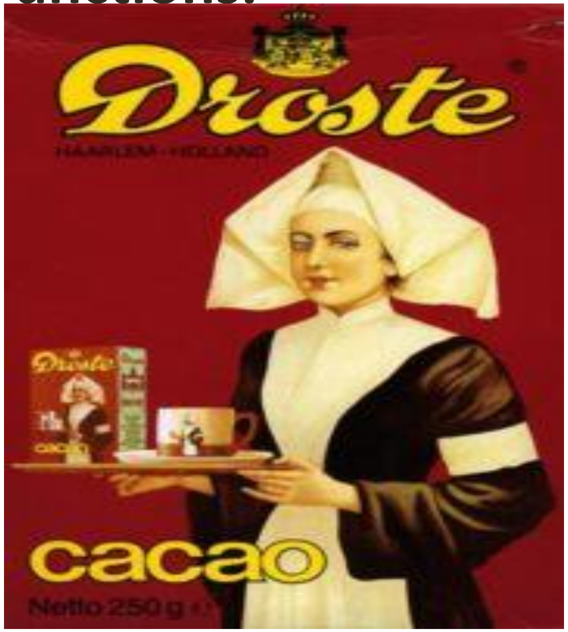
CHAPTER 5: RECURSION

BIBHA STHAPIT
ASST. PROFESSOR
IoE, PULCHOWK CAMPUS

Recursion

- Function may call itself
- Function may call other function and the other function in turn again may call the calling function
- Such functions are called as **recursive Functions.**
- An object contains itself



Recursion : example

Iterative algorithm

$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1 & \text{if } n > 0 \end{cases}$$

Recursive algorithm

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

Iterative Solution

Algorithm iterativeFactorial (n <integer>)

Calculates the factorial of a number using a loop

Pre n is the number to be raised factorially

Return n! is returned

1. i = 1
2. factN = 1
3. loop (i <= n)
    1. factN = factN *i
    2. i = i + 1
4. return factN

End iterativeFactorial

Recursive Solution

Algorithm recursiveFactorial (n <integer>)

Calculates the factorial of a number using recursion

Pre n is the number to be raised factorially

Return n! is returned

1. if (n = 0)

   factN = 1

2. else

   factN = n × recursiveFactorial(n – 1)

3. return factN

End recursiveFactorial

Recursive Solution

Factorial(3) = 3 × Factorial(2)

Factorial(2) = 2 × Factorial(1)

Factorial(1) = 1 × Factorial(0)

Factorial(0) = 1

Factorial(1) = 1 × 1 = 1

Factorial(2) = 2 × 1 = 2

Factorial(3) = 3 × 2 = 6

Recursive solution

- Every recursive solution has two cases:

- 1. Base case
  - In which the problem is simple enough to be solved directly without further call to same function
  - Eg: In factorial, factorial(1)=1 for n=1

- 2. Recursive case
  - In which the function calls itself with simpler sub-parts.
  - Eg: factorial(n)=n $\times$ factorial(n – 1)

Variants of recursion

- Depending on the characterization, the recursive functions are categorized as :

  - **Direct and indirect,**
  - **Linear and tree,**
  - **Non-tail and tail recursions**

Direct and indirect recursion

- Recursion - *when function calls itself*

- *Recursion is said to be direct when functions calls itself* directly and is said to be *indirect when it calls other function that in turn calls it*

- The function factorial we studied is an example of direct recursion

Tail and Non-Tail recursion

- A recursive function is said to be **tail recursive** if there are no pending operations tobe performed on return from a recursive call

- Tail recursion is also used to return the value of the last recursive call as the value of the function

- Tail recursion is advantageous as the amountof information which must be stored on the system stack during computation is independent of the number of recursive calls

Tail and Non-Tail recursion

- The function in **non-tail recursive** whenever there is pending operation to be performed.

- Since the information about each pending operation must be stored, the amount of information directly depends on the number of calls.

