Algorithms and Data Structures 2

4 – Recursive algorithms

Dr Michele Sevegnani

School of Computing Science
University of Glasgow

michele.sevegnani@glasgow.ac.uk

Outline

- **Recursive algorithms**
  - Recursion traces
  - Linear recursion
  - Tail recursion
  - Conversion to non-recursive (iterative) algorithms
  - Binary recursion
  - Recursion trees
- **Algorithm design paradigms**
  - Incremental
  - Divide-and-conquer

Recursion

- **A function is recursive if it refers to itself in its definition**
- **Classic example: the factorial function**
  - n! = n * (n-1) * (n-2) * . . .* 1

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

- **We have implemented the factorial function as a recursive algorithm**
  - Note FACT() is applied to a smaller number every time until it is applied to 1 (stopping case)

In general

- **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)

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)

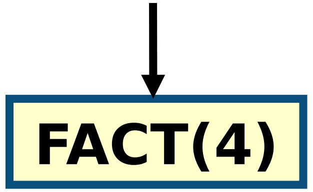
Recursion trace

- **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)**

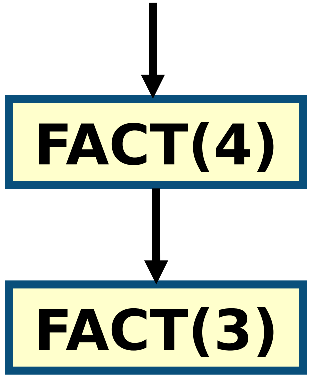



```
FACT(n)
  if n = 1
    return 1
  else
    return n * FACT(n-1)
```

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)**

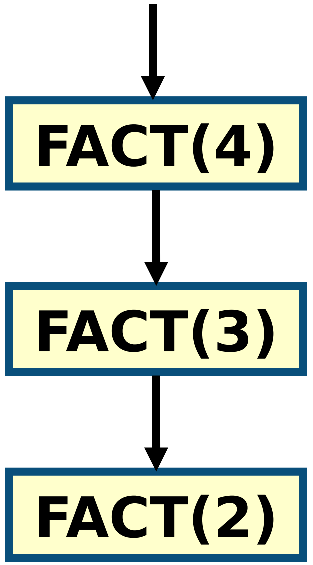
```
FACT(n)
  if n = 1
    return 1
  else
    return n * FACT(n-1)
```



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)**

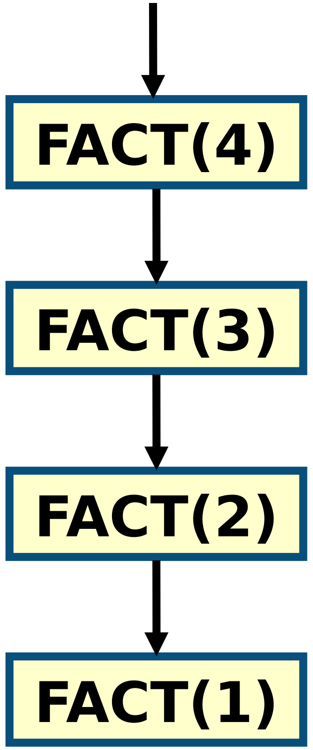
```
FACT(n)
  if n = 1
    return 1
  else
    return n * FACT(n-1)
```



