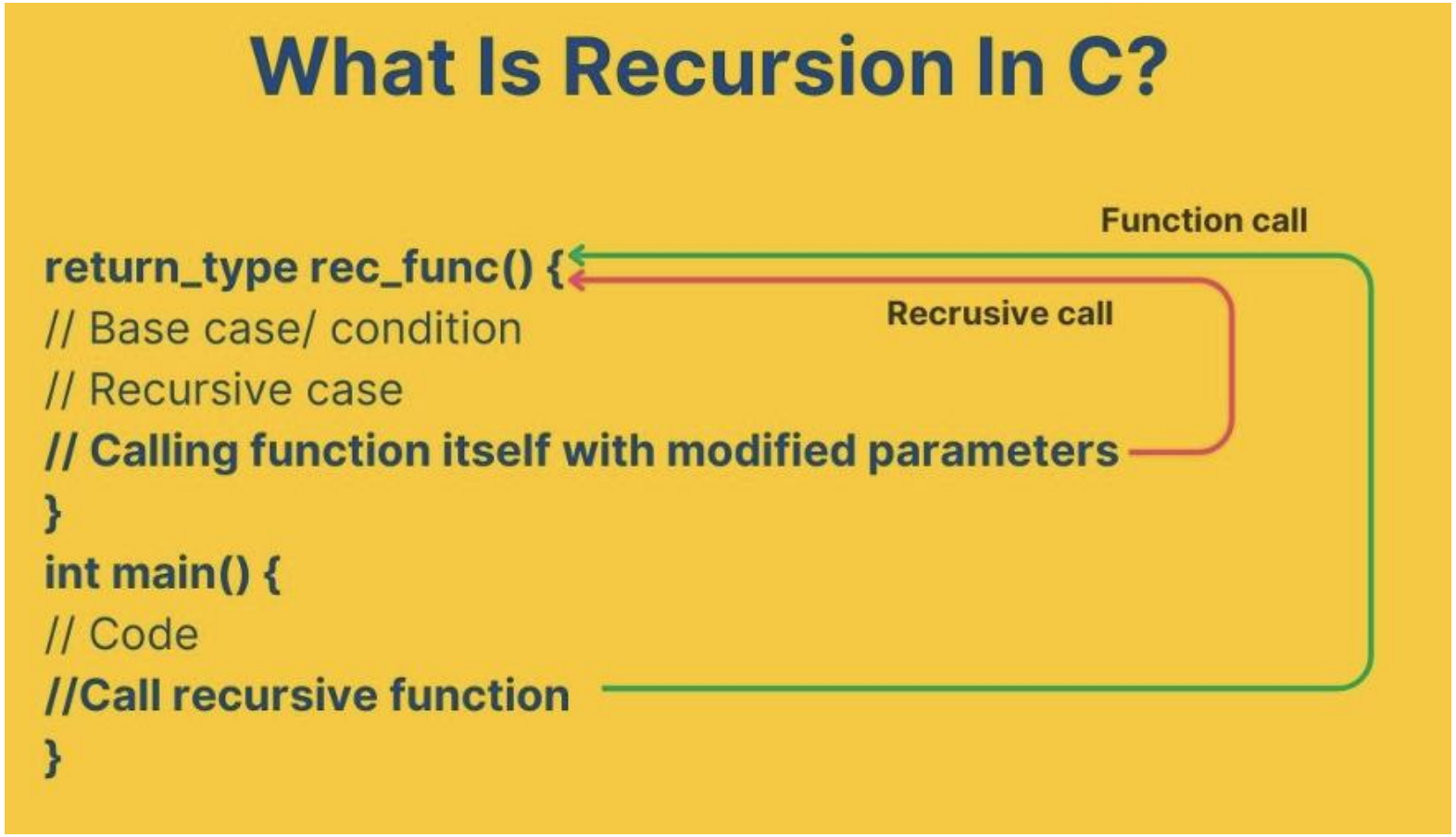
Fundamentals of Computer Programming

CS-110

Course Instructor: Dr. Momina Moetesum

Introduction to Recursion

Week 11



Learning Objectives

01

To understand how recursion works

02

To practice tracing recursive algorithms for indepth understanding

03

To understand different types of Recursion

Introduction to Recursion

- A recursive call is a function call in which the function being called is the same as the one making the call.
  - In many programming languages, any function can call another function
  - A function can even call itself.
  - When a function calls itself, it is making a *recursive* call
  - Recursion is a powerful technique that can be used in the place of iteration(looping)

Why Choose Recursion?

This technique provides a way to break complicated problems down into simple problems which are easier to solve, i.e. the **“Divide and Conquer”** rule

Solves small problems directly

Simplifies large problem into 1 or more smaller sub-problem(s) & solve recursively

Calculates solution from solution(s) for sub-problem

Why Recursion?

Recursion is a powerful problem-solving technique that often produces very clean solutions to even the most complex problems

Recursive solutions can be easier to understand and to describe than iterative solutions

Example

```
int sum(int k) {
  if (k > 0)
     {return k + sum(k - 1);}
  else
     {return 0;}
}

int main() {
  int result = sum(10);
  cout << result;
  return 0;
}
```

10 + sum(9)
10 + (9 + sum(8))
10 + (9 + (8 + sum(7)))
...
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + sum(0)
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + 0

Recursive Definition

A problem can be solved recursively if the following hold.

Base Case: The case for which the solution can be stated non-recursively. The case for which the answer is explicitly known.

General Case: The case for which the solution is expressed in smaller version of itself. Also known as recursive case

Recursion Example - Power

$x^n = x * x * x * x \ldots n \text{ times}$

$3^4 = 3 \times 3 \times 3 \times 3 = 81$

$3^3 = 3 \times 3 \times 3 = 27$

$3^2 = 3 \times 3 = 9$

$3^1 = 3$

$3^0 = 1$

- *Once some power of 3 has been calculated, it can be used to calculate the next power.*
- Given $3^3 = 27$, we can calculate: $3^4 = 3 \times 3^3 = 3 \times 27 = 81$
- And this Value to Calculate

Recursion Example - Power

This approach of calculating powers leads to the following recursive definition of the power function.

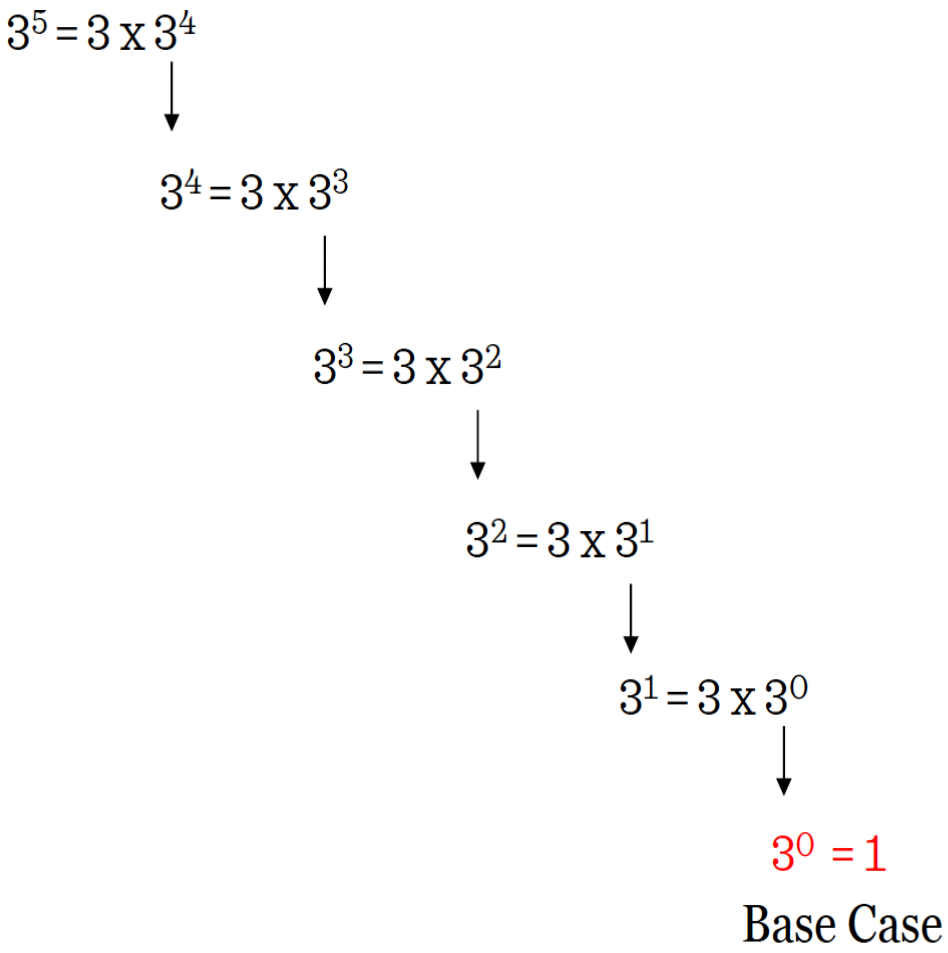
- ✓A Base Case

$$x^0 = 1$$

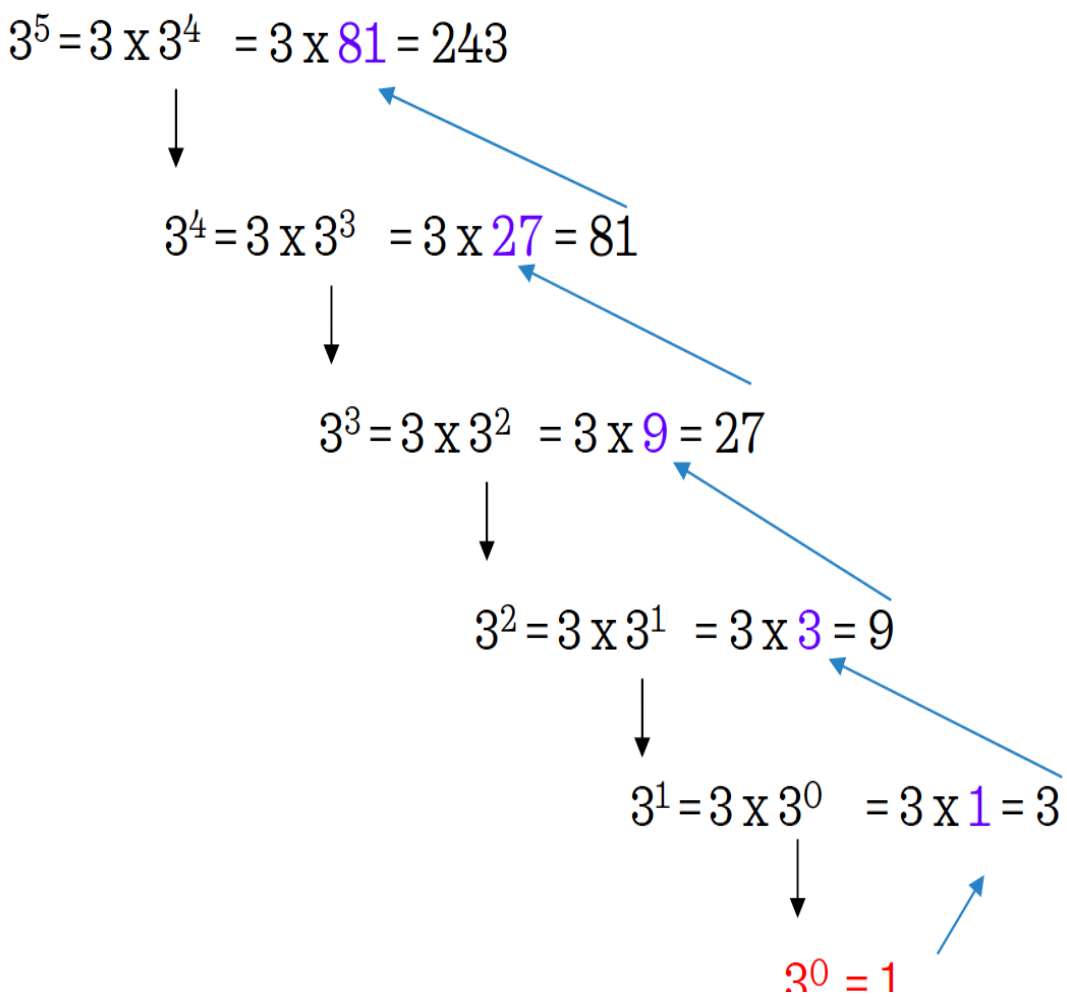
- ✓A recursive step (previously calculated values)

$$\text{For } n > 0,\ x^n = x * x^{n-1}$$

Recursion Example - Power



Recursion Example - Power

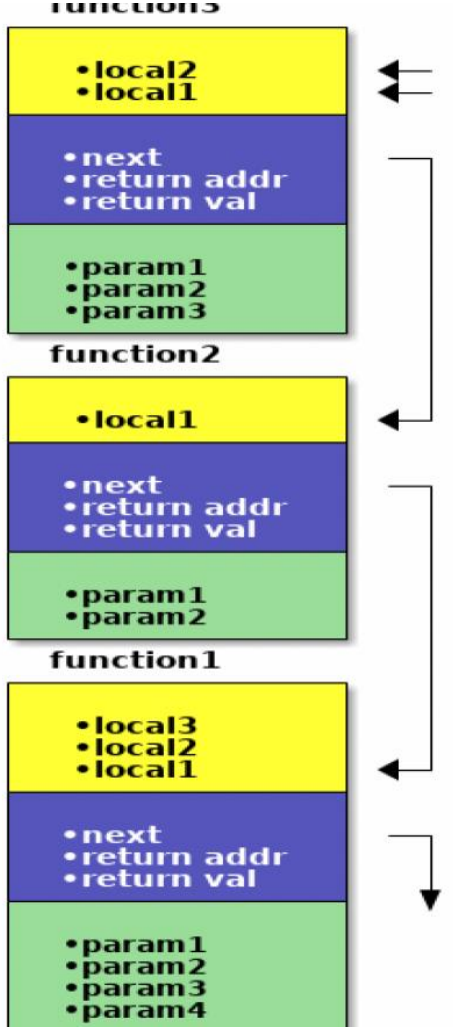


Recursion Example

```
int power (int base , int exp)
{
  if (exp == 0)   //base case
    return 1;
  else
    return base * power(base,exp-1)
                    //recursive step
}
```

How Recursion Works?

