

# **CS100**

# **Introduction to Programming**

## **Lecture 10. Recursion**

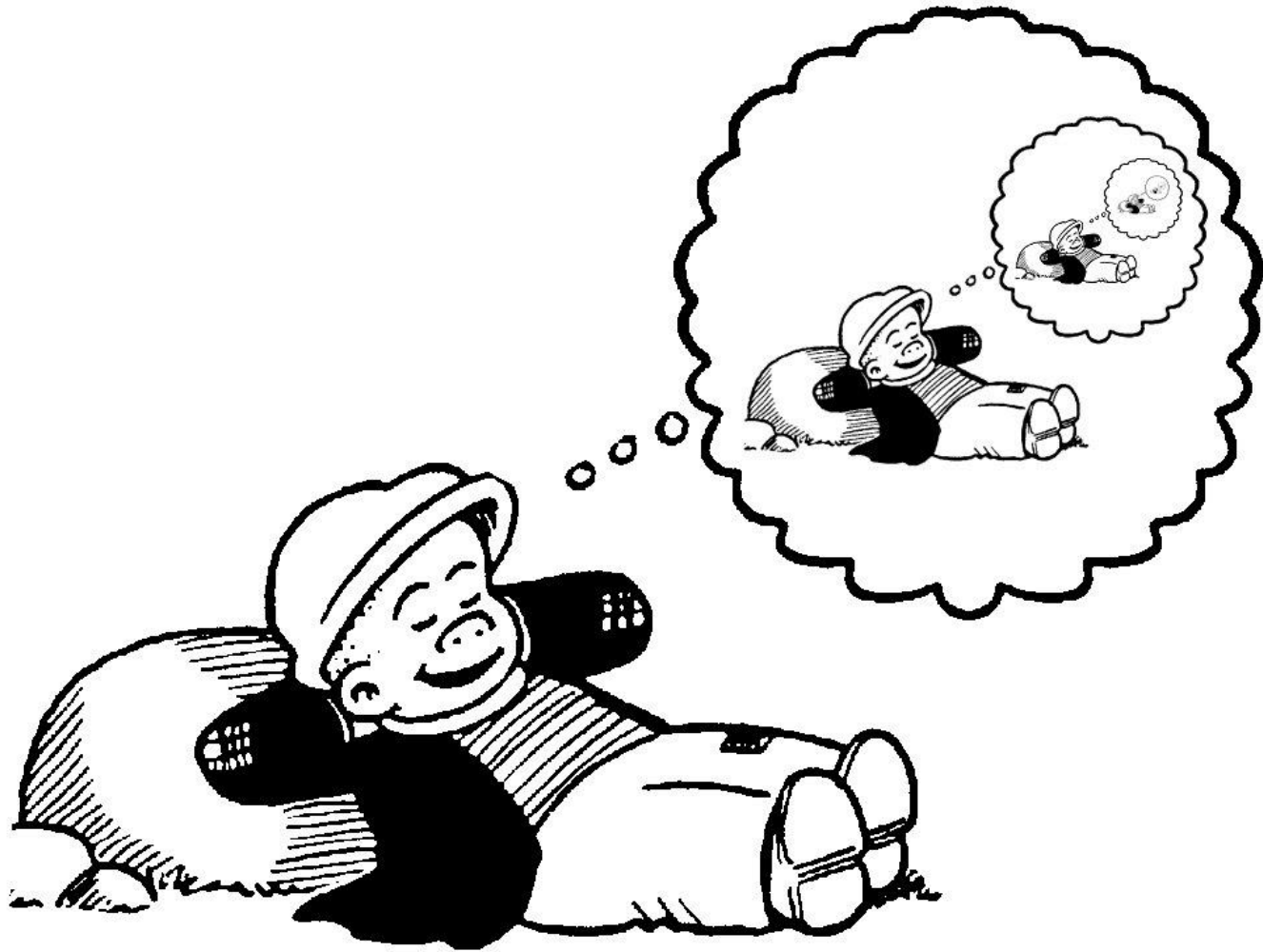
# Learning Objectives

- 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

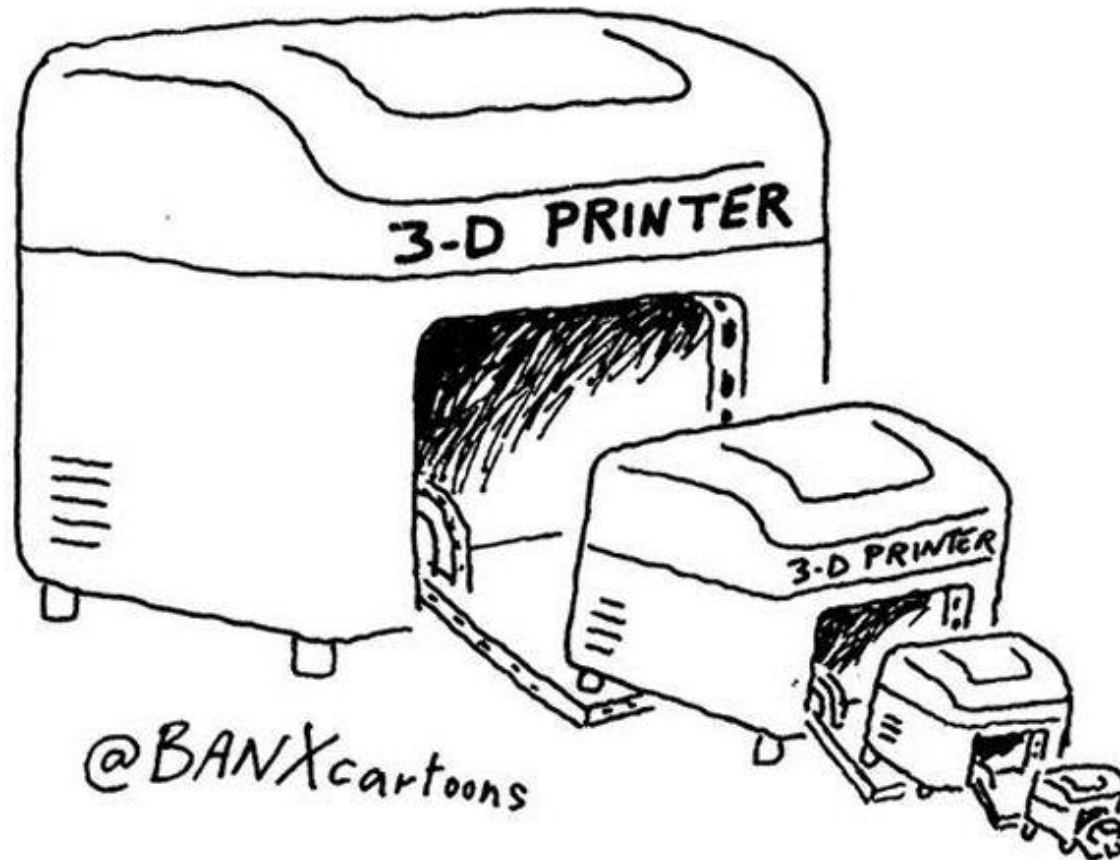
# What Is Recursion?

