

CS100

Introduction to Programming

Lecture 10. Recursion

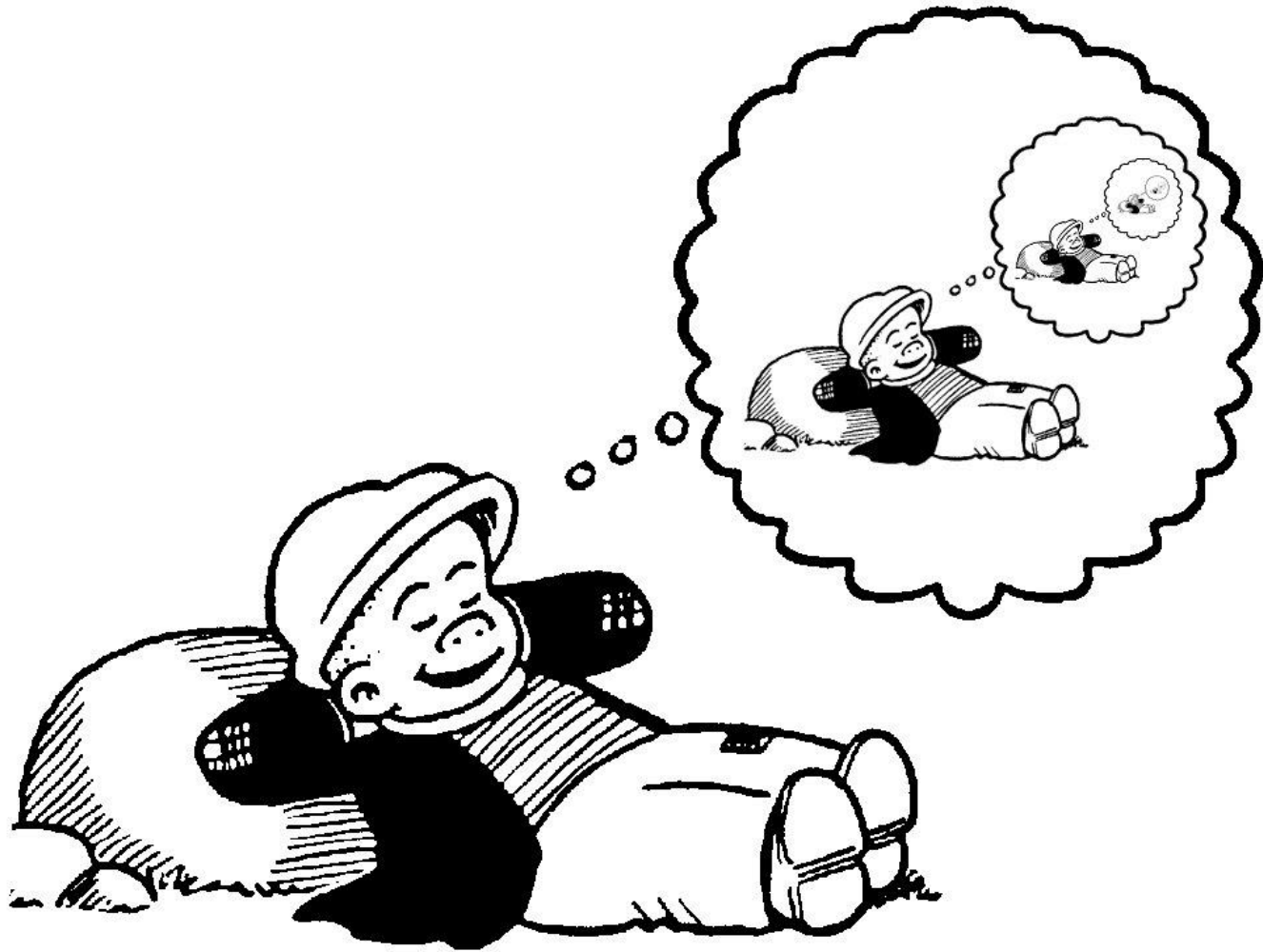
Learning Objectives

- At the end of this lecture, you should be able to understand the following:
 - Recursive Methods
 - Recursive Cheers
 - Recursive PrintSomething
 - Recursive Factorials
 - Recursive Multiplication by Addition
 - Recursive PrintDigits
 - Recursive SumArray
 - Tower of Hanoi

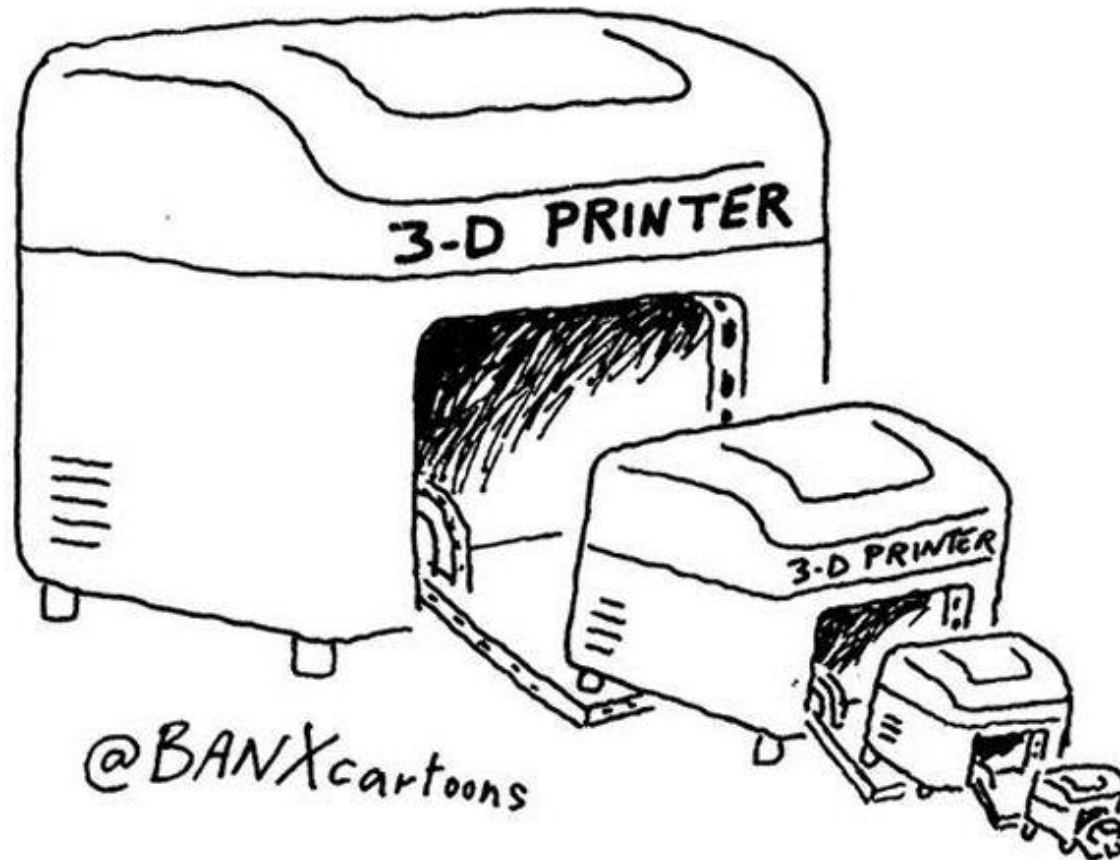
What Is Recursion?

- The method in which a problem is solved by reducing it to smaller cases of the same problem.
- Recursion is the name for the case when
 - a function calls itself, or
 - calls a sequence of other functions, one of which eventually calls the first function again.
- Two parts
 - **Base case** (with terminating condition)
 - **Recursive case** (with recursive condition)

Recursion in Art and Life



Recursion in Art and Life



Recursion in Art and Life



FROM THE MIRROR.

*Life Mirror Recursive (1909),
by Coles Phillips (1880 – 1927)*

Recursion in Art and Life



Example 1: Cheers

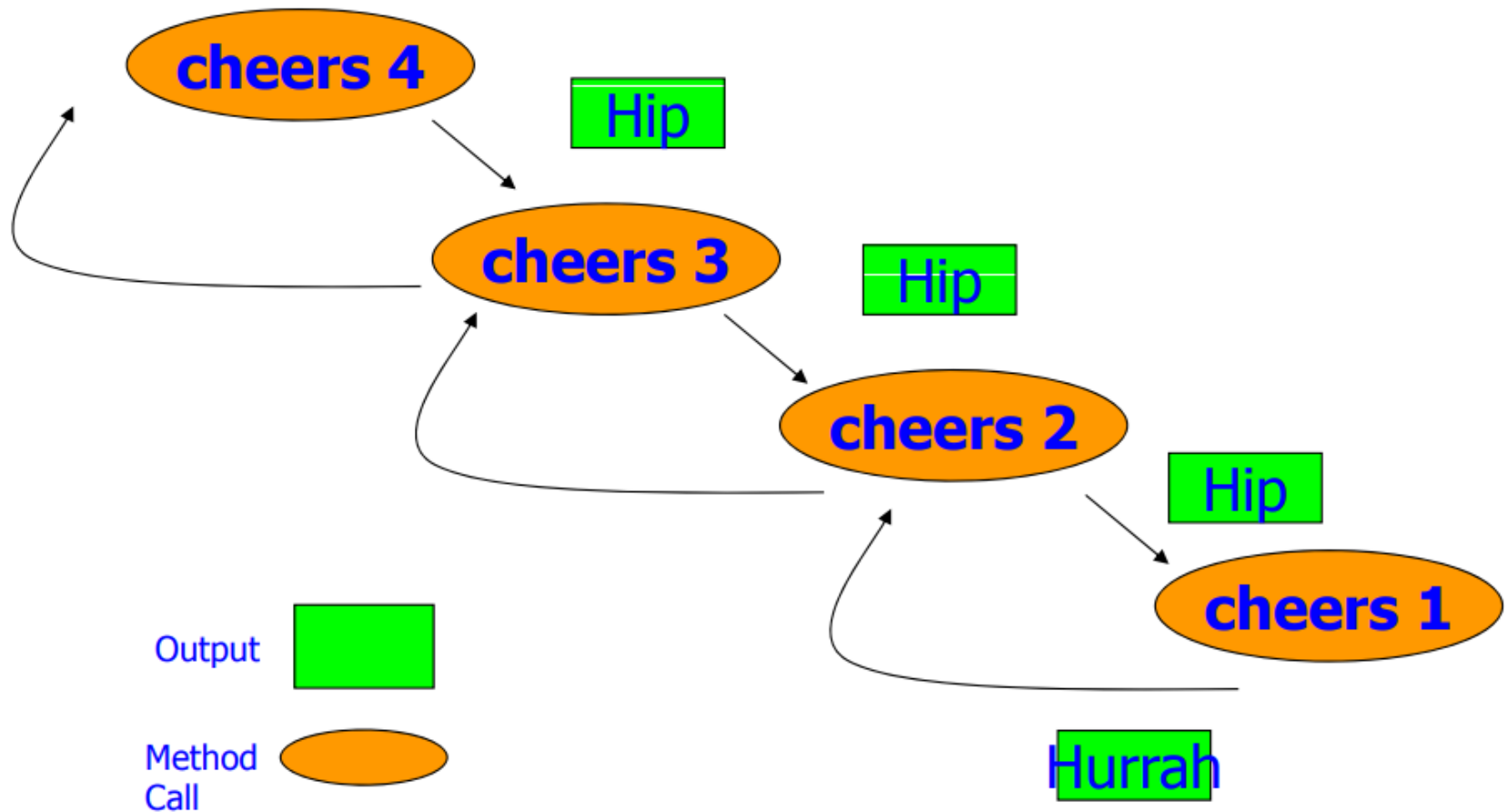
What does the following **recursive** method print when called with `cheers(4)`?

```
void cheers(int n)
{
    if (n <= 1)
        printf("Hurrah \n");
    else {
        printf("Hip \n");
        cheers(n-1);
    }
}
```

Output:

Hip
Hip
Hip
Hurrah

Recursive Cheers – Tracing



Recursive Cheers

```
void cheers(int n)
{
    if (n <= 1)
        printf("Hurrah \n");
    else {
        printf("Hip \n");
        cheers(n-1);
    }
}
```

terminating condition

recursive condition
(n > 1)

Example 2: PrintSomething

What does the following **recursive** method print when called with `printSomething(3)`?

