

Data Structures and Algorithms

Prepared by Prof. Shereen Taie

IT251
Information Technology Department

Lecture 3



Data Structures and Algorithms

Lecture 3

IT251

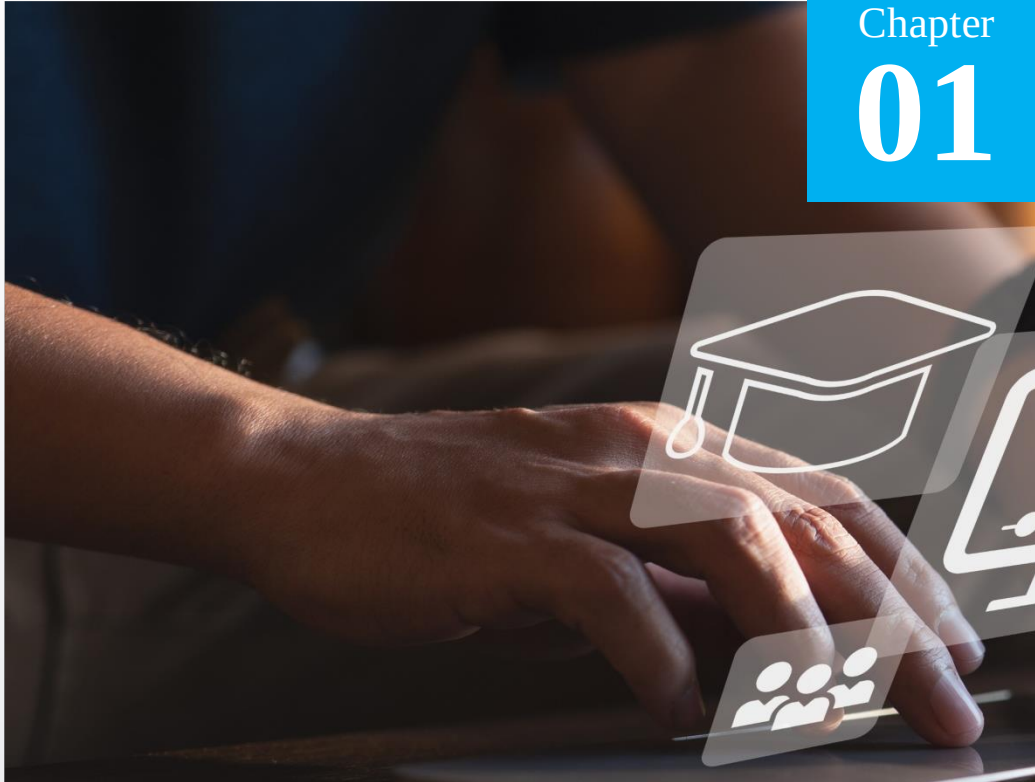
Information Technology Department

Prepared by
Prof. Shereen Taie

www.midocean.ae

Chapter
01

Theoretical analysis of time efficiency



Theoretical analysis of time efficiency

- Time efficiency is analyzed by determining the number of repetitions of the basic operation as a function of input size
- Basic operation: the operation that contributes most towards the running time of the algorithm

$$T(n) \approx c_{op} C(n)$$

Diagram illustrating the components of the time complexity equation $T(n) \approx c_{op} C(n)$:

- $T(n)$: running time
- n : input size
- c_{op} : execution time for basic operation
- $C(n)$: Number of times basic operation is executed

How to compute number of operations:

Statement	Steps	Comment
$A = B + C$	2	Addition and assignment operation
Return (B + C)	2	Addition and return
$A = \text{Array}[1]$	2	Assignment and Array reference
$i < 10 \parallel j > 20$	3	$<, >$ and $\parallel$ operation
$A = \text{factorial}(5)$	2	Assignment and function call
for (i=0; i<n; i=i+1)	$3n + 2$	Initialization 1 i=0
		Comparison n + 1 i<n
		Increment n i+1
		Assignment n i=i+1

Input size and basic operation examples

Problem	Input size measure	Basic operation executed
Searching for key in a list of n items	Number of list's items, i.e. n	Key comparison
Multiplication of two matrices	Matrix dimensions or total number of elements	Multiplication of two numbers
Checking primality of a given integer n	n 's size = number of digits (in binary representation)	Division
Typical graph problem	#vertices and/or edges	Visiting a vertex or traversing an edge