- 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)`?

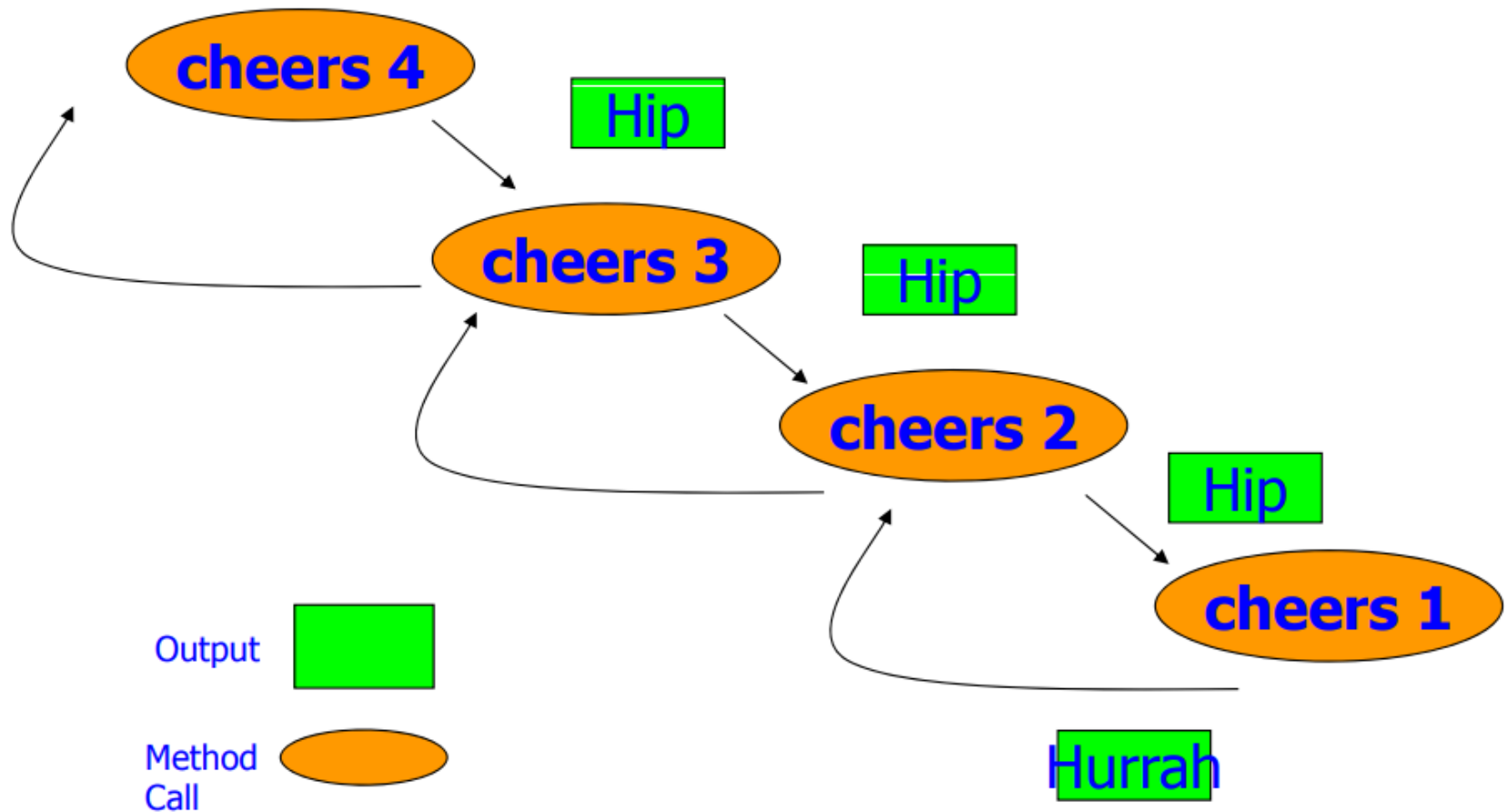
```
void cheers(int n)
{
    if (n <= 1)
        printf("Hurrah \n");
    else {
        printf("Hip \n");
        cheers(n-1);
    }
}
```