1. In recursion, it is essential for a function to call itself, otherwise recursion will not take place.
2. Only user define function can be involved in recursion.
3. To stop the recursive function it is necessary to base the recursion on test condition and proper terminating statement such as *exit()* or *return* must be written using if() statement.
4. When a recursive function is executed, the recursive calls are not implemented instantly. All the recursive calls are pushed onto the stack until the terminating condition is not detected, the recursive calls stored in the stack are popped and executed.
5. During recursion, at each recursive call new memory is allocated to all the local variables of the recursive functions with the same name.



Points to Consider

- When written correctly recursion can be a very efficient and mathematically-elegant approach to programming.
- However, the developer should be very careful with recursion as it can be quite easy to slip into writing a function which:
  - Never terminates, or
  - One that uses excess amounts of memory, or
  - Uses excess amounts of processor power.

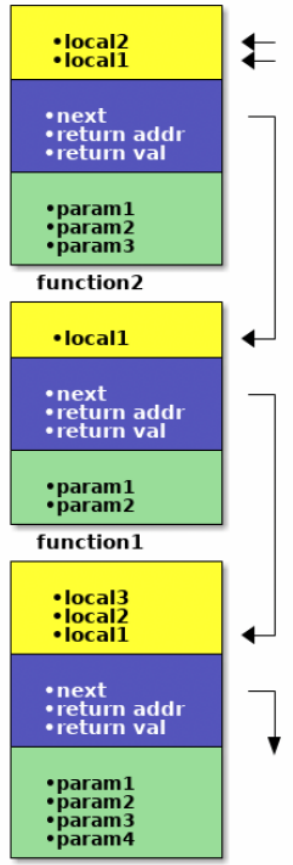
Limitations of Recursion

Just because an algorithm *can* be implemented in a recursive manner doesn't mean that it *should* be implemented in a recursive manner

Recursive solutions may involve extensive overhead because they use calls

When a call is made, it takes time to build a *stackframe* and push it onto the system stack

Conversely, when a return is executed, the *stackframe* must be popped from the stack and the local variables reset to their previous values – this also takes time



Recursion Example - Factorial

Factorial Function

- Given a positive integer n, the factorial of n is defined as the product of all integers between 1 and n, including n

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & else \end{cases}$$

Recursion Example - Factorial

```
int F(int n)
{
      if(n==0)
         return 1;
      else
         return n* F(n-1);

}
```

Find the value of F(n) when n=3

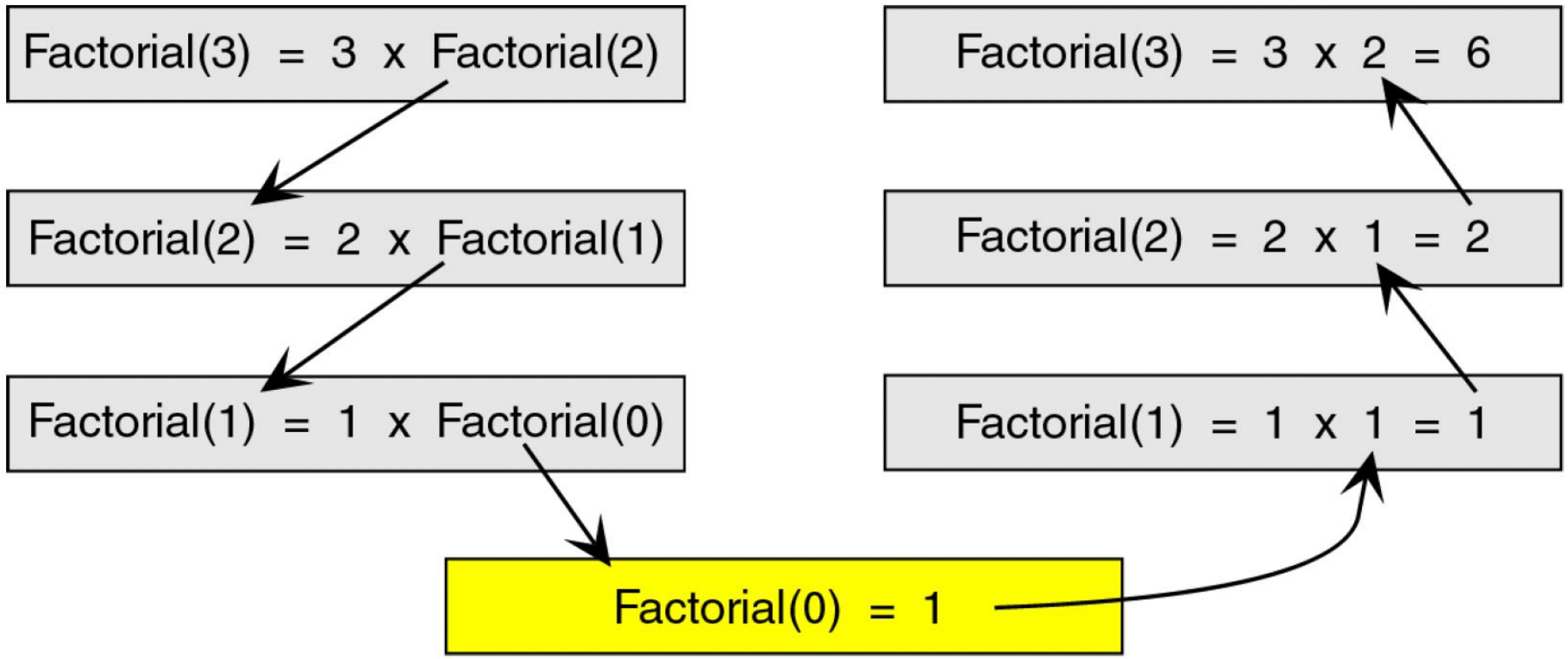
F(3) = 3 * F(2) F(3) = 3*2 =6

F(2) = 2 * F(1) F(2) = 2*1 =2

F(1) = 1 * F(0) F(1) = 1*1 =1

F(0) = 1

Recursion Example - Factorial



Recursion Example – Summation Series

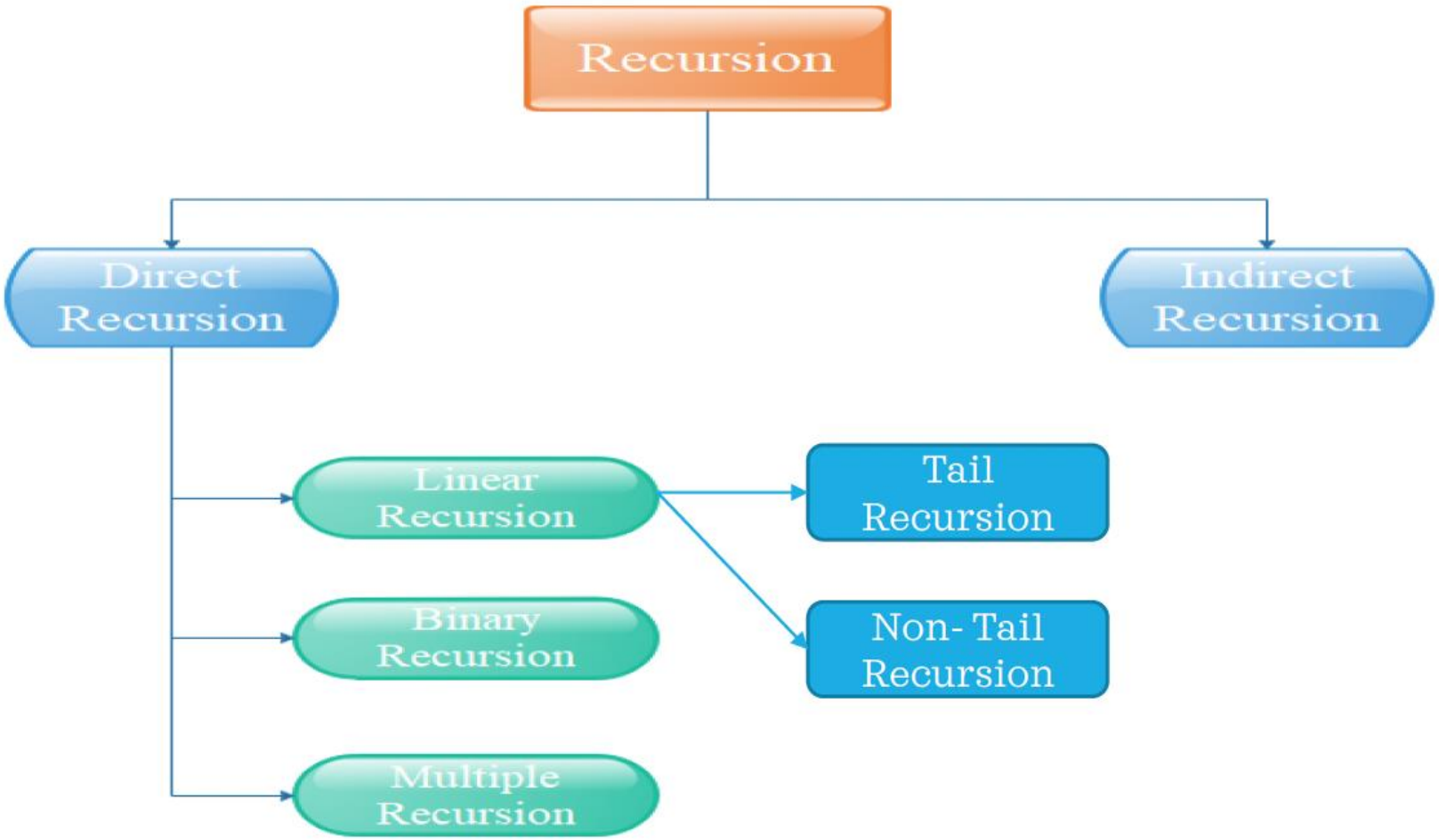
The problem of computing the sum of all the numbers between *1* and any positive integer *N* can be recursively defined as:

$$\sum_{i=1}^{N} = N + \sum_{i=1}^{N-1} = N + (N-1) + \sum_{i=1}^{N-2} \ldots$$

Recursion Example – Summation Series

```
int sum(int n)
{
if(n==1)
  return n;
else
  return n + sum(n-1);
}
```

Find the value of F(n) when n=3



Types of Recursion

Direct vs. Indirect Recursion

A function is *indirectly* recursive if it contains a call to another function which ultimately calls originally calling function.

Indirect Recursion

```
int foo(int x)
 {
   if (x <= 0)
        return x;
   else
       return bar(x);
 }
int bar(int y)
 {
   return foo(y - 1);
 }
```

Direct Recursion

```
int foo(int x)
 {
   if (x <= 0)
        return x;
   else
        return foo(x - 1);
 }
```

Types of Recursion

Linear Recursion

- In linear recursion a function calls exactly once to itself each time the function is invoked, and grows linearly in proportion to the size of the problem.

