

Data Structure and Algorithm

Chapter 01

DSA, Chapter 1: Overview

- Introduction, syllabus, organisation
- Algorithms
- Recursion (principle, trace, factorial, Fibonacci)
- Sorting (bubble, insertion, selection)

What are Algorithms About?

Solving problems in everyday life

- Travel from Bolzano to Munich
- Cook Spaghetti alla Bolognese *(I know, not in Italy, ...)*
- Register for a Bachelor thesis at FUB

For all these problems, there are

- instructions
- recipes
- procedures,

which describe a complex operation in terms of

- elementary operations *("beat well ...")*
- control structures and conditions *("... until fluffy")*

Algorithms

Problems involving numbers, strings, mathematical objects:

- for two numbers, determine their sum, product, ...
- for two numbers, find their greatest common divisor
- for a sequence of strings,
find an alphabetically sorted permutation of the sequence
- for two arithmetic expressions, find out if they are equivalent
- for a program in Java,
find an equivalent program in byte code
- on a map, find for a given house the closest bus stop

We call instructions, recipes, for such problems *algorithms*

*What have algorithms in common with recipes?
How are they different?*

Data Structures, Algorithms, and Programs

- Data structure
 - Organization of data to solve the problem at hand
- Algorithm
 - Outline, the essence of a computational procedure, step-by-step instructions
- Program
 - implementation of an algorithm in some programming language

Overall Picture

Using a computer to help solve problems:

- Precisely specifying the problem
- Designing programs
 - architecture
 - algorithms
- Writing programs
- Verifying (testing) programs

Data Structure and Algorithm Design Goals

Correctness



Efficiency



Implementation Goals

Robustness



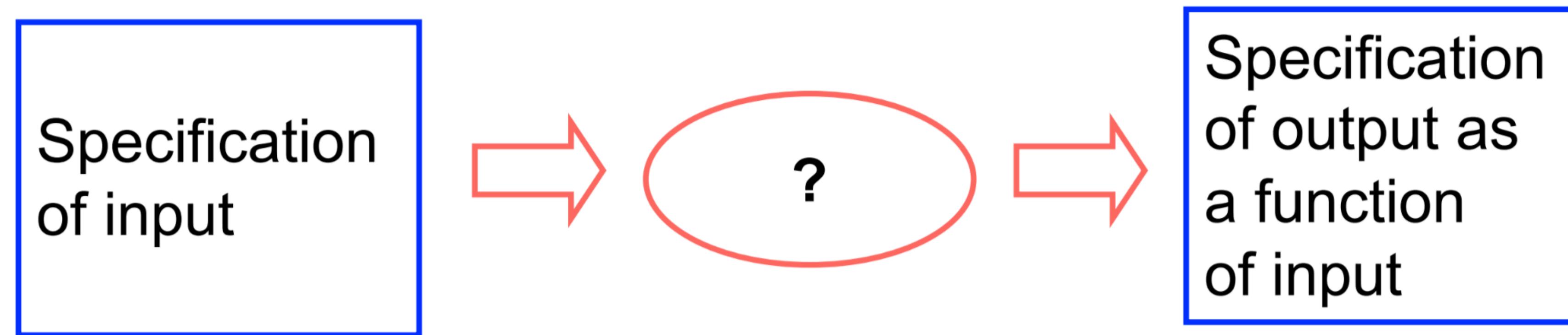
Adaptability



Reusability



Algorithmic Problem

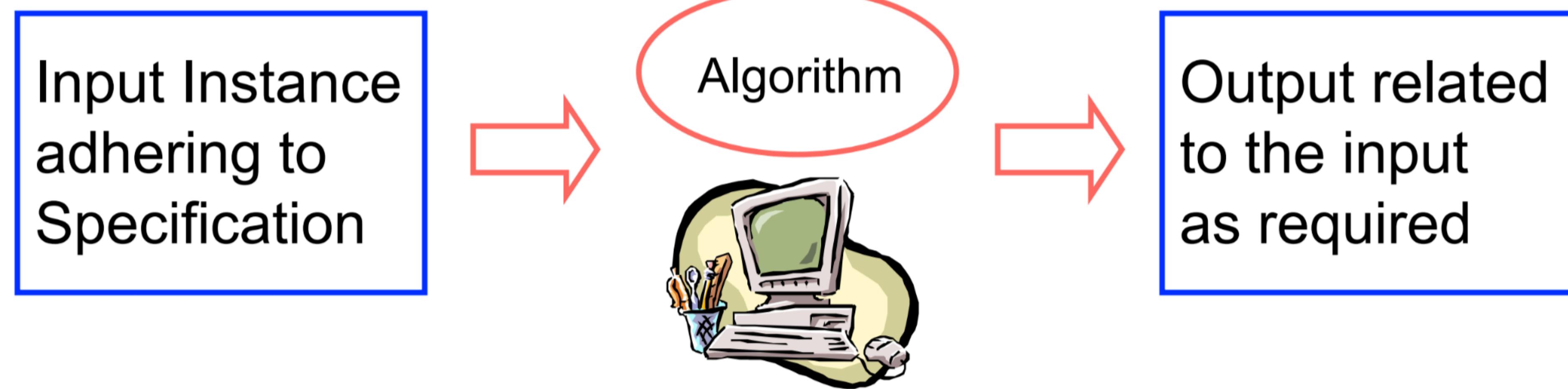


There is an infinite number of possible input *instances* satisfying the specification.

For example: A sorted, non-decreasing sequence of natural numbers, on nonzero, finite length:

1, 20, 908, 909, 100000, 1000000000.

Algorithmic Solution



- Algorithm describes actions on the input instance
- There may be many correct algorithms for the same algorithmic problem.

Definition

An **algorithm** is a sequence of *unambiguous* instructions for solving a problem, i.e.,

- for obtaining a *required output*
 - for any *legitimate input*
- in a finite amount of time.

→ This presumes a mechanism to execute the algorithm

Properties of algorithms:

- Correctness, Termination, (Non-)Determinism, Run Time, ...

How to Develop an Algorithm

- Precisely define the problem.
Precisely specify the input and output.
Consider all cases.
- Come up with a simple plan
to solve the problem at hand.
 - The plan is independent of a (programming) language
 - The precise problem specification influences the plan.
- Turn the plan into an implementation
 - The problem representation (data structure)
influences the implementation

Example 1: Searching

INPUT

- A - (un)sorted sequence of n ($n > 0$) numbers
- q - a single number

$a_1, a_2, a_3, \dots, a_n; q$



2 5 6 10 11; 5

2 5 6 10 11; 9

OUTPUT

- index of number q in sequence A, or *NIL*

j



2

NIL

Searching/2, search1

search1

INPUT: $A[1..n]$ (un)sorted array of integers, q an integer.