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)**

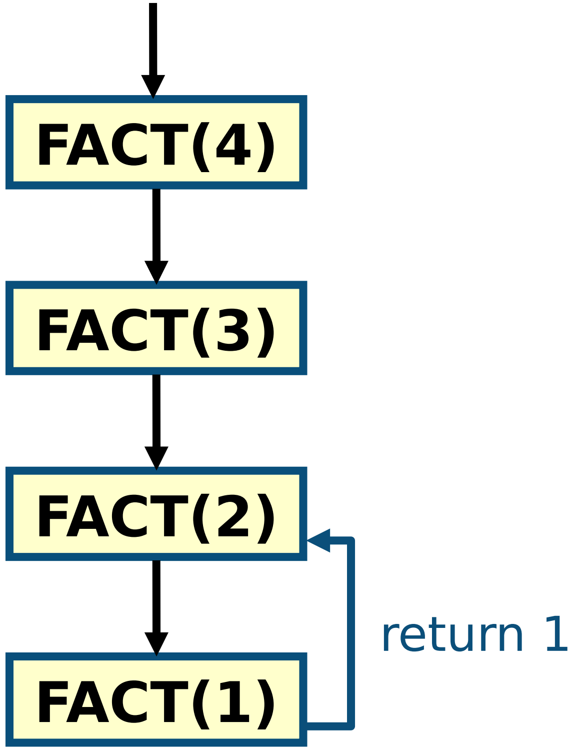
```
FACT(n)
  if n = 1
    return 1
  else
    return n * FACT(n-1)
```



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)**

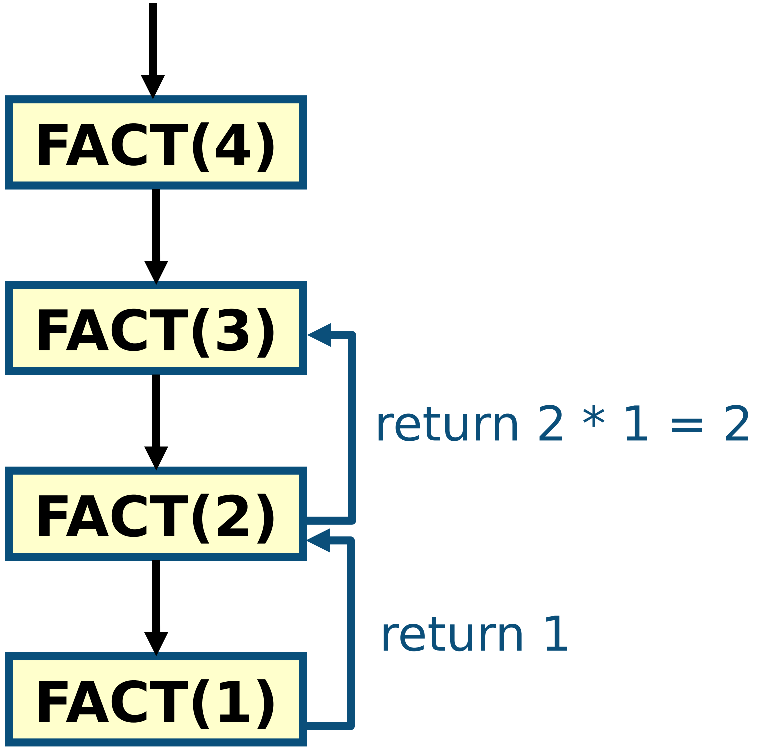
```
FACT(n)
  if n = 1
    return 1
  else
    return n * FACT(n-1)
```



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)**

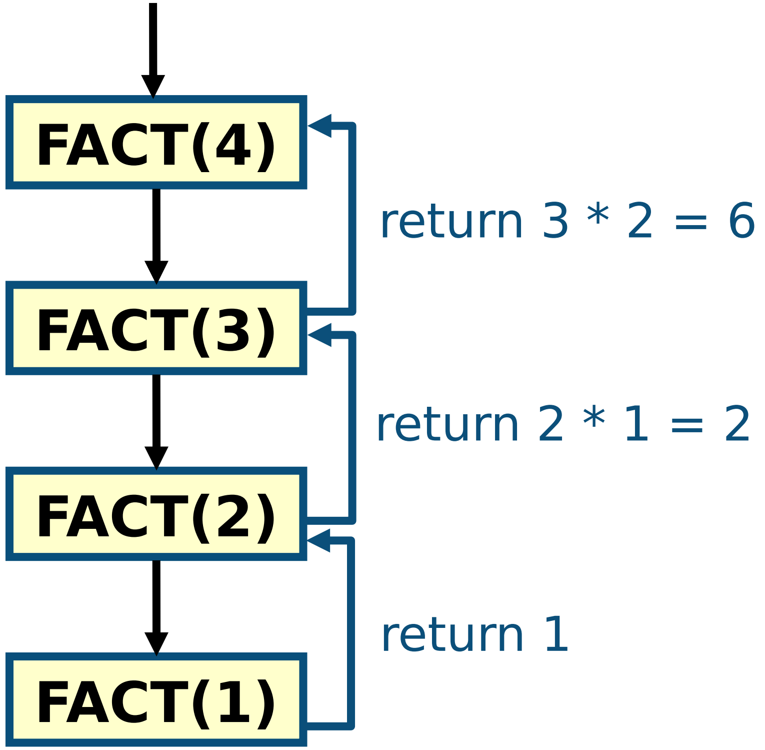
```
FACT(n)
  if n = 1
    return 1
  else
    return n * FACT(n-1)
```