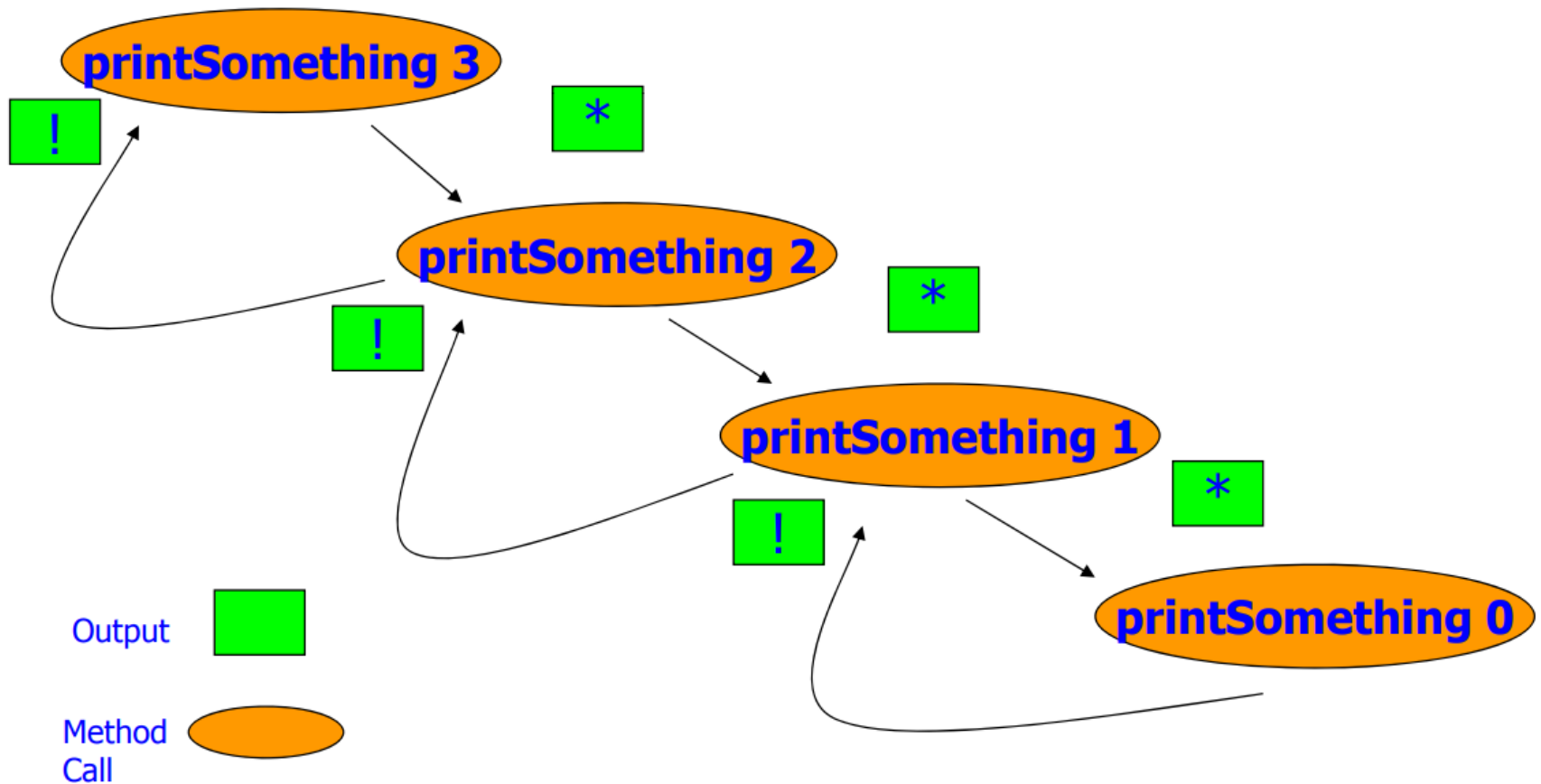
```
void printSomething(int n)
{
    if (n > 0) {
        printf("*");
        printSomething(n-1);
        printf("!");
    }
}
```

What condition is this?
Continue looping condition

Output:
***!!!

Where is the other condition?

Recursive PrintSomething – Tracing



Example 3: Factorials

- **Problem:** Find the factorial of a non-negative integer number.
- The factorial function of a positive integer is usually defined by the formula

$$n! = n \times (n - 1) \times \dots \times 1$$

- A more precise definition (recursive definition)

$$n! = 1 \quad \text{if } n = 0$$

$$n! = n \times (n - 1)! \quad \text{if } n > 0$$

Non-recursive Factorials

The following program will do the job, using a **for** loop:

```
#include <stdio.h>
int factorial(int n);
int main(void)
{
    int num;
    printf("Enter an integer:");
    scanf("%d", &num);
    printf("n! = %d\n",
        factorial(num));
    return 0;
}
```

```
int factorial(int n)
{
    int i;
    int temp = 1;
    for (i=n; i > 0; i--)
        temp *= i;
    return temp;
}
```

Output:

Enter an integer: 4
n! = 24

Recursive Factorials

- Suppose we wish to calculate $4!$

As $4 > 0$, $4! = 4 \times 3!$

- But we do not know what is $3!$

As $3 > 0$, $3! = 3 \times 2!$

As $2 > 0$, $2! = 2 \times 1!$

As $1 > 0$, $1! = 1 \times 0!$

- What is $0!$ equal to?

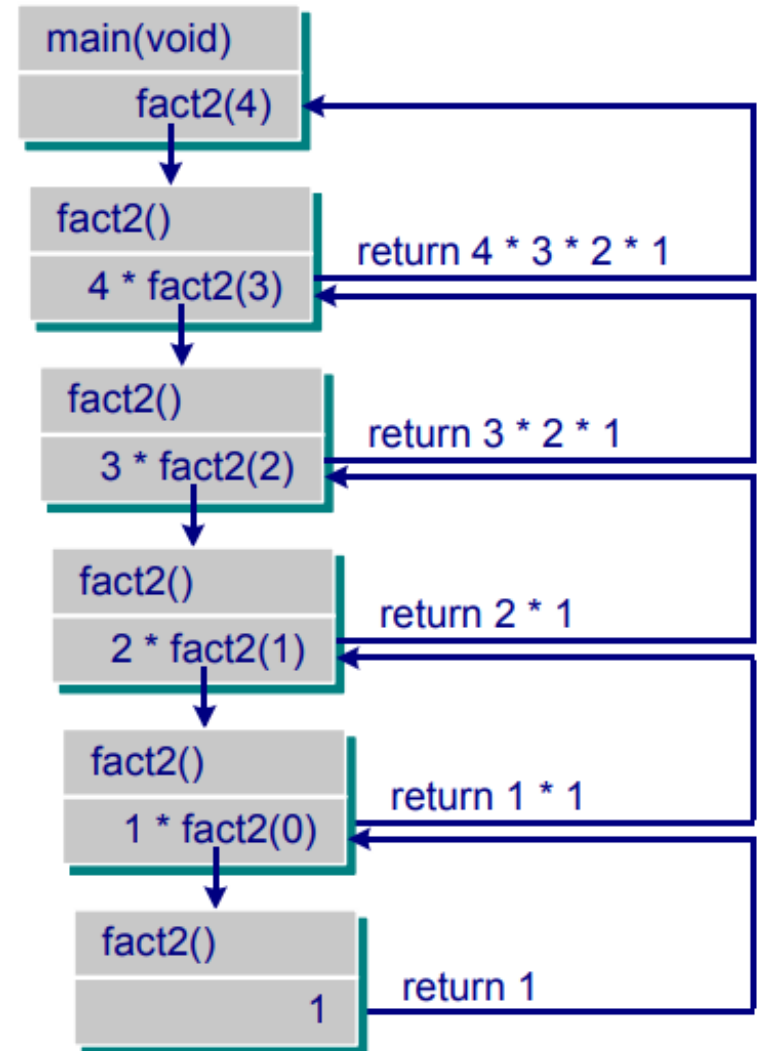
$$\begin{aligned}4! &= 4 \times 3! \\&= 4 \times (3 \times 2!) \\&= 4 \times (3 \times (2 \times 1!)) \\&= 4 \times (3 \times (2 \times (1 \times 0!))) \\&= 4 \times (3 \times (2 \times (1 \times 1))) \\&= 4 \times (3 \times (2 \times 1)) \\&= 4 \times (3 \times 2) \\&= 4 \times 6 \\&= 24\end{aligned}$$

Recursive Factorials

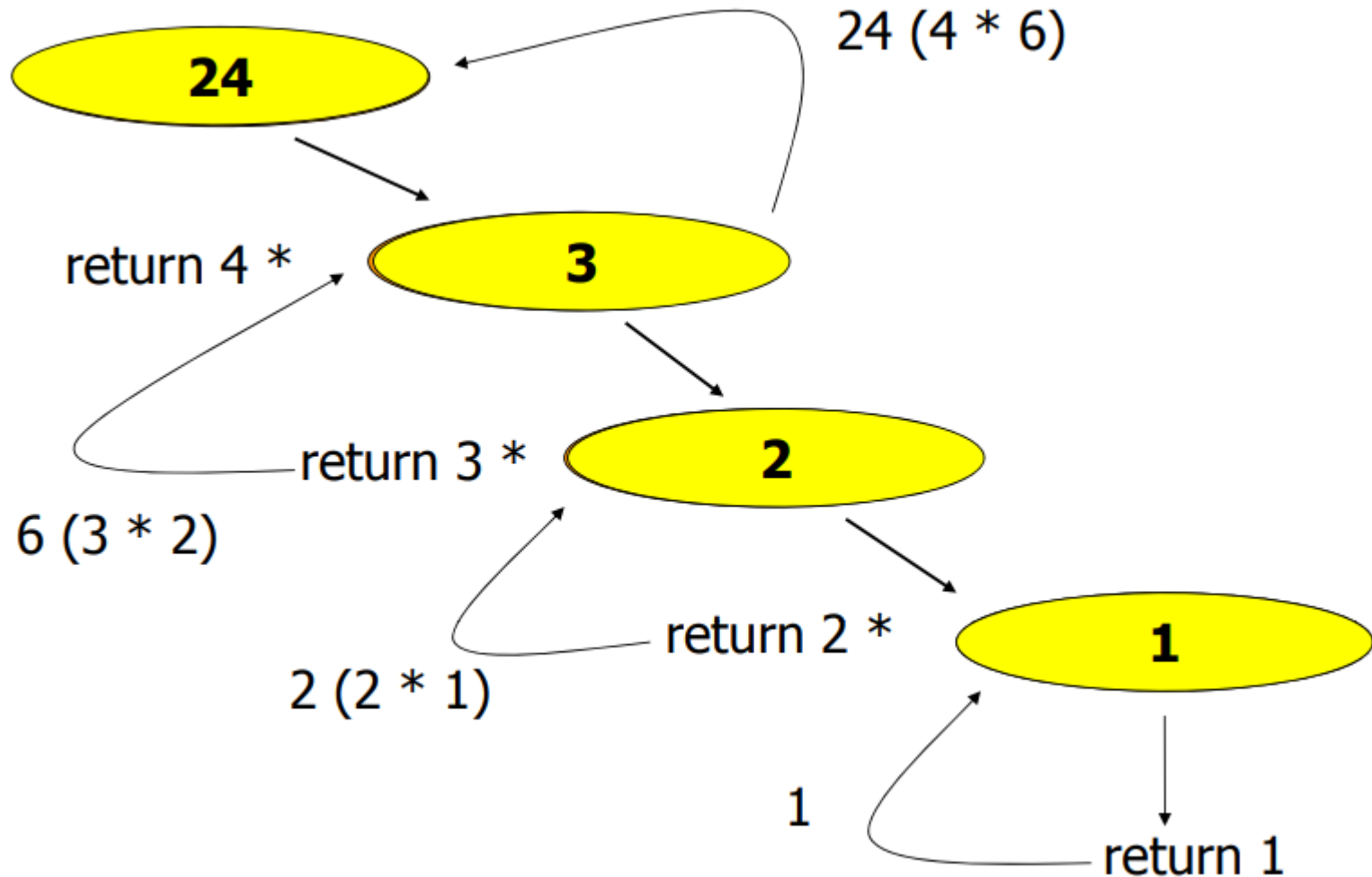
```
int factorial(int n)
{
    if (n == 0) {
        // terminating condition
        return 1;
    } else {
        // recursive condition
        return n*factorial(n-1);
    }
}
```

Output:

Enter an integer: 4
n! = 24



Recursive Factorials: Tracing



Example 4: Multiplication by Addition

- The multiplication operation

$$\text{multi}(\textcolor{red}{m}, \textcolor{red}{n}) = m \times n$$

can be defined recursively as

$$\text{multi}(\textcolor{red}{m}, \textcolor{red}{n}) = \textcolor{red}{m} \quad \text{if } n = 1$$

$$\text{multi}(\textcolor{blue}{m}, \textcolor{blue}{n}) = \textcolor{blue}{m} + \text{multi}(\textcolor{blue}{m}, n-1) \quad \text{if } n > 1$$

Using Call by Value

```
/* multiplication by addition, pass parameter using  
call by value */  
#include <stdio.h>  
int multi1(int, int);  
int main(void)  
{  
    printf("5 * 3 = %d\n", multi1(5, 3));  
    return 0;  
}  
int multi1(int m, int n)  
{  
    if (n == 1)                // terminating condition  
        return m;  
    else                        // recursive condition  
        return m + multi1(m, n-1);  
}
```

Output:

5 * 3 = 15

Using Call by Reference

```
/* multiplication by addition, pass parameter using call by  
reference */  
#include <stdio.h>  
void multi2(int, int, int*);  
int main(void)  
{  
    int result;  
    multi2(5, 3, &result);  
    printf("5 * 3 = %d\n", result);  
    return 0;  
}  
void multi2(int m, int n, int *product)  
{  
    if (n == 1)                // terminating condition  
        *product = m;  
    else {                    // recursive condition  
        multi2(m, n-1, product);  
        *product = *product + m;  
    }  
}
```

Output:

5 * 3 = 15

Recursive Call by Reference

```
#include <stdio.h>
void fn(int x, int y, int *z);
```

```
int main(void)
{
    int n1=20, n2=15, n3=0;
    fn(n1, n2, &n3);
    printf("n1 = %d, n2 = %d, n3 = %d\n", n1, n2, n3);
    return 0;
}

void fn(int x, int y, int *z)
{
    if (x > y) {
        x = x - 2;
        y = y - 1;
        *z = x * y;
        fn(x, y, z);
    }
    printf("x = %d, y = %d, *z = %d\n", x, y, *z);
}
```

Output:

```
x = 10, y = 10, *z = 100
x = 12, y = 11, *z = 100
x = 14, y = 12, *z = 100
x = 16, y = 13, *z = 100
x = 18, y = 14, *z = 100
n1 = 20, n2 = 15, n3 = 100
```

Example 5: Printing Digits

- **Problem:** Given a number, say 2345, print each digit of the number, in the order from left to right, one per line.
- Recursive Solution:
 - Look for the simplest case (**terminating condition**). If the number is a single digit, just print that digit.
 - Look into reducing the problem into a smaller but same problem (**recursive condition**).