How to compute number of operations:

- Example : Factorial Function

```
factorial (n) {  
    f = 1;  
    if ( n > 0 )  
        for (i = 1 to n)  
            f = f * i;  
    return f ; }
```

Ex. factorial (3)

```
f(1)=1  
i=1  
f(1)=1  
  
i=2  
f(2)=1*2  
  
i=3  
f(3)=1*2*3  
  
return f(3)=6
```

- Let $T(n)$ = Number of multiplications.
- For a given n , then $T(n) = n$ (always)

How to compute number of operations:

- Example: Sum Function

```
Sum(a, b) {  
    return (a+b)  
}
```

- Number of steps: 1 for return and 1 for addition

Every-case Time Complexity

- **Every-Case:**

The basic operation is **always done the same number of times for every instance of size n .**

- $T(n)$ is called the ***every-case time complexity*** of the algorithm, and
- The determination of $T(n)$ is called an ***every-case time complexity analysis***.

Examples

- **Add Array Members:**

- Basic operation: the addition of an item in the array to *sum*.
- Input size: n , the number of items in the array.

$$T(n) = n.$$

Chapter 2

Recurrences

Recurrence: an equation that **describes a function in terms of its value on smaller functions**

Recurrence relation uses **recursion** to define a sequence.

Recursion is used to compute the factorial function, to determine whether a word is a palindrome, to compute powers of a number, ...etc.

Recursive Algorithms

Definition: An algorithm is called *recursive* if it solves a problem by reducing it to an instance of the same problem with smaller input.

For the *algorithm to terminate*, the instance of the problem *must eventually be reduced to some initial case* for which the solution is known.

Recursive algorithms

Recursive solution: solve a smaller version of the problem and combine the smaller solutions.

Example: to find the largest element in an array $A[1..n]$

- If n is 1, the problem is easily solvable. **Just return $A[n]$**
- If $n > 1$, then find the largest element of $A[1..n-1]$, compare it to $A[n]$ and return the largest.

Recursive algorithms

Recursive solutions consist of:

- **Base case(s)**
 - The problem is explicitly solved for certain values of the input (generally small values)
- **Recursive step**
 - Divide the problem into one or more simpler or smaller parts.
 - Solve each part recursively
 - Combine the solutions of the parts into a solution to the problem

1. If this is missing, or not implemented correctly, **your program will not terminate.**

2. If the "part" is as big as the whole, **your program will not terminate**

Recursive algorithms:

Rules for successful recursion:

- Handle the base case(s) first
- Make sure that the recursive step is applied on a smaller version of the problem.
- "smaller" means closer to the base case
- Make certain that the base case will be reached.

Recursive Algorithms

A subroutine which calls itself, with different parameters.

- Need to evaluate factorial(n)

$$\begin{aligned}\text{factorial}(n) &= n \cdot \underline{(n-1) \dots 2 \cdot 1} \\ &= n * \text{factorial}(n-1)\end{aligned}$$

- Suppose routine factorial(p) can find factorial of p for all $p < m$. Then factorial(m+1) can be solved as follows:

$$\text{factorial}(m+1) = (m+1) * \text{factorial}(m)$$

Ex. factorial (3)

f(1)=1

i=1

f(1)=1

i=2

f(2)=2*1=2*f(1)

i=3

f(3)=3*2*1= 3*f(2)

return f(3)=3*2*1=6

Factorial(m)

{

 If $m = 0$, Factorial(m) = 1;

 Else Factorial(m) = $m * \text{Factorial}(m-1)$

}

Rules:

1. Have a base case for which the subroutine need not call itself.
2. For general case, the subroutine does some operations, calls itself, gets result and does some operations with the result
3. To get result, subroutine should progressively move towards the base case.