Tail and Non-Tail recursion

- Non-tail recursion

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

}
```

- Tail recursion

```
int fact(int n)
  {
          return tail_fact(n,1);
   }

int tail_fact(int n, int res)
{
if (n==1)
          return res;
else
          return tail_fact(n-1,n*res);
}
```

Linear and tree recursion

- Depending on the way in which recursion grows is classified as linear or tree

- A recursive function is said to be linearly recursive when no pending operation involves another recursive call

- The simplest form of recursion is *linear recursion*. *It occurs* where an action has a simple repetitive structure consisting of some basic step followed by the action again

- Factorial function is example of linear recursion

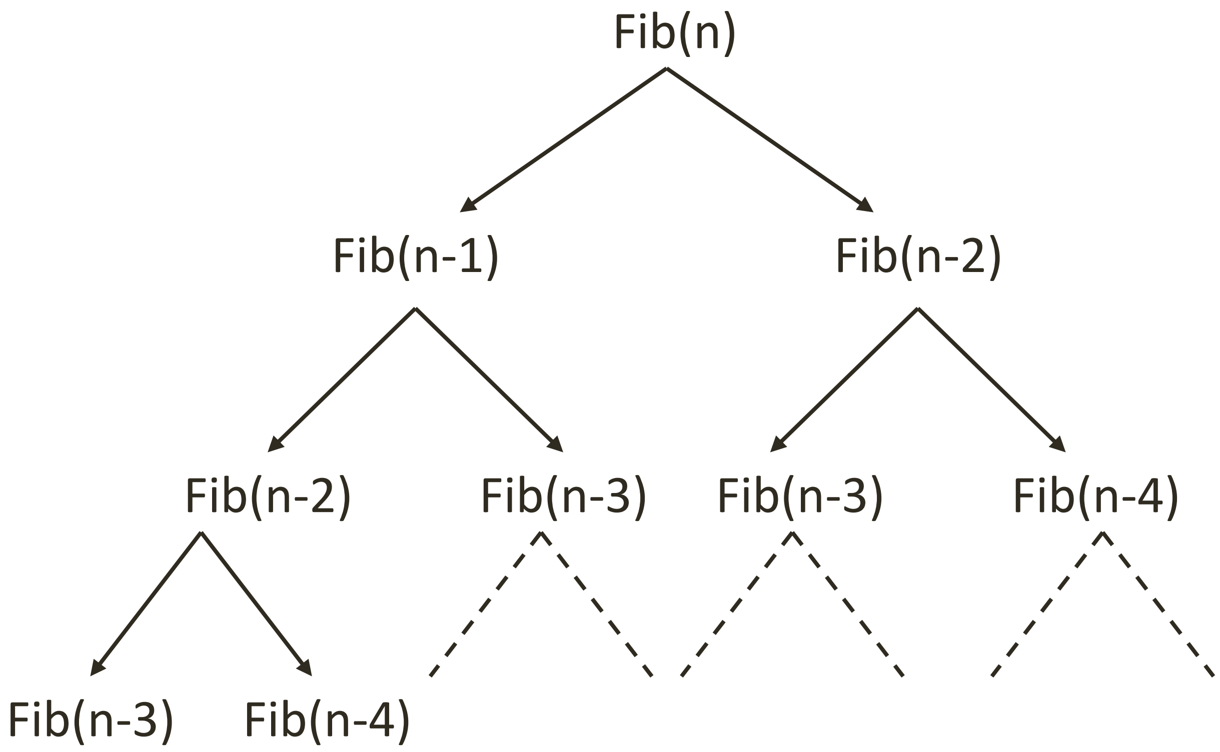
Linear and tree recursion

- In recursive function, if there is another set of operations to be completed after the recursion is over, then the recursive call is called a *tree recursion*
- Fibonacci function is the example of tree recursion

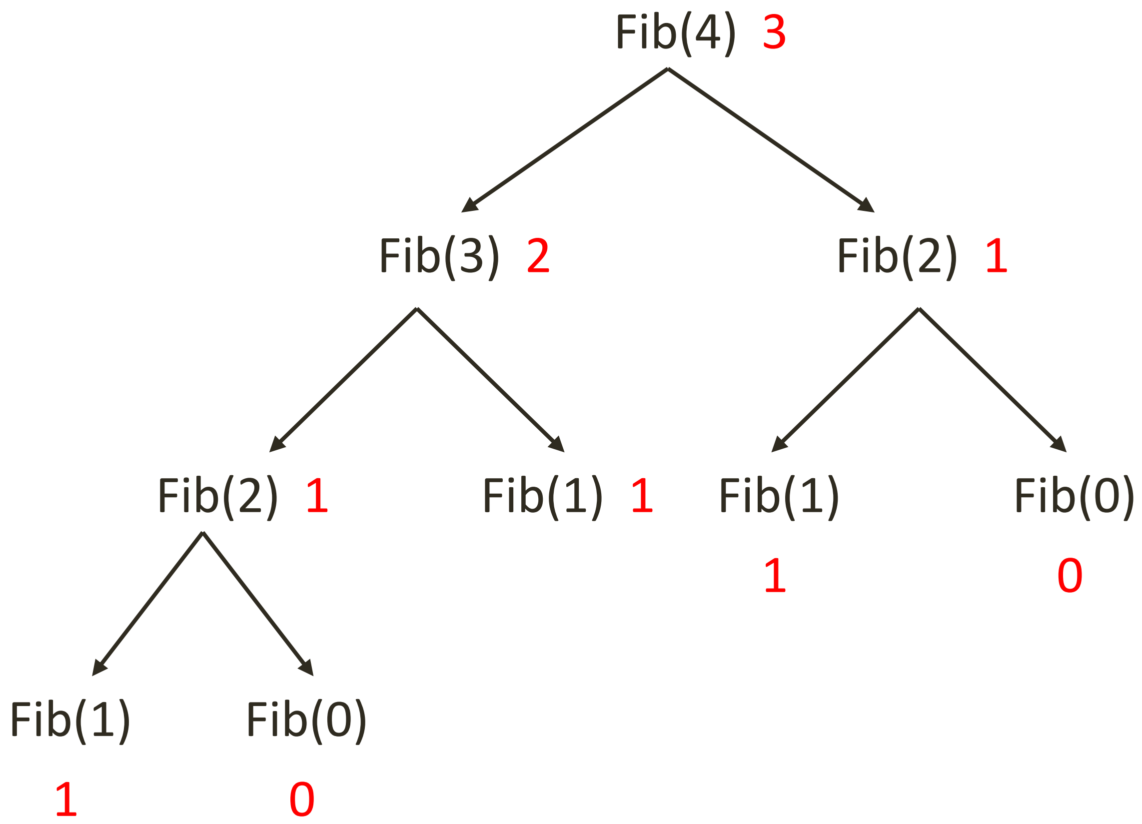
Fibonacci numbers

0 1 1 2 3 5 8 13 21 34

- Recursive case: Fib(n) = Fib(n – 1) + Fib(n – 2)
- Stopping case: Fib(0) = 0 Fib(1) = 1



Fibonacci numbers



Fibonacci Numbers

Algorithm fib (num <integer>)

Calculates the nth Fibonacci number

Pre num is the ordinal of the Fibonacci number

Post returns the nth Fibonacci number

if (num = 0 OR num = 1) stopping case

return num

return (fib(n - 1) + fib(n - 2)) recursive case

End fib

Fibonacci Numbers

