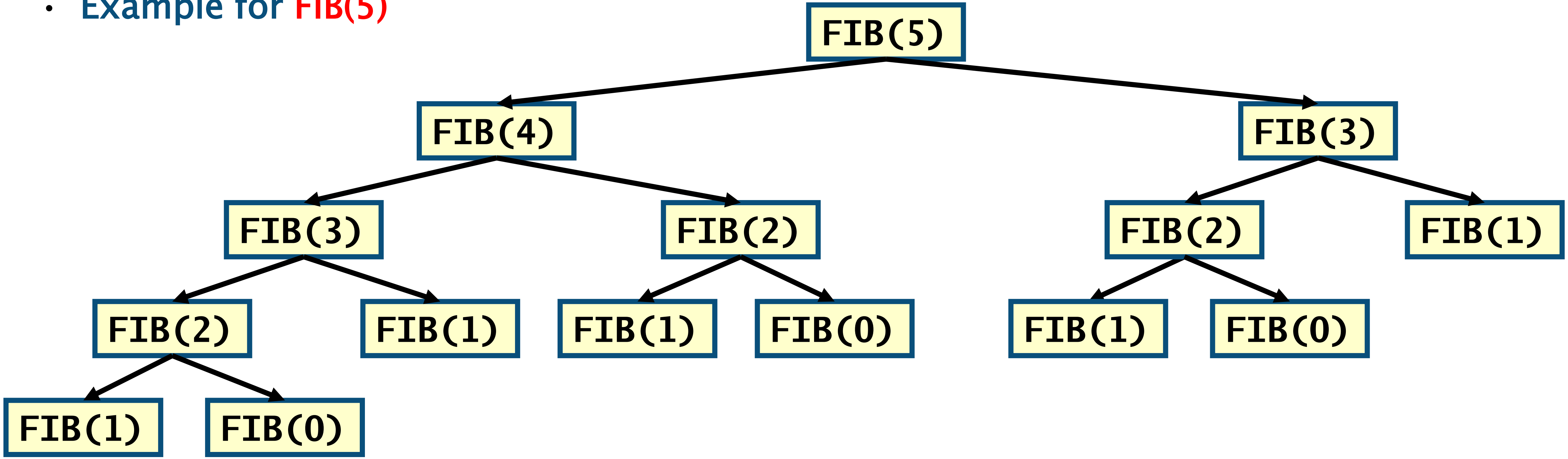
```
FIB(n)
  if n ≤ 1                            // base cases
    return n
  else
    return FIB(n-1) + FIB(n-2)   //binary recursion
```

Let’s get coding

- yeah yeah yeah...

Recursion tree

- Visualising each recursive call in an algorithm using binary recursion results in a (binary) recursion tree
- Example for FIB(5)



Other BINARY recursive algorithms we have seen already (in the world of Binary Trees)

```
INORDER(node)
  if node != NIL
    INORDER(node.left)
    print node.key
    INORDER(node.right)
```

- Because:
  - ... a binary tree is a *recursive* data structure (you can define a binary tree in term of smaller binary trees),
- ...so:
  - ...the algorithms for operations on them can often be defined recursively too

Tail Recursion

Tail recursion

- Recursion is useful tool for designing algorithms with short, elegant definitions

- Recursion has a cost
  - Need to use memory to keep track of the state of each recursive call (boxes in recursion traces)

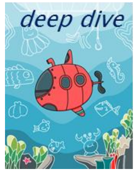
- Overcoming the cost:
  - When memory is of primary concern, useful to be able to derive non-recursive algorithms from recursive ones
    - Using iterations (e.g. for or while loops)
  - In some cases, we can gain memory efficiency by simply using tail recursion
    - Less to store for each recursive call

- An algorithm uses tail recursion when:
  - recursion is linear and
  - recursive call is its very last operation

Why Tail Recursion is "good" (and easy to replace with a loop)