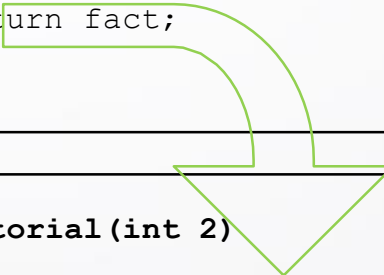
factorial Method:

```
int factorial(int n)
{
    int fact;
    if (n==0) //base case fact = 1;
    else // recursive case (decomposition)
        fact = factorial(n - 1) * n; // composition

    return fact;
}
```

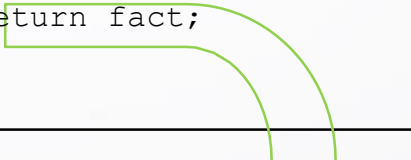
```
int factorial(int 3)
{
    int fact;
    if (n==0) fact = 1;
    else
        fact = factorial(2) * 3; return fact;
}
```

```
int factorial(int 3)
{
    int fact;
    if (n ==0) fact =1;
    else
        fact = factorial(2) * 3; return fact;
}
```



```
int factorial(int 2)
{
    int fact;
    if (n==0) fact =1; else
        fact = factorial(1) * 2;
    return fact;
}
```

```
int factorial(int 3)
{
    int fact;
    if (n ==0) fact =1;
    else
        fact = factorial(2) * 3; return fact;
}
```

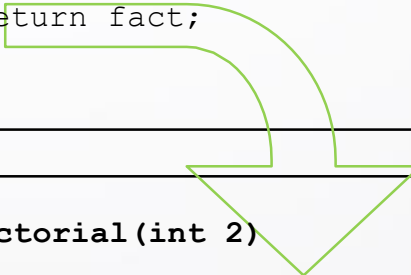


```
int factorial(int 2)
{
    int fact;
    if (n==0) fact =1; else
        fact = factorial(1) * 2;
    return fact;
}
```

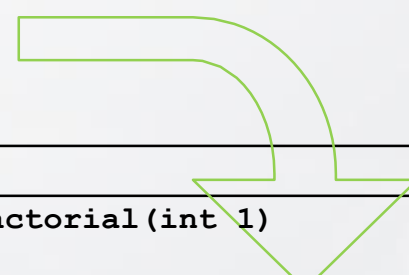


```
int factorial(int 1)
{
    int fact;
    if (n==0)fact =1; else
        fact=factorial(1 - 1) * 1; return fact;
}
```

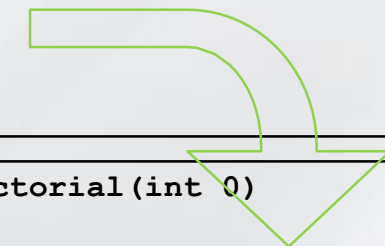
```
int factorial(int 3)
{
    int fact;
    if (n == 0) fact = 1;
    else
        fact = factorial(2) * 3; return fact;
}
```



```
int factorial(int 2)
{
    int fact;
    if (n == 0) fact = 1; else
        fact = factorial(1) * 2;
    return fact;
}
```



```
int factorial(int 1)
{
    int fact;
    if (n == 0) fact = 1; else
        fact = factorial(1 - 1) * 1; return fact;
}
```

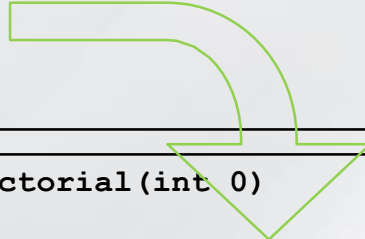


```
int factorial(int 0)
{
    int fact;
    if (n == 0) fact = 1;
    else
        fact = factorial(n - 1) * n;
    return fact;
}
```

```
int factorial(int 3)
{
    int fact;
    if (n ==0) fact =1;
    else
        fact = factorial(2) * 3; return fact;
}
```

```
int factorial(int 2)
{
    int fact;
    if (n==0) fact =1; else
        fact = factorial(1) * 2;
    return fact;
}
```

```
int factorial(int 1)
{
    int fact;
    if (n==0) fact =1; else
        fact=factorial(1 - 1) * 1; return fact;
}
```



```
int factorial(int 0)
{
    int fact;
    if (n==0) fact =1;
    else
        fact=factorial(n - 1) * n;
    return fact;
}
```

```
int factorial(int 3)
{
    int fact;
    if (n ==0) fact =1;
    else
        fact = factorial(2) * 3; return fact;
}
```

```
int factorial(int 2)
{
    int fact;
    if (n==0) fact =1; else
        fact = factorial(1) * 2;
    return fact;
}
```

```
int factorial(int 1)
{
    int fact;
    if (n==0) fact =1; else
        fact=factorial(1 - 1) * 1; return fact;
}
```