Non-Tail Recursion

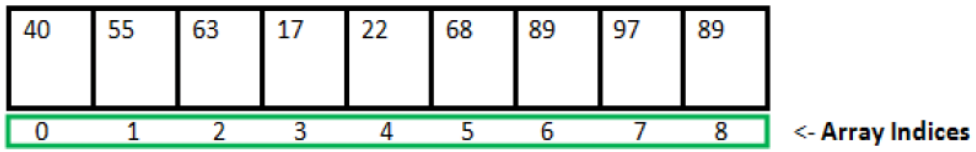
- Non-Tail recursion is another form of linear recursion, where the function does not make a recursive call as its very last operation. A function will be said non-tail recursive if the recursive call is not the last thing the function performs. ***Printing an array is an example of linear recursion.***

Tail Recursion

- Tail recursion is another form of linear recursion, where the function makes a recursive call as its very last operation. A function will be said tail recursive if the recursive call is the last thing the function performs. ***Printing an array in reverse is an example of tail recursion.***

Example of Non-Tail Recursion:Printing an array using recursion

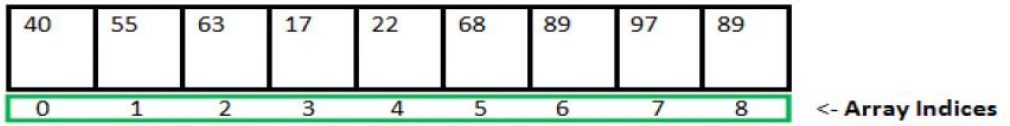
```
void printArray(int array[], int size){

        if(size==0)
                return;
        else
        {
                printArray(array,size-1);
                cout<<array[size-1]<<endl;
        }
}
```



Array Length = 9
First Index = 0
Last Index = 8

Example of Tail Recursion: Printing an array in reverse order using recursion

```
void printArray(int array[],int size)
{
	if(size==0)
		return;
	else
	{
		cout<< array[size-1]<<endl;
		printArray(array,size-1);
	}
}
```



Array Length = 9
First Index = 0
Last Index = 8

Types of Recursion

Binary Recursion

In binary recursion a function makes two recursive calls to itself when invoked, it uses binary recursion. ***Fibonacci series is a very nice example to demonstrate binary recursion.***

Multiple Recursion

Multiple recursion can be treated a generalized form of binary recursion. When a function makes multiple recursive calls possibly more than two, it is called multiple recursion.

Binary Recursion Example: Fibonacci Series

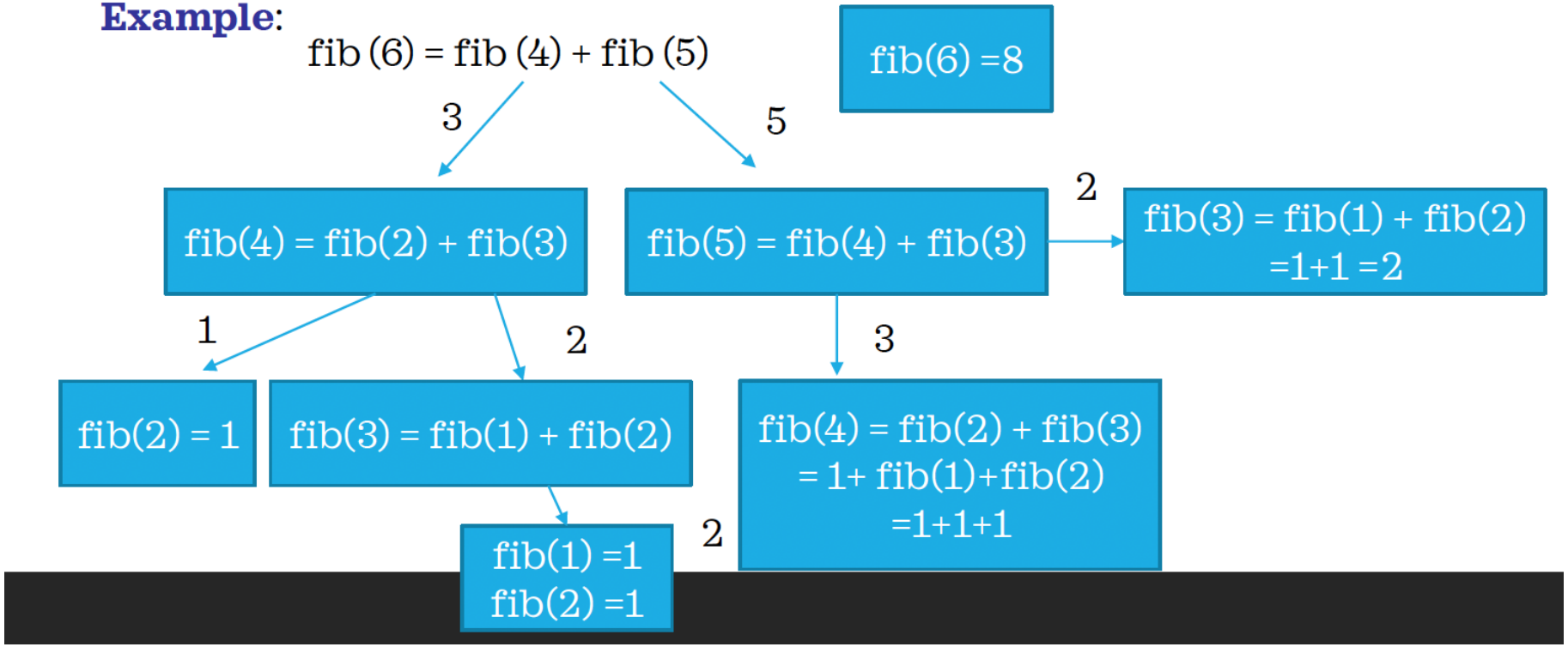
The sequence 1, 1, 2, 3, 5, 8, 13, 21, 34, ... is called the *Fibonacci* sequence 8

Definition:

fib (n) = 1 if n = 1 or n = 2

fib (n) = fib (n-2) + fib (n-1) if n > 2

Example:

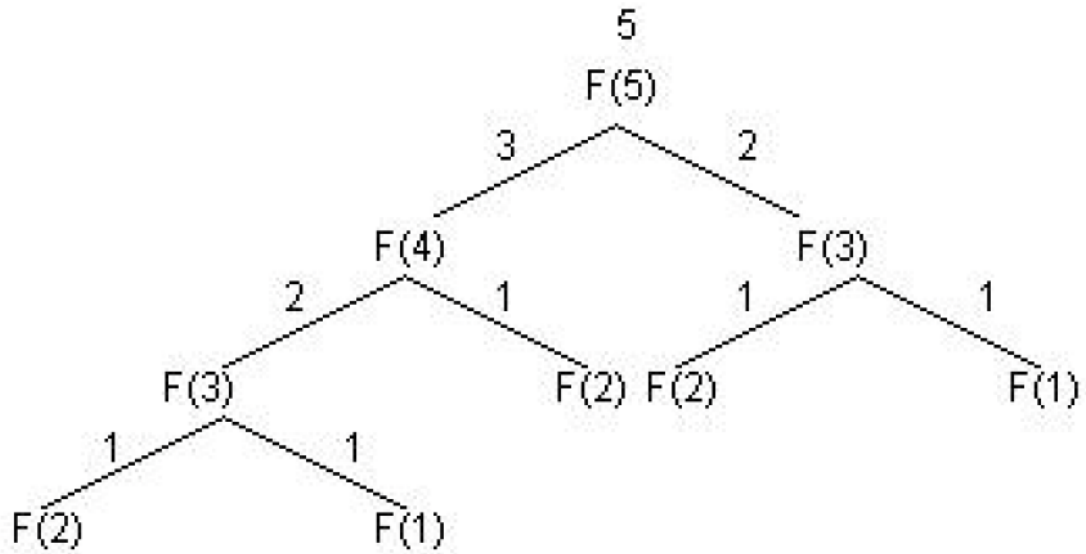


Recursive Example – Fibonacci Series

```
int fib( int n )
{
  if (n <= 2) return 1;
  else
  return fib(n-1) + fib(n-2);
}
```

Recursive Example – Fibonacci Series

```
int fib( int n )
{
   if (n <= 2) return 1;
   else
   return fib(n-1) + fib(n-2);
}
```



Recursive Example – Ackermann Function

```
int Ackermann(int m, int n)
{       if(m==0)
                return n+1;
        else if (m>0 && n==0)
                return Ackermann(m-1,1);
        else if (m>0 && n>0)
                return Ackermann( m-1, Ackermann(m, n-1));
}
```

Recursive Example – Ackermann Function

Evaluate A(m,n) where m=1 and n=2

$$A(m,n) = \begin{cases} n+1 & m = 0 \\ A(m-1,\ 1) & \text{for } m > 0 \text{ and } n = 0 \\ A(m-1,\ A(m,n-1)) & \text{for } m > 0 \text{ and } n > 0 \end{cases}$$

Recursive Example – Ackermann Function

Ackermann function

$$A(m,n) = \begin{cases} n+1 & m = 0 \\ A(m-1, 1) & \text{for } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{for } m > 0 \text{ and } n > 0 \end{cases}$$

- Find the terms $A(1,1), A(2,1), A(1,2)$

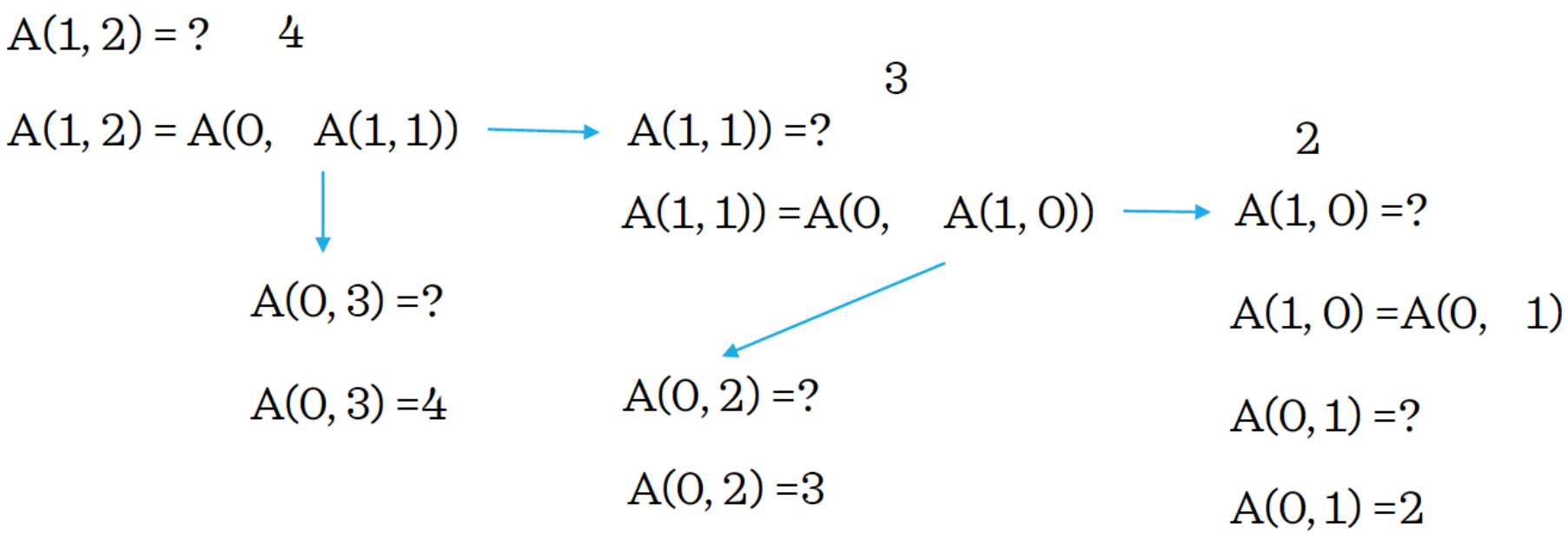
$$A(1,1) = A(0, A(1,0)) = A(0, A(0,1)) = A(0,2) = 3$$