Tail Recursion

Not Tail Recursion



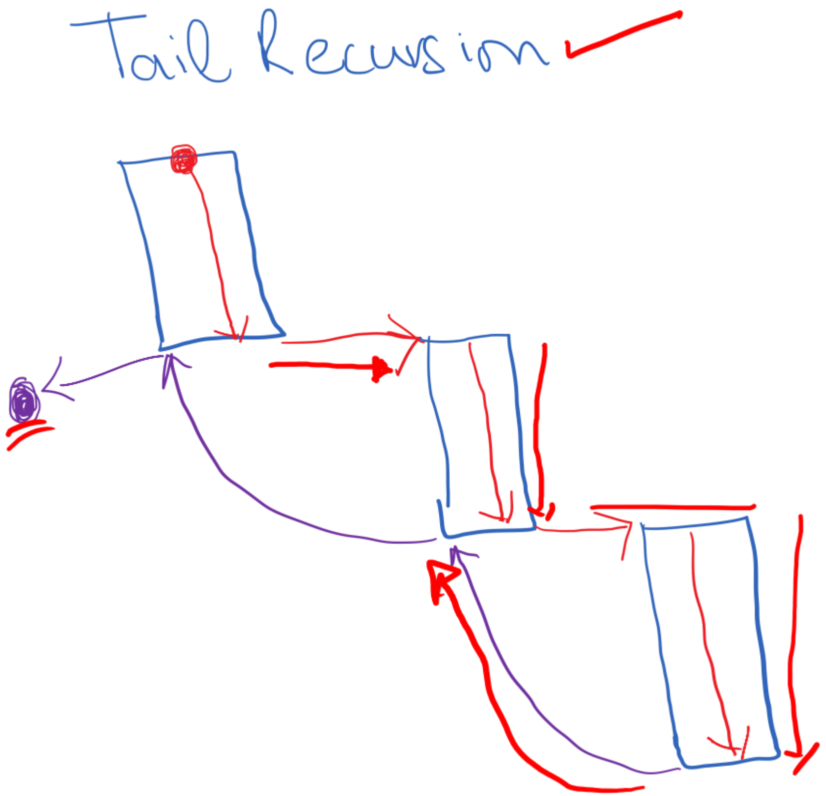
Tail Recursion
https://en.wikipedia.org/wiki/Tail_call

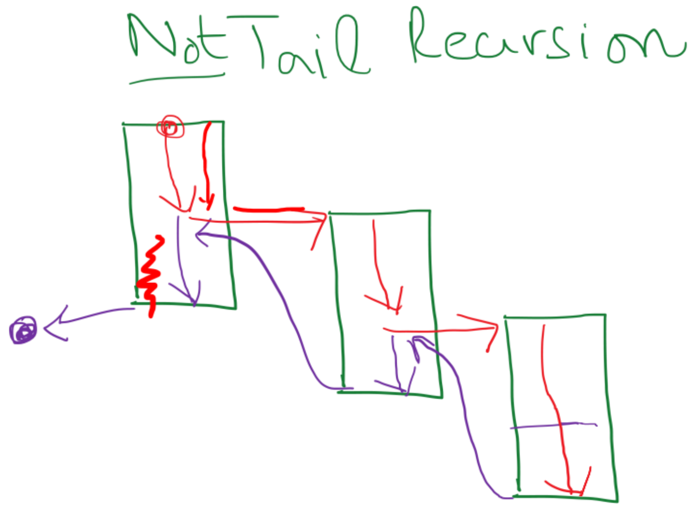
https://www.geeksforgeeks.org/tail-recursion/

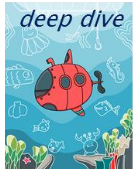
Tricks of the trade: Recursion to Iteration, Part 1: The Simple Method, secret features, and accumulators

http://blog.moertel.com/posts/2013-05-11-recursive-to-iterative.html

Why Tail Recursion is "good" (and easy to replace with a loop)