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)**

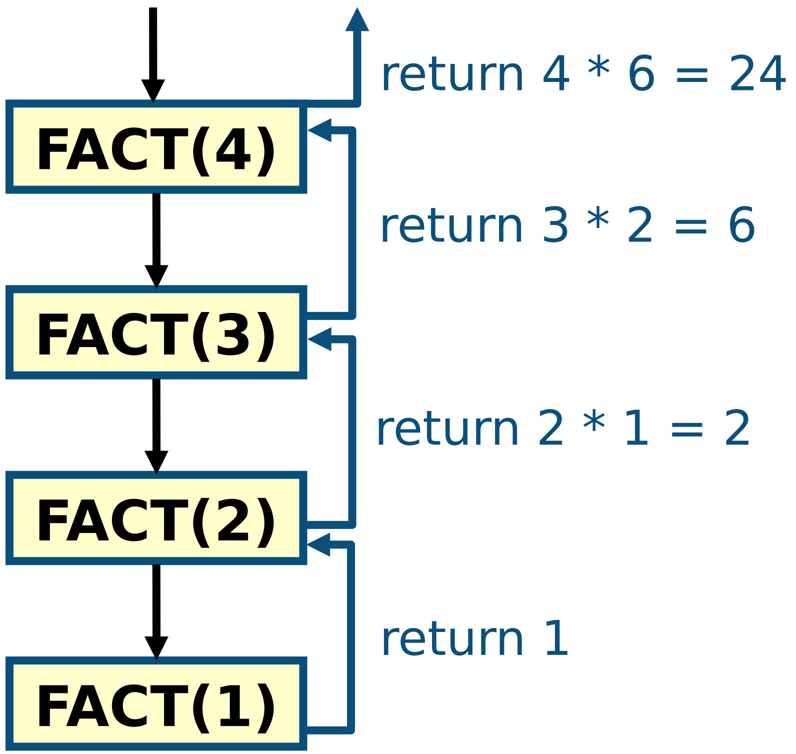
```
FACT(n)
  if n = 1
    return 1
  else
    return n * FACT(n-1)
```



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 recursion

- **With linear recursion a method is defined so that it makes at most one recursive call each time it is invoked**
  - Useful when we view an algorithmic problem in terms of a first and/or last element plus a remaining set with same structure as original set
- **The amount of space needed to keep track of the nested calls, grows linearly with n (the size of the input)**
- **Example: FACT(n)**
  - One recursive call FACT(n-1)
  - See the recursion trace for space requirements

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

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

- **Does LINEAR-SUM satisfy Rules 1 and 2?**
  - Rule 1: input reduced at each recursive call
  - Rule 2: stopping case

Example: summing the elements of an array

- **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 Rules 1 and 2?**
  - Rule 1: input reduced at each recursive call
  - Rule 2: stopping case