OUTPUT: index j such that $A[j]=q$ or NIL if $A[j] \neq q$ for all j ($1 \leq j \leq n$)

```
j := 1
while j ≤ n and A[j] ≠ q do j++
if j ≤ n then return j
else return NIL
```

- The code is written in *pseudo-code* and INPUT and OUTPUT of the algorithm are specified.
- The algorithm uses a *brute-force* technique, i.e., scans the input sequentially.

Preconditions, Postconditions

Specify preconditions and postconditions of algorithms:

Precondition:

- what does the algorithm get as input?

Postcondition:

- what does the algorithm produce as output?
- ... how does this relate to the input?

Make sure you have considered the special cases:

- empty set, number 0, pointer nil, ...

Pseudo-code

Like Java, Pascal, C, or any other imperative language

- Control structures:
(**if then else** , **while**, and **for** loops)
- Assignment: **:=**
- Array element access: **A[i]**
- Access to **element** of composite type (**record** or **object**):
A.b

Searching, Java Solution

```
import java.io.*;

class search {
    static final int n = 5;
    static int j, q;
    static int a[] = { 11, 1, 4, -3, 22 };

    public static void main(String args[]) {
        j = 0; q = 22;
        while (j < n && a[j] != q) { j++; }
        if (j < n) { System.out.println(j); }
        else { System.out.println("NIL"); }
    }
}
```

Searching, C Solution

```
#include <stdio.h>
#define n 5

int j, q;
int a[n] = { 11, 1, 4, -3, 22 };
int main() {
    j = 0; q = -2;
    while (j < n && a[j] != q) { j++; }
    if (j < n) { printf("%d\n", j); }
    else { printf("NIL\n"); }
}

// compilation: gcc -o search search.c
// execution: ./search
```

Searching/3, search2

Another idea:

Run through the array
and set a pointer if the value is found.

search2

INPUT: A[1..n] (un)sorted array of integers, q an integer.

OUTPUT: index j such that $A[j]=q$ or *NIL* if $A[j] \neq q$ for all j ($1 \leq j \leq n$)

```
ptr := NIL;  
for j := 1 to n do  
  if a[j] = q then ptr := j  
return ptr;
```

Does it work?

search1 vs search2

Are the solutions equivalent?

- No!

Can one construct an example such that, say,

- search1 returns 3
- search2 returns 7 ?

But both solutions satisfy the specification (or don't they?)

Searching/4, search3

An third idea:

Run through the array and
return the index of the value in the array.

search3

INPUT: A[1..n] (un)sorted array of integers, q an integer.

OUTPUT: index j such that $A[j]=q$ or NIL if $A[j] \neq q$ for all j ($1 \leq j \leq n$)

```
for j := 1 to n do
    if a[j] = q then return j
return NIL
```

Comparison of Solutions

Metaphor: shopping behavior when buying a beer:

- **search1**: scan products;
stop as soon as a beer is found and go to the exit.
- **search2**: scan products until you get to the exit;
if during the process you find a beer,
put it into the basket
(instead of the previous one, if any).
- **search3**: scan products;
stop as soon as a beer is found
and exit through next window.

Comparison of Solutions/2

- **search1** and **search3** return *the same result* (index of the **first occurrence** of the search value)
- **search2** returns the index of the **last occurrence** of the search value
- **search3 does not finish the loop** (as a general rule, you better avoid this)

Beware: Array Indexes in Java/C/C++

- In pseudo-code, array indexes range from **1** to **length**
- In Java/C/C++, array indexes range from **0** to **length-1**
- Examples:

– Pseudo-code

```
for j := 1 to n do
```

Java:

```
for (j=0; j < a.length; j++) { ...}
```

– Pseudo-code

```
for j := n to 2 do
```

Java:

```
for (j=a.length-1; j >= 1; j--) { ...}
```

Suggested Exercises

- Implement the three variants of search
 - Create random arrays for different lengths
 - Compare the results
 - Add a counter for the number of cycles and return it, compare the result
- Implement them to scan the array in reverse order

Recursion

An object is **recursive** if

- a part of the object **refers to the entire object**, or
- one part **refers to another part** and **vice versa**

(mutual recursion)

Recursion/2

- A **recursive definition**: a concept is defined by referring to itself.

E.g., arithmetical expressions (like $(3 * 7) - (9 / 3)$):

$$\text{EXPR} := \text{VALUE} \mid (\text{EXPR} \text{ OPERATOR } \text{EXPR})$$

- A **recursive procedure**: a procedure that calls itself

Classical example: **factorial**, that is $n! = 1 * 2 * 3 * \dots * n$

$$n! = n * (n-1)!$$

... or is there something missing?