**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)`?

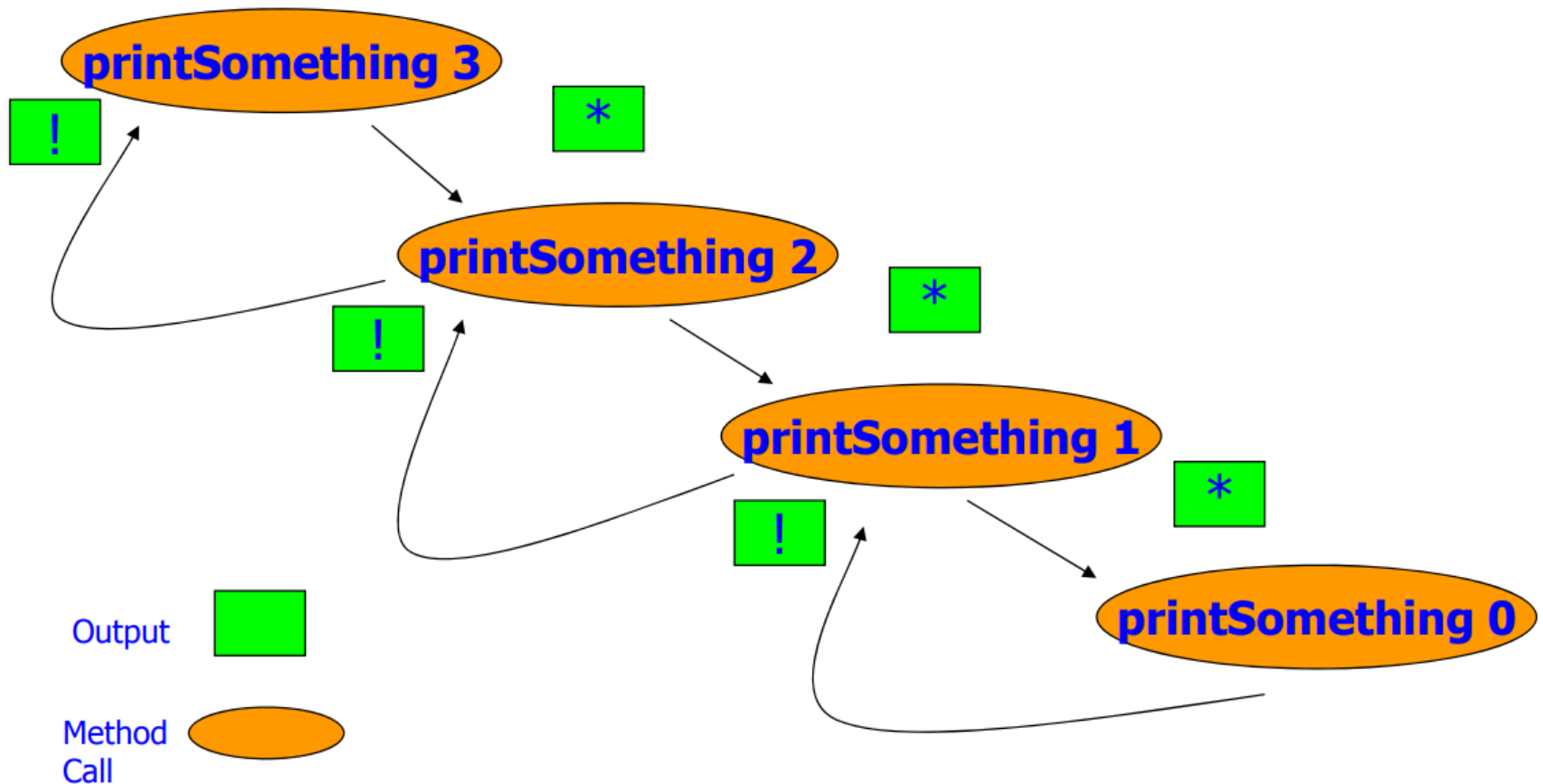
```
void printSomething(int n)
{
    if (n > 0) {
        printf("*");
        printSomething(n-1);
        printf("!");
    }
}
```

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?