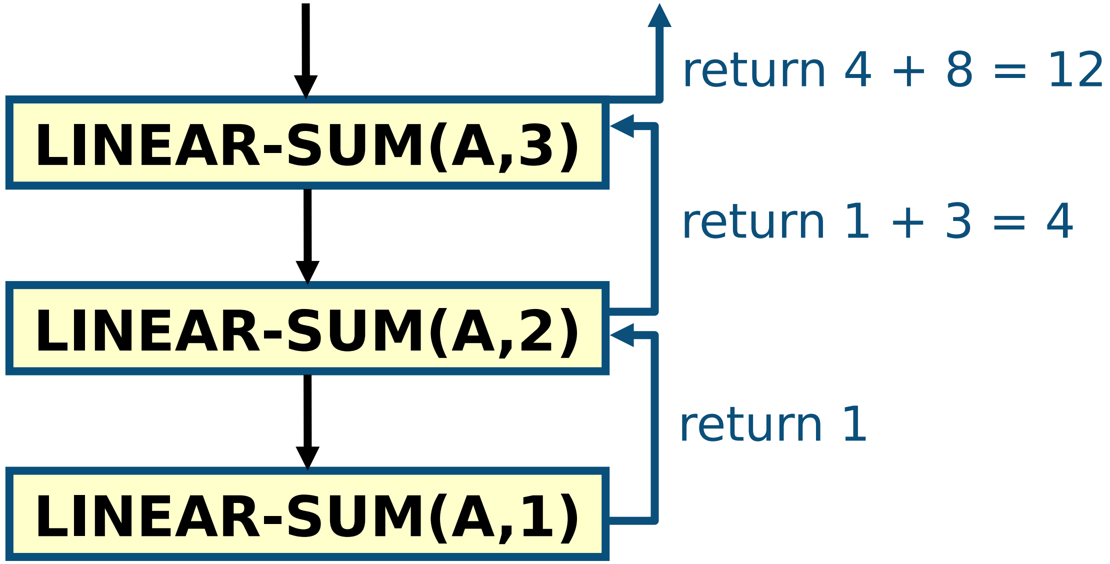
Yes: return statement

Yes: if statement

Recursion trace

- **For LINEAR-SUM(A,3) with A = [1,3,8,6,4,3]**

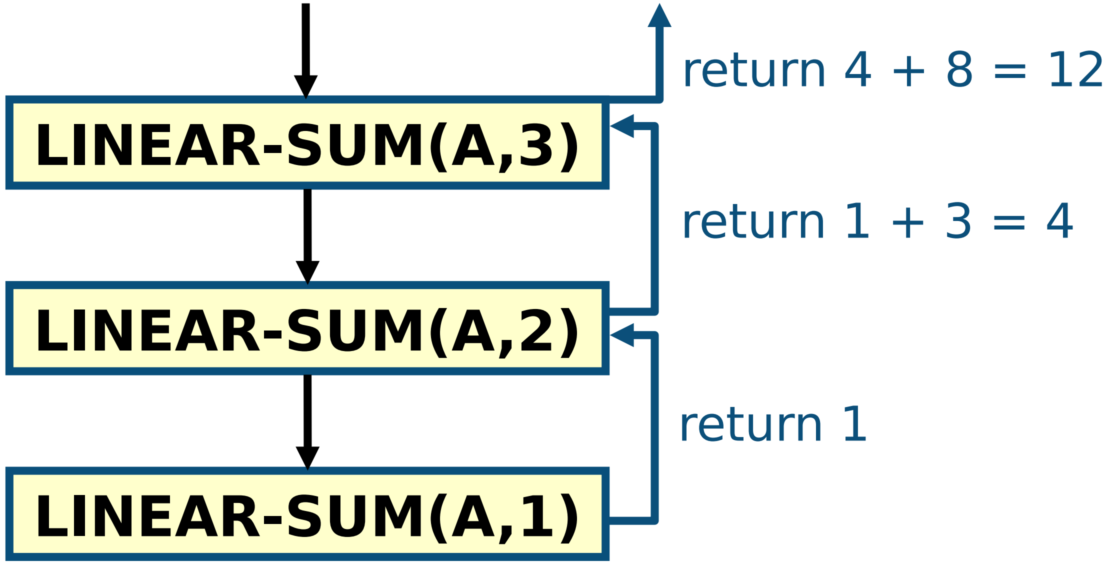
```
LINEAR-SUM(A,n)
  if n = 1 then
    return A[0]
  else
    return LINEAR-SUM(A,n-1) + A[n-1]
```