No	Calls	Time	No	Calls	Time
1	1	< 1 sec.	11	287	< 1 sec.
2	3	< 1 sec.	12	465	< 1 sec.
3	5	< 1 sec.	13	753	< 1 sec.
4	9	< 1 sec.	14	1,219	< 1 sec.
5	15	< 1 sec.	15	1,973	< 1 sec.
6	25	< 1 sec.	20	21,891	< 1 sec.
7	41	< 1 sec.	25	242,785	1 sec.
8	67	< 1 sec.	30	2,692,573	7 sec.
9	109	< 1 sec.	35	29,860,703	1 min.
10	177	< 1 sec.	40	331,160,281	< 13 min.

Execution of recursive calls

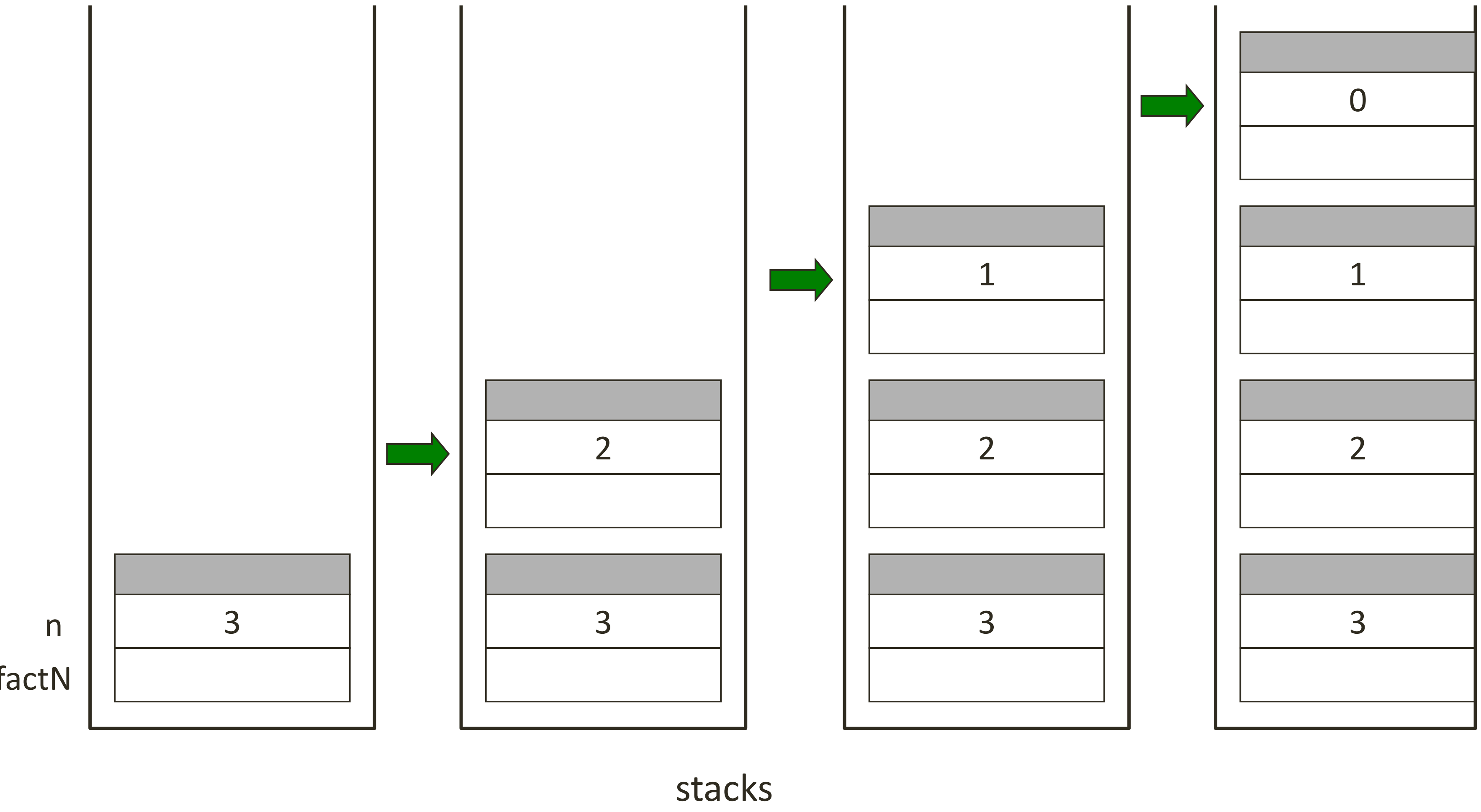
- At every recursive call, all reference parameters and local variables are pushed onto the stack along with function value and return address
- These data are conceptually placed in a **stack frame is pushed onto the** system stack
- A stack frame contains four different elements:
    - The reference parameters to be processed by the called function