Printing Digits

PrintDigit 2345
PrintDigit 234

output 5

PrintDigit 23

output 4

PrintDigit 2

output 3

output 2

Reduce to a smaller problem

The simple case of 1 digit

Printing Digits: Recursive Solution

```
#include <stdio.h>
void printDigit(int);

int main(void) {
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);
    printDigit(num);
    return 0;
}
```

```
void printDigit(int num) {
    if (num < 10)                // terminating condition
        printf("%d\n", num);
    else {                       // recursive condition
        printDigit(num/10);
        printf("%d\n", num%10);
    }
}
```

Output:

Enter a number: 2345

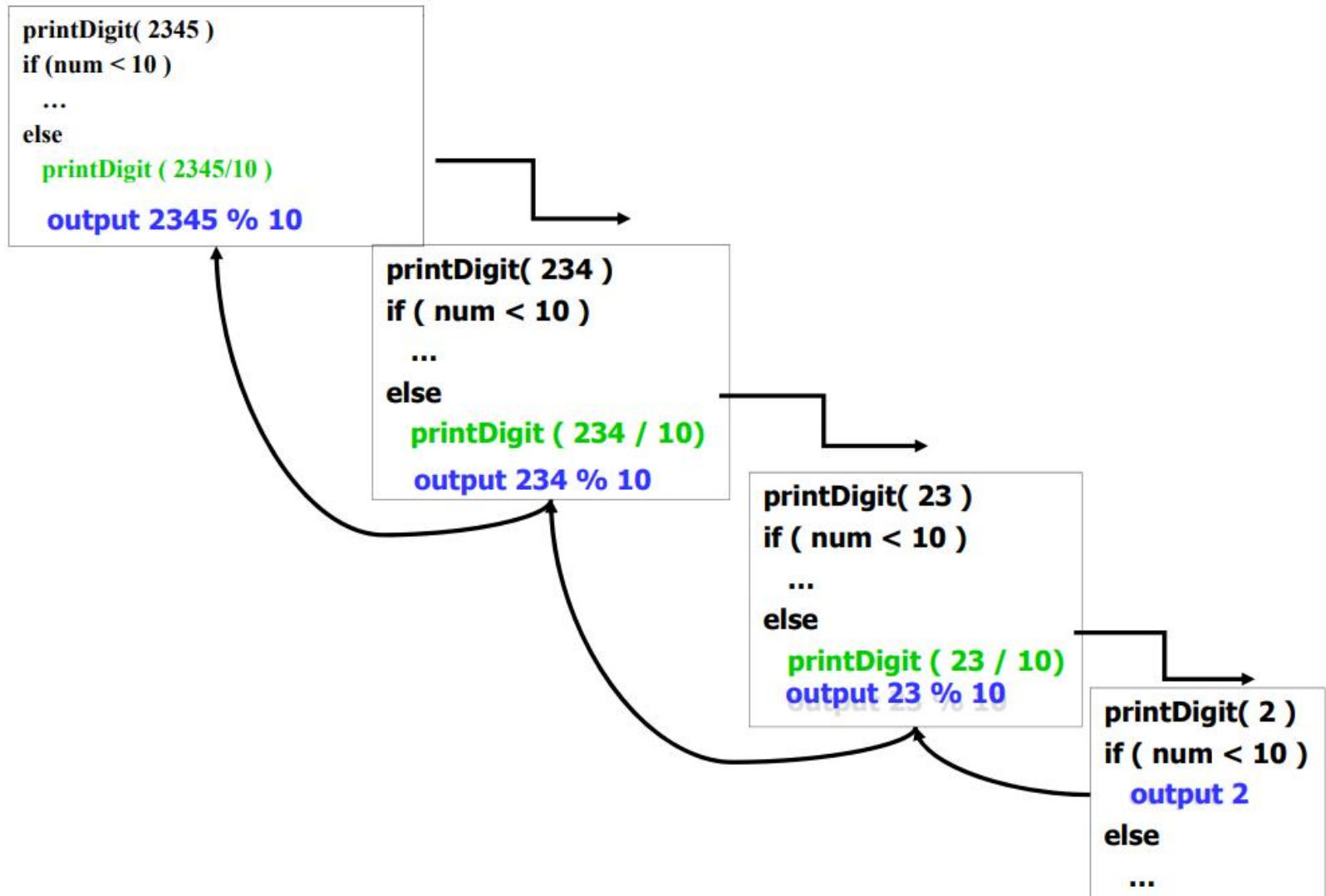
2

3

4

5

printDigit(2345);



Recursion: Summary

- To obtain the answer to a larger problem, a general method is used that
 - reduces the large problem to one or more problems of a similar nature but a smaller size.
- Recursion continues until the size of the problem is reduced to the smallest, i.e. base case
 - where the solution is given directly without using further recursion.
- As such, recursive methods consist of two parts:
 - A smallest, **base case** that is processed without recursion (**terminating condition**).
 - A general method that reduces a particular case to one or more of the smaller cases (**recursive condition**).

Recursion: Summary

- Each function makes a **call to itself** with an argument which is closer to the terminating condition.
- Each call to a function has its **own set of values/arguments** for the formal arguments and local variables.
- When a recursive call is made, control is transferred from the calling point to the first statement of the recursive function. When a call at a certain level is finished, control returns to the calling point **one level up**.

Example 6: Summing Array of Integers

```
int sumArray(int a[], int size)
{
    int sum = 0;
    for (int i = 0; i < size; i++)
    {
        sum = sum + a[i];
    }
    return sum;
}

int recursiveSum(int a[], int size)
{
    if (size == 1)
        return a[0];
    else
        return a[size - 1] +
            recursiveSum(a, size - 1);
}
```

- Given an array of integers, write a method to calculate the sum of all the integers.

a

1	2	3	4
[0]	[1]	[2]	[3]

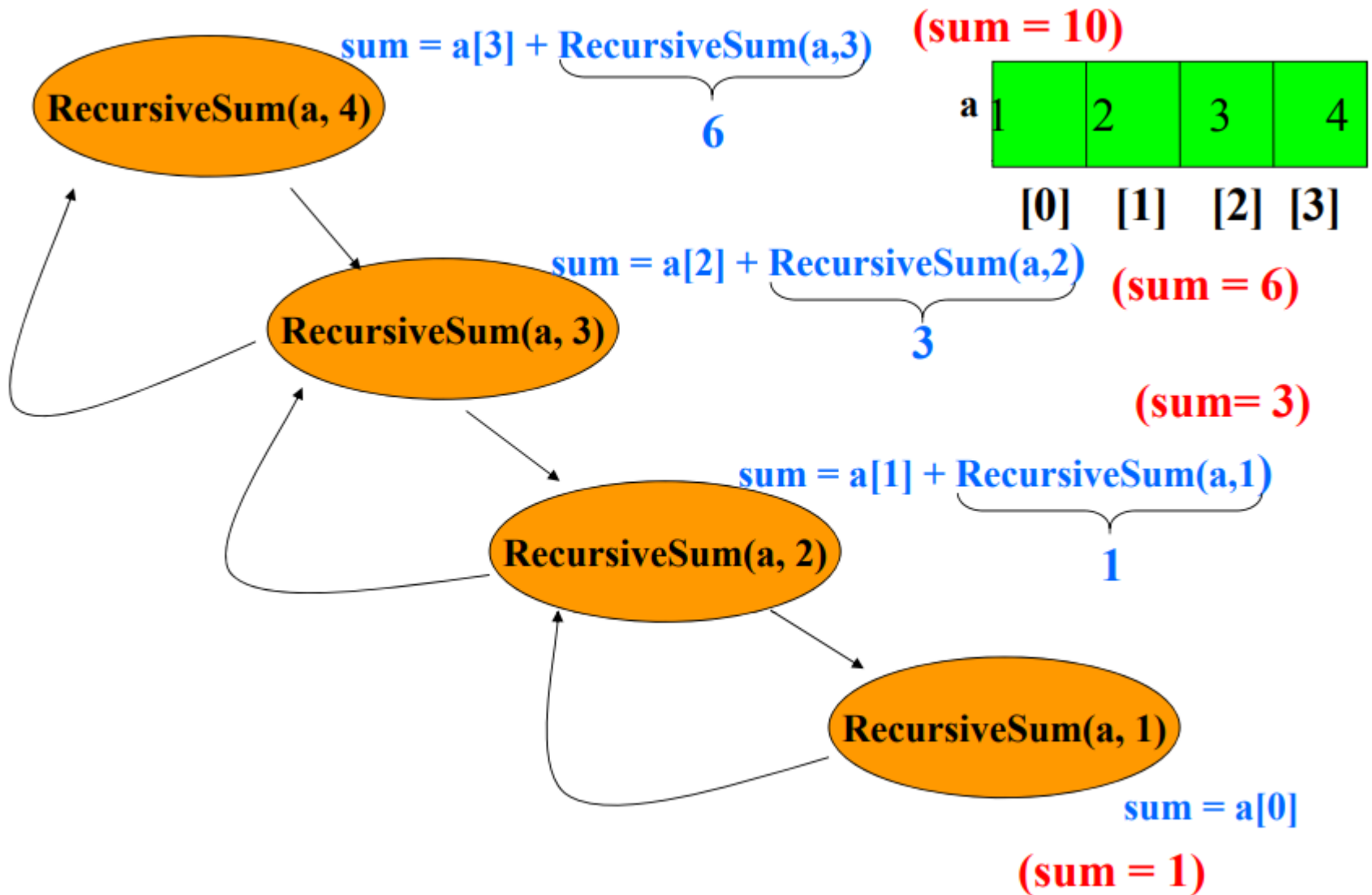
Input:

$a[] = \{1, 2, 3, 4\}$

Output:

Sum = 10

Recursive Sum – Tracing

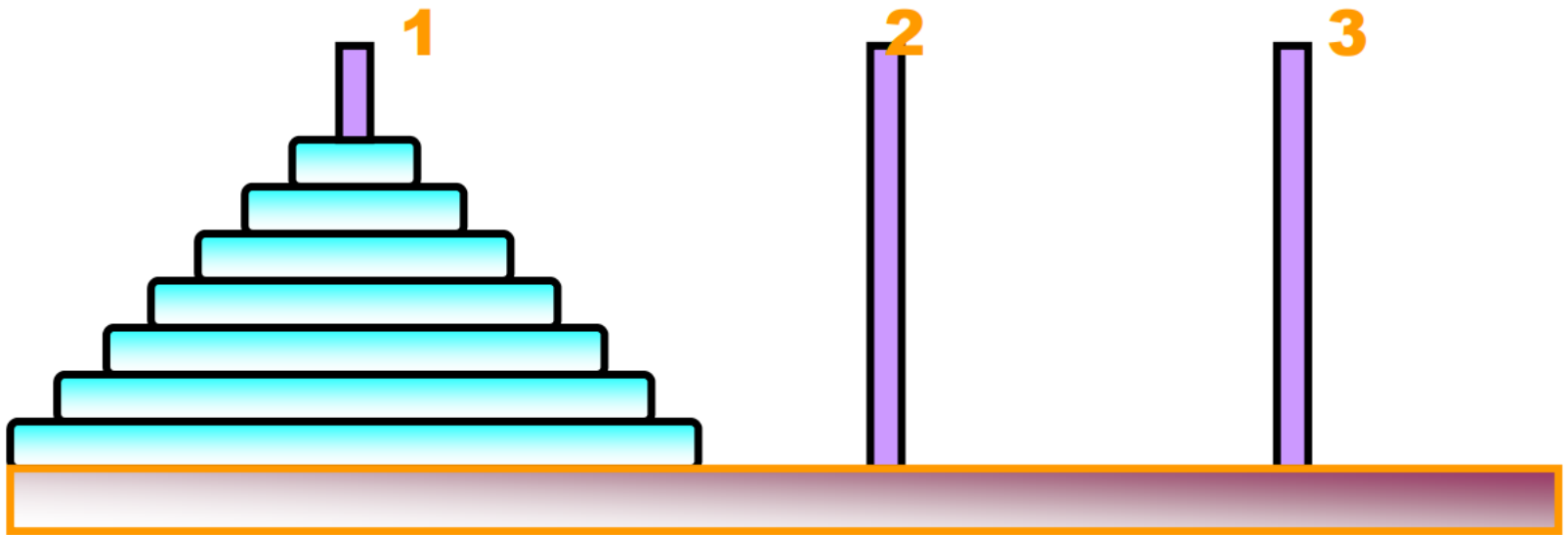


Example 7: Tower of Hanoi

- In the 19th century, a game called the **Towers of Hanoi** appeared in Europe.
- The game represents a task underway in the **Temple of Brahma**.
- At the creation of the world, the priests were given a brass platform on which were **3 diamond needles**.
- On the **first** needle were stacked **64 golden disks**.
- Each one slightly **smaller** than the one under it.
- The priest were assigned the task of moving all the golden disks from the **first** needle to the **third**
 - **Condition:** Only one disk can be moved at a time, and no disk is allowed to be placed on top of a smaller disk.