Recursion trace

- **For LINEAR-SUM(A,3) with 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]
```



- **What is the complexity of LINEAR-SUM?**
- **And FACT?**

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)
- **When memory is of primary concern, useful to be able to derive non-recursive algorithms from recursive ones**
  - Can use a stack data structure to do this (we will cover this in the next lectures)
  - Using iterations (e.g. for or while loops)
- **In some cases, we can gain memory efficiency by simply using tail recursion**
- **An algorithm uses tail recursion when recursion is linear and recursive call is its very last operation**

Example: reversing the elements of an array

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

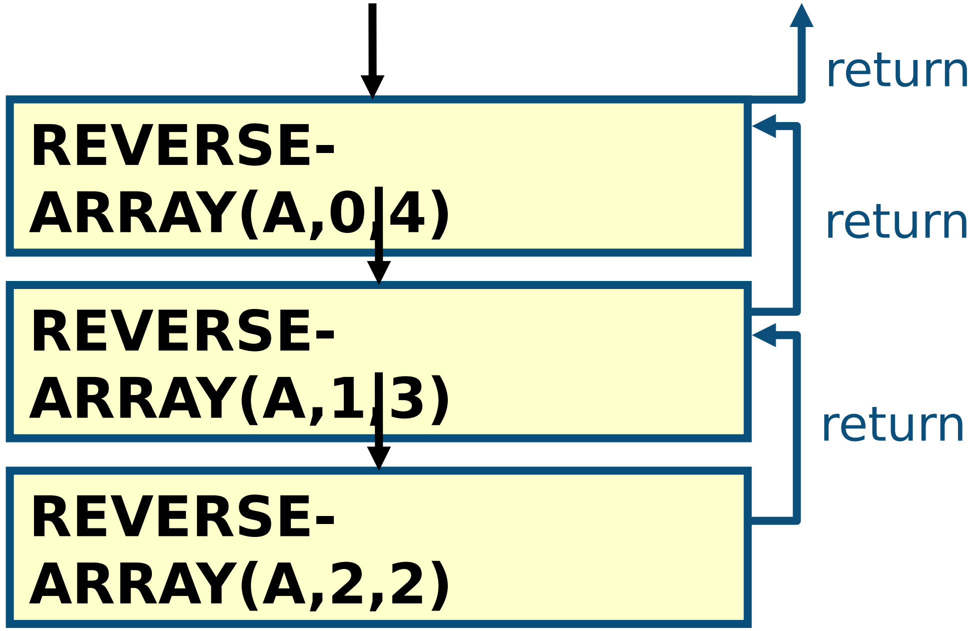
```
REVERSE-ARRAY(A,i,j)
  if i < j then
    SWAP(A[i],A[j])
    REVERSE-ARRAY(A,i+1,j-1)
```

- **Recursive call is the last operation**
- **Is LINEAR-SUM tail recursive?**
- **And FACT?**

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)
```



- **No operations performed on the blue (return) arrows**

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 recursive calls rather than calling them explicitly**
- **In general, we can always replace recursive algorithm with an iterative one, but often the recursive solution is shorter and easier to understand**

```
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
```