The Factorial Function

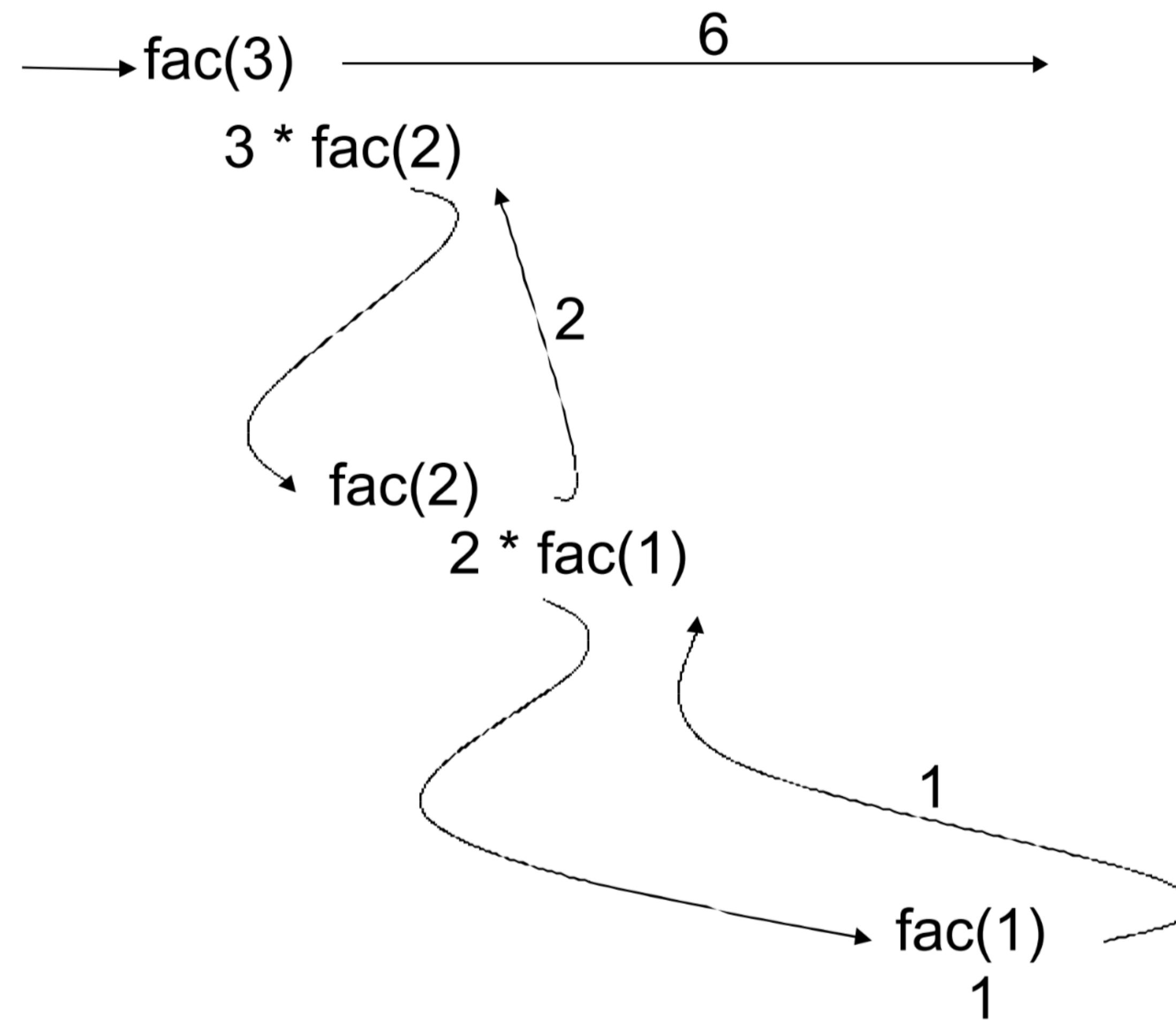
Pseudocode of factorial:

```
fac1  
INPUT: n - a natural number.  
OUTPUT: n! (factorial of n)  
  
fac1(n)  
  if n < 2 then return 1  
  else return n * fac1(n-1)
```

This is a recursive procedure. A recursive procedure has

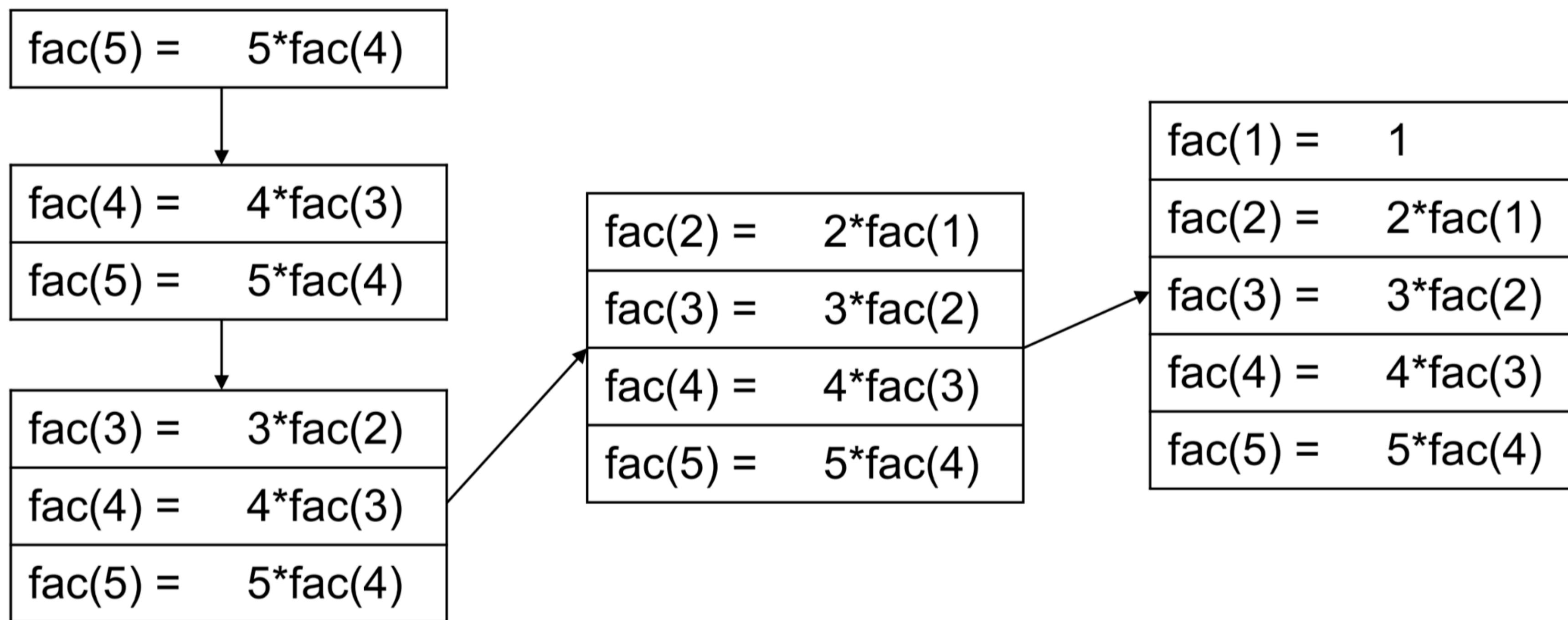
- a termination condition
(determines when and how to stop the recursion).
- one (or more) recursive calls.