    - Local variables in the calling function
    - The return address
    - The expression that is to receive the return value, if any

Execution of recursive calls

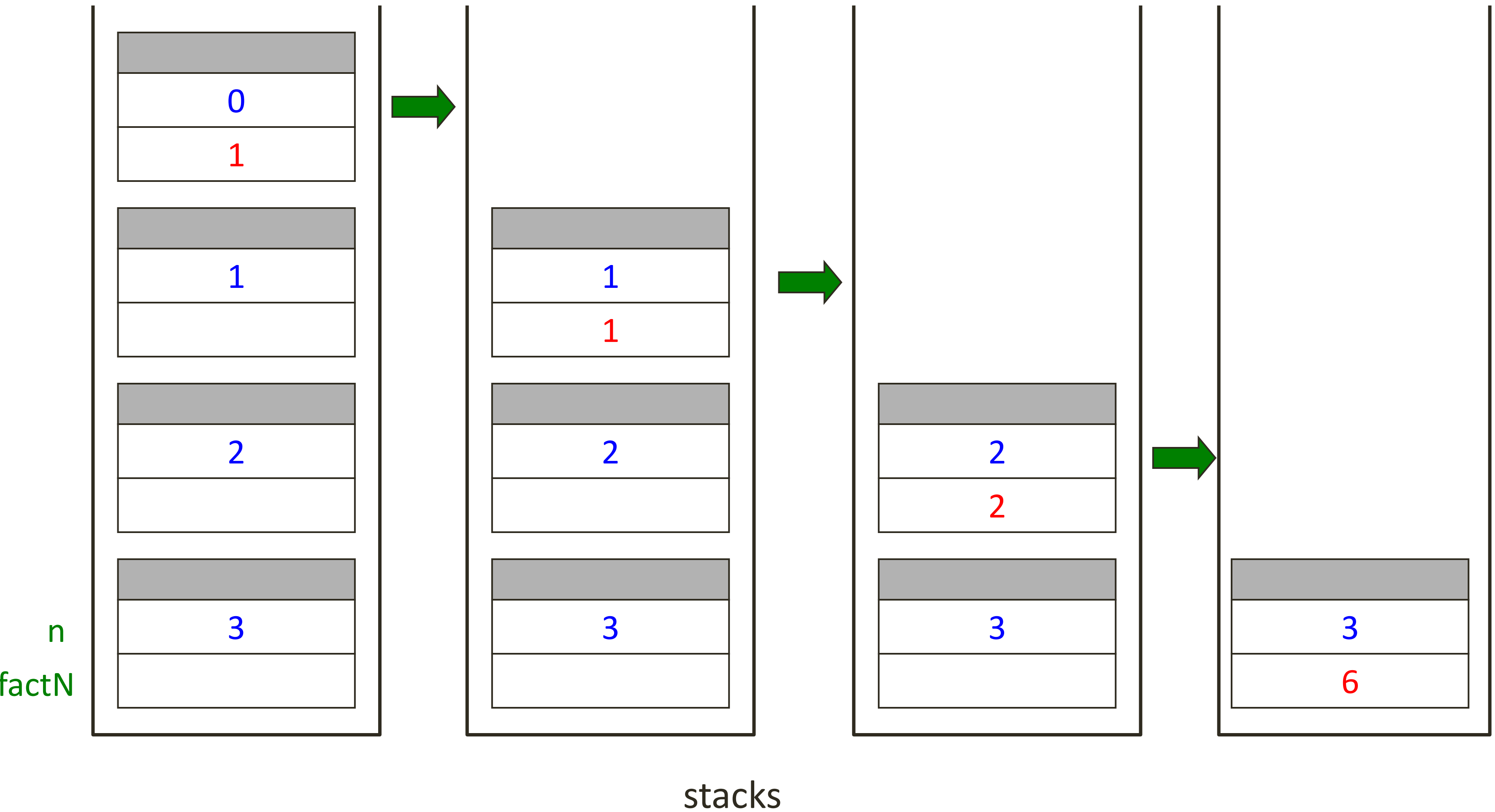
- At the end condition, when no more recursive calls are made, the following steps are performed

- If the stack is empty then execute a normal return
- Otherwise POP the stack frame, that is, take the values of all parameters which are on the top of the stack and assign these values to the corresponding variables
- **Use the return address to locate the place where the call was made**
- Execute all the statements from that place (address) where the call was made

Recursive Solution



Recursive Solution



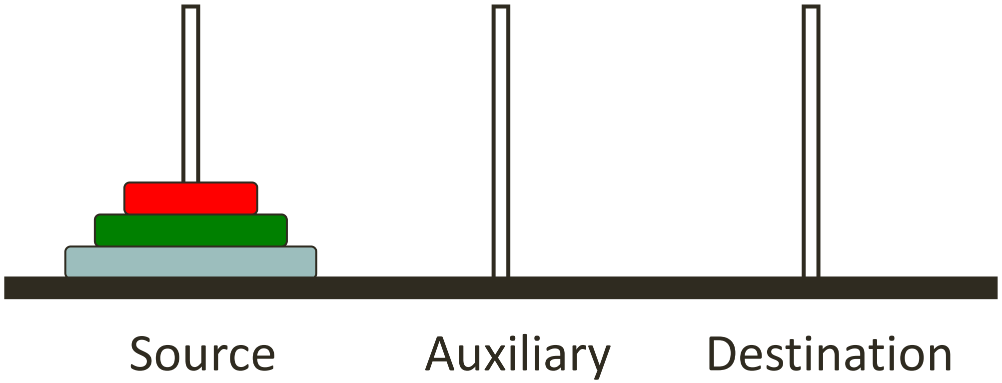
Writing recursive functions