$$4! = 4 \times 3!$$

$$= 4 \times (3 \times 2!)$$

$$= 4 \times (3 \times (2 \times 1!))$$

$$= 4 \times (3 \times (2 \times (1 \times 0!)))$$

$$= 4 \times (3 \times (2 \times (1 \times 1)))$$

$$= 4 \times (3 \times (2 \times 1))$$

$$= 4 \times (3 \times 2)$$

$$= 4 \times 6$$

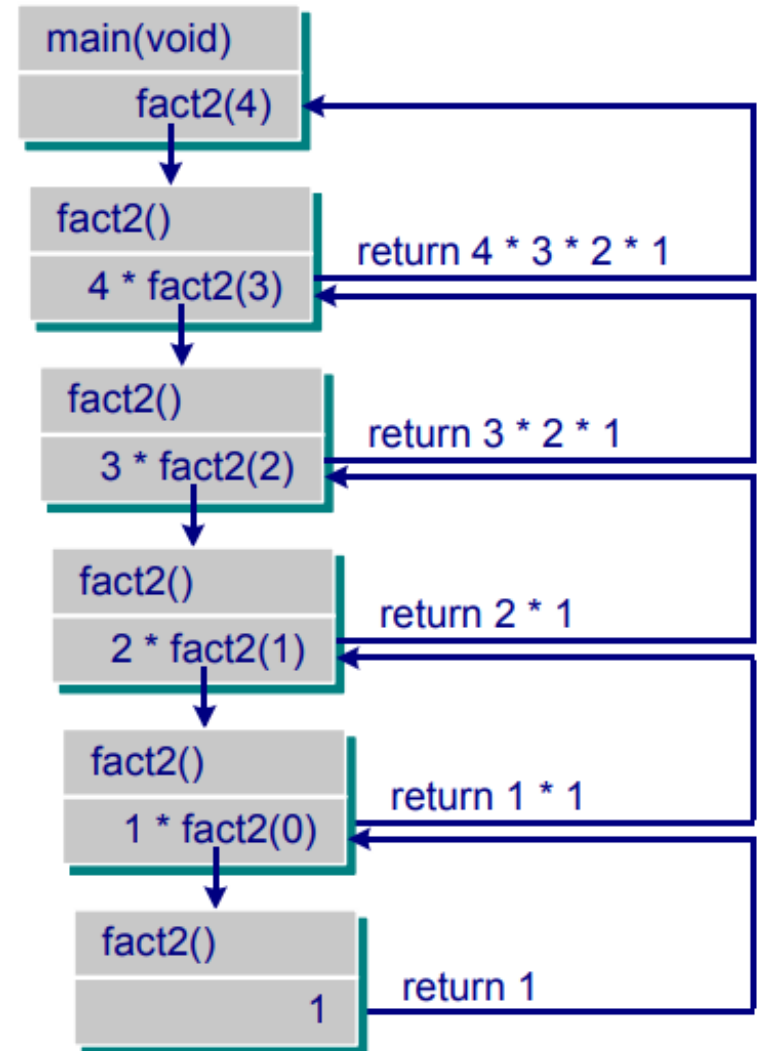
$$= 24$$

# Recursive Factorials

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