$$A(1,2) = A(0, A(1,1)) = A(0, 3) = 4$$

$$A(2,1) = A(1, A(2,0)) = A(1, A(1,1)) = A(1, 3)$$
$$= A(0, A(1,2)) = A(0, 4) = 5$$

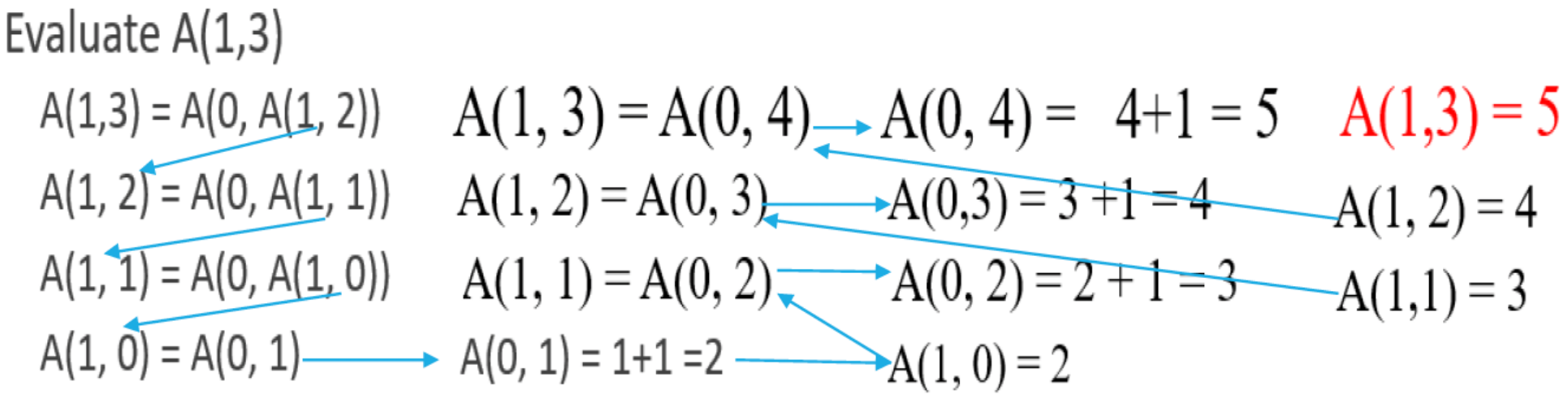
Recursive Example – Ackermann Function

Evaluate A(m,n) where m=1 and n=2



A(1, 2) =4

Recursive Example – Ackermann Function

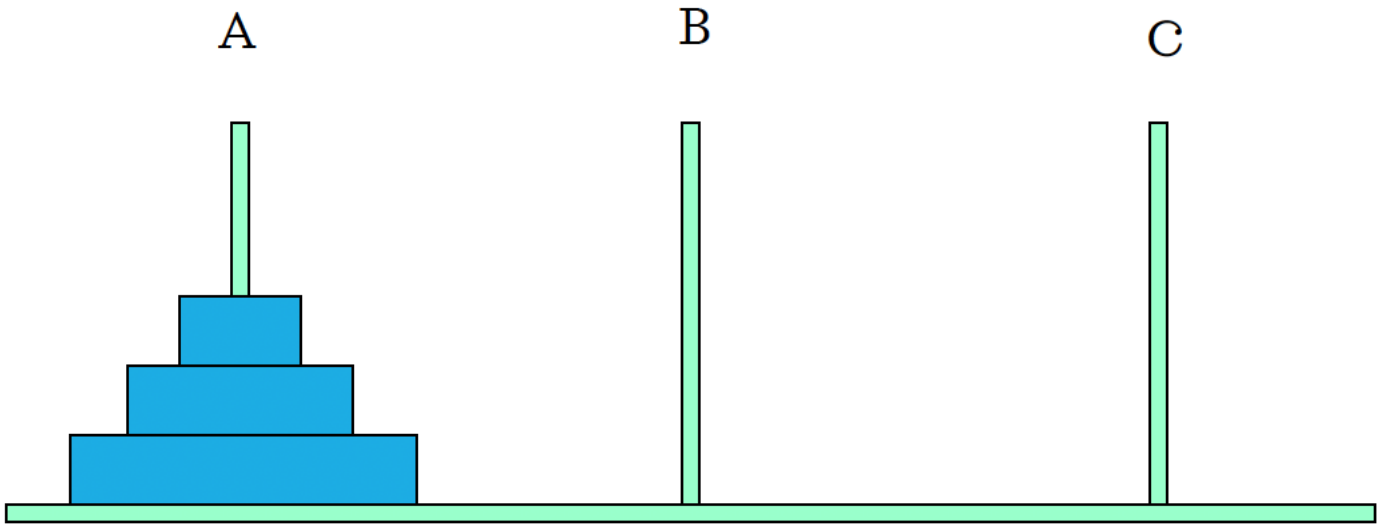


Recursive Example – Tower of Hanoi

A Mathematical puzzle invented by a French Mathematician, Edouard Lucas, in 1883. Inspiration: A temple story, where priests in the temple were given a stack of 64 gold disks, each one a little smaller than the one beneath it. (*Many other stories*)

Their assignment was to transfer the 64 disks from one of the three poles to another, with one important provision that a large disk could never be placed on top of a smaller one.

Recursive Example – Tower of Hanoi



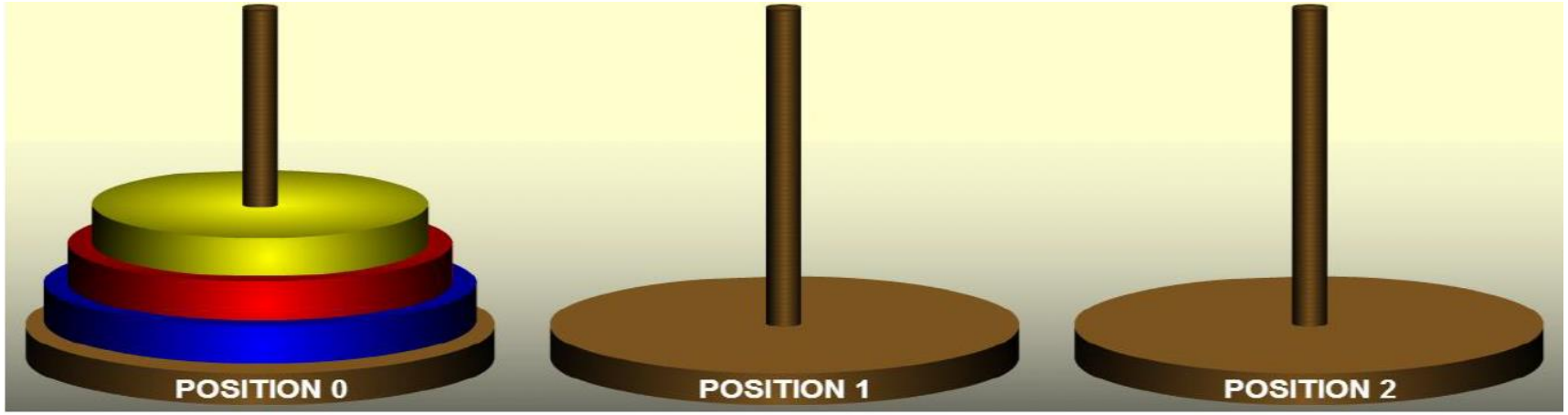
Recursive Example – Tower of Hanoi

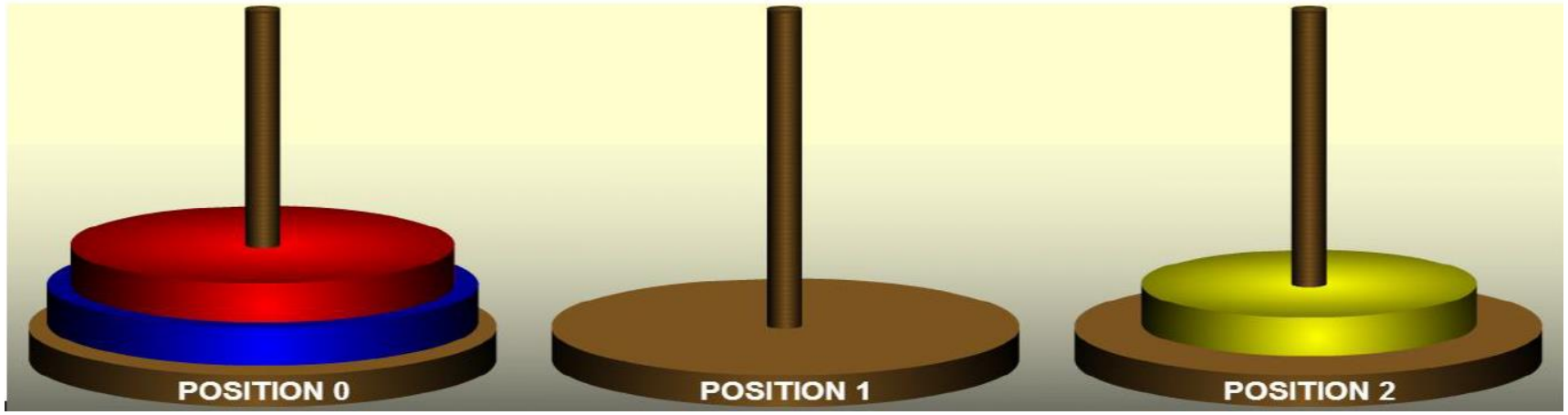
Problem:

Move the disks from the left peg to the right peg according to the following rules:

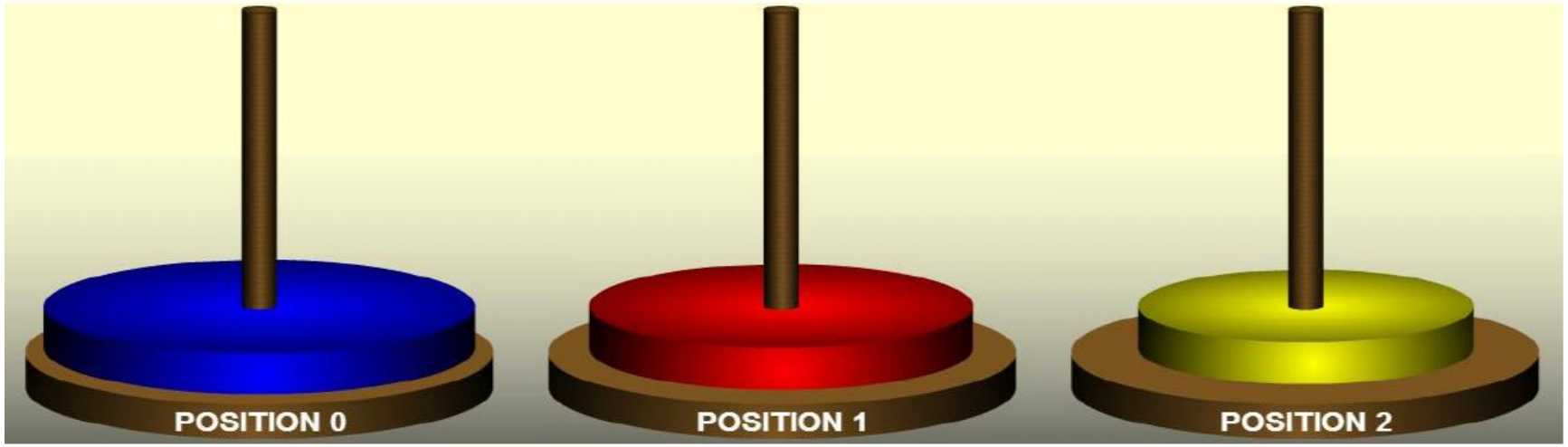
- When a disk is moved, it must be placed on one of the three pegs.
- Only one disk may be moved at a time, and it must be the top disk on one of the pegs.
- A large disk may never be placed on top of a smaller one.