Tracing the Execution



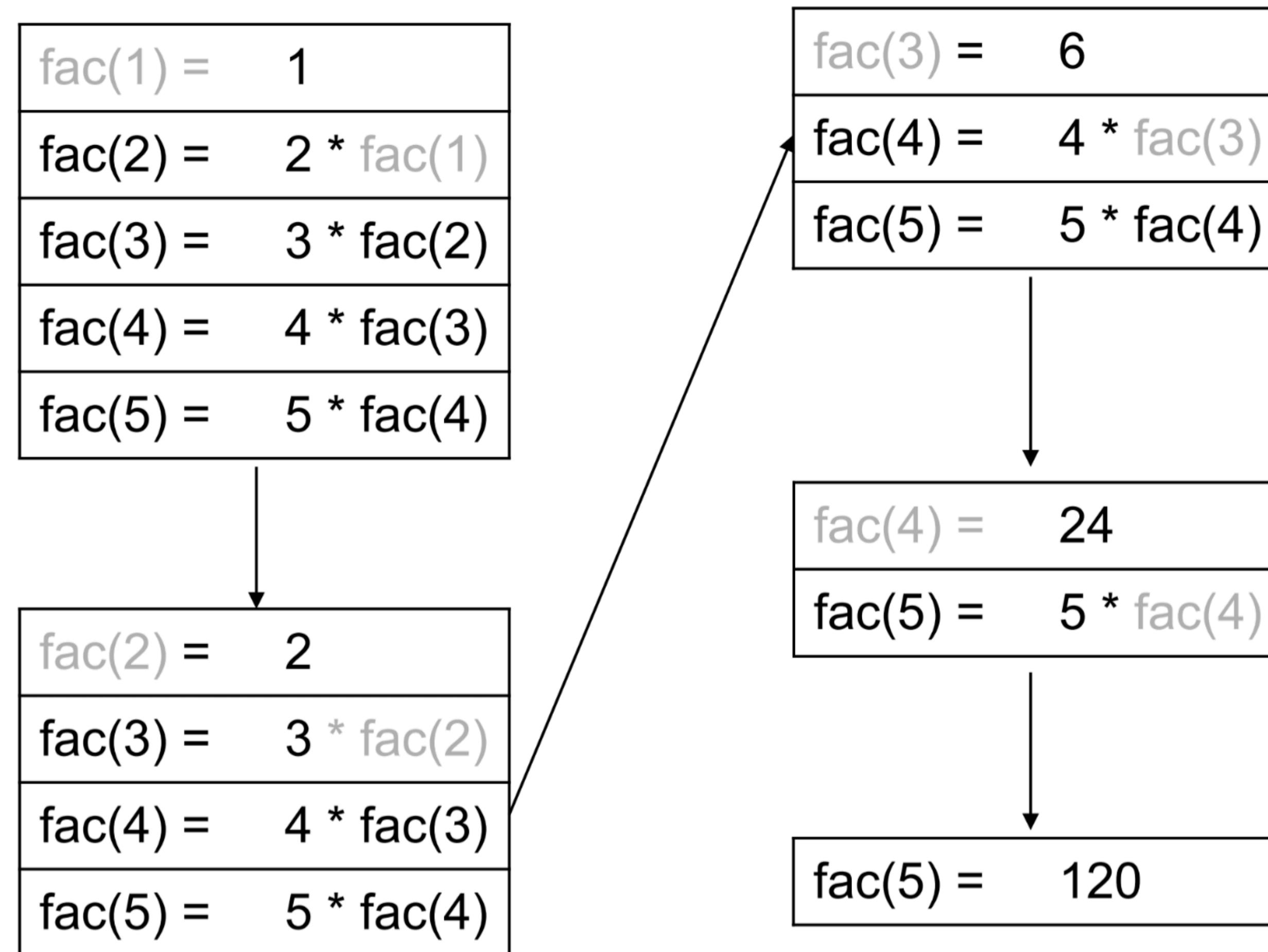
Bookkeeping

The computer maintains an **activation stack** for active procedure calls (\rightarrow compiler construction).
Example for $\text{fac}(5)$. The stack is built up.



Bookkeeping/2

Then the activation stack is reduced



Variants of Factorial

fac2

INPUT: n - a natural number.

OUTPUT: n! (factorial of n)

fac2(n)

if n = 0 **then return** 1

return n * fac2(n-1)

fac3

INPUT: n - a natural number.

OUTPUT: n! (factorial of n)

fac3(n)

if n = 0 **then return** 1

return n * (n-1) * fac3(n-2)

Analysis of the Variants

fac2 is correct

- The return statement in the if clause terminates the function and, thus, the entire recursion.

fac3 is incorrect

- Infinite recursion.
- The termination condition is never reached if n is odd:

$$\begin{aligned}\text{fact}(3) &= 3 * 2 * \text{fact}(1) \\ &= 3 * 2 * 1 * 0 * \text{fact}(-1) \\ &= \dots\end{aligned}$$

Variants of Factorial/2

fac4

INPUT: n – a natural number.

OUTPUT: n! (factorial of n)

fac4(n)

if n <= 1 **then return** 1

return n * (n-1) * fac4(n-2)

fac5

INPUT: n – a natural number.

OUTPUT: n! (factorial of n)

fac5(n)

return n * fac5(n-1)

if <= 1 **then return** 1

Analysis of the Variants/2

fac4 is correct

- The return statement in the if clause terminates the function and, thus, the entire recursion.