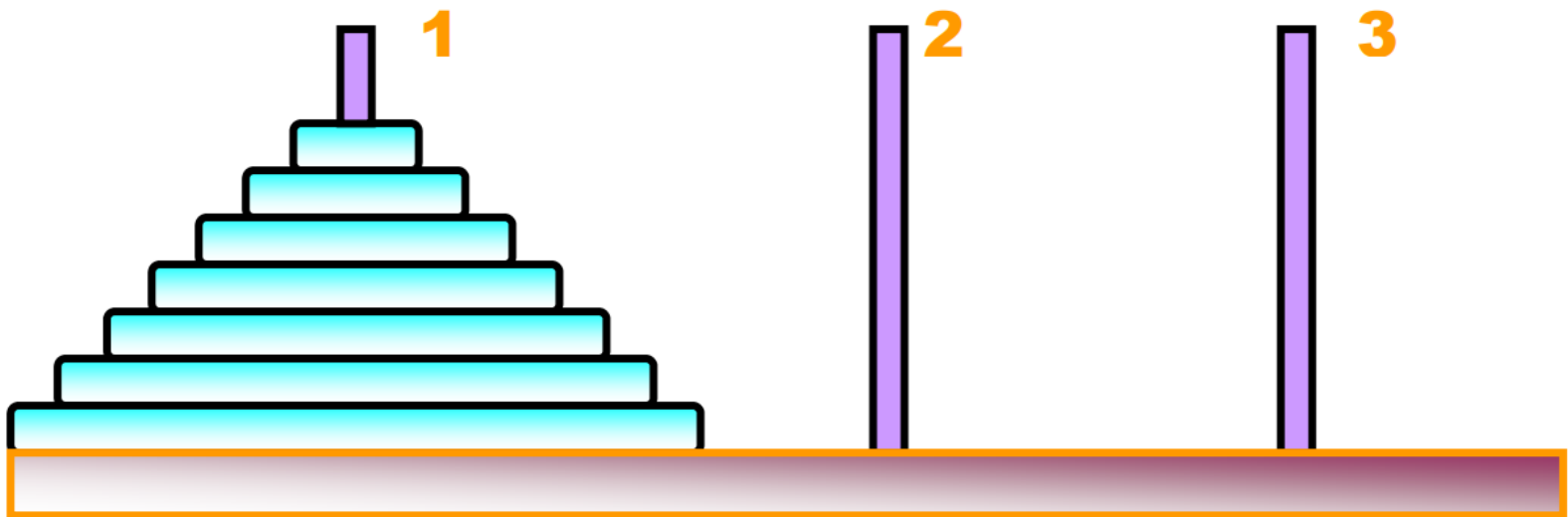
Tower of Hanoi

- **Problem:** The priests were told that, when they had finished moving the 64 disks from tower 1 to tower 3, it would signify the **end of the world!!!**



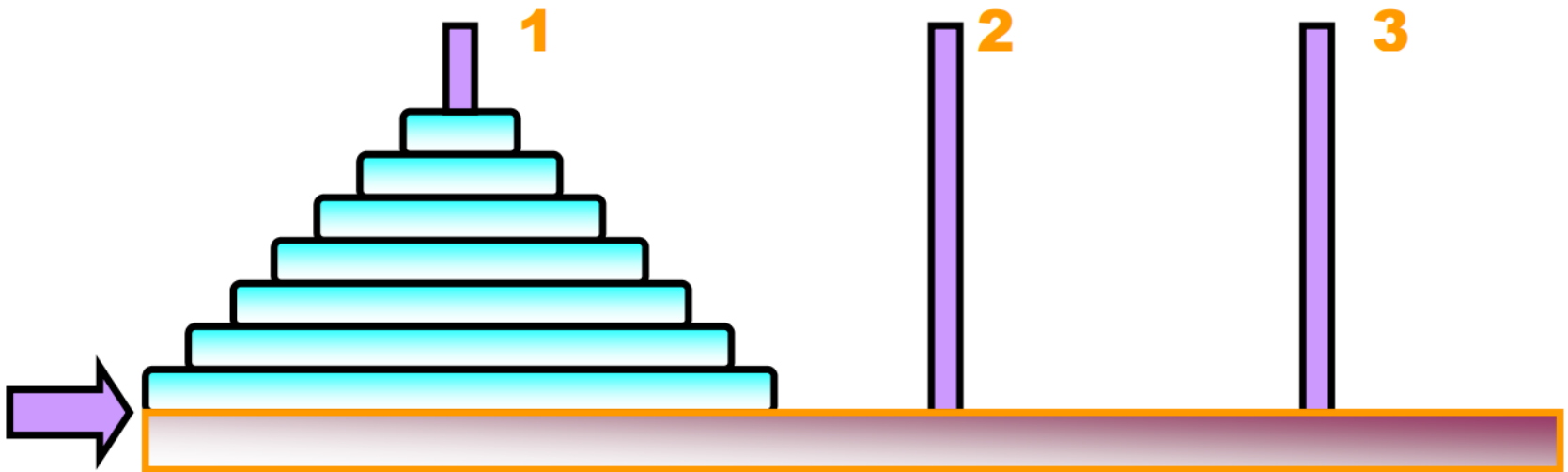
The Problem

- Write a computer program that will type out a list of instructions for the priests
- **move(64, 1, 3, 2)**
 - Move 64 disks from tower 1 to tower 3 using tower 2 as temporary storage



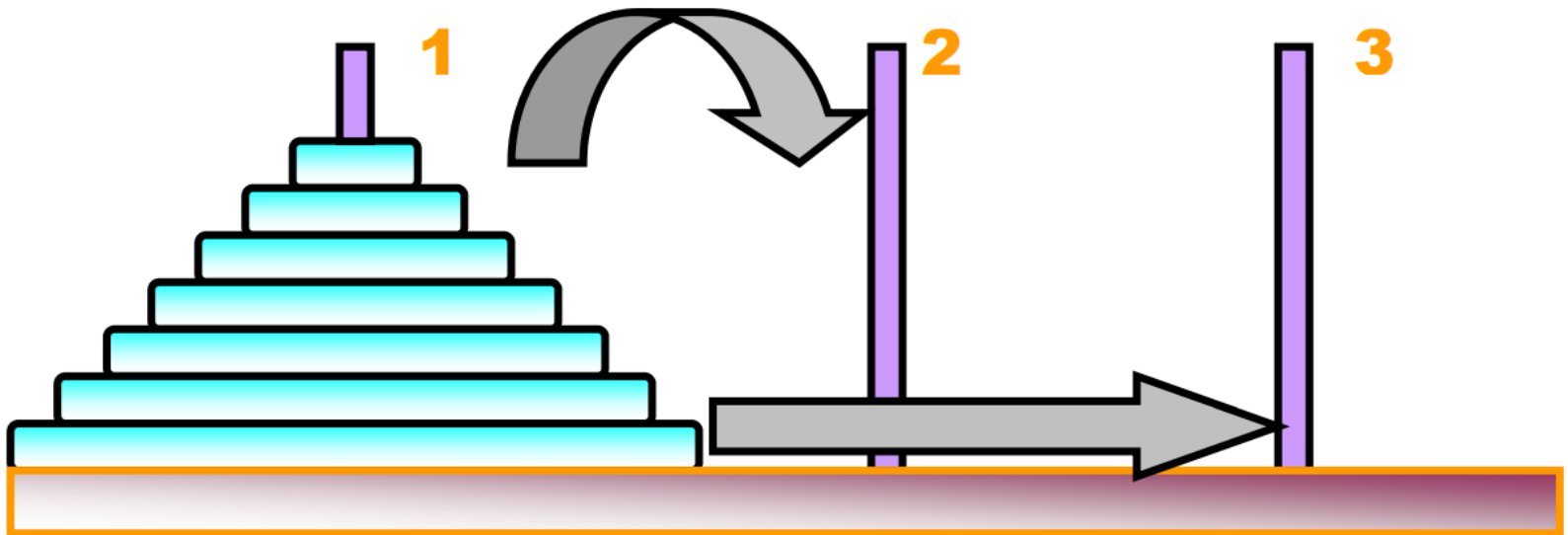
The Solution

- Concentrate our attention not on the first step
 - Move the top disk anywhere
- Rather, on the hardest step
 - Moving the bottom disk



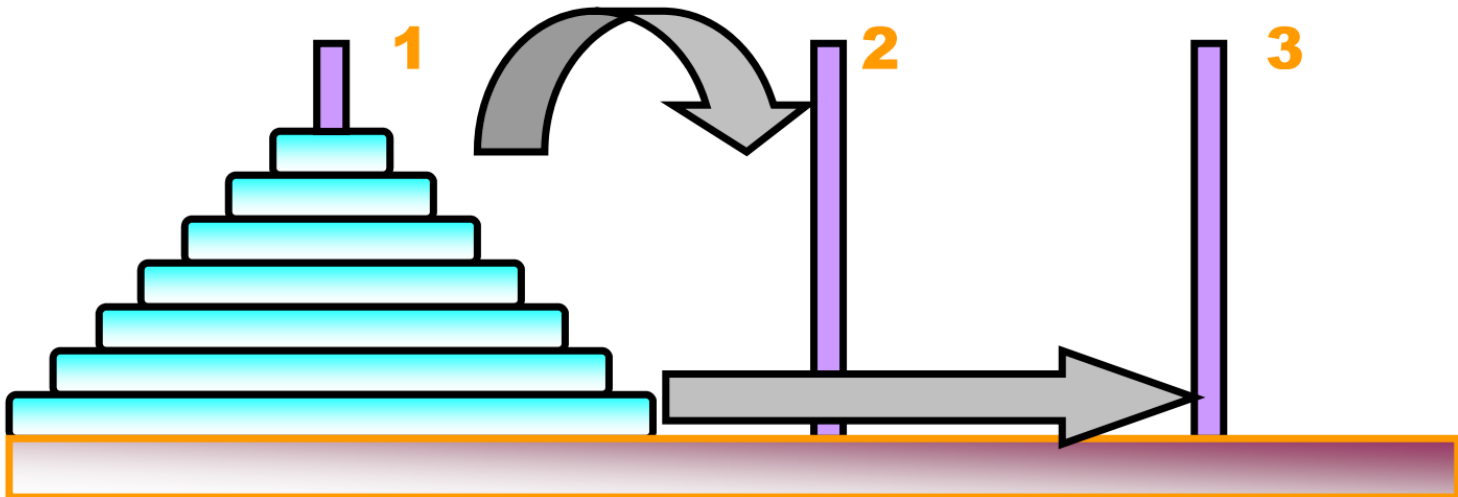
The Solution

- How to reach the bottom disk?
 - The top 63 disks have to be moved to tower 2
 - The bottom disk can then be moved from tower 1 to 3
- There cannot be any other disk in tower 3



The Solution

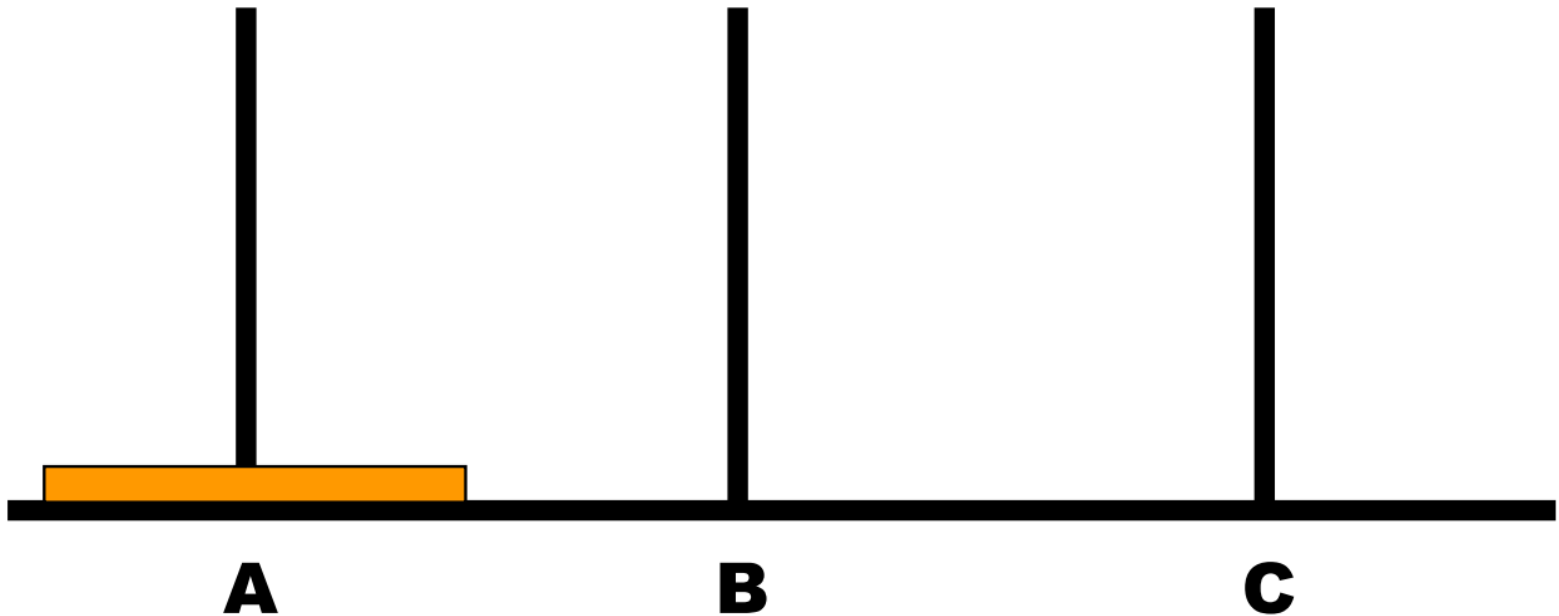
- `move(63, 1, 2, 3);` // Move 63 disks from tower 1 to tower 2 (using tower 3 as temporary)
- Display “Move disk 64 from tower 1 to 3”
- `move (63, 2, 3, 1);` // Move 63 disks from tower 2 to tower 3 (using tower 1 as temporary)