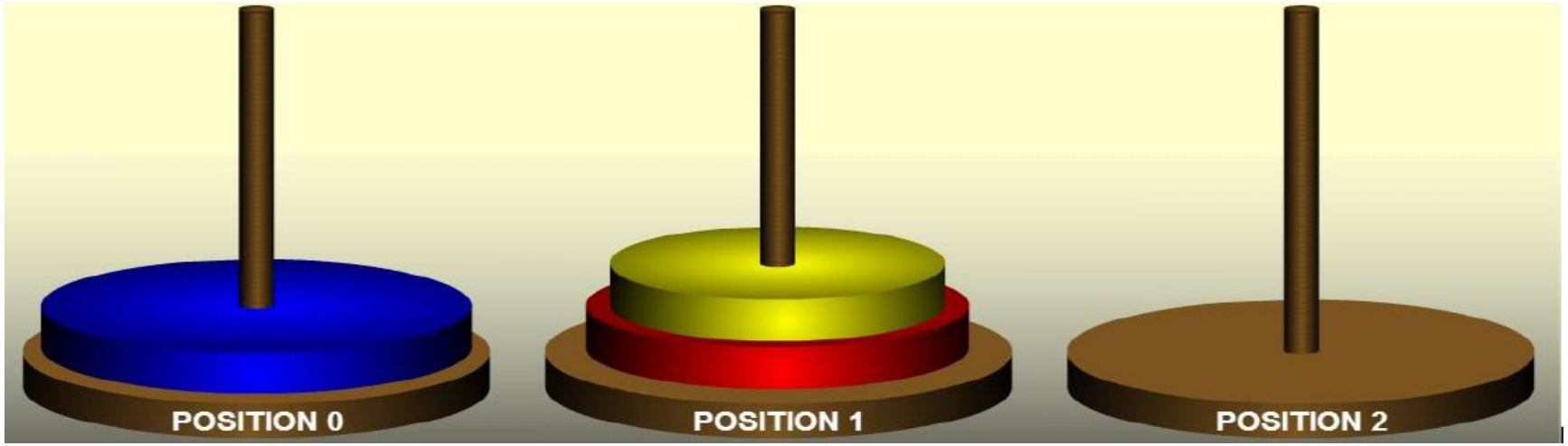
Recursive Example – Tower of Hanoi



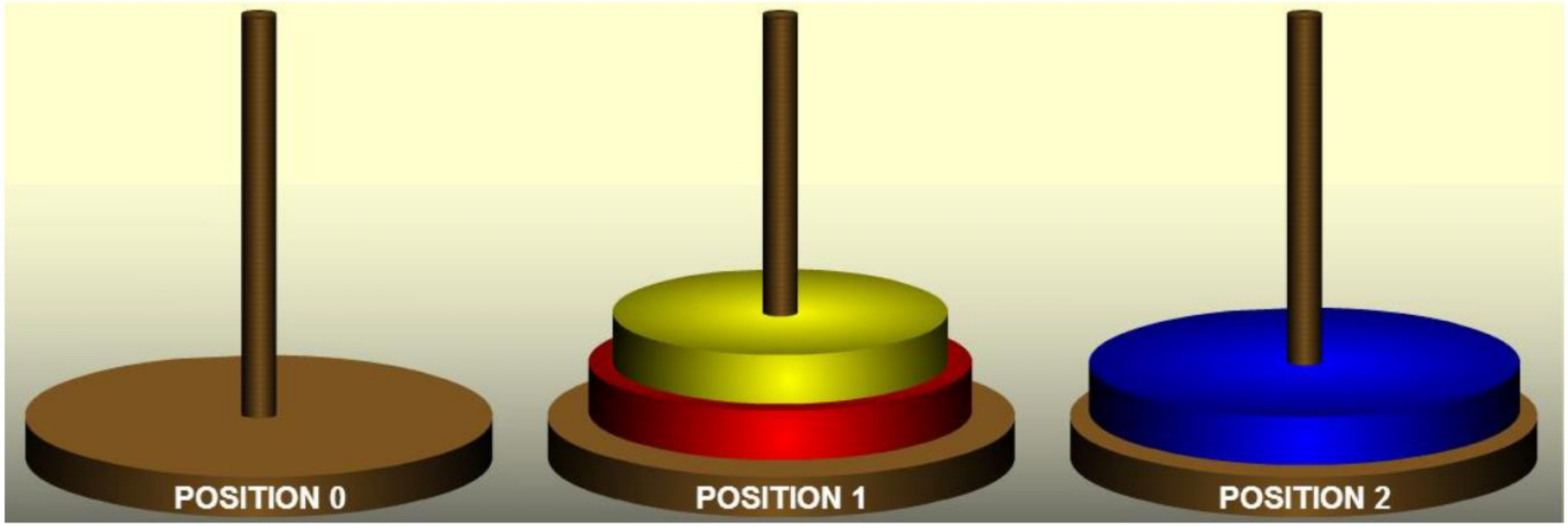


Recursive Example – Tower of Hanoi

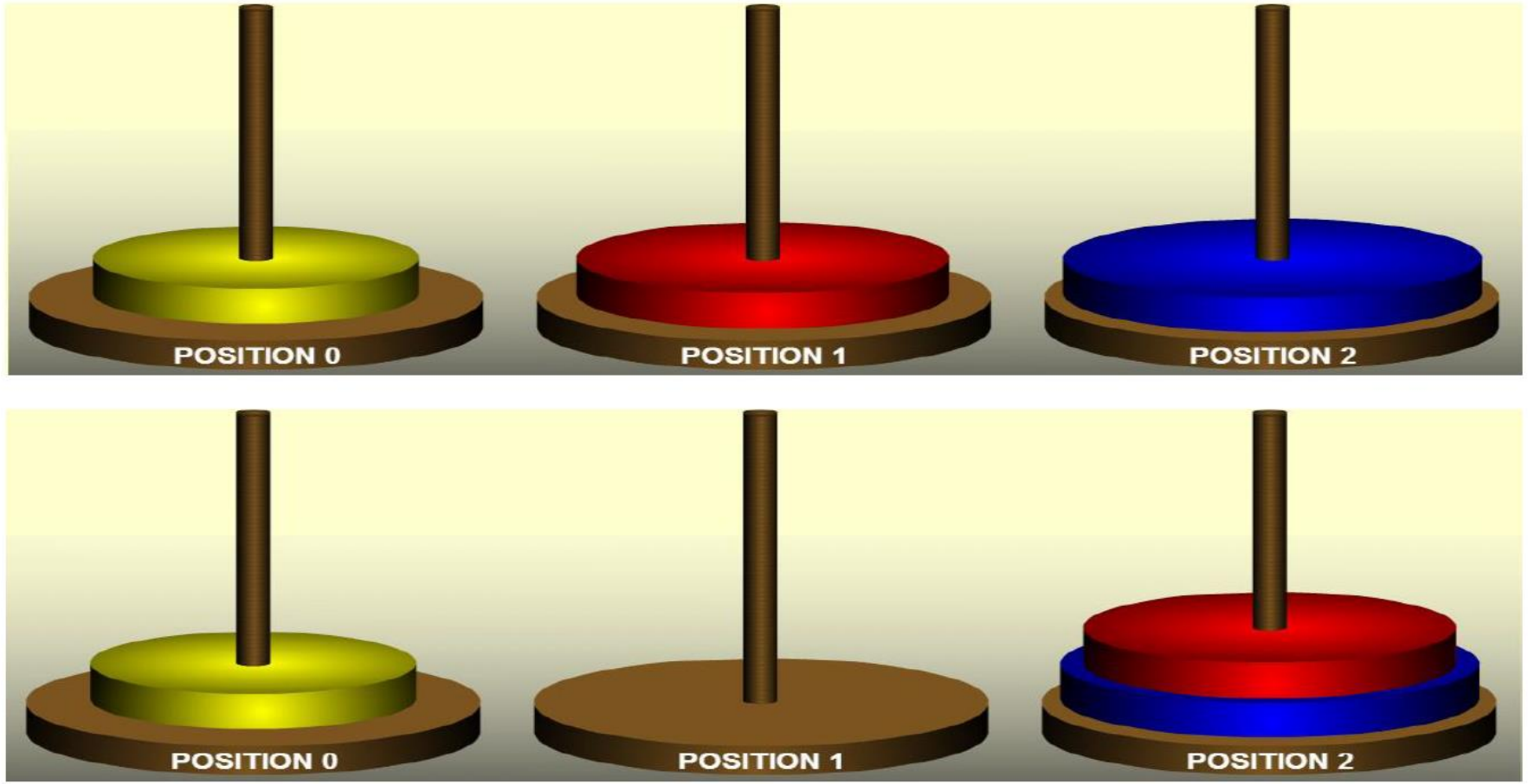




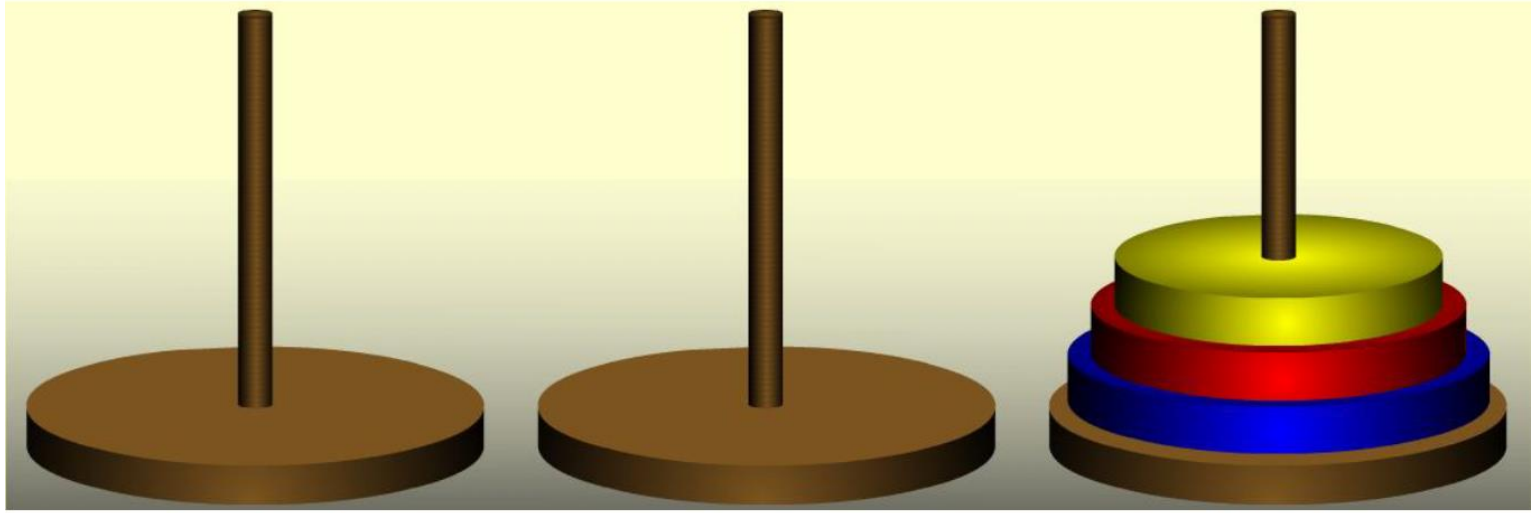
Recursive Example – Tower of Hanoi



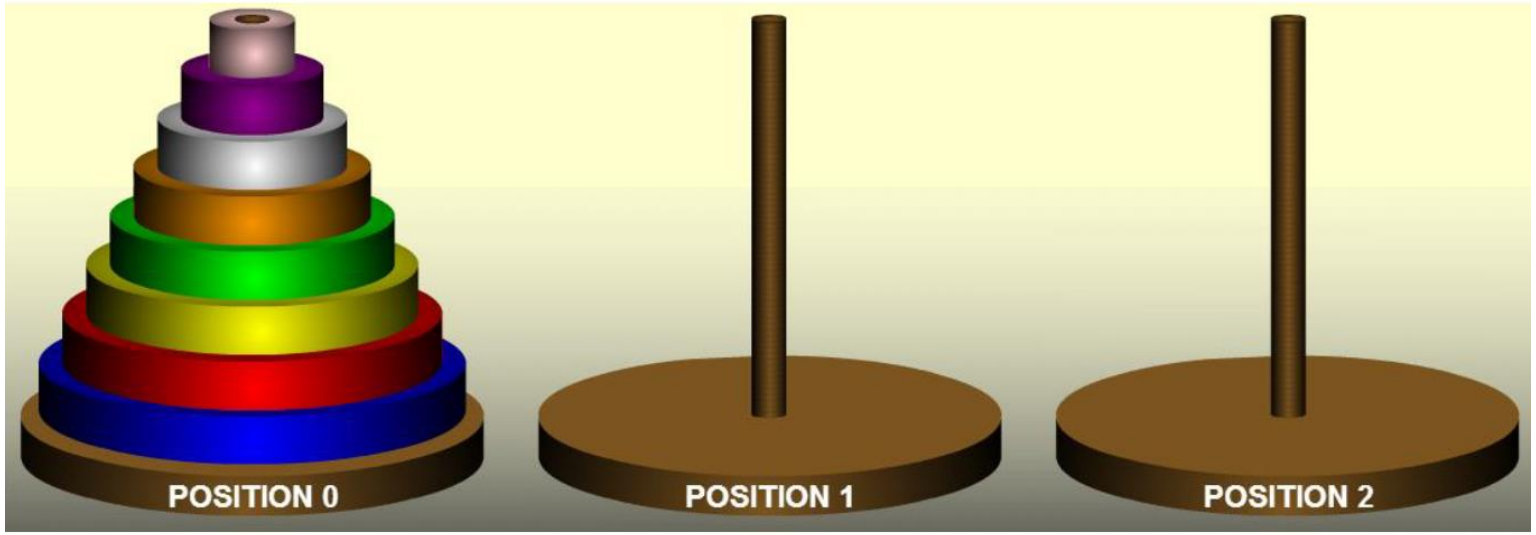
Recursive Example – Tower of Hanoi



Recursive Example – Tower of Hanoi



How to solve for larger number of disks?