fac5 is incorrect

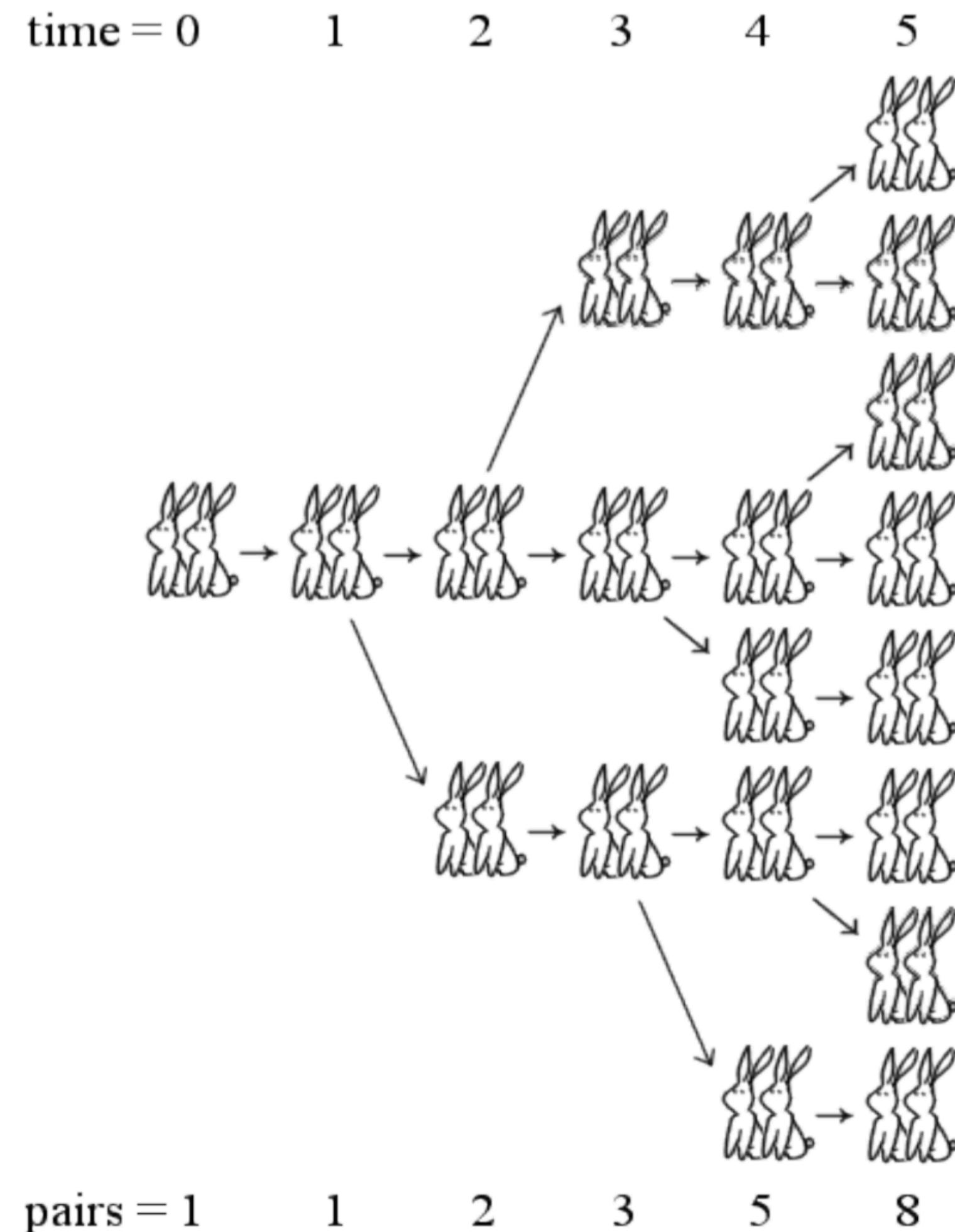
- Infinite recursion.
The termination condition is never reached.

Counting Rabbits

*Someone placed a pair of rabbits
in a certain place,
enclosed on all sides by a wall,
so as to find out
how many pairs of rabbits
will be born there in the course of one year,
it being assumed
that every month
a pair of rabbits produces another pair,
and that rabbits begin to bear
young two months after their own birth.*

*Leonardo di Pisa ("Fibonacci"),
Liber abacci, 1202*

Counting Rabbits/2



Source: <http://www.jimloy.com/algebra/fibo.htm>

Fibonacci Numbers

Definition

- $\text{fib}(1) = 1$
- $\text{fib}(2) = 1$
- $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2), n > 2$

Numbers in the series:

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Fibonacci Procedure

fib

INPUT: n – a natural number larger than 0.

OUTPUT: fib(n), the nth Fibonacci number.

fib(n)

if n ≤ 2 **then return** 1

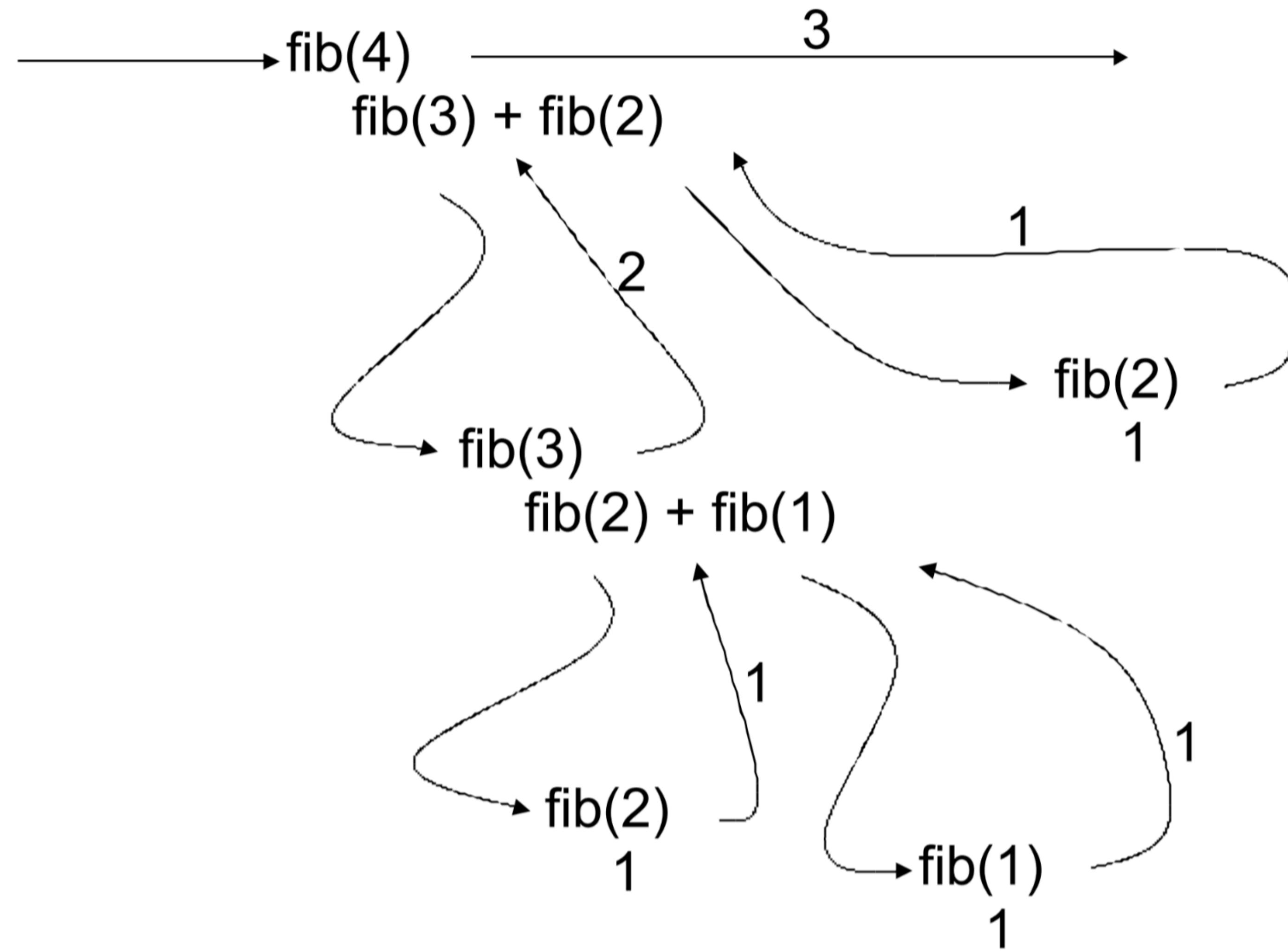
else return fib(n-1) + fib(n-2)