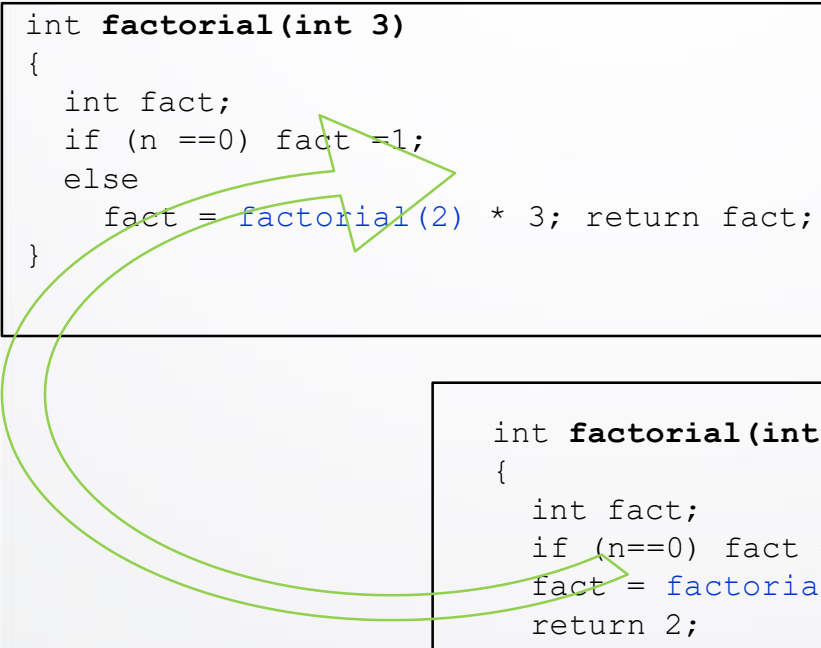
```
int factorial(int 0)
{
    int fact;
    if (n==0) fact =1;
    else
        fact=factorial(n - 1) * n;
    return 1;
}
```

```
int factorial(int 3)
{
    int fact;
    if (n ==0) fact =1;
    else
        fact = factorial(2) * 3; return fact;
}
```

```
int factorial(int 2)
{
    int fact;
    if (n==0) fact =1; else
        fact = factorial(1) * 2;
    return fact;
}
```

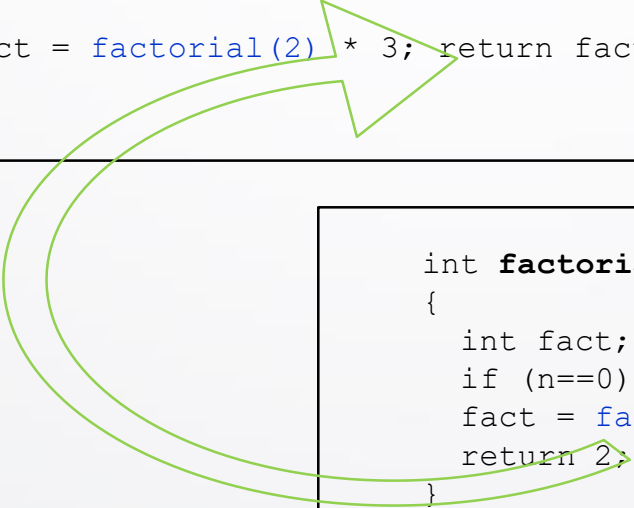
```
int factorial(int 1)
{
    int fact;
    if (n==0)fact =1; else
        fact=factorial(1 - 1) * 1;
    return 1;
}
```

```
int factorial(int 3)
{
    int fact;
    if (n == 0) fact = 1;
    else
        fact = factorial(2) * 3; return fact;
}
```



```
int factorial(int 2)
{
    int fact;
    if (n == 0) fact = 1; else
        fact = factorial(1) * 2;
    return 2;
}
```

```
int factorial(int 3)
{
    int fact;
    if (n == 0) fact = 1;
    else
        fact = factorial(2) * 3; return fact;
}
```