Recursive Example – Tower of Hanoi

Base Case: *If there is one disk, move it from peg A to peg C.*

Recursive Steps:

For n > 1, we assume that a solution exists for n –1 disks:

1. *Move the topmost n-1 disks from peg A to peg B, using peg C for temporary storage.*

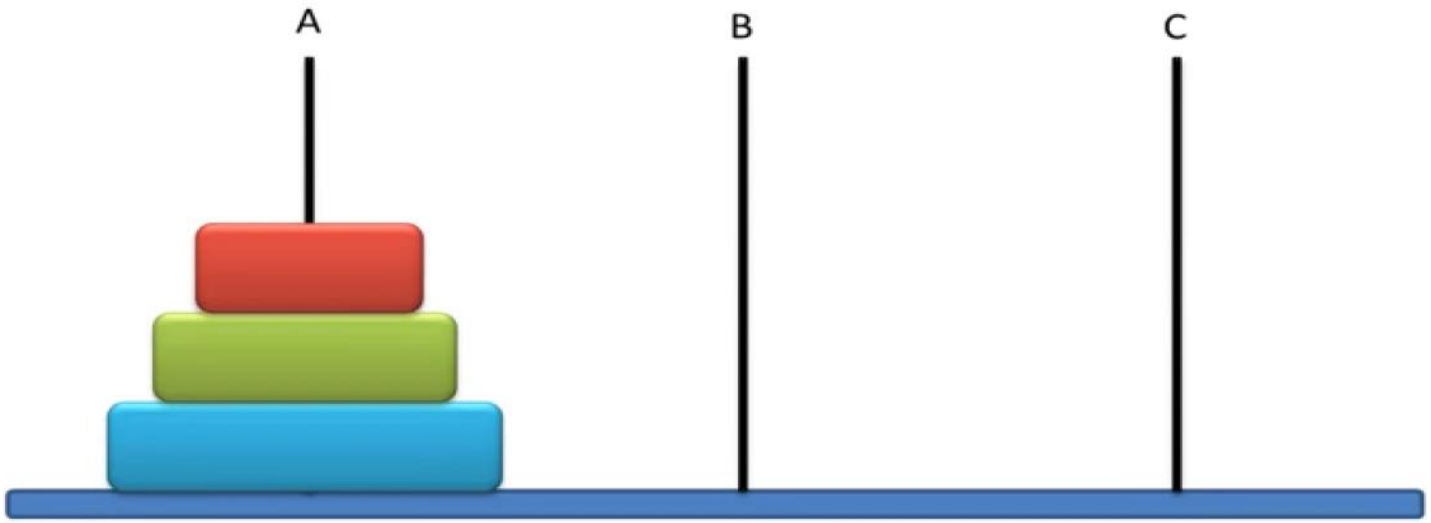
2. *Move the final disk remaining on peg A to peg C.*

3. *Move the n-1 disks from peg B to peg C, using peg A for temporary storage.*

Recursive Example – Tower of Hanoi

```
void Move(int n, char Beg, char Aux, char End)
{
if (n == 1)
        cout << "Move the top disk from " << Beg
         << " to " << End << endl;
 else
 {

   Move(n-1, Beg, End, Aux);

   Move(1, Beg, Aux, End);

   Move(n-1, Aux, Beg, End);
 }
}
```

Recursive Example – Tower of Hanoi



- Three Pegs A, B,C
- Three disks, N = 3
- Start with : `Move(3,A,B,C)`

Recursive Example – Tower of Hanoi

- Move(3,A,B,C)
- Follow the following three steps to solve the problem

Step 1: Move(n-1, Beg, End, Aux);

Step 2: Move(1, Beg, Aux, End);

Step 3: Move(n-1, Aux, Beg, End);

N = 3
Beg = A
Aux = B
End = C

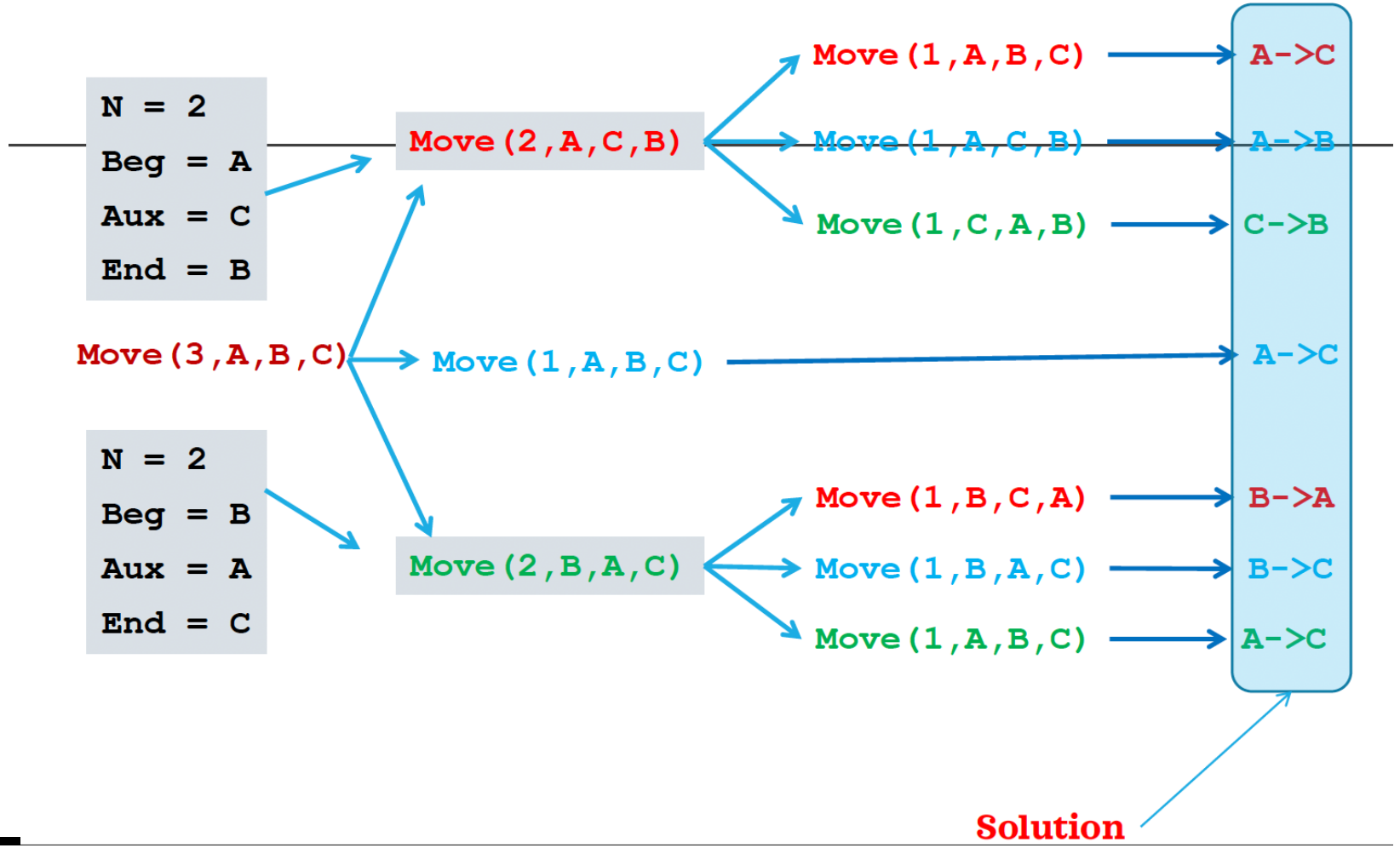
Plug in these values

Move(3,A,B,C)

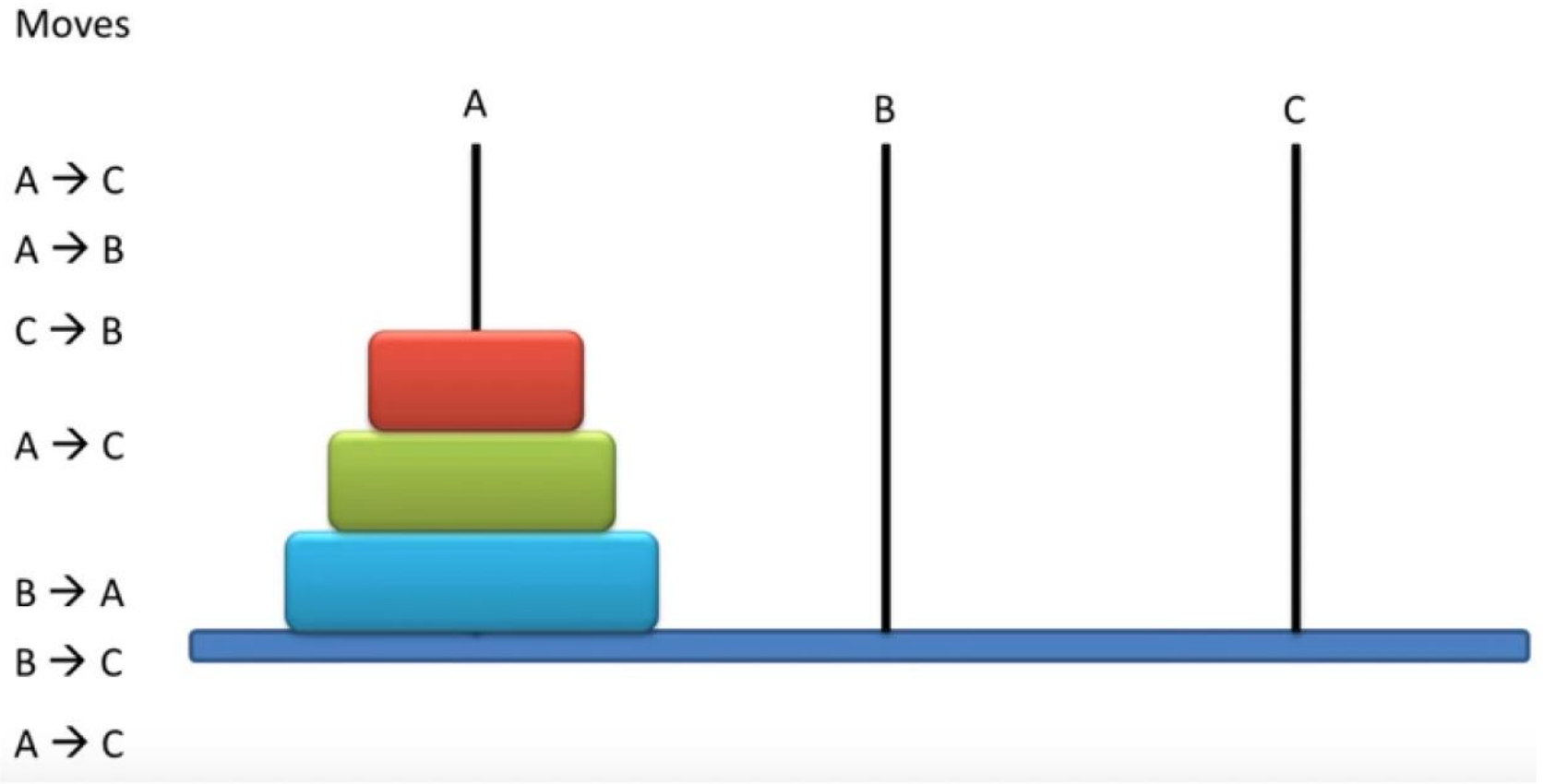
- Move(2,A,C,B)
- Move(1,A,B,C)
- Move(2,B,A,C)

Recursive Example – Tower of Hanoi

```
Step 1: Move(n-1, Beg, End, Aux);
Step 2: Move(1, Beg, Aux, End);
Step 3: Move(n-1, Aux, Beg, End);
```