A procedure with **multiple** recursive calls

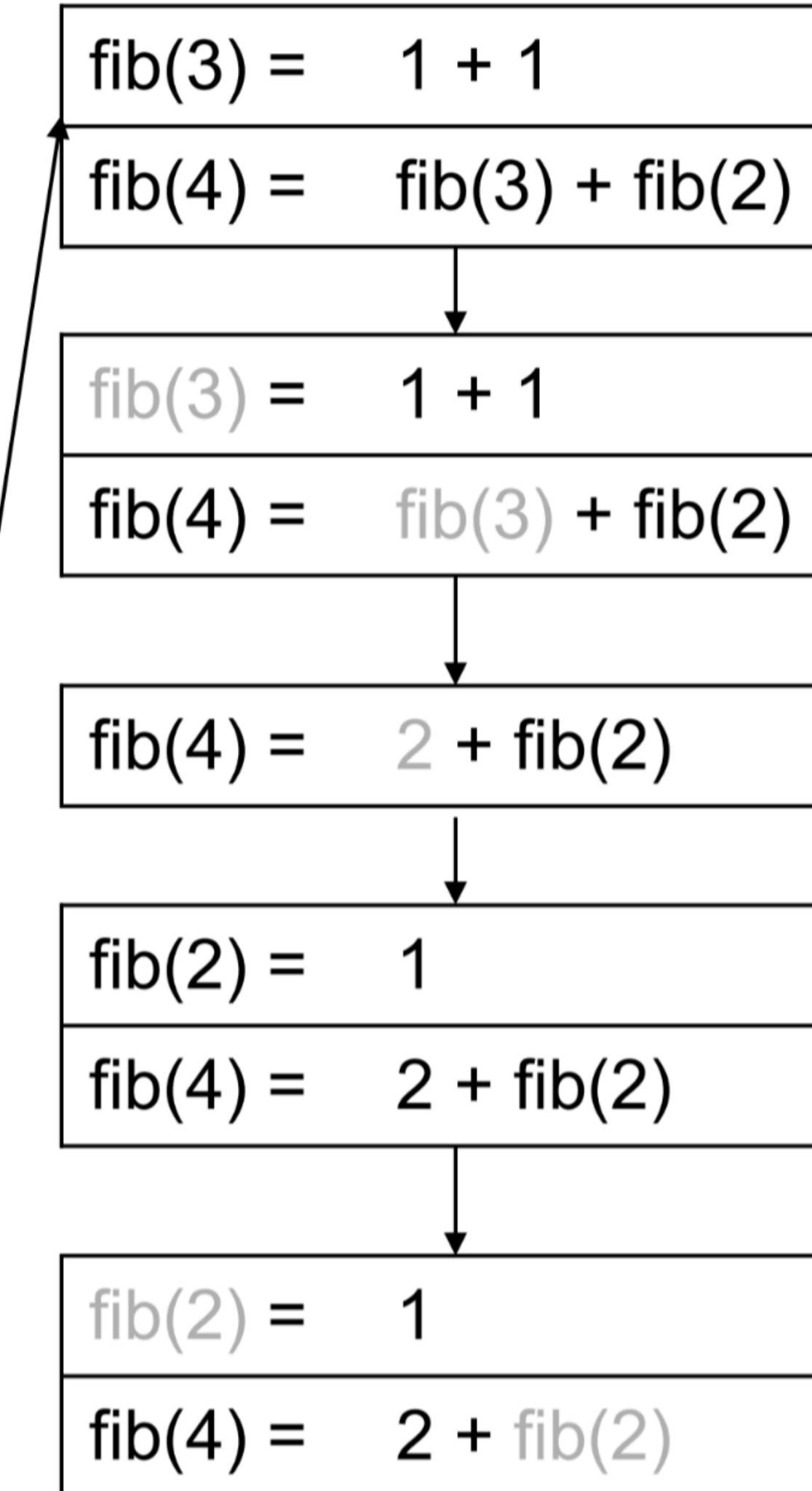
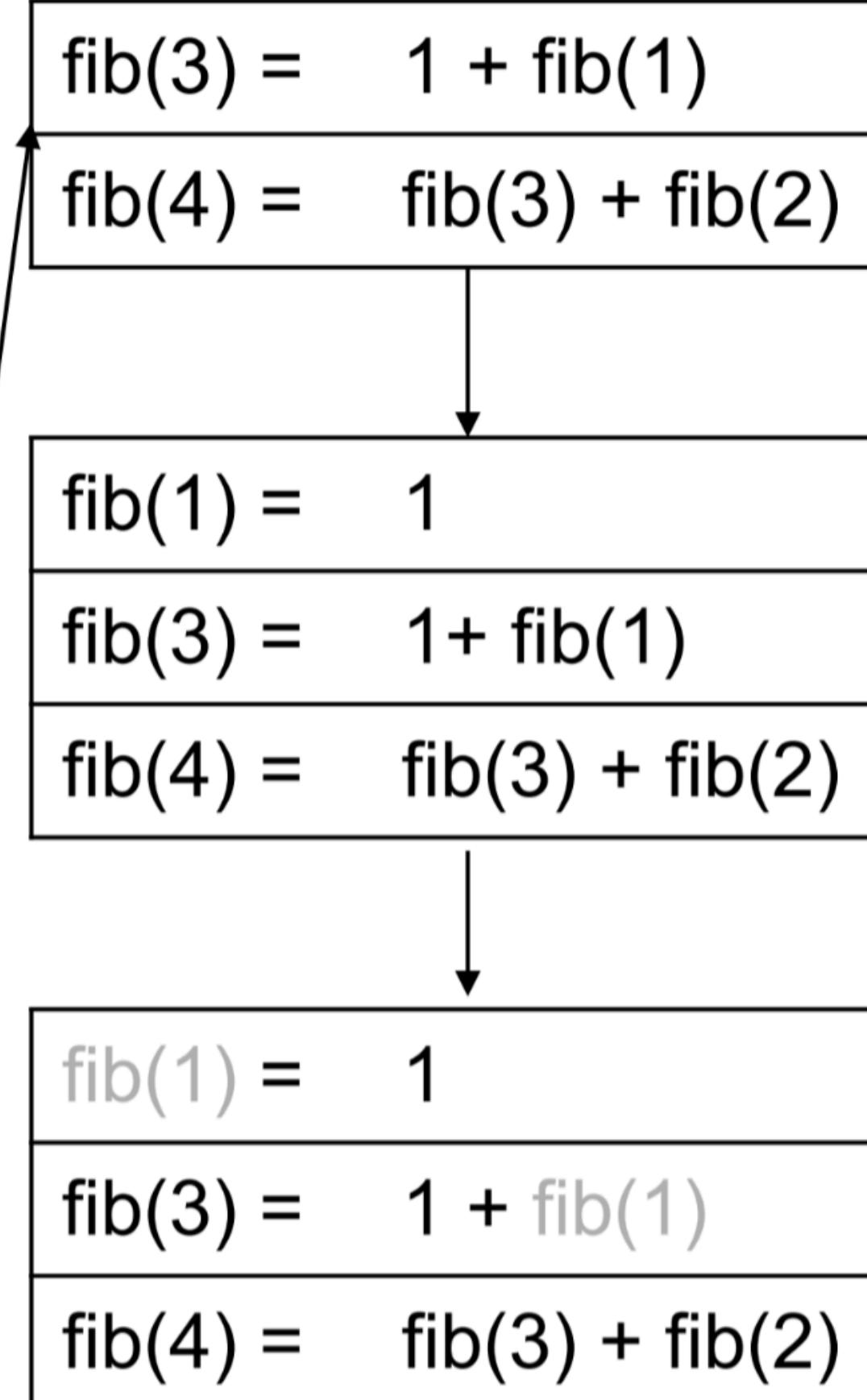
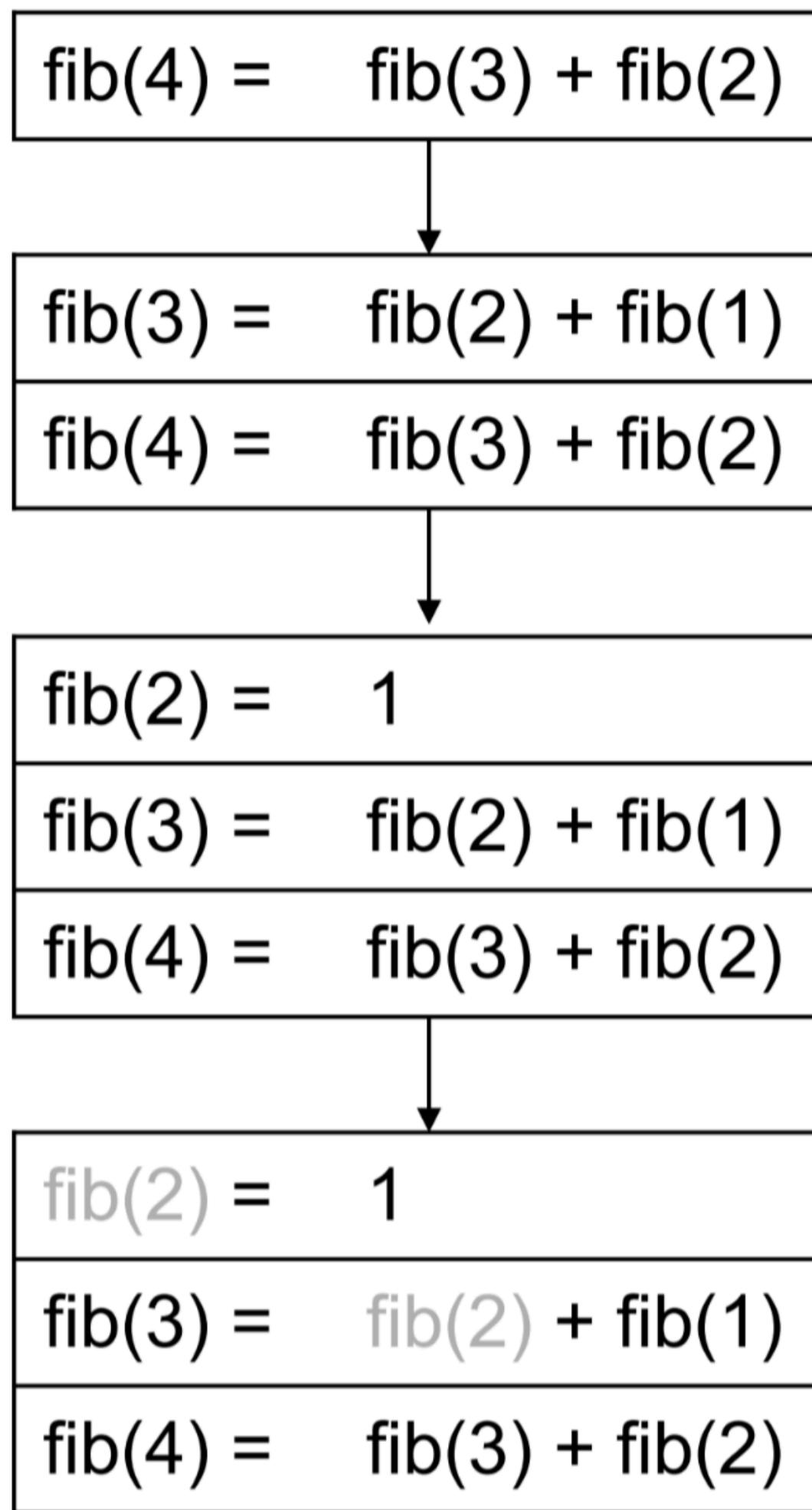
Fibonacci Procedure/2

```
public class fibclassic {  
  
    static int fib(int n) {  
        if (n <= 2) {return 1;}  
        else {return fib(n - 1) + fib(n - 2);}  
    }  
  
    public static void main(String args[]) {  
        System.out.println("Fibonacci of 5 is "  
                           + fib(5));  
    }  
}
```

Tracing fib(4)



Bookkeeping



Questions

- How many recursive calls are made to compute $\text{fib}(n)$?
- What is the maximal height of the recursion stack during the computation?
- How are number of calls and height of the stack related to the size of the input?
- Can there be a procedure for fib with fewer operations?
- How is the size of the result $\text{fib}(n)$ related to the size of the input n ?