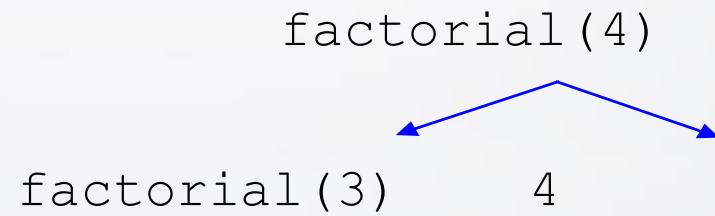


```
int factorial(int 2)
{
    int fact;
    if (n == 0) fact = 1; else
        fact = factorial(1) * 2;
    return 2;
}
```

```
int factorial(int 3)
{
    int fact;
    if (n ==0) fact =1;
    else
        fact = factorial(2) * 3; return 6;
}
```

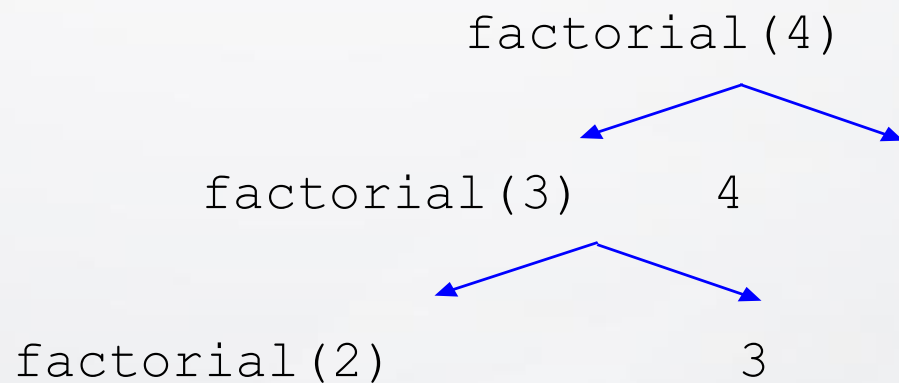
Execution Trace (decomposition):

```
int factorial(int n)
{
    int fact;
    if (n==0) fact = 1; else
        fact = factorial(n-1) * n; return
    fact;
}
```

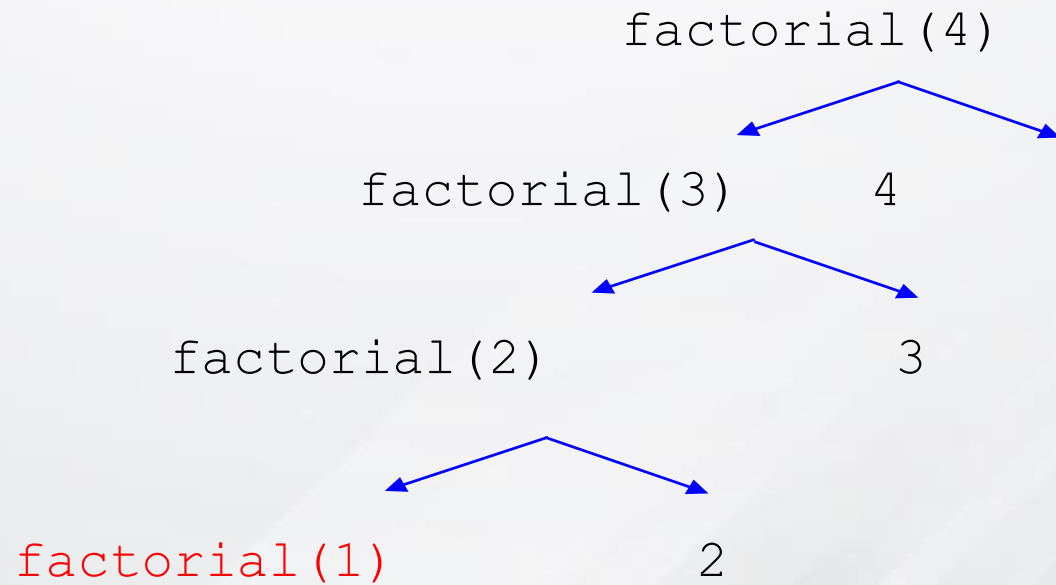


Execution Trace (decomposition):

```
public static int factorial(int n)
{
    int fact;
    if (n > 1) // recursive case (decomposition) fact =
        factorial(n - 1) * n; (composition)
    else // base case fact = 1;
        return fact;
}
```