Tail Recursion
https://en.wikipedia.org/wiki/Tail_call

https://www.geeksforgeeks.org/tail-recursion/

Tricks of the trade: Recursion to Iteration, Part 1: The Simple Method, secret features, and accumulators

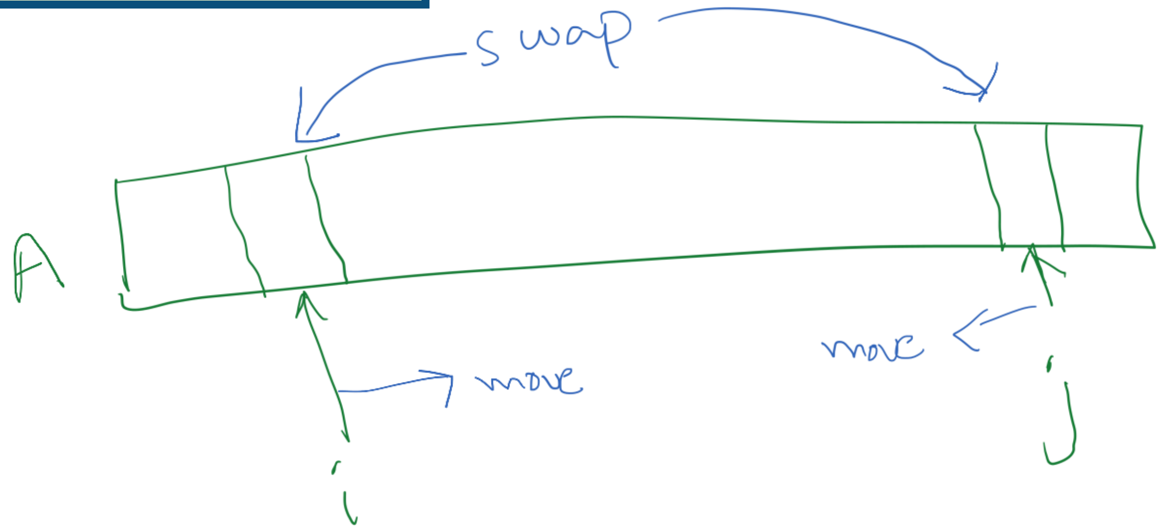
http://blog.moertel.com/posts/2013-05-11-recursive-to-iterative.html

Example: reversing the elements of an array

- Input: An array A and integer indices i,j ≥ 1
- Output: The reversal of the elements in A starting at index i and ending at j

```
REVERSE-ARRAY(A,i,j)
  if i < j then
    SWAP(A[i],A[j])
    REVERSE-ARRAY(A,i+1,j-1)
```