Recursive Example – Tower of Hanoi



Recursive Example – Tower of Hanoi

Moves

A → C ✓

A → B

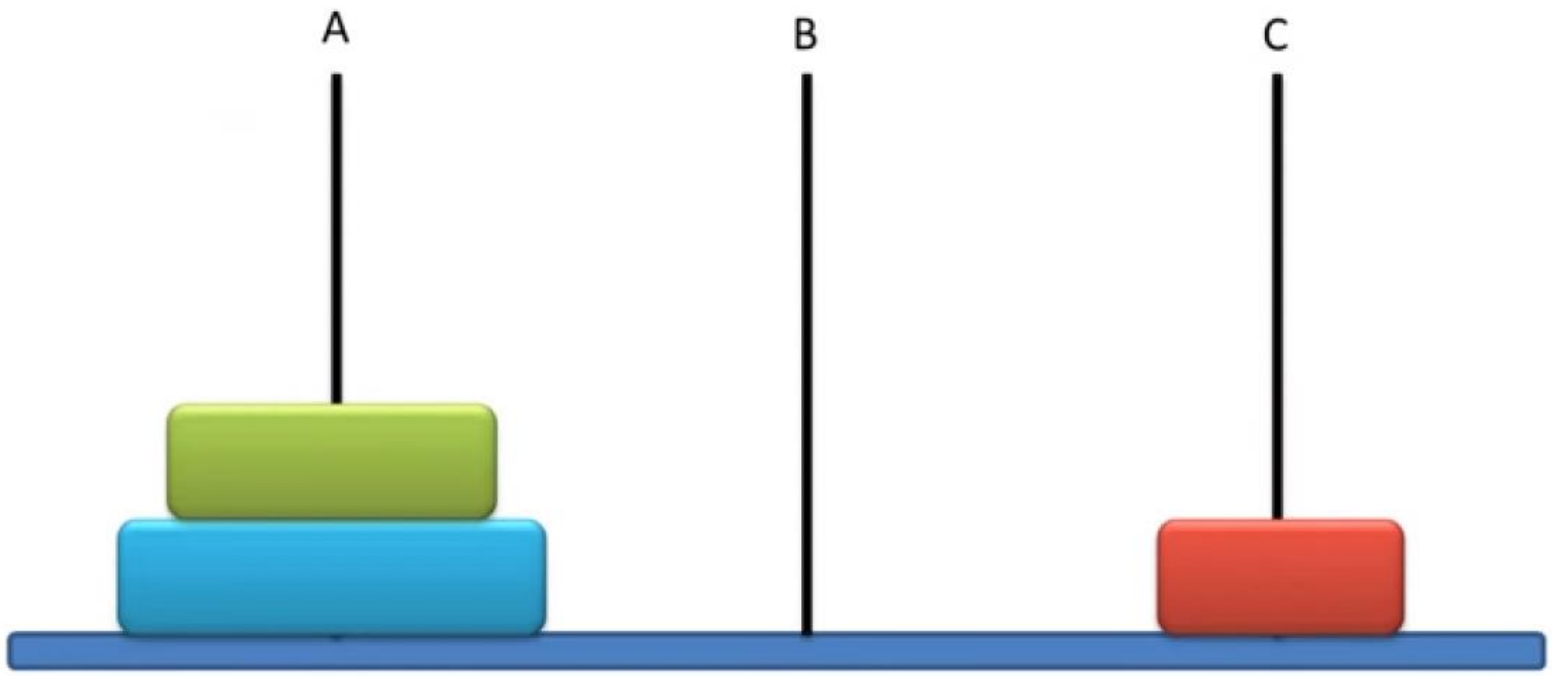
C → B

A → C

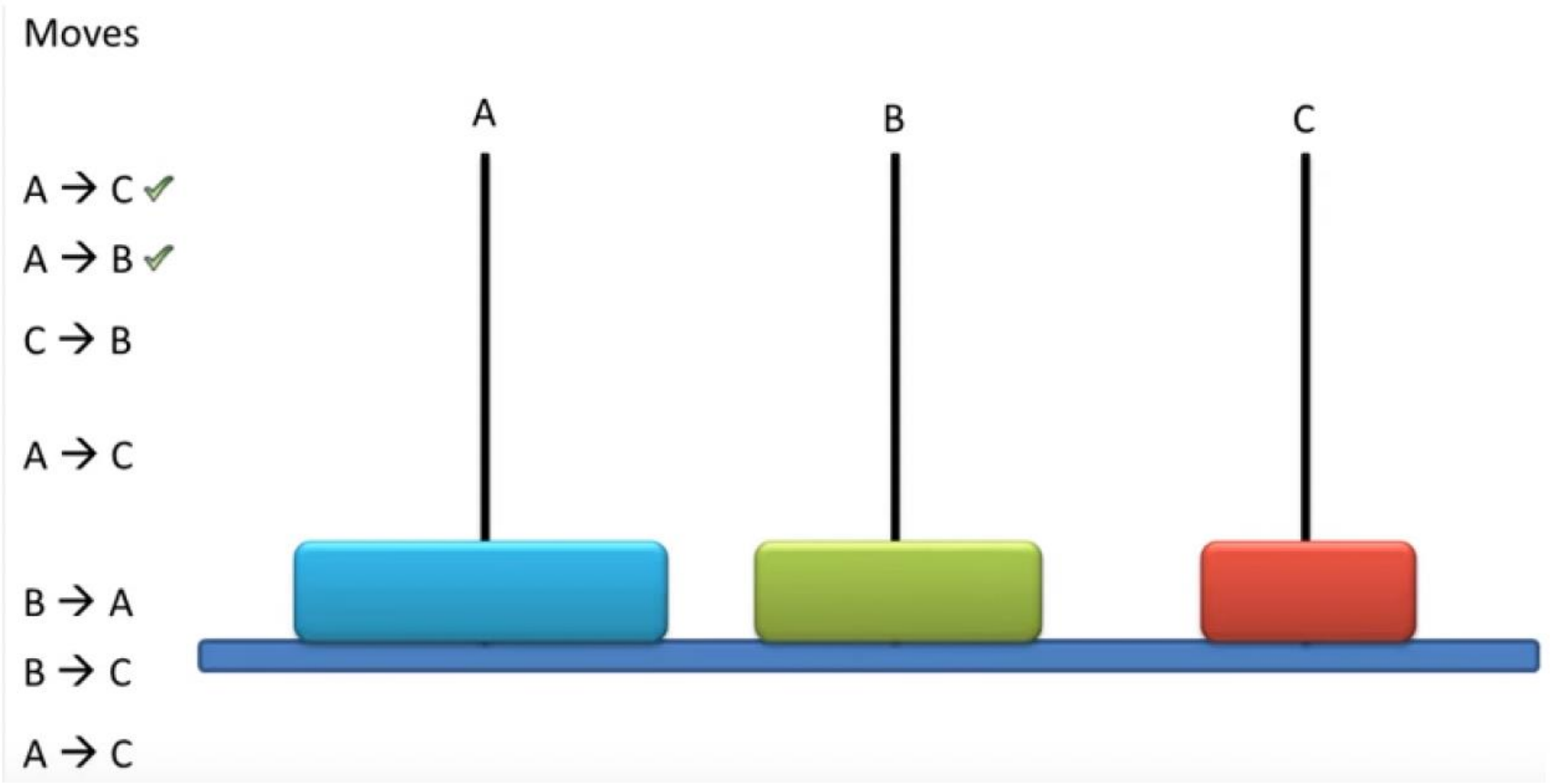
B → A

B → C

A → C



Recursive Example – Tower of Hanoi



Recursive Example – Tower of Hanoi

Moves

A → C ✓

A → B ✓

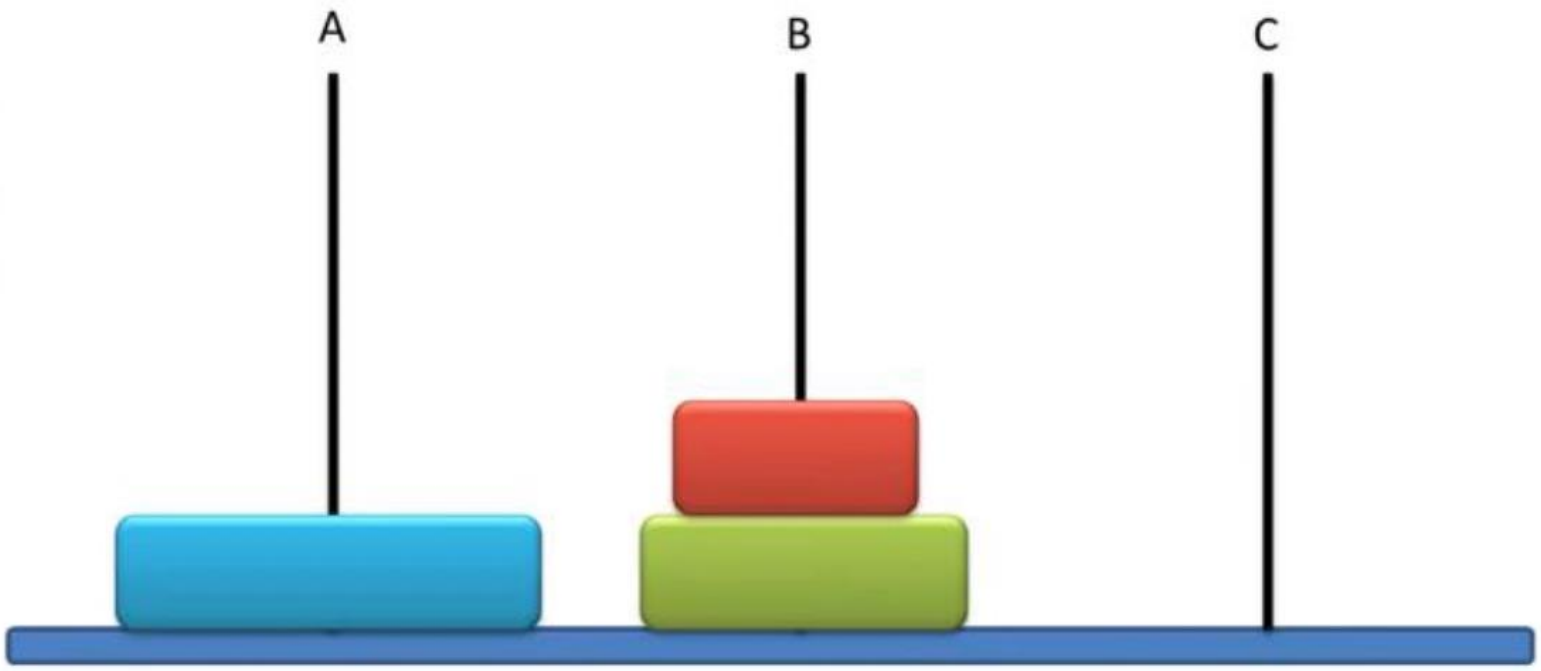
C → B ✓

A → C

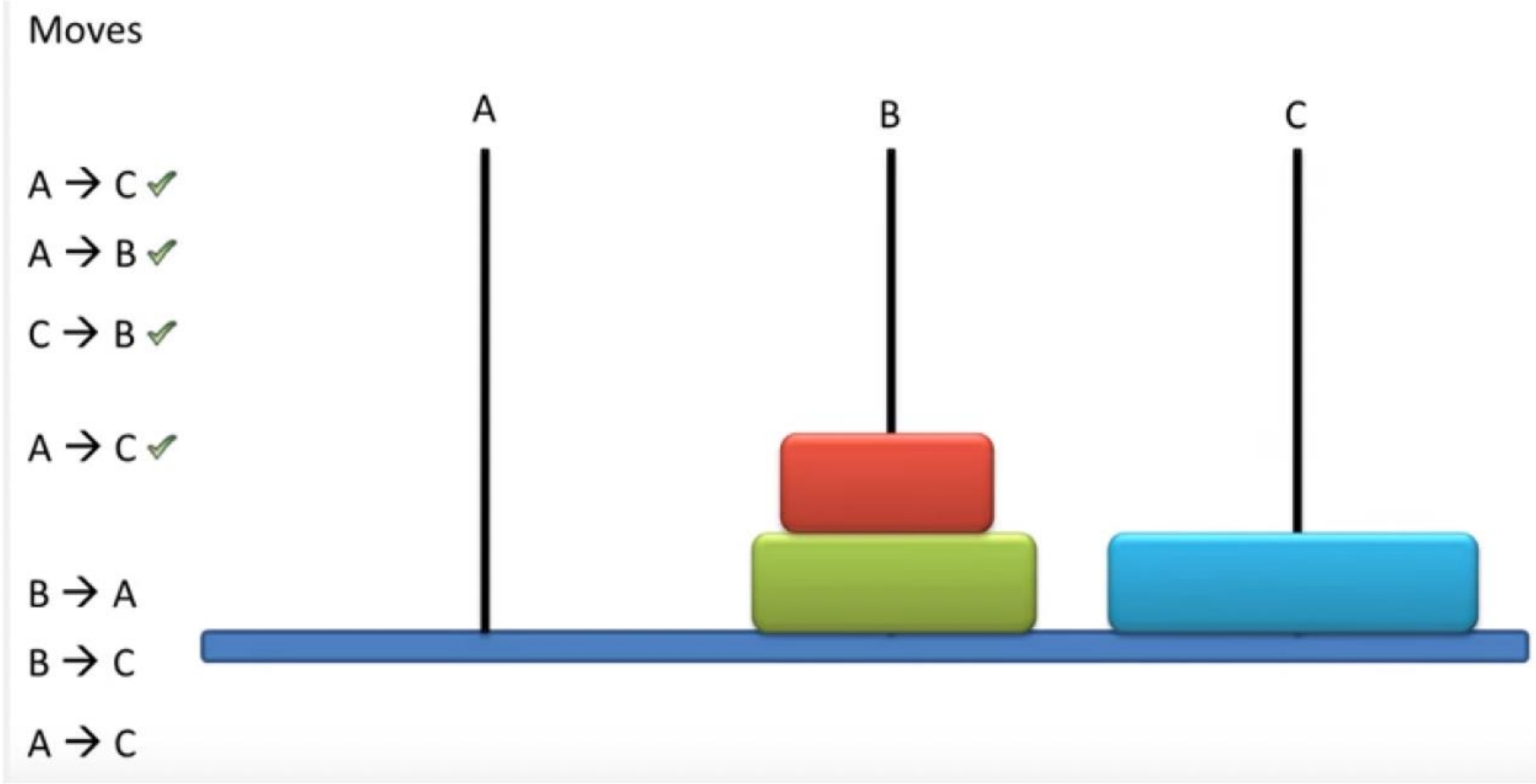
B → A

B → C

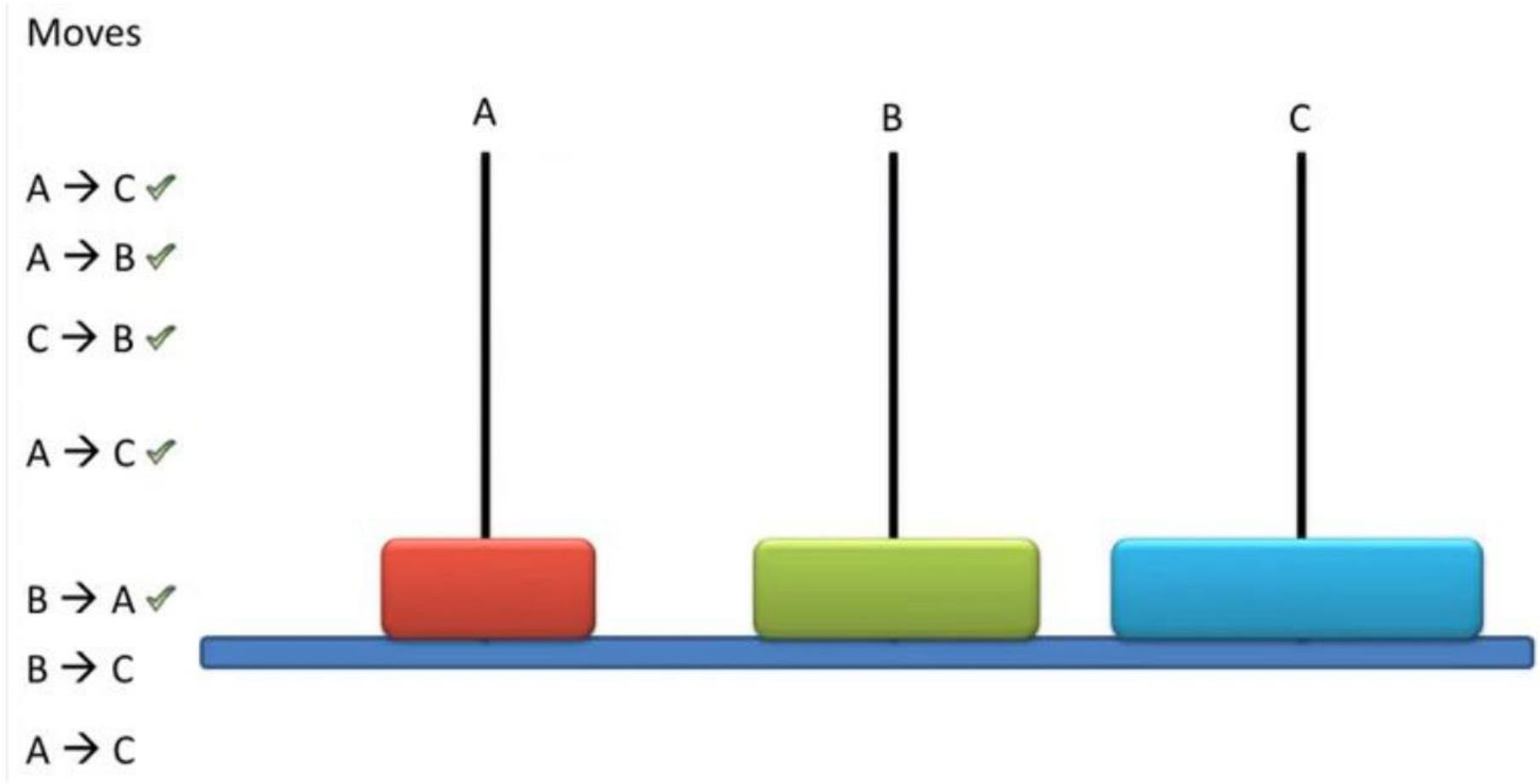
A → C



Recursive Example – Tower of Hanoi



Recursive Example – Tower of Hanoi



Recursive Example – Tower of Hanoi

Moves

A → C ✓

A → B ✓

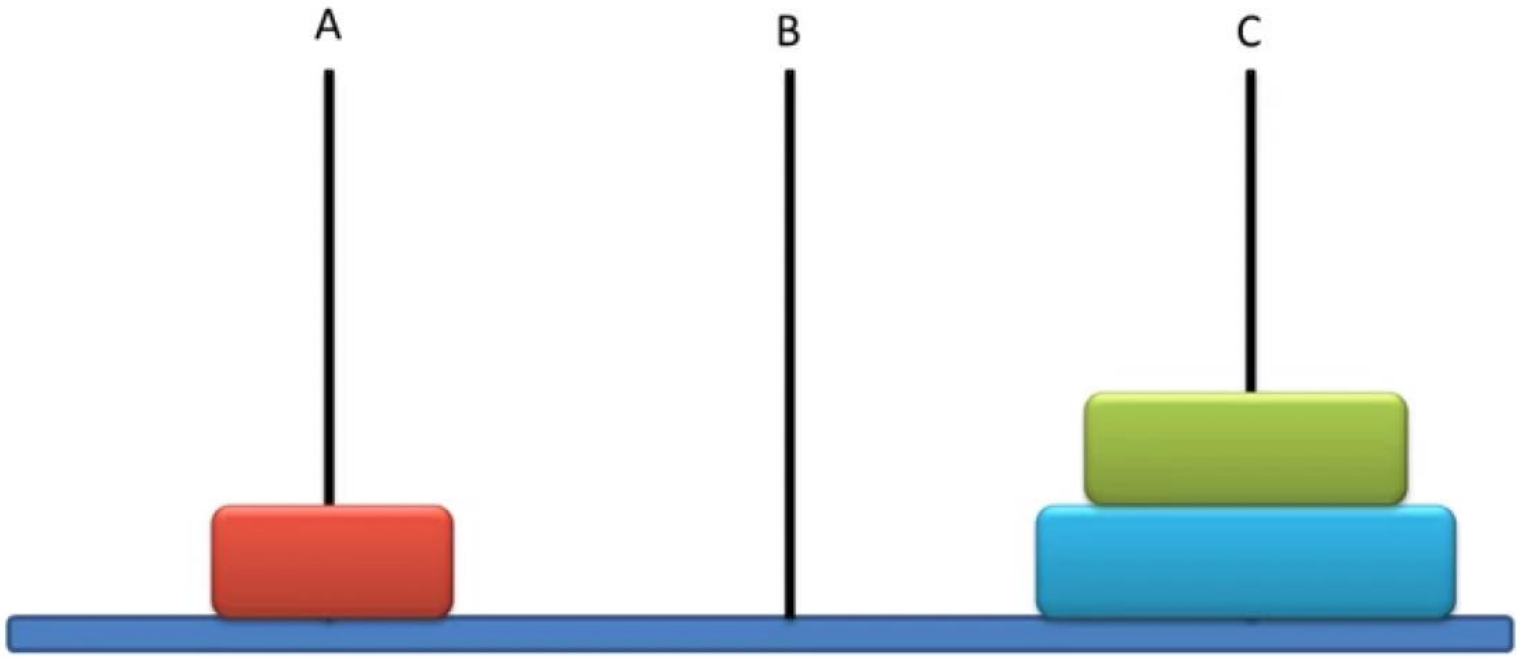