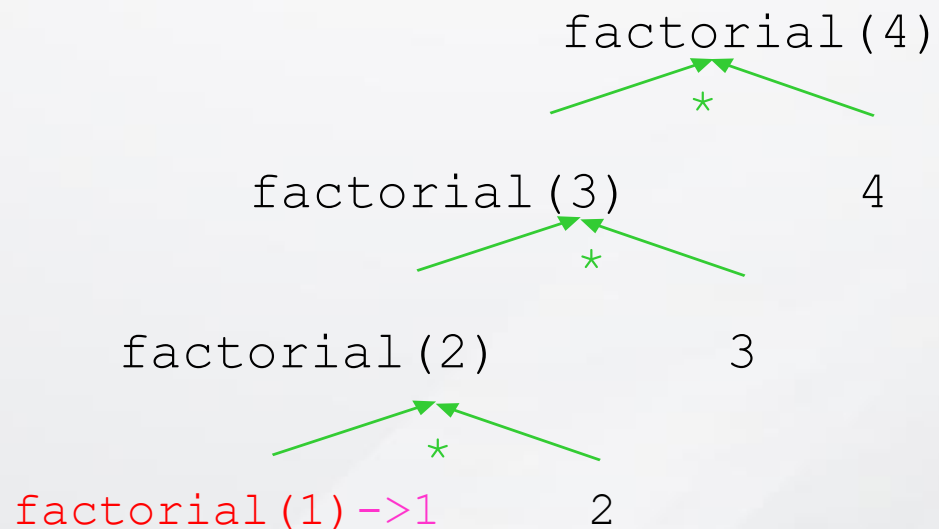


```
public static int factorial(int n)
{
    int fact;
    if (n > 1) // recursive case (decomposition) fact =
        factorial(n - 1) * n; (composition)
    else // base case fact = 1;
        return fact;
}
```



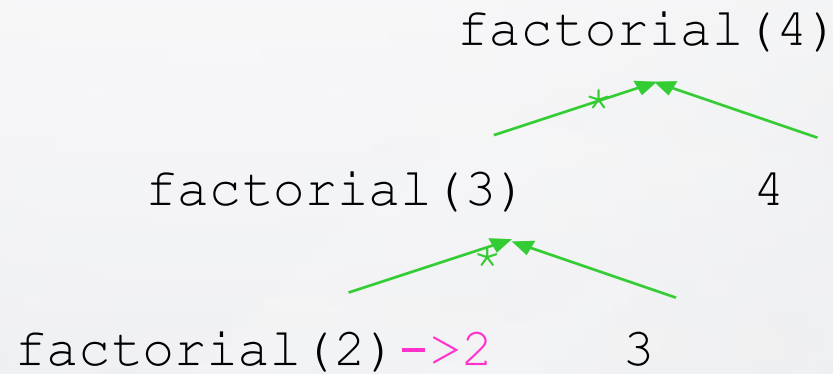
Execution Trace (composition):

```
public static int factorial(int n)
{
    int fact;
    if (n > 1) // recursive case (decomposition) fact =
        factorial(n - 1) * n; (composition)
    else // base case fact = 1;
        return fact;
}
```



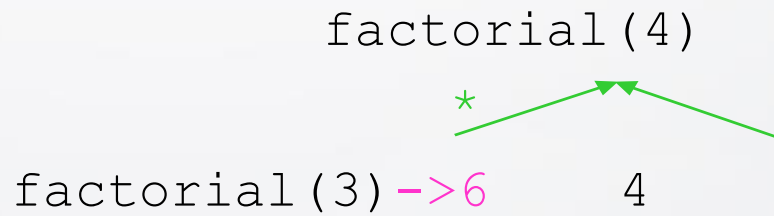
Execution Trace (composition):

```
public static int factorial(int n)
{
    int fact;
    if (n > 1) // recursive case (decomposition) fact =
        factorial(n - 1) * n; (composition)
    else // base case fact = 1;
        return fact;
}
```