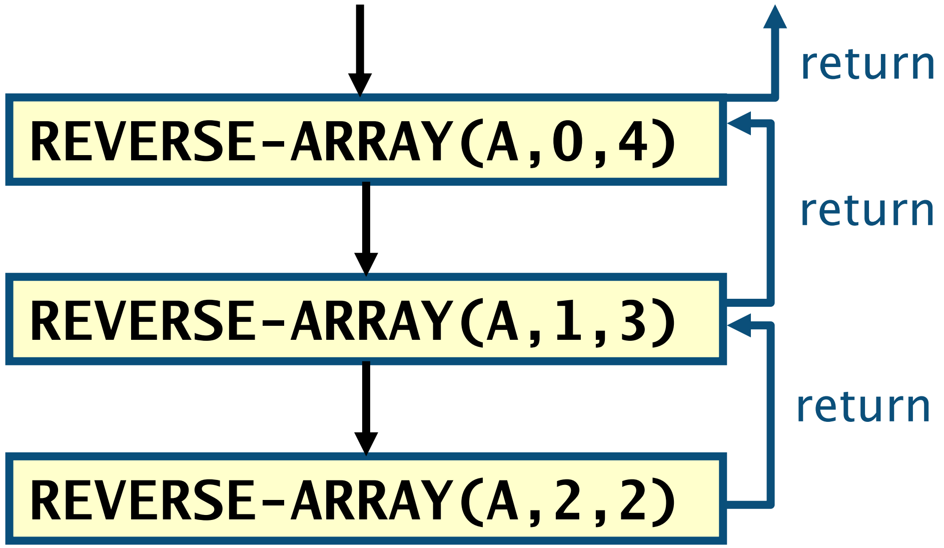
- Recursive call is the last operation



Recursion trace

- For REVERSE-ARRAY(A,0,4) with A = [3,4,6,1,0]

```
REVERSE-ARRAY(A,i,j)
  if i < j then
    SWAP(A[i],A[j])
    REVERSE-ARRAY(A,i+1,j-1)
```



- <u>Note: No operations performed on the blue (return) arrows</u>
  - This is a sign of TAIL RECURSION
  - Because there is no operation on the return path, we can easily replace a tail-recursion with loops