C → B ✓

A → C ✓

B → A ✓

B → C ✓

A → C



Recursive Example – Tower of Hanoi

Moves

A → C ✓

A → B ✓

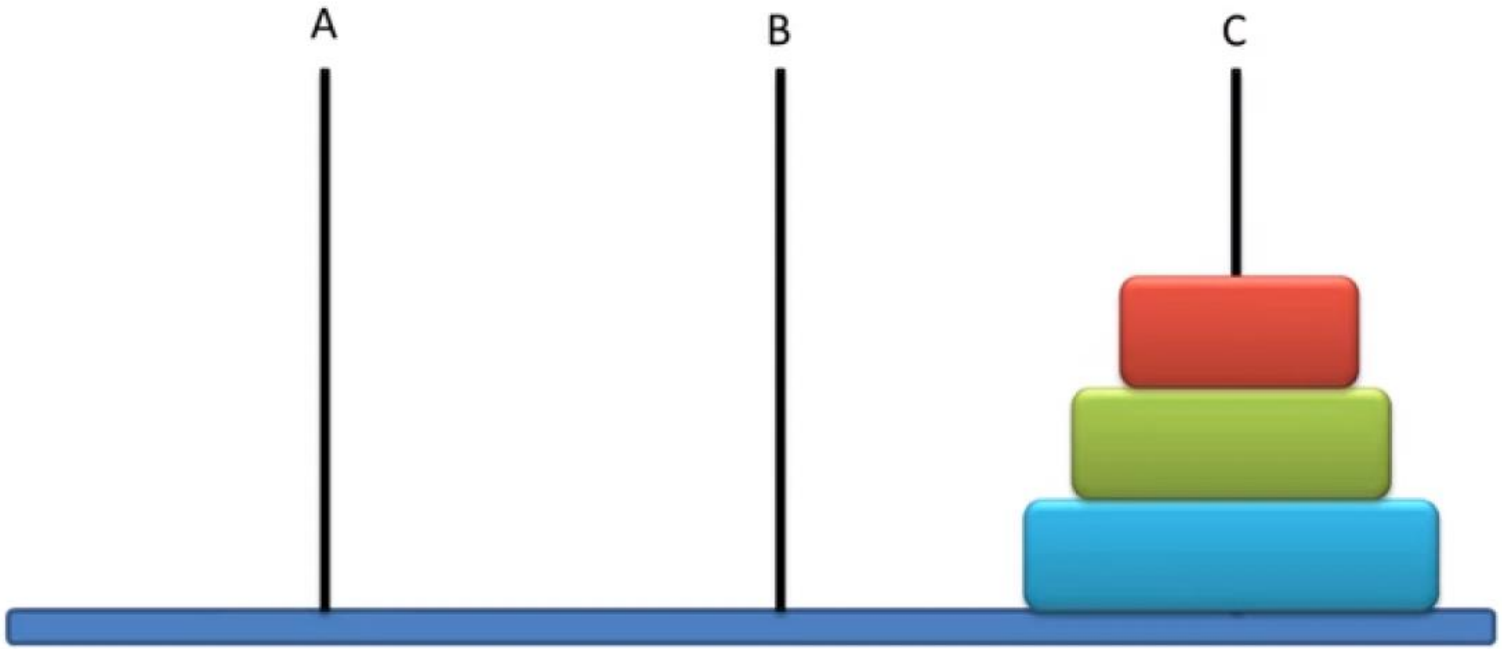
C → B ✓

A → C ✓

B → A ✓

B → C ✓

A → C ✓



Conclusion

- Easier to solve problems which are difficult otherwise
- Takes less code to write comparative to iterative version – cleaner and compact
- Provides no storage or time saving
- Push and Pop of stack frame for each function call/return

Acknowledgment

- Content of these slides are taken from:
  - https://www.geeksforgeeks.org/
  - https://www.tutorialspoint.com/
  - https://www.programiz.com/
  - https://www.w3schools.com/