Execution Trace (composition):

```
public static int factorial(int n)
{
    int fact;
    if (n > 1) // recursive case (decomposition) fact =
        factorial(n - 1) * n; (composition)
    else // base case fact = 1;
        return fact;
}
```



Execution Trace (composition):

```
public static int factorial(int n)
{
    int fact;
    if (n > 1) // recursive case (decomposition) fact =
        factorial(n - 1) * n; (composition)
    else // base case fact = 1;
        return fact;
}
```

factorial(4) $\rightarrow$ 24

$$T(n) = T(n-1) + 1$$

Fibonacci Numbers:

The N^{th} Fibonacci number is the sum of the previous two Fibonacci numbers

0, 1, 1, 2, 3, 5, 8, 13, ...

Recursive Design:

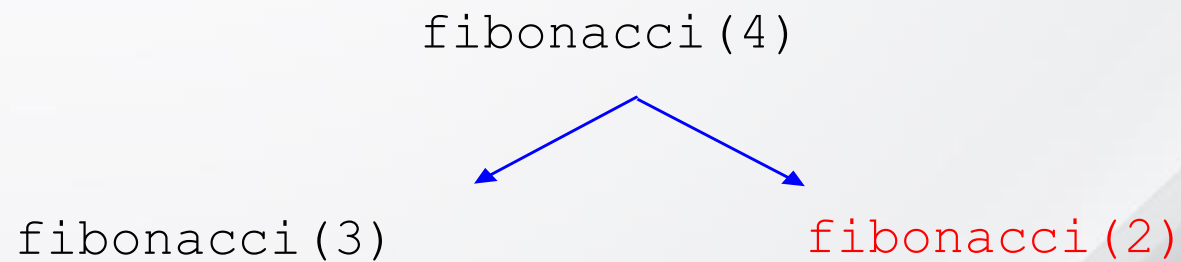
- Decomposition & Composition
 - $\text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2)$
- Base case:
 - $\text{fibonacci}(1) = 0$
 - $\text{fibonacci}(2) = 1$

fibonacci Method:

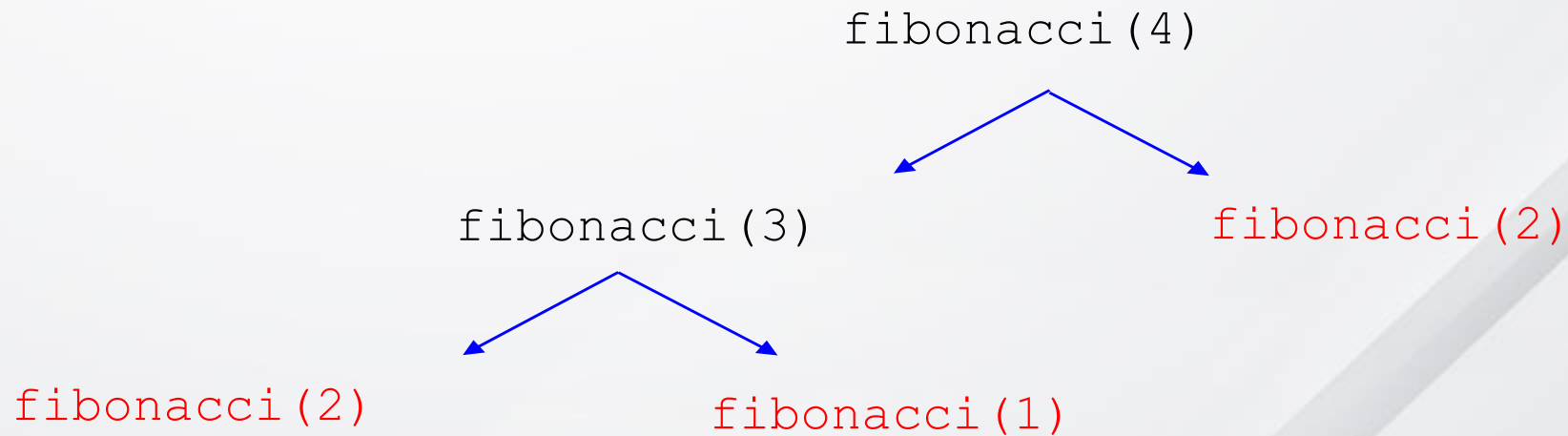
Example II: Recursive Fibonacci Algorithm

```
fibonacci(n: nonnegative integer) if n = 0 then  
    fibo(0) := 0 else if n = 1 then fibo(1) := 1  
else fibo(n) := fibo(n - 1) + fibo(n - 2)
```