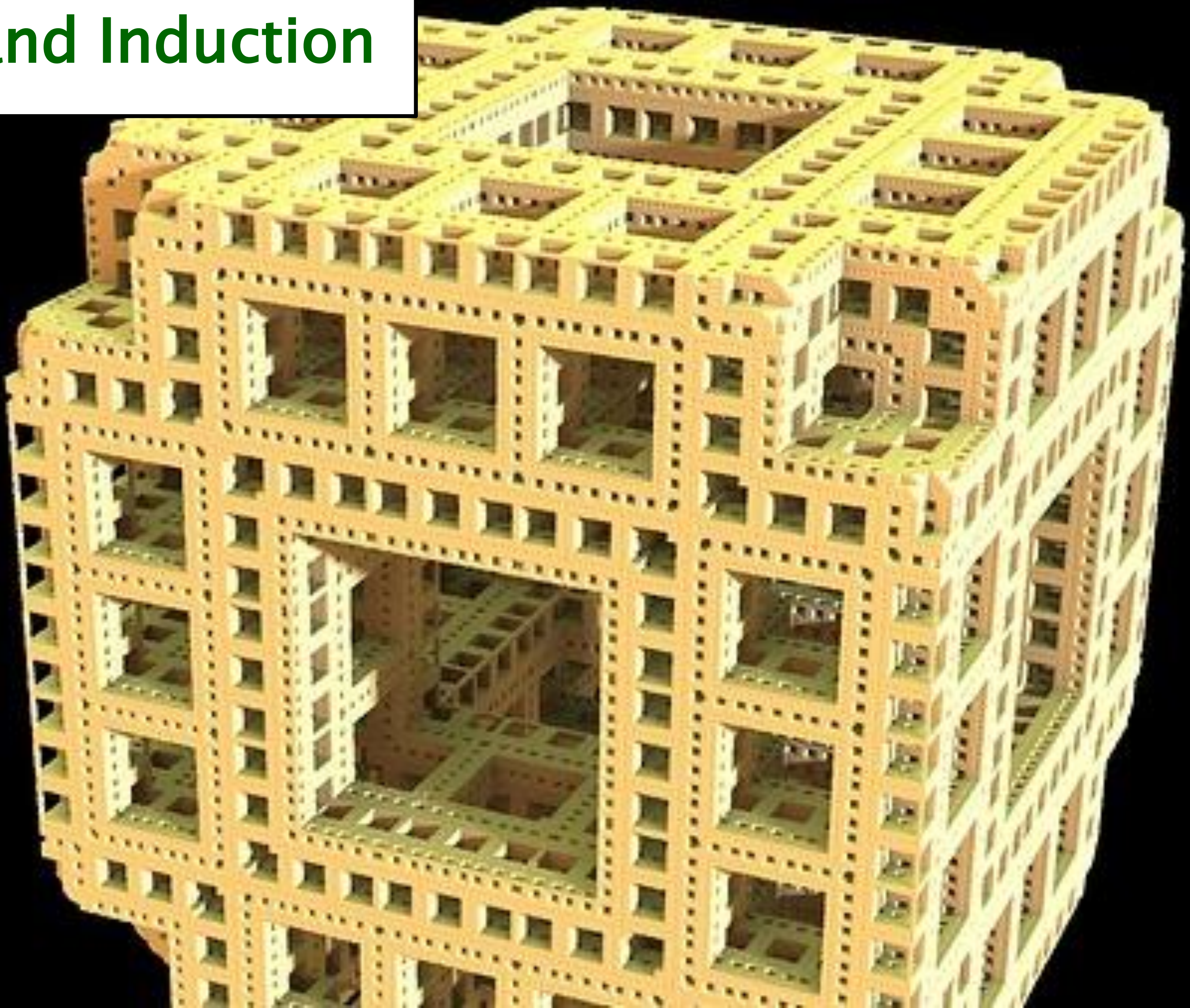
Conversion to non-recursive algorithm

- Non-recursive algorithm are also called iterative
- Algorithms using tail recursion can be converted to a non-recursive algorithm by iterating through repeated operations of functions, rather than calling the function again and again explicitly
- In general, we can always replace recursive algorithm with an iterative one, but often (not always) the recursive solution is shorter and easier to understand
- Example

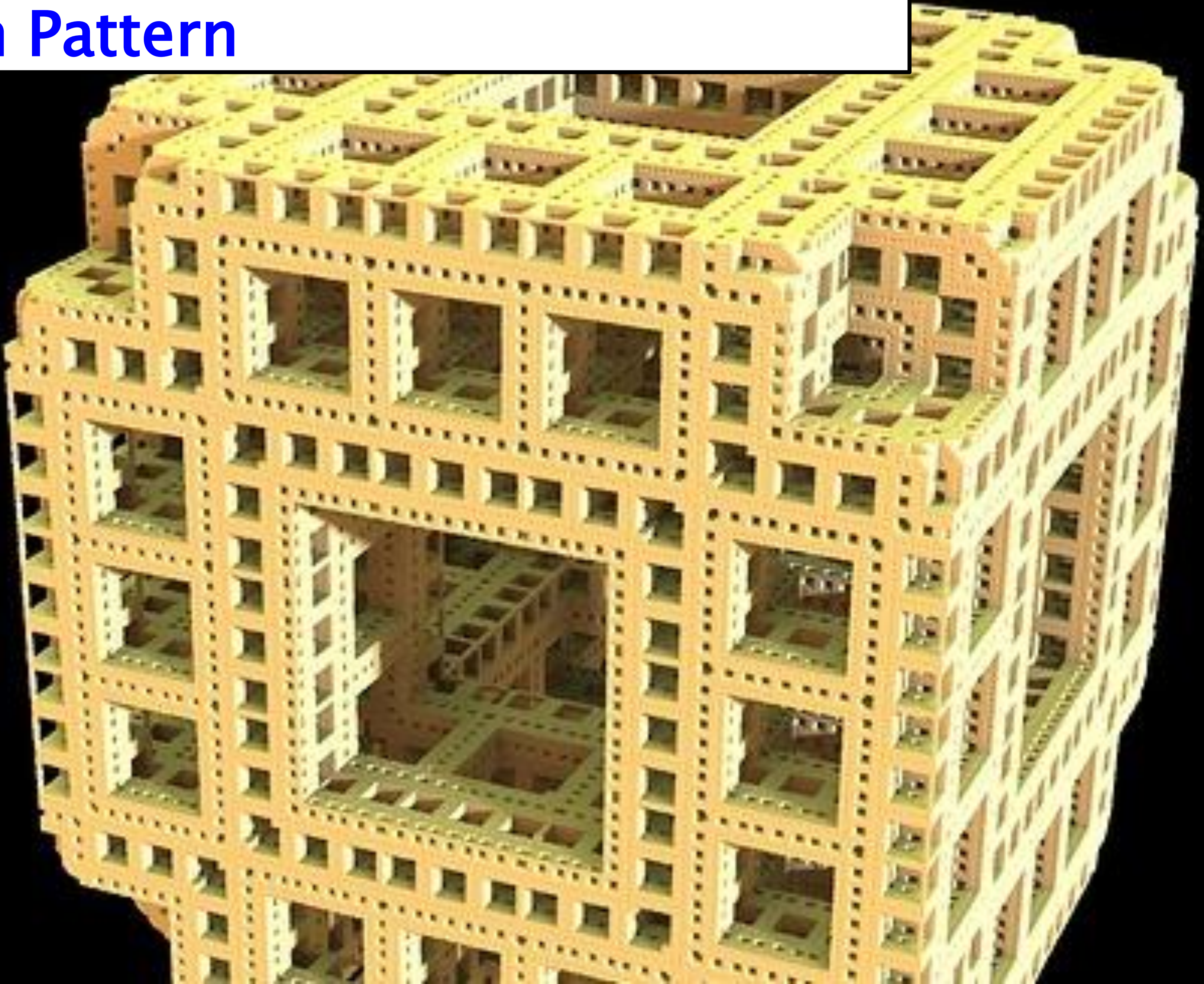
```
REVERSE-ARRAY(A,i,j)
  if i < j then
    SWAP(A[i],A[j])
    REVERSE-ARRAY(A,i+1,j-1)
```

```
REVERSE-ARRAY-ITER(A,i,j)
  while i < j
    SWAP(A[i],A[j])
    i := i + 1
    j := j - 1
```