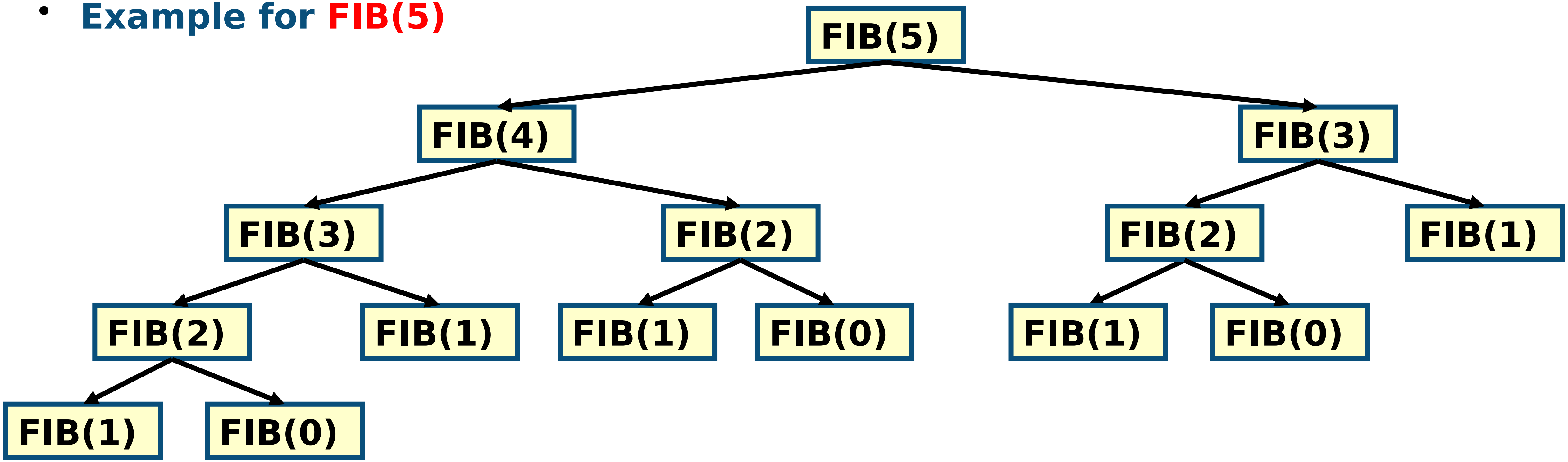
Binary recursion

- **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**
  - $F_n = F_{n-1} + F_{n-2}$ for $n > 1$ with $F_0 = 0$ and $F_1 = 1$
- **In pseud**