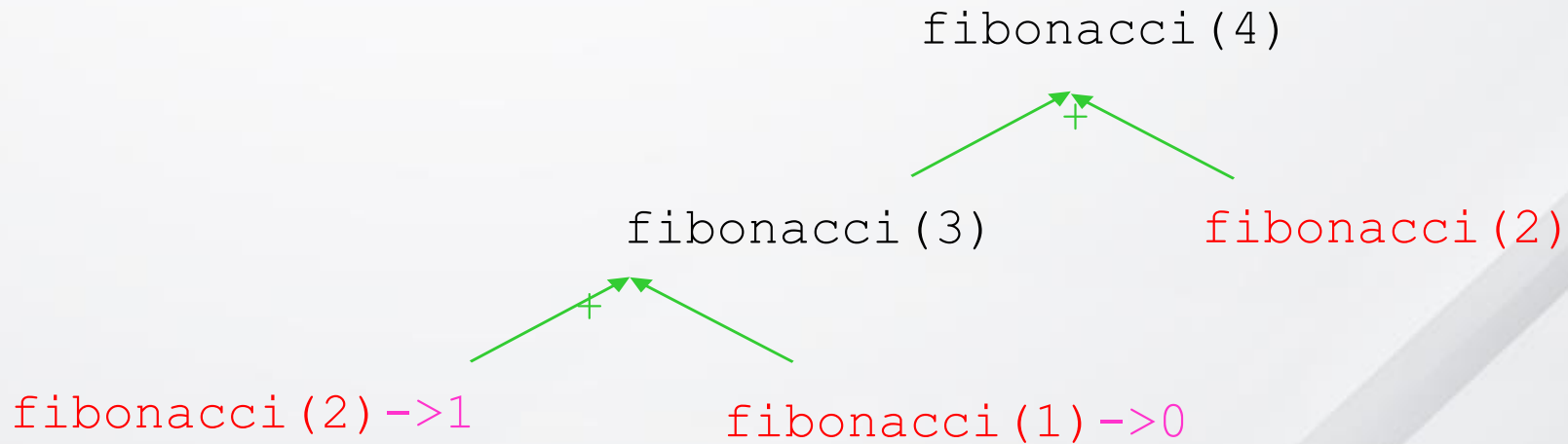
Execution Trace (decomposition)



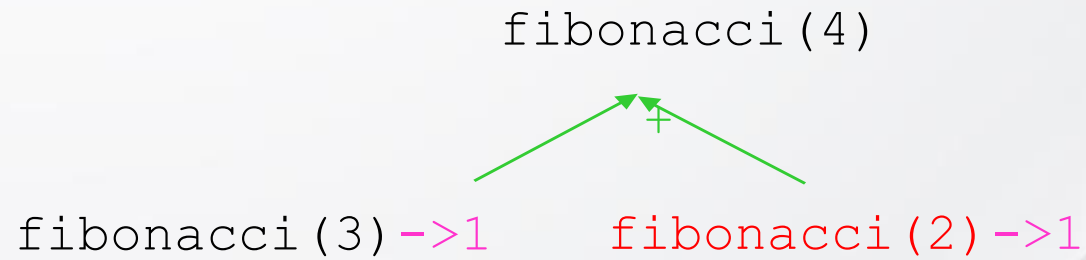
Execution Trace (decomposition)



Execution Trace (**composition**)



Execution Trace (**composition**)



Execution Trace (composition)

`fibonacci(4) -> 2`

$$T(n) = T(n-1) + T(n-2) + 1$$



Thank You