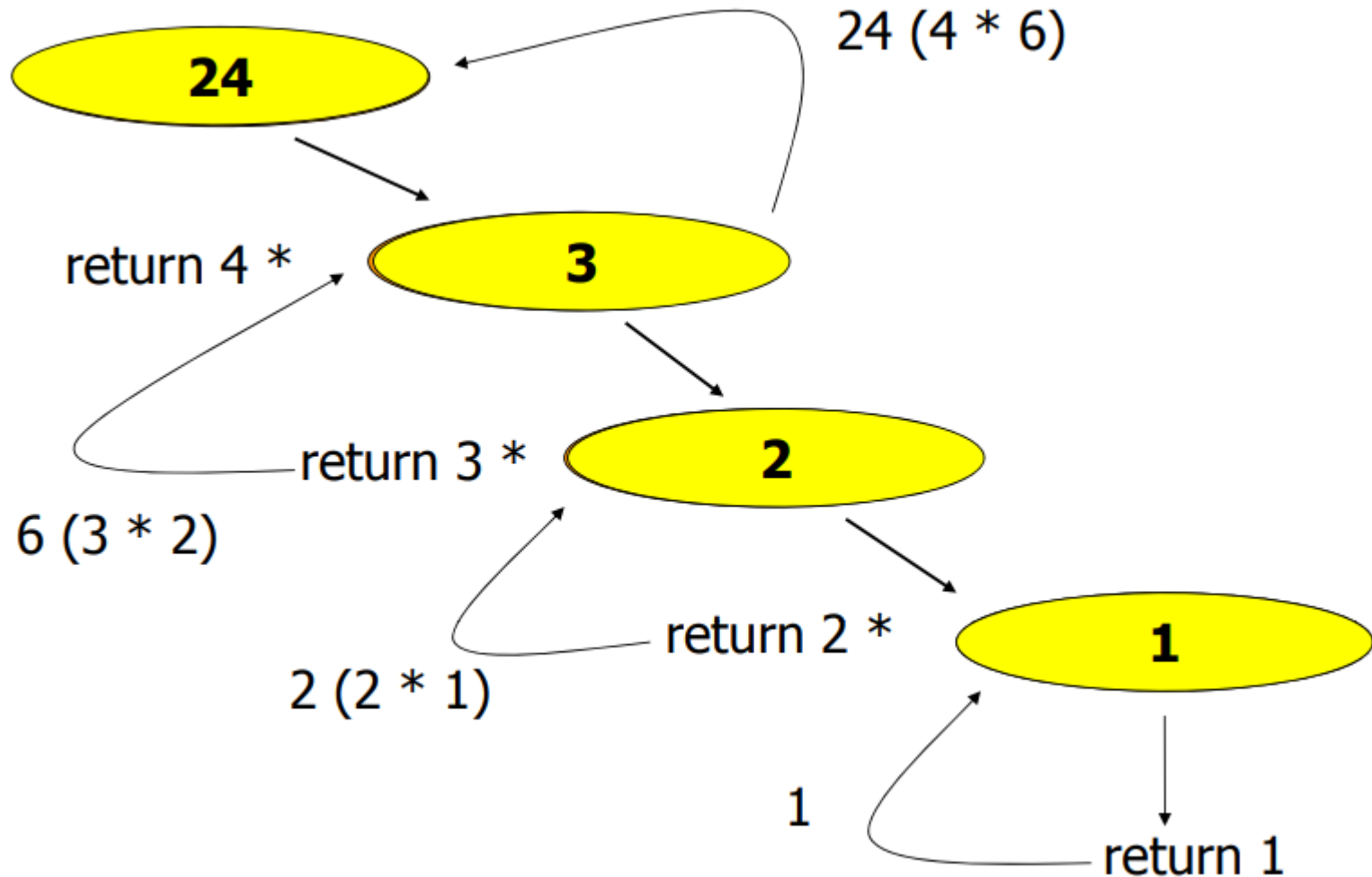
**Output:**

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

Reduce to a smaller problem

**output 5**

1

PrintDigit 23

2

**output 4**

7

PrintDigit 2

3

**output 3**

6

**output 2**

4

5

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

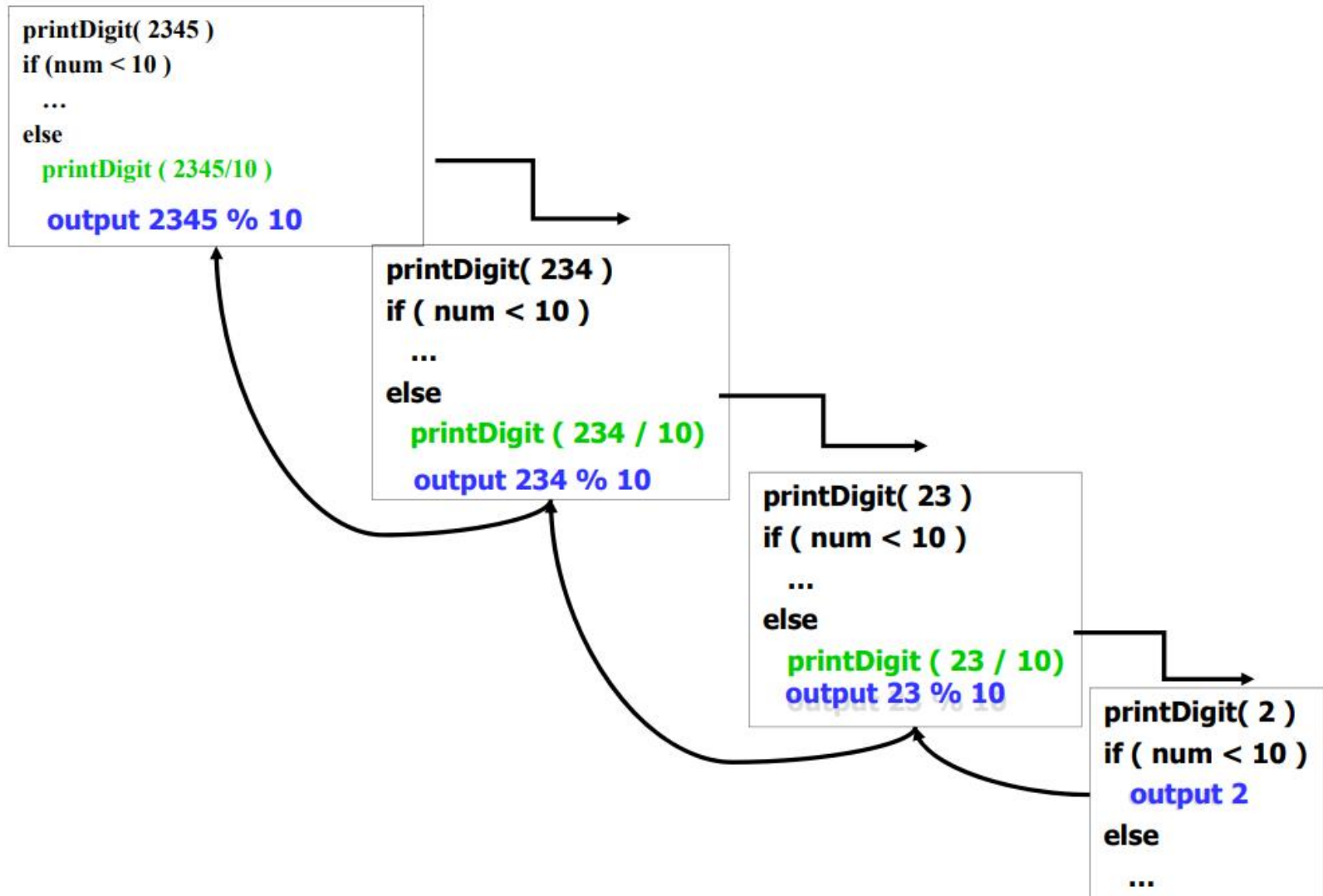
3

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]