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

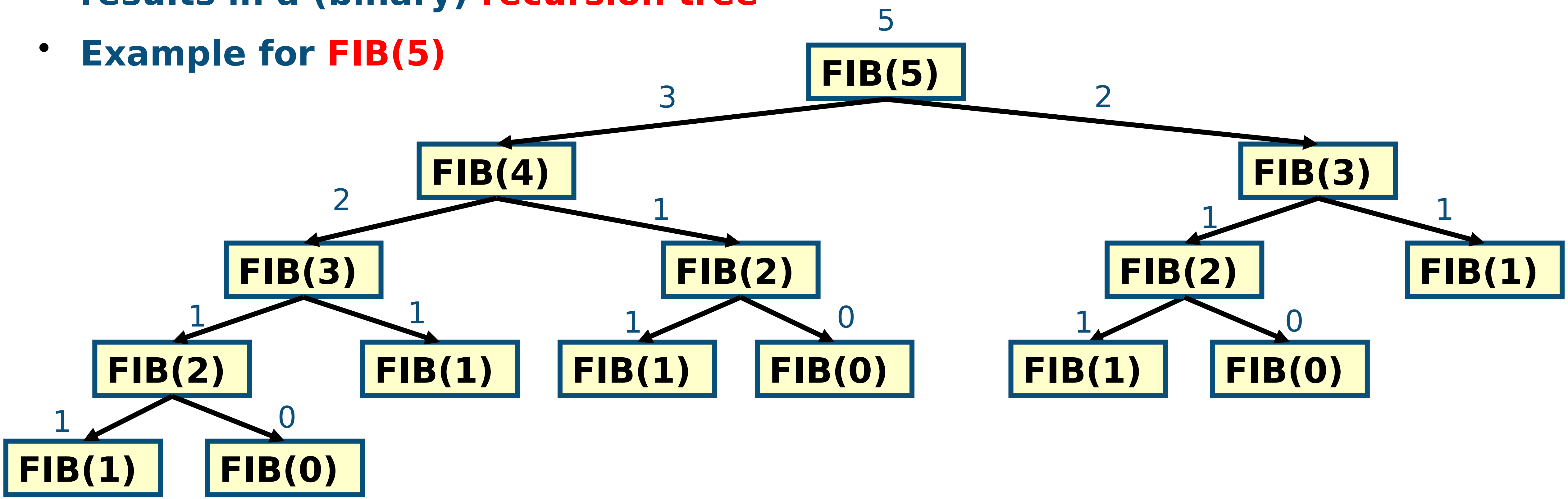
Recursion tree

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



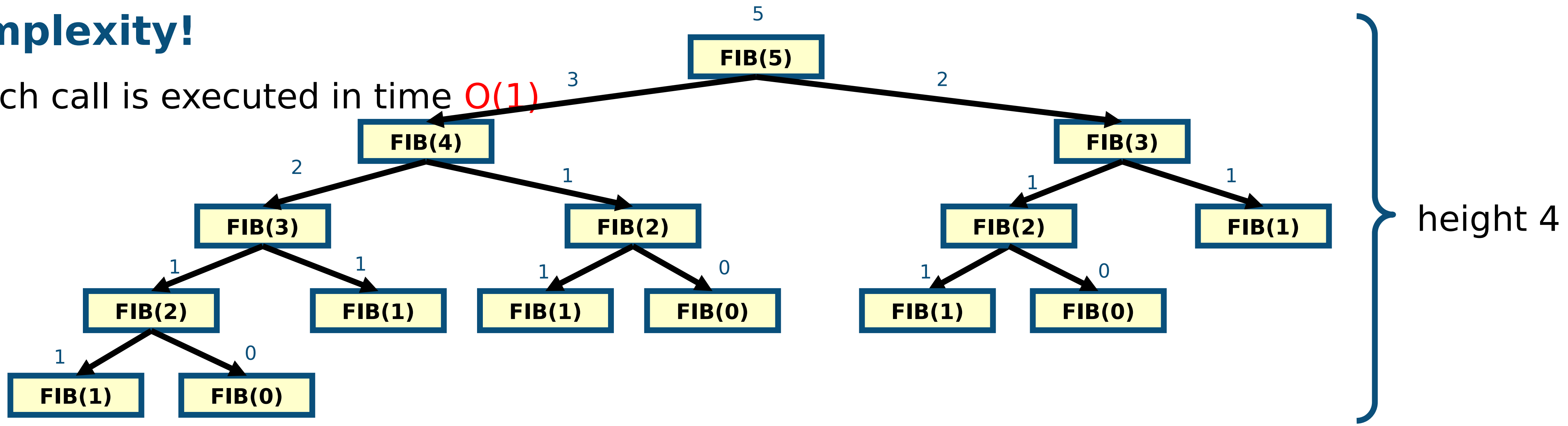
Recursion tree

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



Recursion tree

- **Recall from AF2**
  - The number of nodes n in a full binary tree is at most $n = 2^{h+1} - 1$ where h is the height of the tree
- **Here height is $O(n)$ therefore there are $O(2^n)$ recursive calls: exponential complexity!**
  - Each call is executed in time $O(1)$



Incremental approach

- **Popular algorithm design approach in which the solution of a problem is built incrementally**
  - One element at a time
- **Example: INSERTION-SORT at each iteration**
  - Subarray A[1..j-1] is assumed sorted
  - Element A[j] is inserted in the correct position to obtain sorted subarray A[1..j]

```
INSERTION-SORT(A)
  for j = 1 to n-1
      key := A[j]
      i := j-1
      while i ≥ 0 and A[i] > key
        A[i+1] := A[i]
        i := i-1
      A[i+1] := key
```

Divide-and-conquer approach

- **An algorithm design paradigm based on recursion**
- **It involves three steps at each level of the recursion**
  - Divide the problem into several smaller subproblems (that are smaller instances of the same problem)
  - Conquer: Solve subproblems recursively (until you hit the base)
  - Combine the solutions to the subproblems to create a solution to the original problem
- **We will use this approach to define some efficient sorting algorithms**
  - MERGE-SORT
  - QUICK-SORT

Summary

- **Recursive algorithms**
  - Linear recursion
  - Binary recursion
  - Recursion trace and trees
  - Tail recursion
  - Conversion to non-recursive algorithm
- **Algorithm design paradigms**
  - Incremental
  - Divide-and-conquer