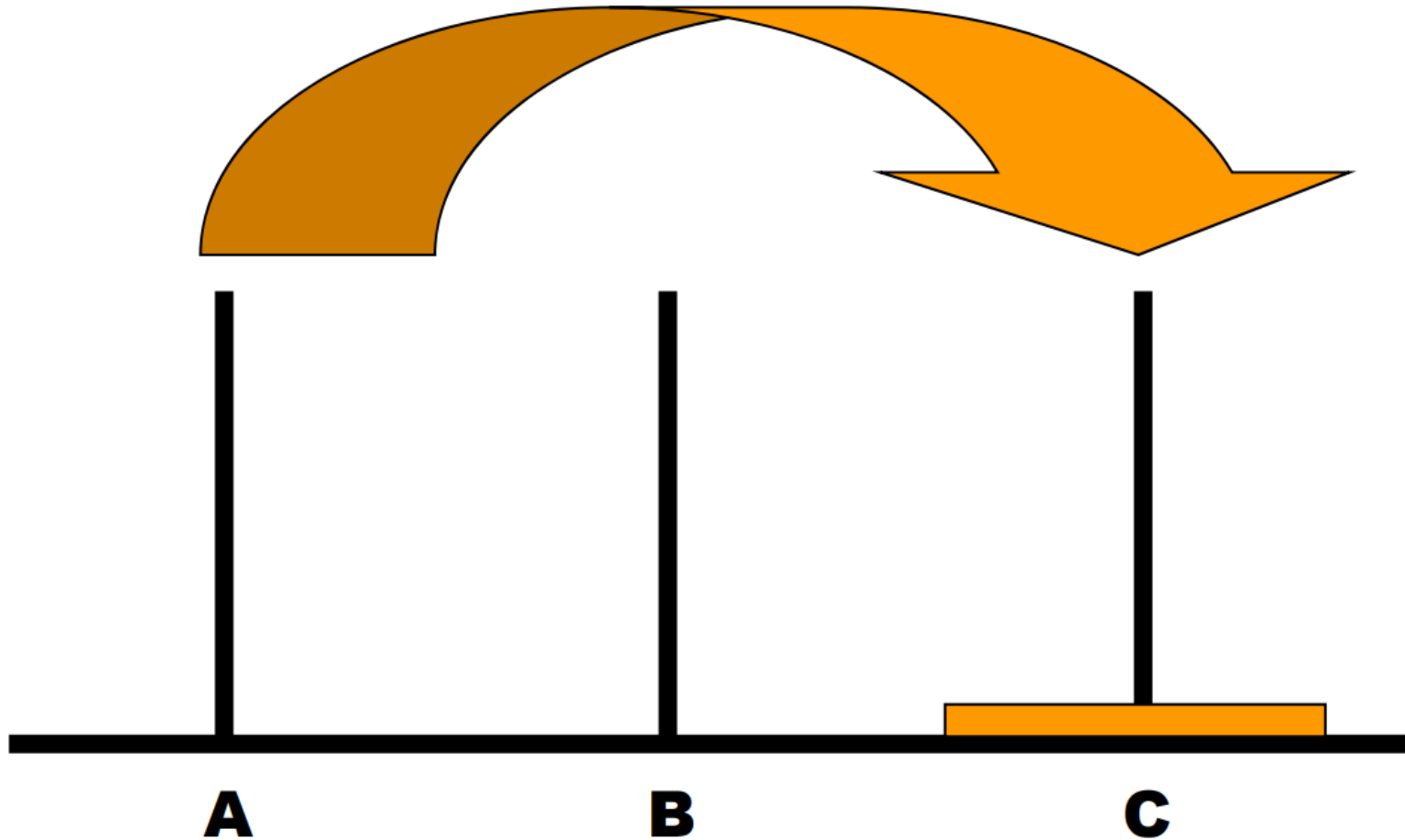
The Solution

- The remaining 63 disks can be moved in the same way
- Divide and conquer
 - To solve a problem we split it into smaller parts which are easier to handle.