- Recursion is most of the times viewed as a somewhat mystical technique which only is useful for some very special class of problems, such as computing factorials or Fibonacci series
- Practically any function that is written using iterative code can be converted into recursive code
- Of course, this does not guarantee that the resulting program will be easy to understand but often the program results in a compact and readable code

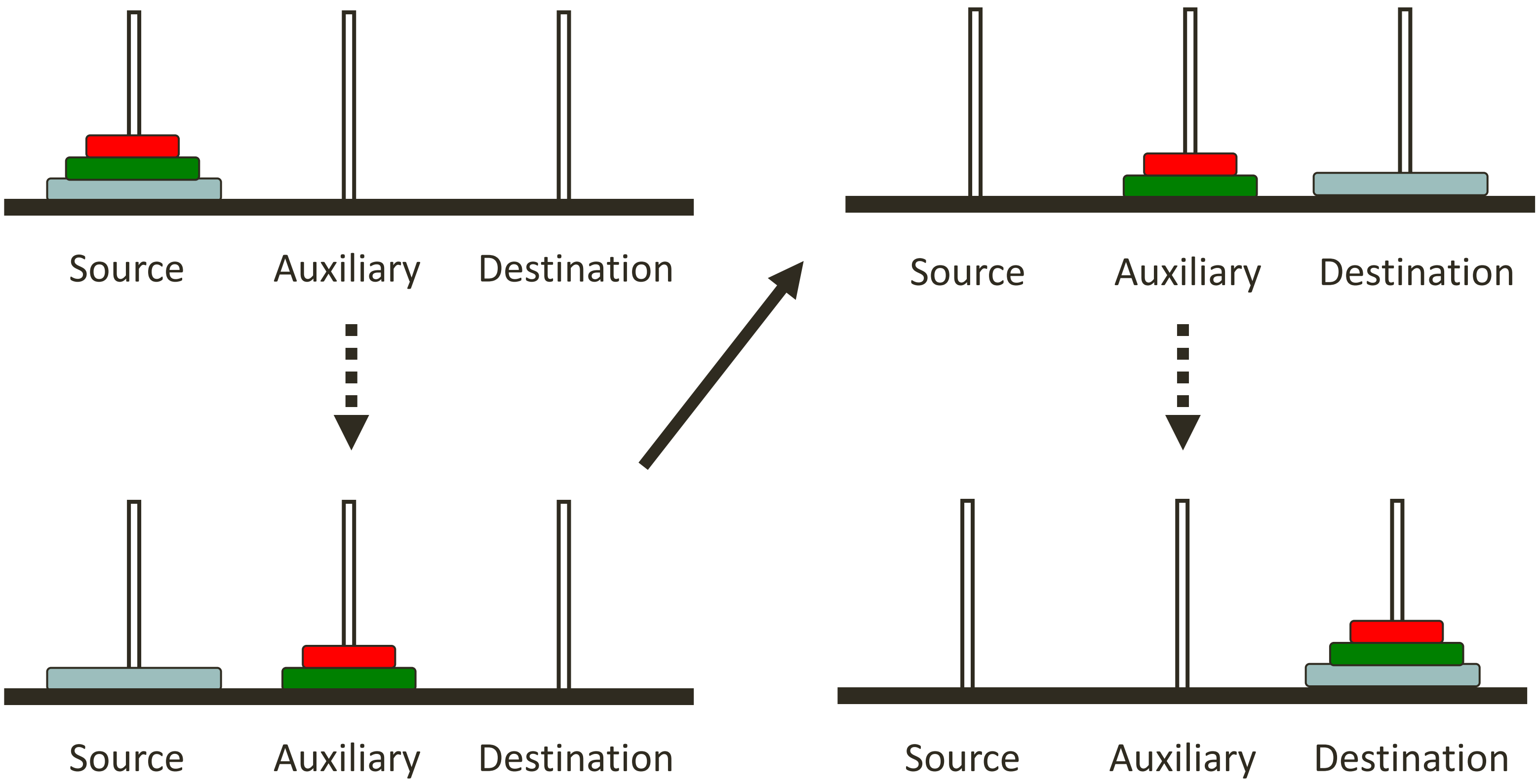
The Towers of Hanoi

Problem statement:

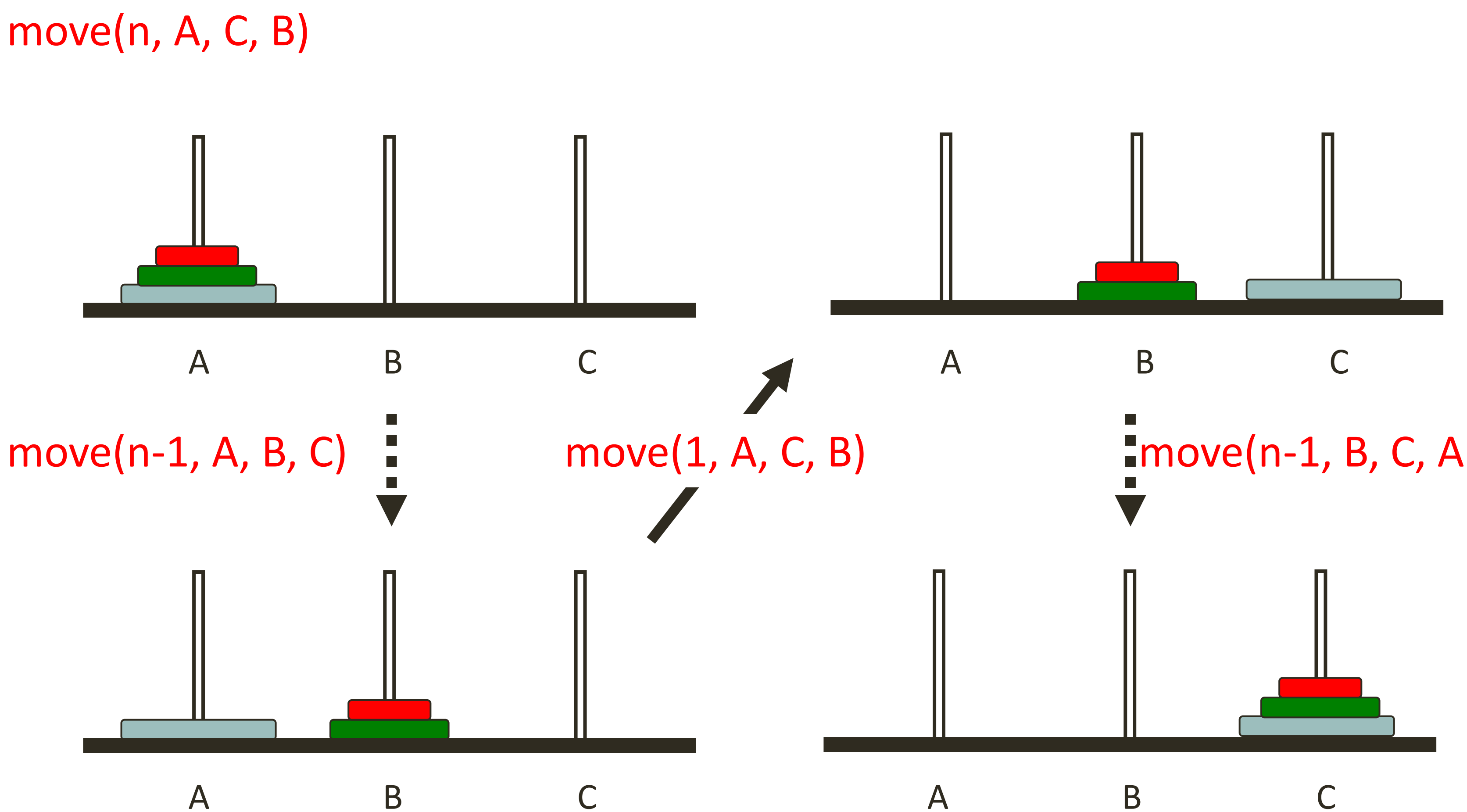
1. Only one disk could be moved at a time.
2. A larger disk must never be stacked above a smaller one.
3. Only one auxiliary needle could be used for the intermediate storage of disks.



The Towers of Hanoi