Mutual Recursion Example

- Problem: Determine whether a natural number is even
- Definition of even:
 - 0 is even
 - N is even if $N - 1$ is odd
 - N is odd if $N - 1$ is even

Implementation of even

even

INPUT: n – a natural number.

OUTPUT: true if n is even; false otherwise

odd(n)

```
if n = 0 then return FALSE  
return even(n-1)
```

even(n)

```
if n = 0 then return TRUE  
else return odd(n-1)
```

- How can we determine whether N is odd?

Is Recursion Necessary?

- Theory: You can always resort to iteration and explicitly maintain a recursion stack.
- Practice: Recursion is elegant and in some cases the best solution by far.
- In the previous examples recursion was never appropriate since there exist simple iterative solutions.
- Recursion is more expensive than corresponding iterative solutions since bookkeeping is necessary.

Sorting

- Sorting is a classical and important algorithmic problem.
 - For which operations is sorting needed?
 - Which systems implement sorting?
- We look at sorting **arrays**
(in contrast to files, which restrict random access)
- A key constraint are the restrictions on the **space**:
in-place sorting algorithms (no extra RAM).
- The **run-time comparison** is based on
 - the number of **comparisons** (C) and
 - the number of **movements** (M).

Sorting

- Simple sorting methods use roughly $n * n$ comparisons
 - Insertion sort
 - Selection sort
 - Bubble sort
- Fast sorting methods use roughly $n * \log n$ comparisons
 - Merge sort
 - Heap sort
 - Quicksort

What's the point of studying those simple methods?

Example 2: Sorting

INPUT

sequence of n numbers

$a_1, a_2, a_3, \dots, a_n$

2 5 4 10 7



OUTPUT

a permutation of the input sequence of numbers

$b_1, b_2, b_3, \dots, b_n$

2 4 5 7 10



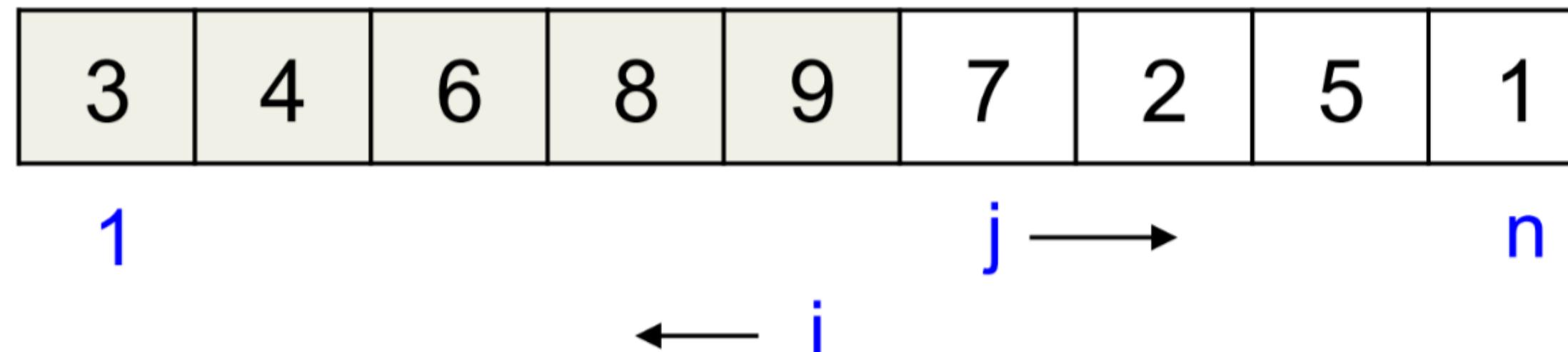
Correctness (requirements for the output)

For any given input the algorithm halts with the output:

- $b_1 \leq b_2 \leq b_3 \leq \dots \leq b_n$
- $b_1, b_2, b_3, \dots, b_n$ is a permutation of $a_1, a_2, a_3, \dots, a_n$

Insertion Sort

A



Strategy

- Start with one sorted card.
- Insert an unsorted card at the correct position in the sorted part.
- Continue until all unsorted cards are inserted/sorted.

44	55	12	42	94	18	06	67
44	55	12	42	94	18	06	67
12	44	55	42	94	18	06	67
12	42	44	55	94	18	06	67
12	42	44	55	94	18	06	67
12	18	42	44	55	94	06	67
06	12	18	42	44	55	94	67
06	12	18	42	44	55	67	94

Insertion Sort/2

INPUT: A[1..n] – an array of integers

OUTPUT: permutation of A s.t. A[1]≤A[2]≤...≤A[n]

```
for j := 2 to n do // A[1..j-1] sorted
    key := A[j]; i := j-1;
    while i > 0 and A[i] > key do
        A[i+1] := A[i]; i--;
    A[i+1] := key
```

The number of comparisons during the j th iteration is

– at least 1: $C_{\min} = \sum_{j=2}^n 1 = n - 1$

– at most $j-1$: $C_{\max} = \sum_{j=2}^n (j-1) = (n^*n - n)/2$

Insertion Sort/2

INPUT: A[1..n] – an array of integers

OUTPUT: permutation of A s.t. A[1]≤A[2]≤...≤A[n]

```
for j := 2 to n do // A[1..j-1] sorted
    key := A[j]; i := j-1;
    while i > 0 and A[i] > key do
        A[i+1] := A[i]; i--;
    A[i+1] := key
```

The number of comparisons during the j th iteration is

– at least 1: $C_{\min} = \sum_{j=2}^n 1 = n - 1$

– at most $j-1$: $C_{\max} = \sum_{j=2}^n (j-1) = (n^*n - n)/2$