Recursion and Induction



RECURSIVE (i.e. INDUCTIVE) DEFINTIONS
A Common Pattern



Have you noticed?

- Recall: Recursion in Functions
  - Define a function in terms in a way that it calls itself, while every time moving towards a *smaller* problem size
  - Define a *stopping case* where the function is defined directly, without any recursive calls.

```
FACT(n)
  if n = 1
    return 1
  else
    return n * FACT(n-1)
```

Have you noticed?

- **Recall: Recursion in Functions**
  - Define a function in terms in a way that it calls itself, while every time moving towards a *smaller* problem size
  - Define a *stopping case* where the function is defined directly, without any recursive calls.

```
FACT(n)
  if n = 1
    return 1
  else
    return n * FACT(n-1)
```

- **Recall: Sequences defined via Recurrence Relations**
  - We specify the first (or first few elements), and then
  - Define a recurrence relation on how to calculate subsequent terms using previous terms.

$$a_n = a_{n-1} + 2 \quad \text{(Recurrence Relation)}$$
$$a_0 = 0 \quad \text{(Initial Condition)}$$
$$\{a_n\} = 0,2,4,6,\ldots \quad \text{(Resulting Sequence, i.e. "Solution")}$$

Have you noticed?

- **Recall: Recursion in Functions**
  - Define a function in terms in a way that it calls itself, while every time moving towards a *smaller* problem size
  - Define a *stopping case* where the function is defined directly, without any recursive calls.

```
FACT(n)
  if n = 1
    return 1
  else
    return n * FACT(n-1)
```

- **Recall: Sequences defined via Recurrence Relations**
  - We specify the first (or first few elements), and then
  - Define a recurrence relation on how to calculate subsequent terms using previous terms.

$$a_n = a_{n-1} + 2$$ (Recurrence Relation)

$$a_0 = 0$$ (Initial Condition)

$$\{a_n\} = 0,2,4,6,\ldots$$ (Resulting Sequence, i.e. "Solution")

- **Mathematical Induction:**
  - Prove something for a *base case*
  - Prove that you can go from a generic case P(k) to its next one, P(k+1)

To prove that P(n) is true for all positive integers n, we complete these steps

- *Basis Step*: Show that P(1) is true.
- *Inductive Step*: Show that P(k) → P(k + 1) is true for all positive integers *k*.