The Towers of Hanoi



The Towers of Hanoi

move(3, A, C, B)

move(2, A, B, C)

move(1, A, C, B)

4. A → C

move(2, B, C, A)

move(1, A, C, B)

1. A → C

move(1, A, B, C)

2. A → B

move(1, C, B, A)

3. C → B

move(1, B, A, C)

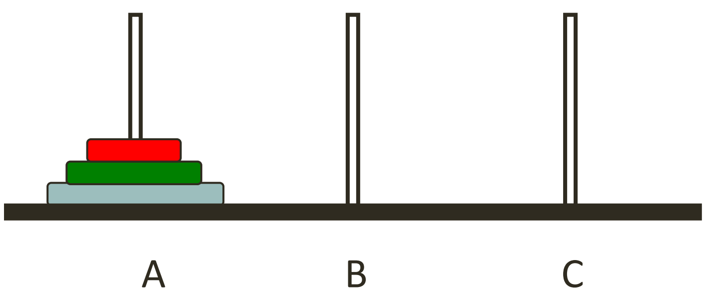
5. B → A

move(1, B, C, A)

6. B → C

move(1,A , C, B)

7. A → C



The Towers of Hanoi

- Base case: if n=1
  - Move the ring from A to C using B as auxiliary

- Recursive case:
  - Move n-1 rings from A to B using C as auxiliary
  - Move the remaining one ring from A to C
  - Move n-1 rings from B to C using A as auxiliary

The Towers Algorithm

Algorithm move(disks <integer>, source <character>, dest <character>, auxiliary <character>)

Pre disks is the number of disks to be moved

Post moves printed