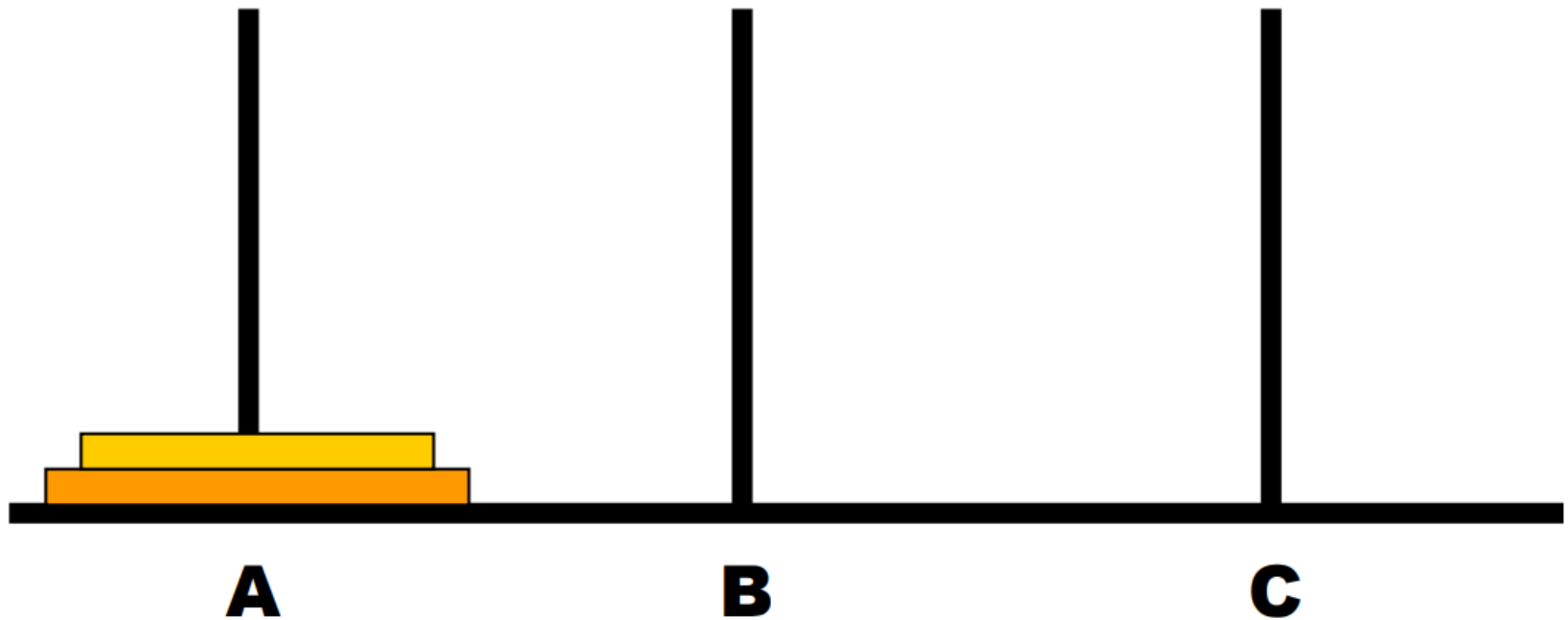
A Single Disk Tower



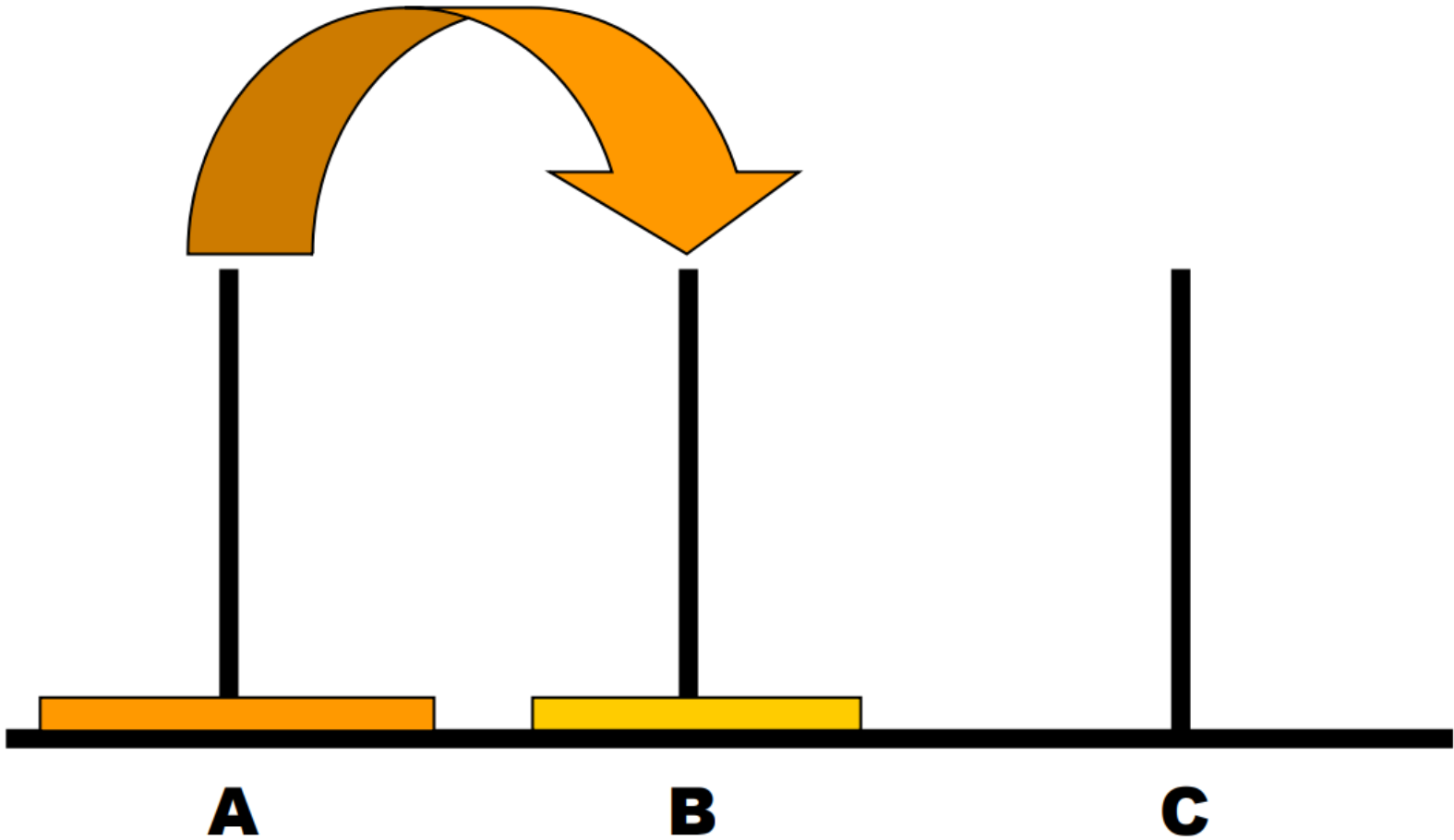
A Single Disk Tower



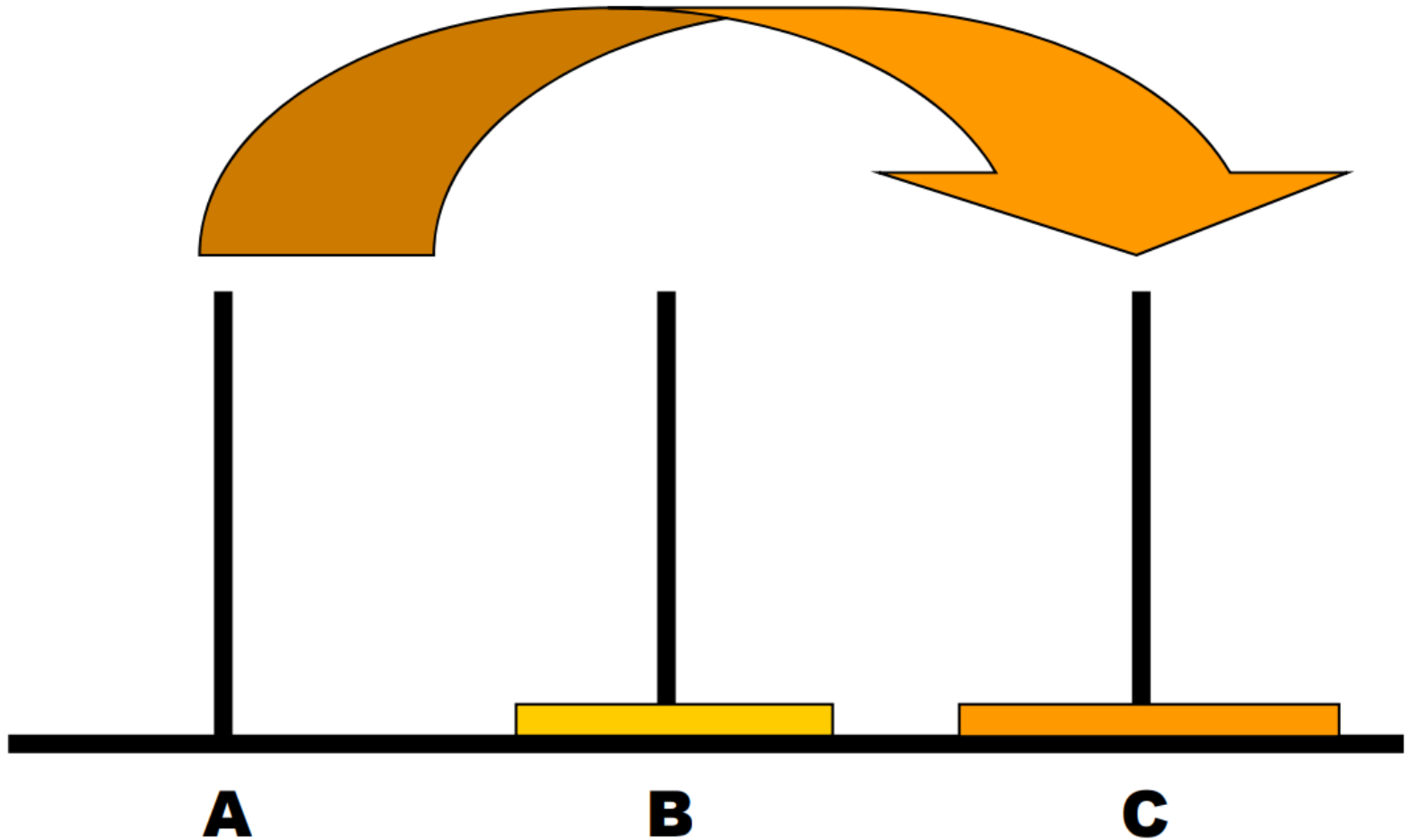
A 2-Disk Tower



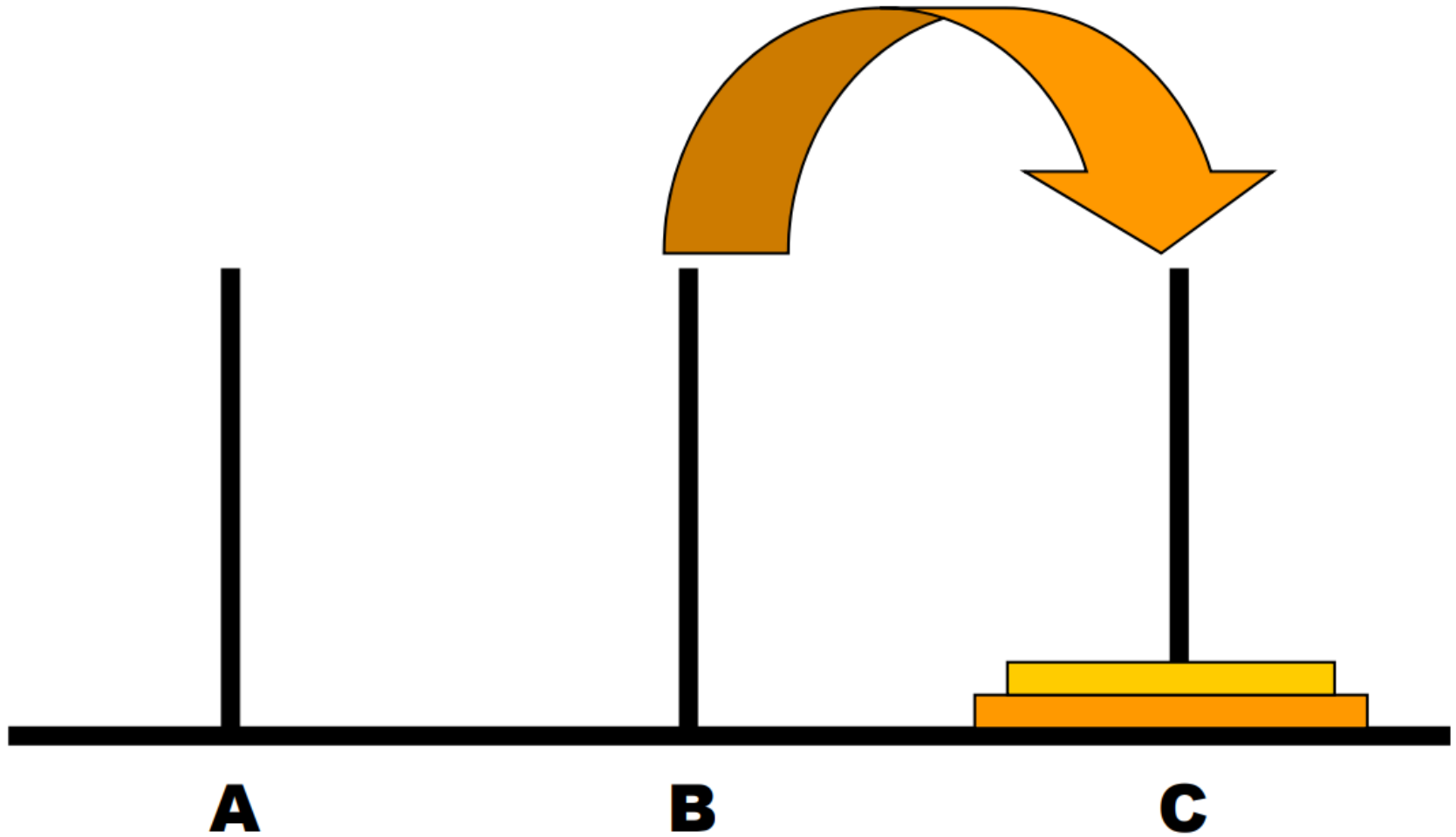
Move 1



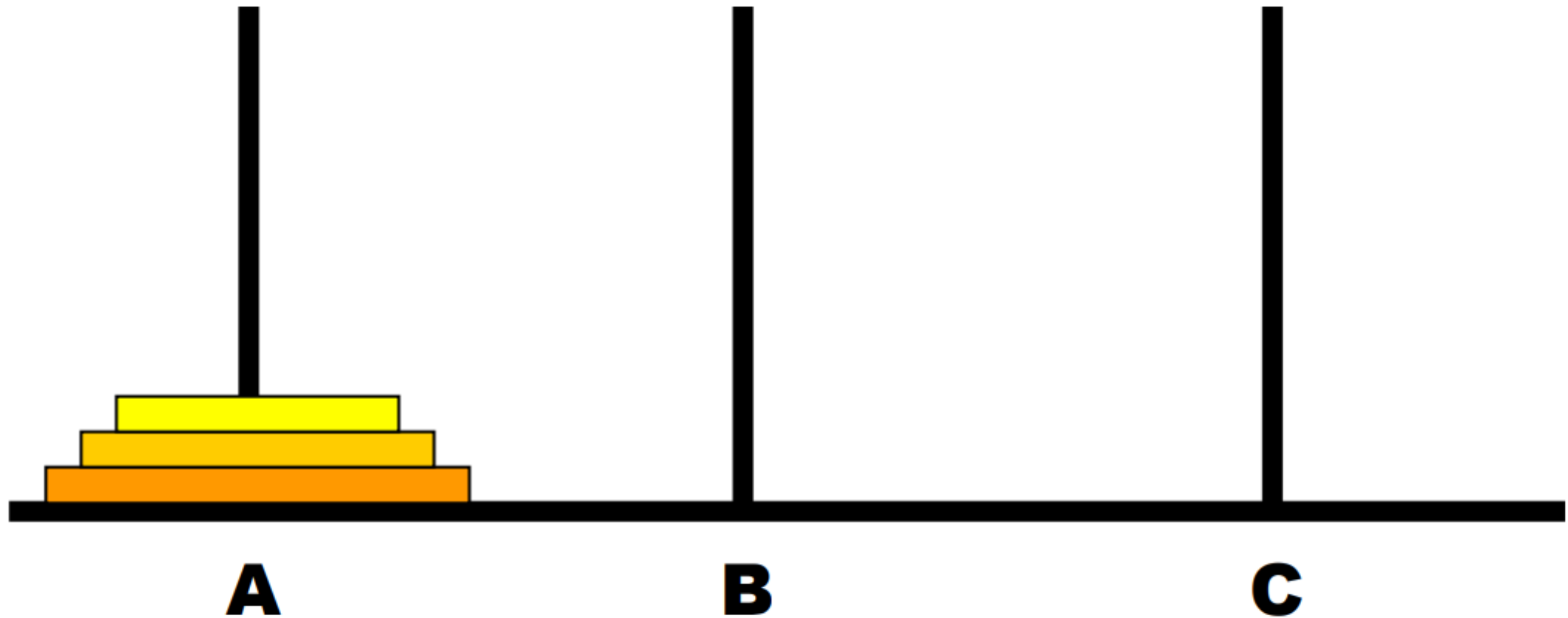
Move 2



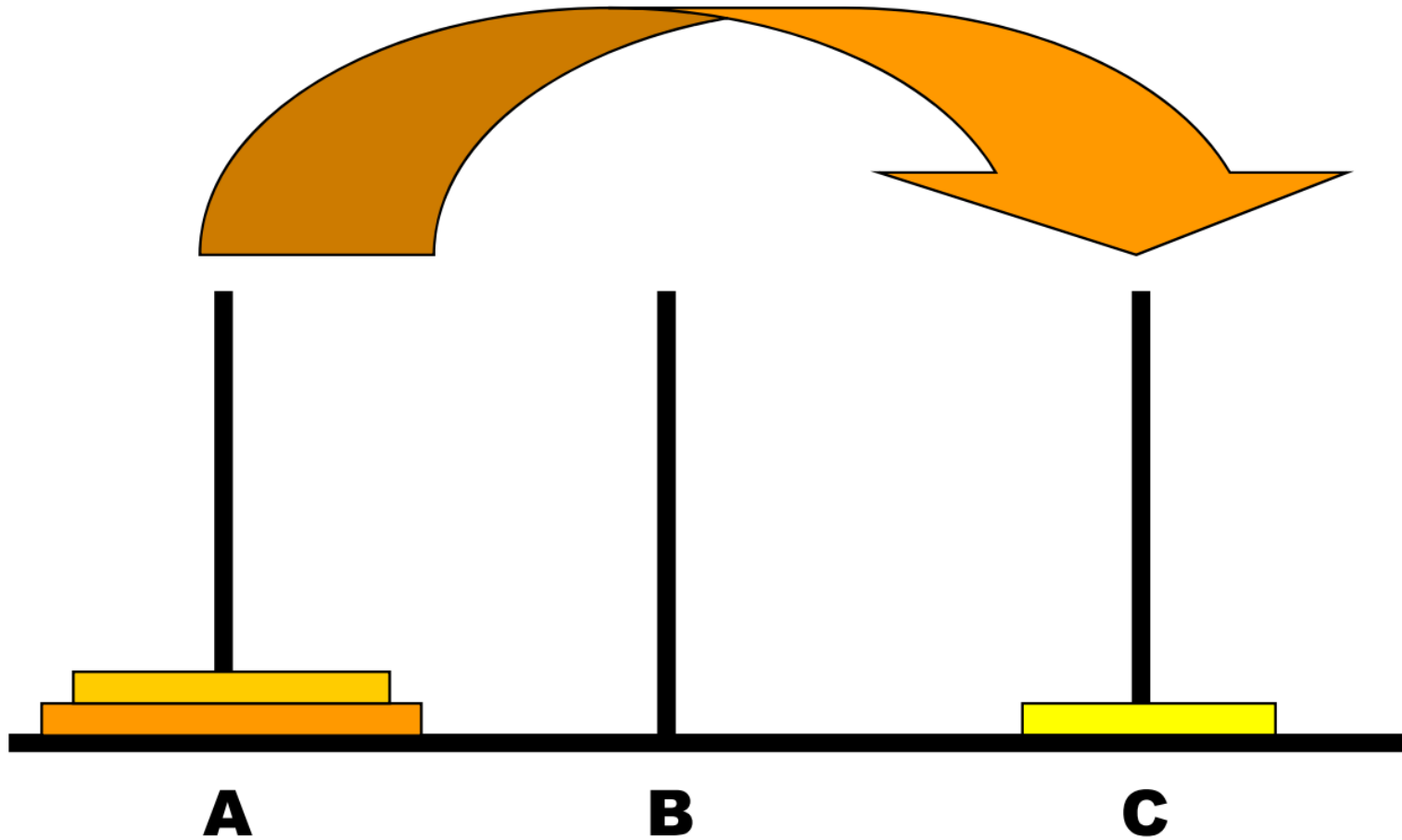
Move 3



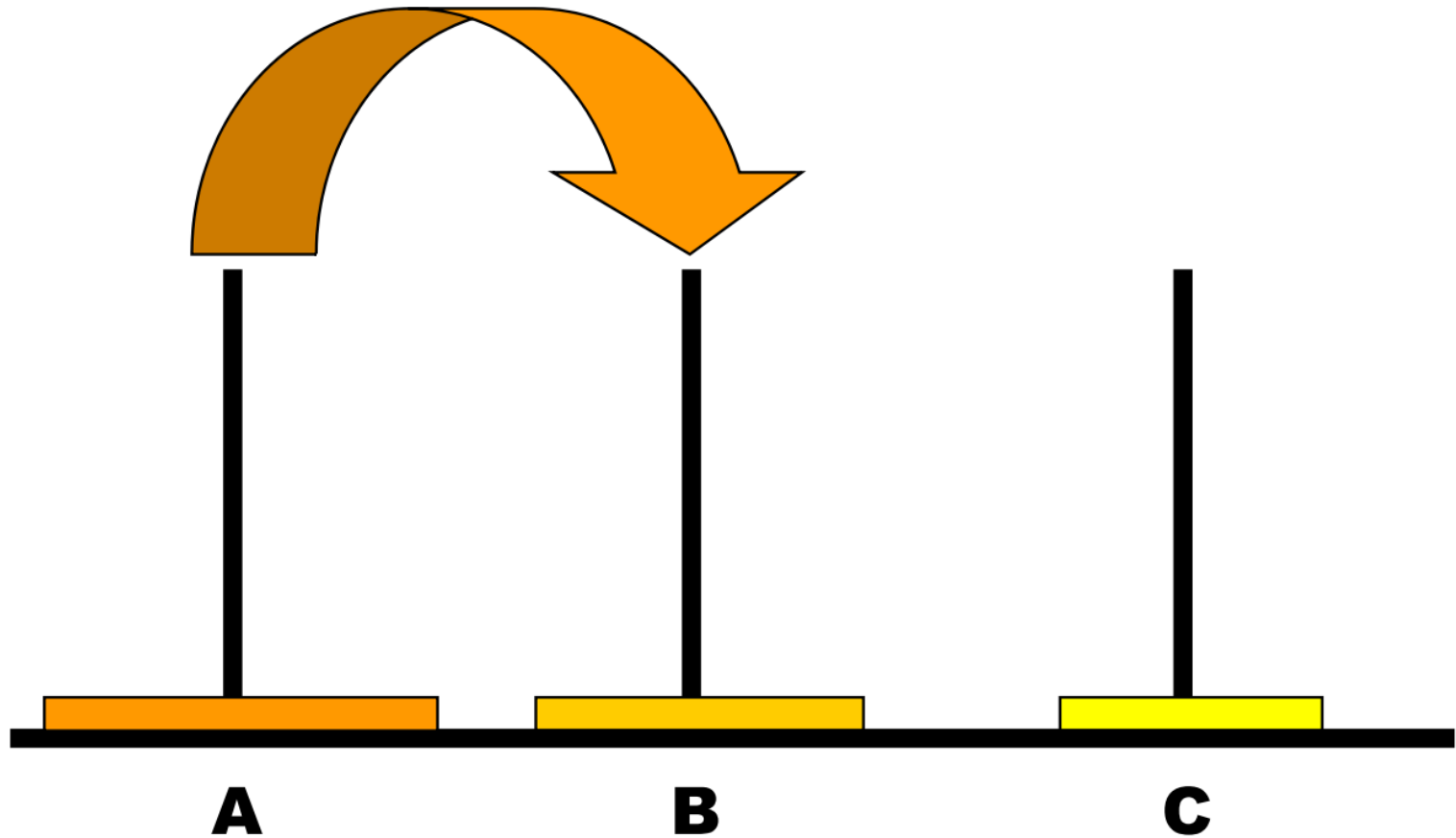
A 3-Disk Tower



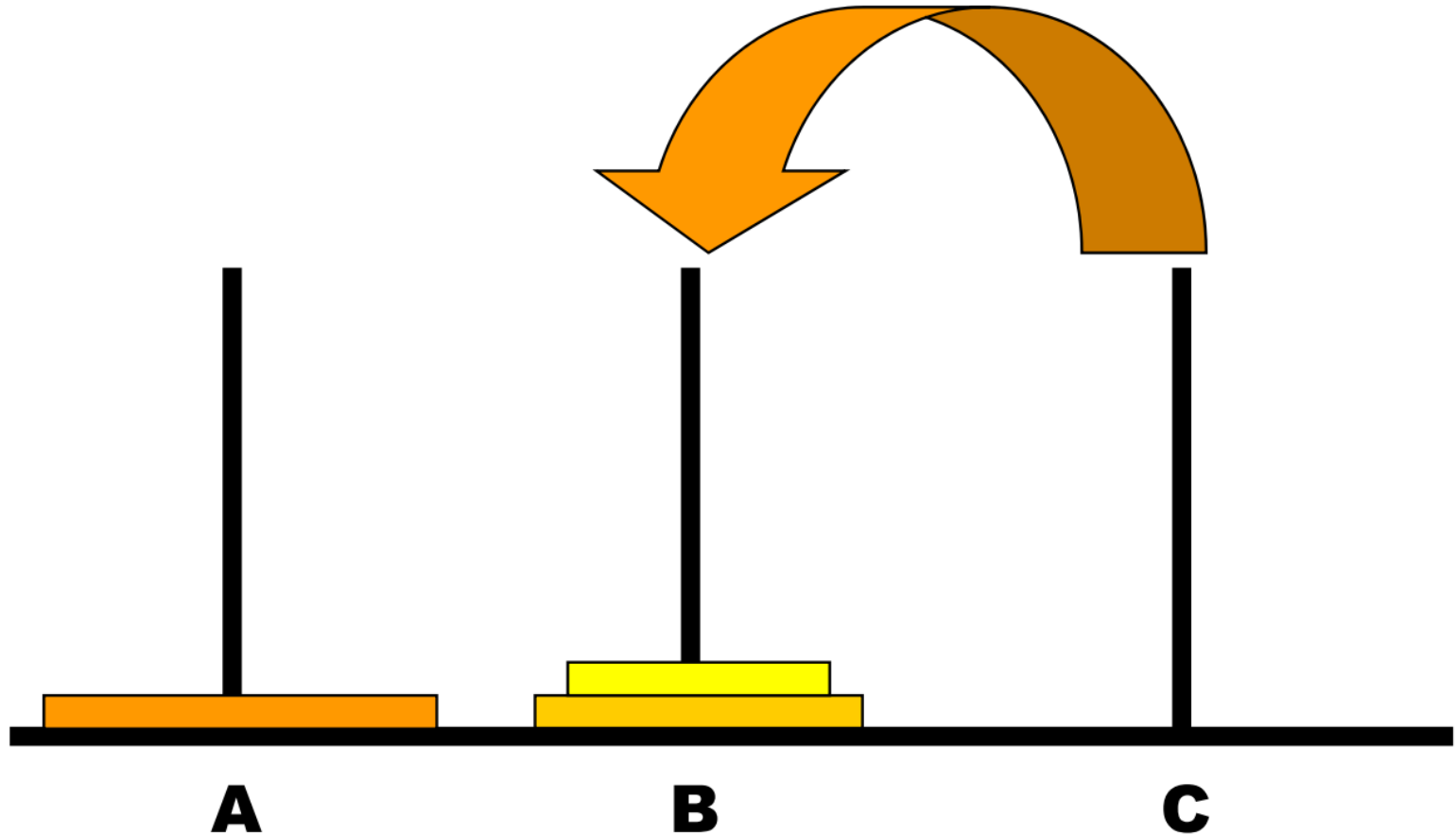
Move 1



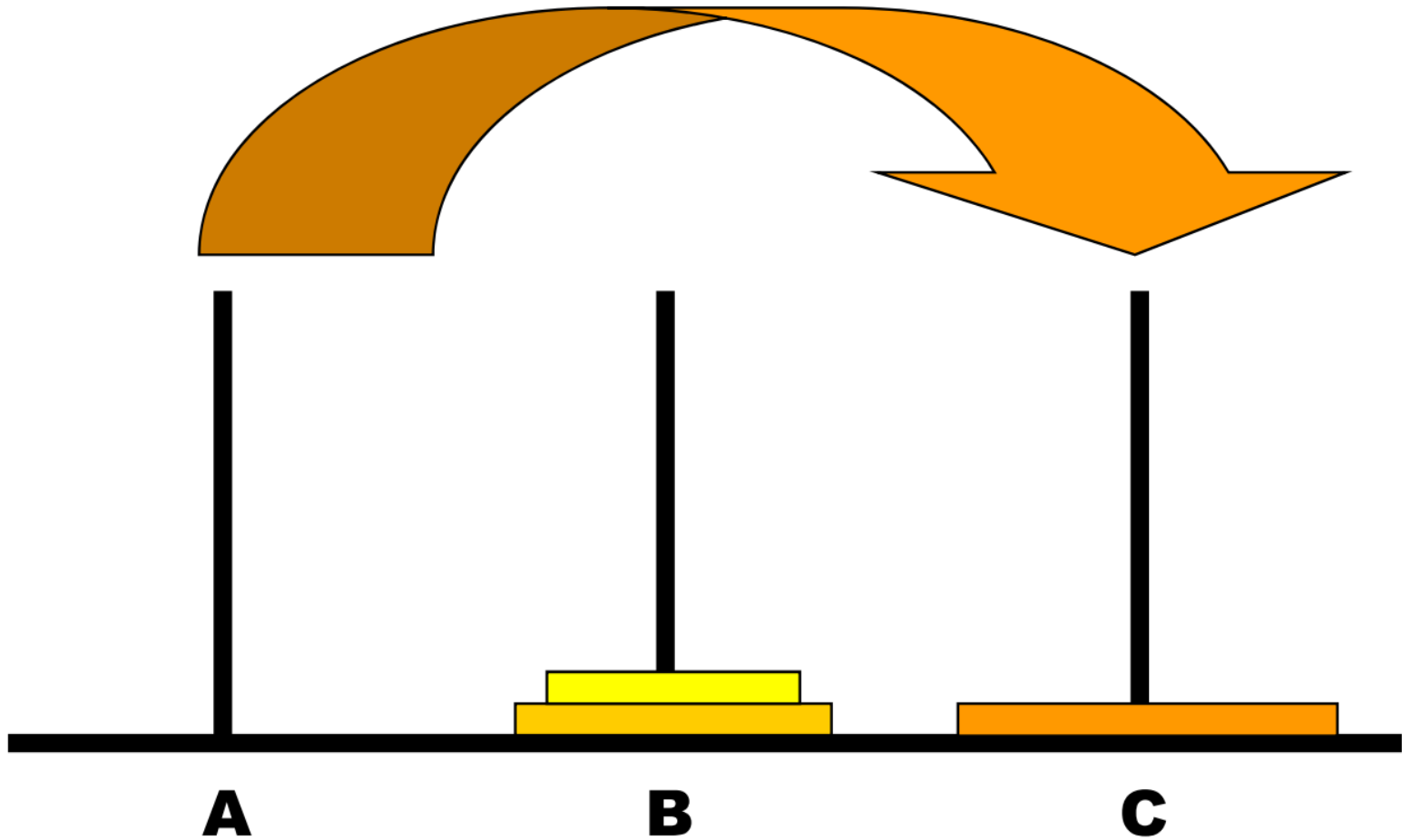
Move 2



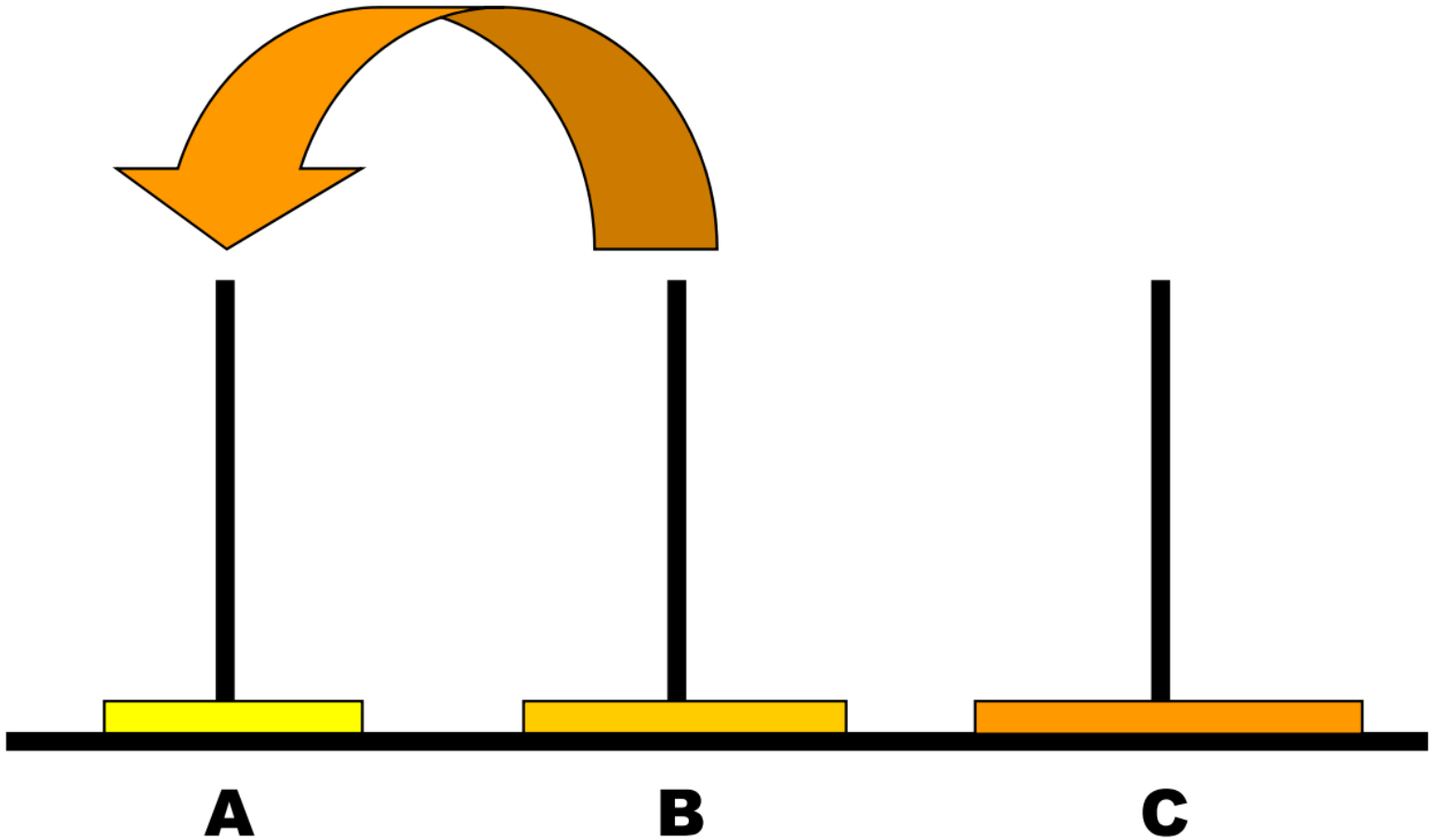
Move 3



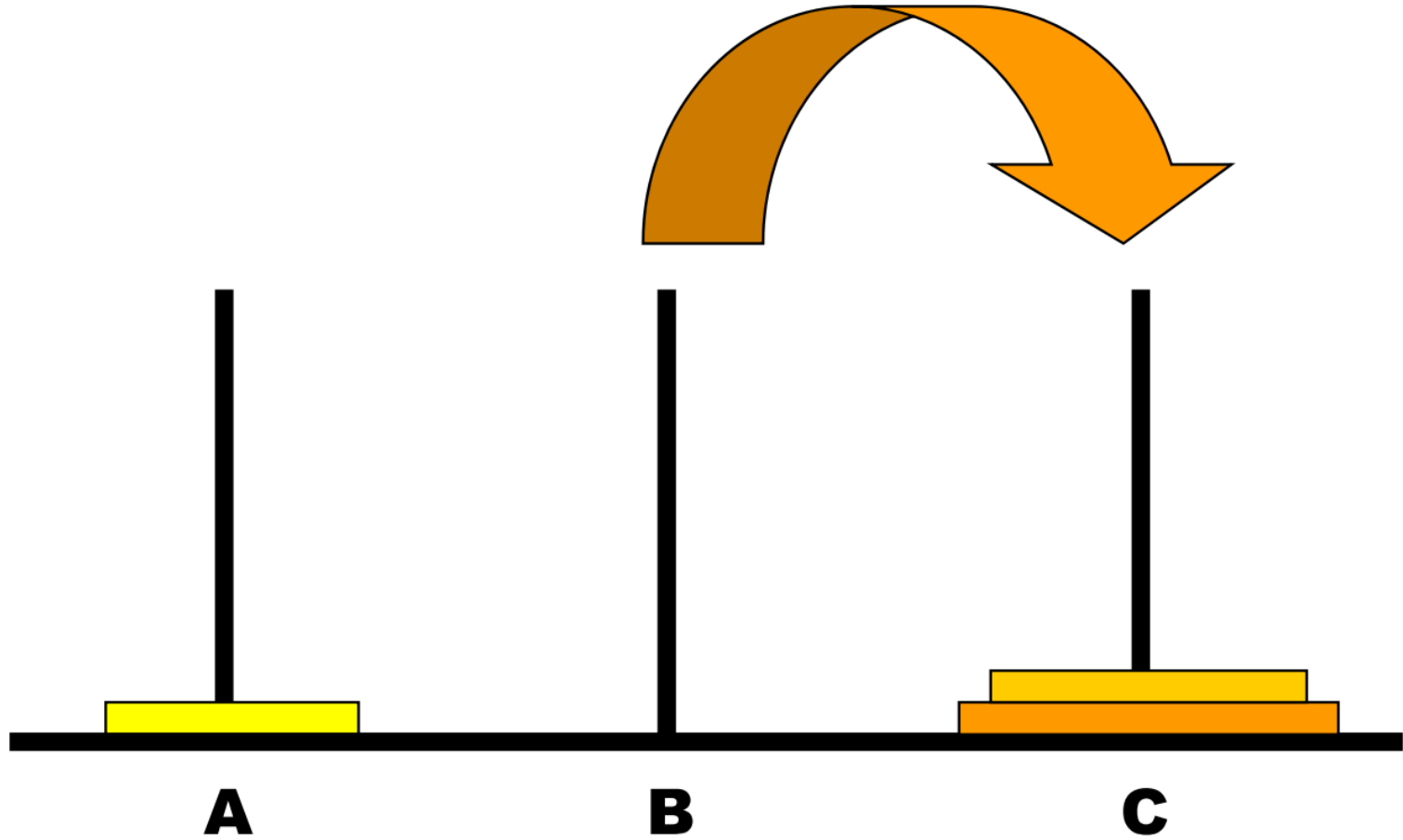
Move 4



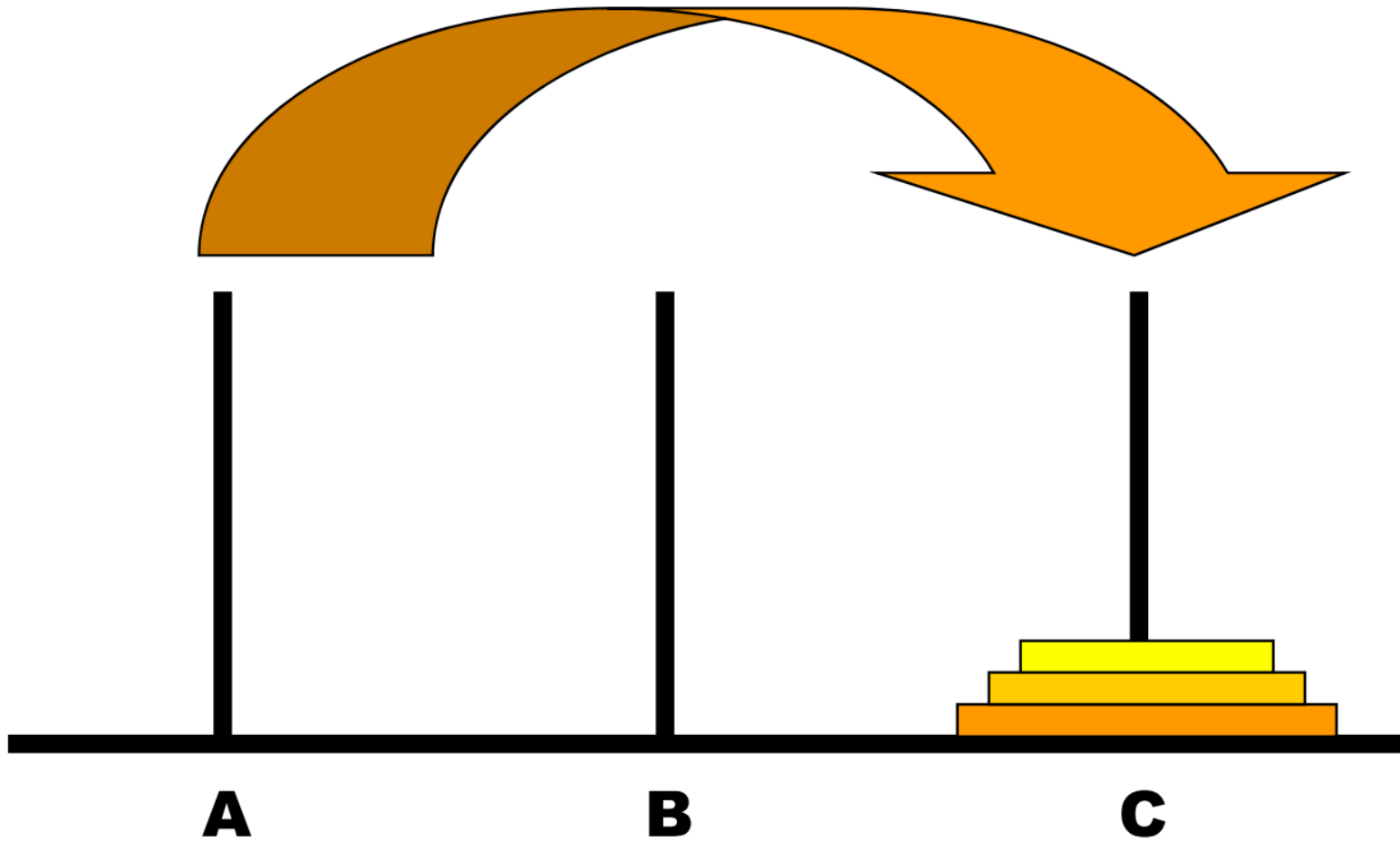
Move 5



Move 6



Move 7



The Algorithm

- **void move(int count, int start, int finish, int temp)**
- Precondition
 - There are at least **count** disks on the tower start
- Postcondition
 - The top **count** disks on start have been moved to tower **finish**;
 - **Tower temp** (used for temporary storage) has been returned to its starting position

The Algorithm

```
void move(int count, int start, int finish, int temp);
int main() {
    int disks = 3;
    printf("Disks: %d\n", disks);
    move(disks, 1, 3, 2);
    return 0;
}