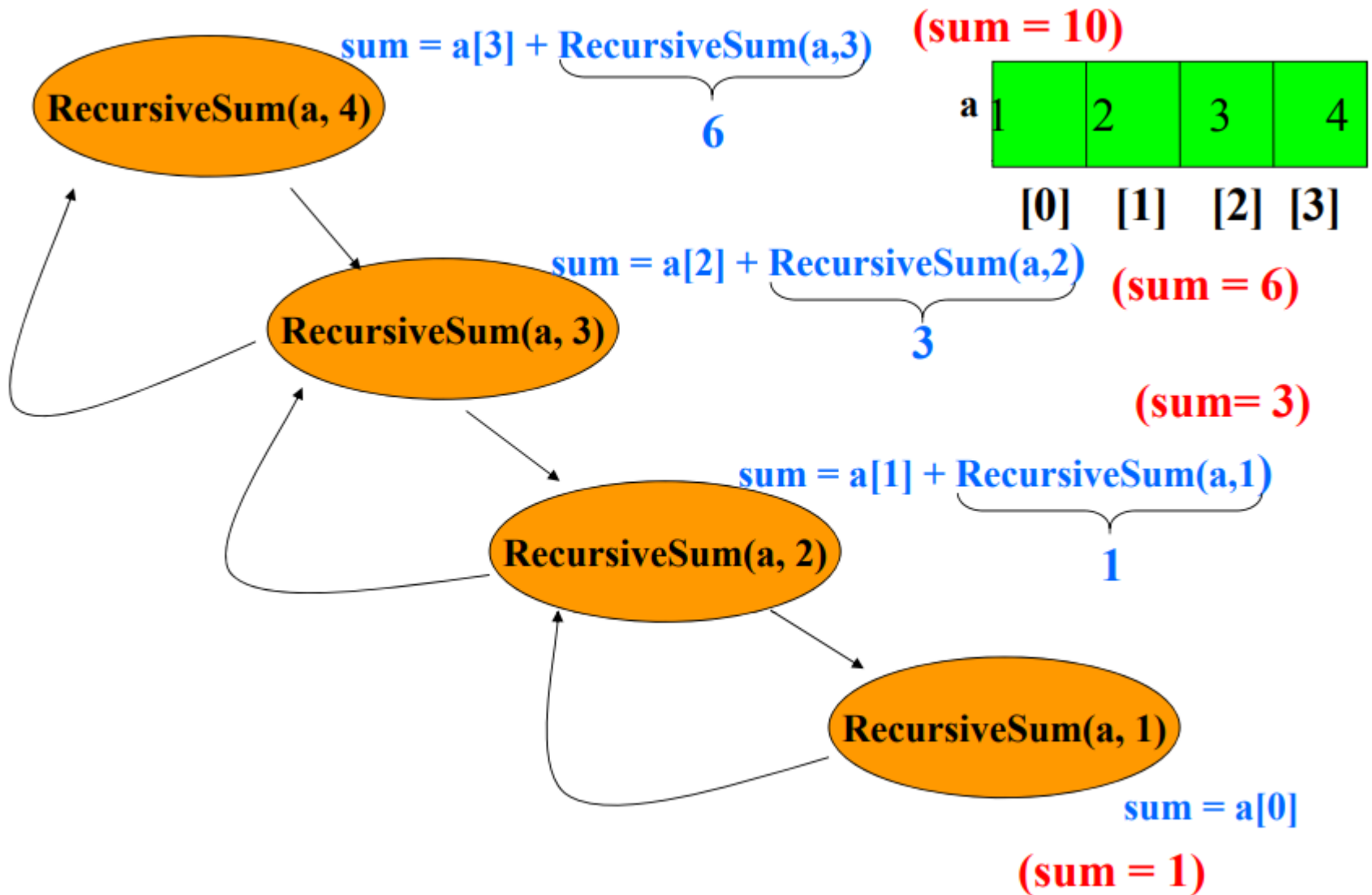
**Input:**

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

**Output:**

Sum = 10

# Recursive Sum – Tracing

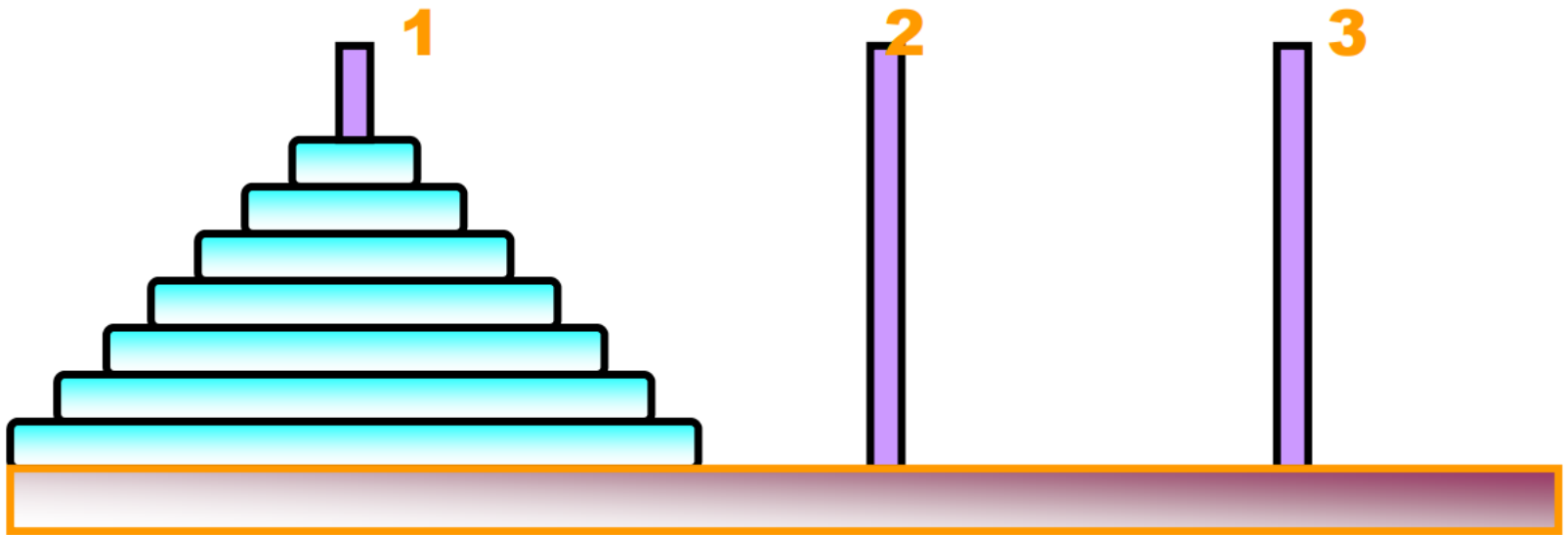


# Example 7: Tower of Hanoi

- In the 19<sup>th</sup> 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.