```
if (disks = =1)
    print ("Move from", source, "to", dest)
else
    move (disks - 1, source, auxiliary, dest)
    move (1, source, dest, auxiliary)
    move (disks - 1, auxiliary, dest, source)
```

End move

Iteration vs. recursion

- Recursion is a top down approach to problem solving. It divides the problem into pieces or selects out one key step, postponing the rest.
-
- **Iteration** is more of a bottom up approach. It begins with what is known and from this constructs the solution step by step
- It is hard to say that the non-recursive version is better than the recursive one or vice versa
- The non-recursive version is more efficient as the overhead of parameter passing in most compilers is heavy

Demerits of recursive algorithm

- Many programming languages do not support recursion; hence recursive mathematical function is to be implemented **using iterative methods**

- Even though mathematical functions can be easily implemented using **recursion** it is always at the cost of additional execution time and memory space

- A recursive function can be called from within or outside itself and to ensure its proper functioning it has to save in some order the return addresses so that, a return to the proper location will result when the return to a calling statement is made

Demerits of iterative algorithm

- Mathematical functions such as **factorial and Fibonacci series** generation can be easily implemented using recursion than iteration

- In **iterative techniques** looping of statement is very much necessary and needs complex logic

- **The iterative code may result into lengthy code**

Applications of recursion

- Following are the major areas in which the process of recursion can be applied
- Search techniques
- Game playing
- Expert Systems
- Pattern Recognition and Computer Vision
- Robotics
- Artificial Intelligence

THANK YOU!