void move(int count, int start, int finish, int temp) {
    if (count == 1) {
        printf("%s %d %s %d %s %d\n", "move disk", count,
            "from tower", start, "to tower", finish);
        return;
    }
    move(count - 1, start, temp, finish);
    printf("%s %d %s %d %s %d\n", "move disk", count,
        "from tower", start, "to tower", finish);
    move(count - 1, temp, finish, start);
}
```

The Algorithm

Output:

Disks: 3

Move disk 1 from tower 1 to tower 3

Move disk 2 from tower 1 to tower 2

Move disk 1 from tower 3 to tower 2

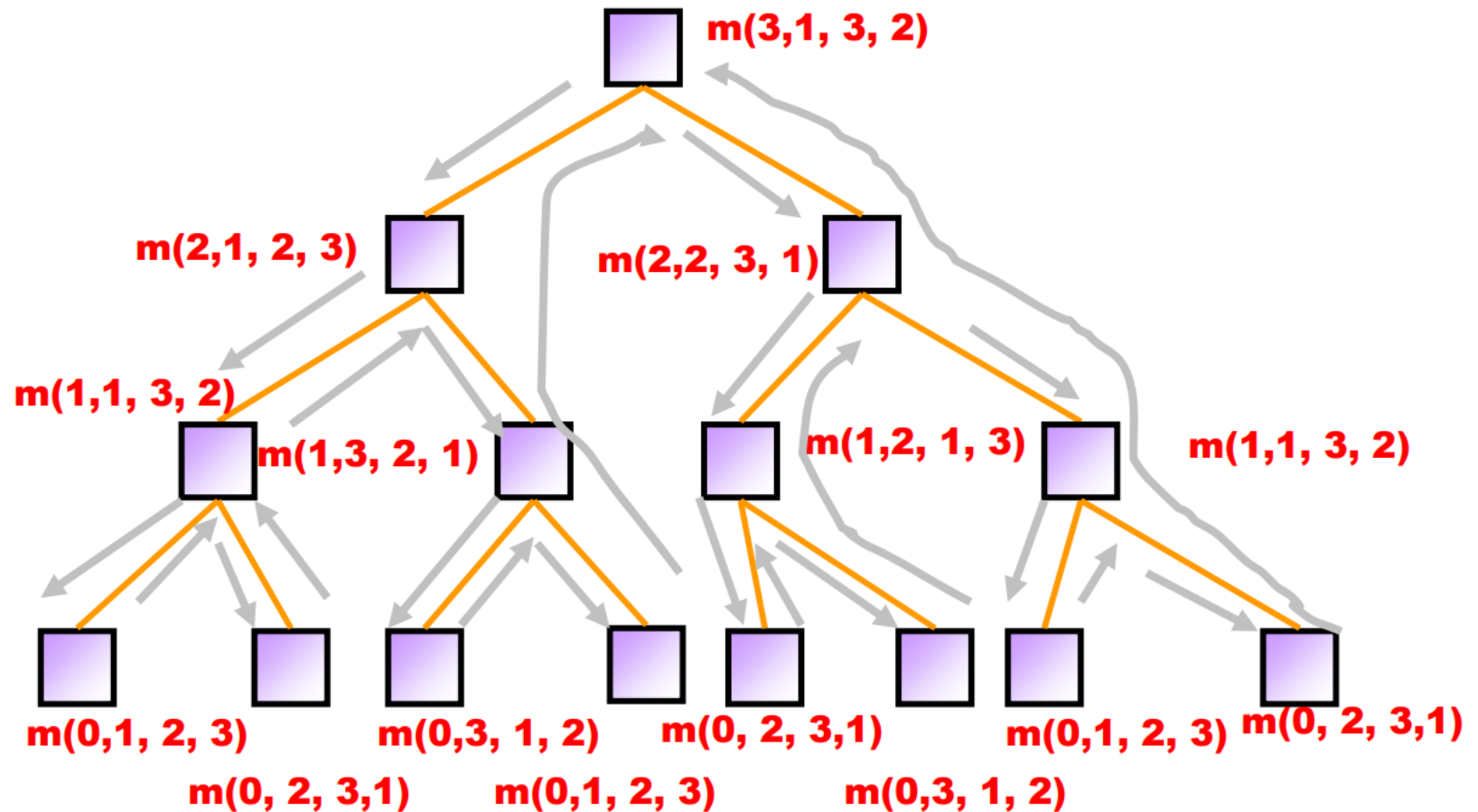
Move disk 3 from tower 1 to tower 3

Move disk 1 from tower 2 to tower 1

Move disk 2 from tower 2 to tower 3

Move disk 1 from tower 1 to tower 3

The Recursion Tree



Designing Recursive Algorithms

- Find the **key step**
 - How can this problem be divided into parts?
 - How will the key step in the middle be done?
- Find a **stopping rule**
 - Small, special case that is trivial or easy to handle without recursion
- Outline your algorithm
 - Combine the stopping rule and the key step, using an **if** statement to select between them
- Check termination
 - Verify that the **recursion always terminates**

Recursion or Iteration ?

- The main advantage of using recursive functions:
When the problem is recursive in nature, a recursive function results in **short, clear** code.
- The main disadvantage of using recursive functions:
recursion is more **expensive** than iteration.
- Any problem that can be solved recursively can also be solved iteratively (by using **loops**).
- A recursive approach is normally chosen in preference to an iterative approach when the recursive approach more **naturally** mirrors the problem and results in a program that is easier to understand and debug.

Summary

- Study several simple examples to see whether recursion should be used and how it will work.
- Attempt to formulate a method that will work more generally
 - How can this problem be divided into parts?
- Ask whether the remainder of the problem can be done in the same or simple way.
- Find a stopping rule.
- Algorithm must terminate.

Recap

- The following concepts have been covered in this chapter:
 - Recursive Methods
 - Recursive Cheers
 - Recursive PrintSomething
 - Recursive Factorials
 - Recursive Multiplication by Addition
 - Recursive PrintDigits
 - Recursive SumArray
 - Tower of Hanoi