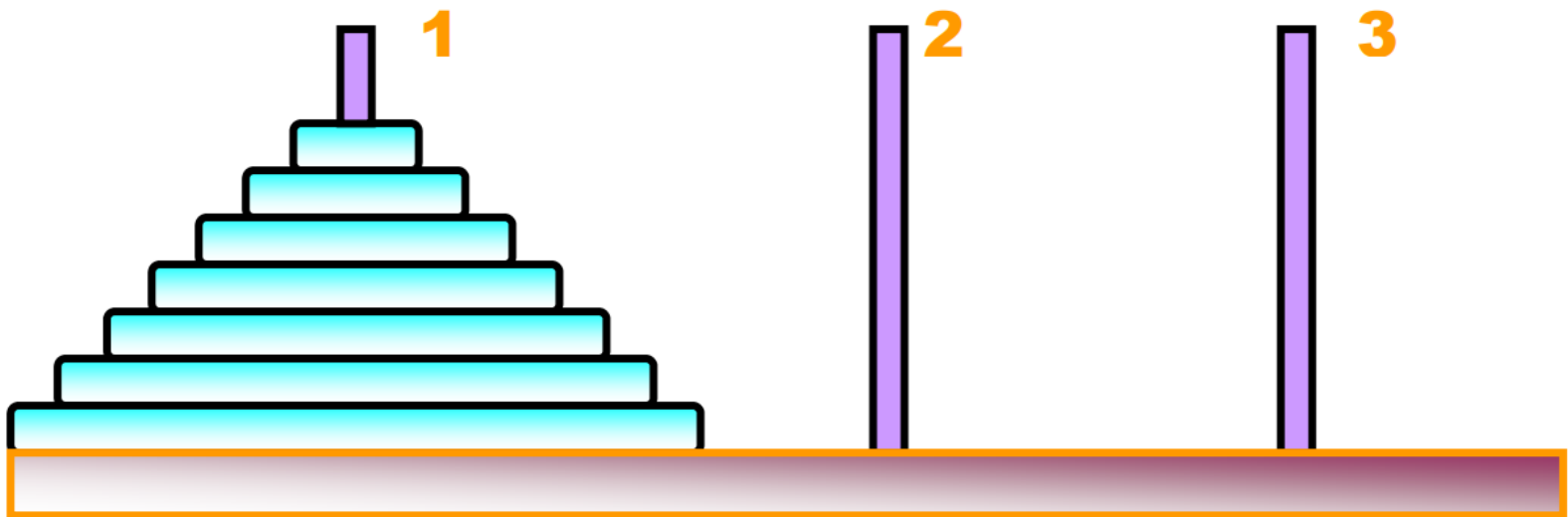
# Tower of Hanoi

- **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

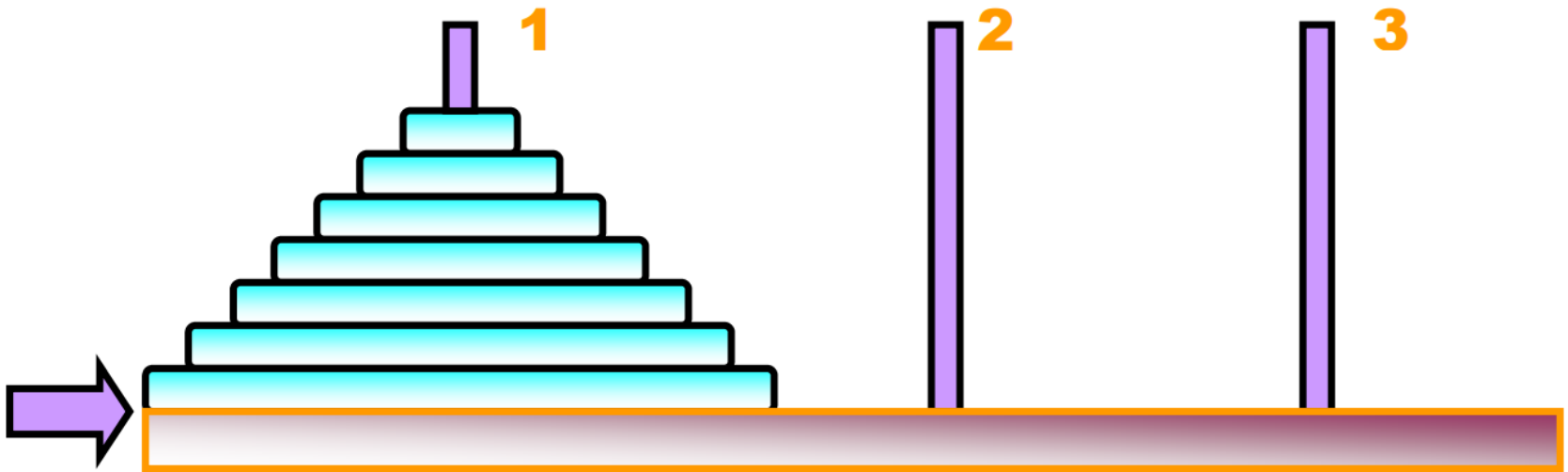
- Write a computer program that will type out a list of instructions for the priests
- **move(64, 1, 3, 2)**
  - Move 64 disks from tower 1 to tower 3 using tower 2 as temporary storage





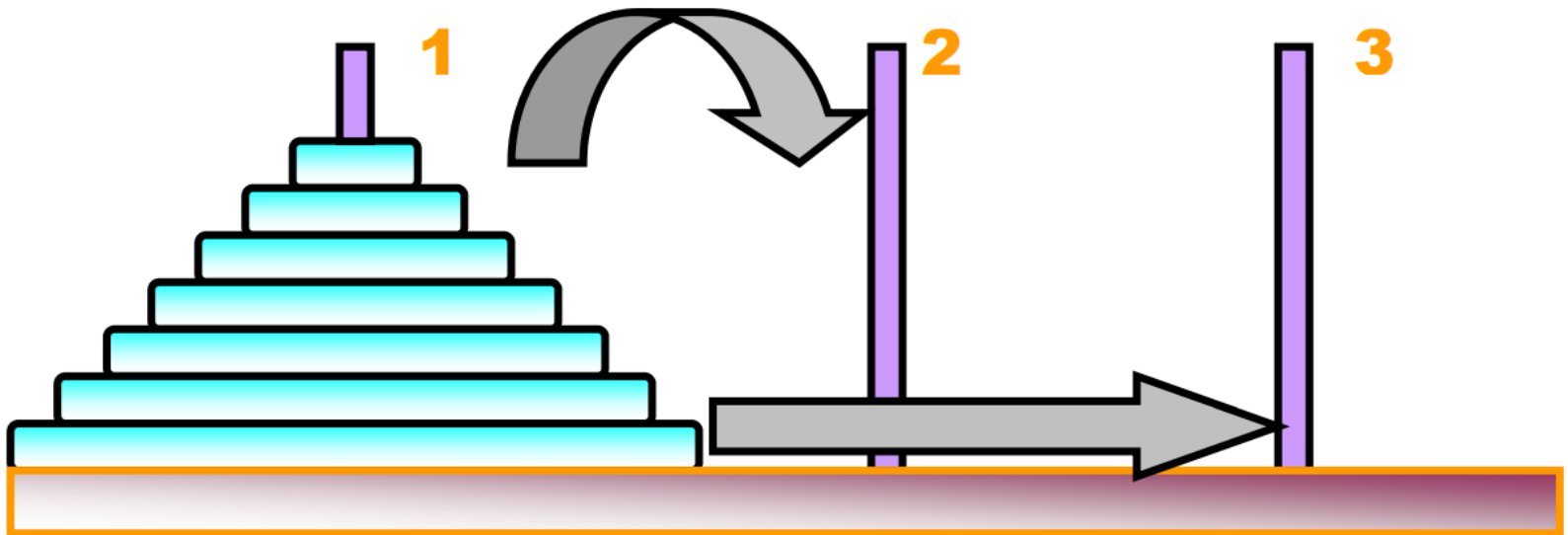
# The Solution

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



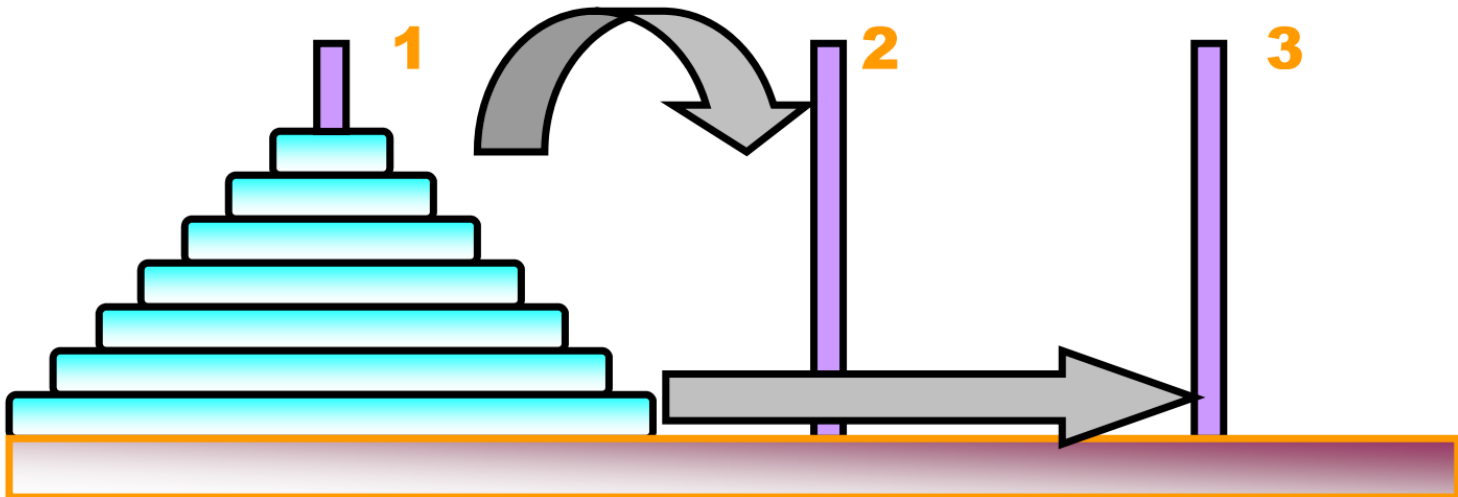
# The Solution

- 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

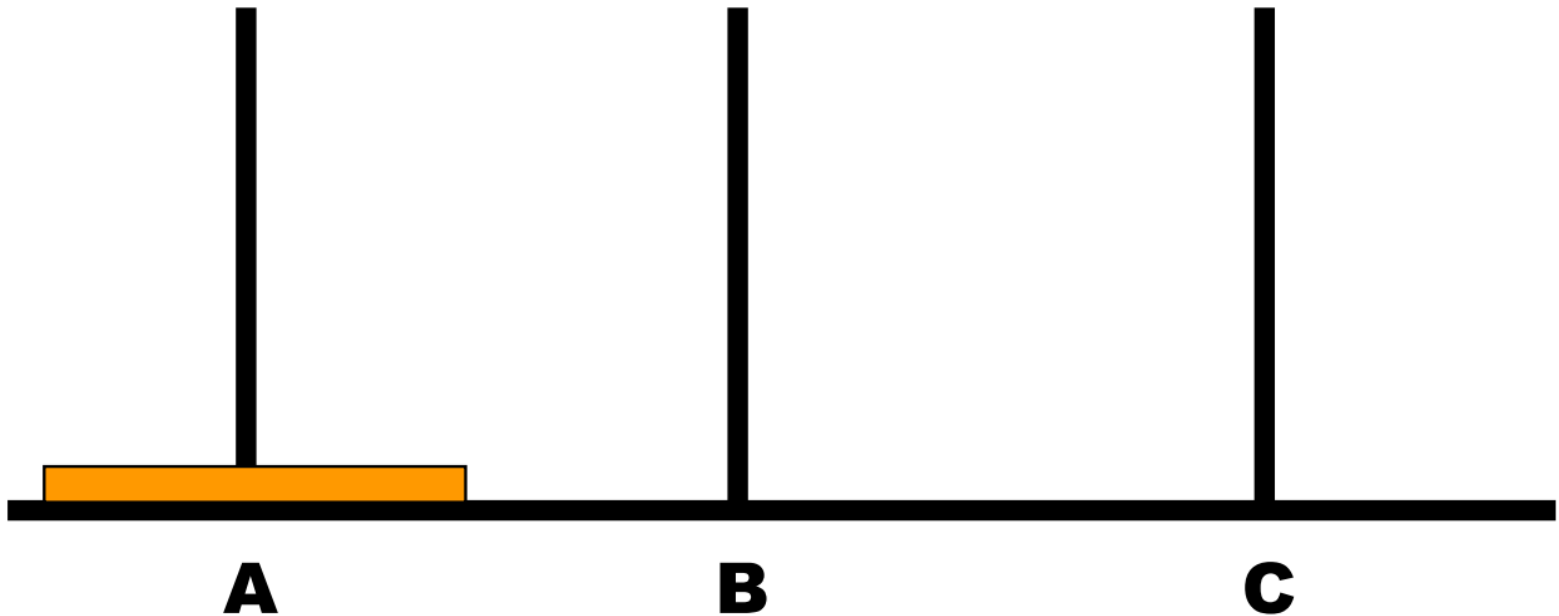
- `move(63, 1, 2, 3);` // Move 63 disks from tower 1 to tower 2 (using tower 3 as temporary)
- Display “Move disk 64 from tower 1 to 3”
- `move (63, 2, 3, 1);` // Move 63 disks from tower 2 to tower 3 (using tower 1 as temporary)



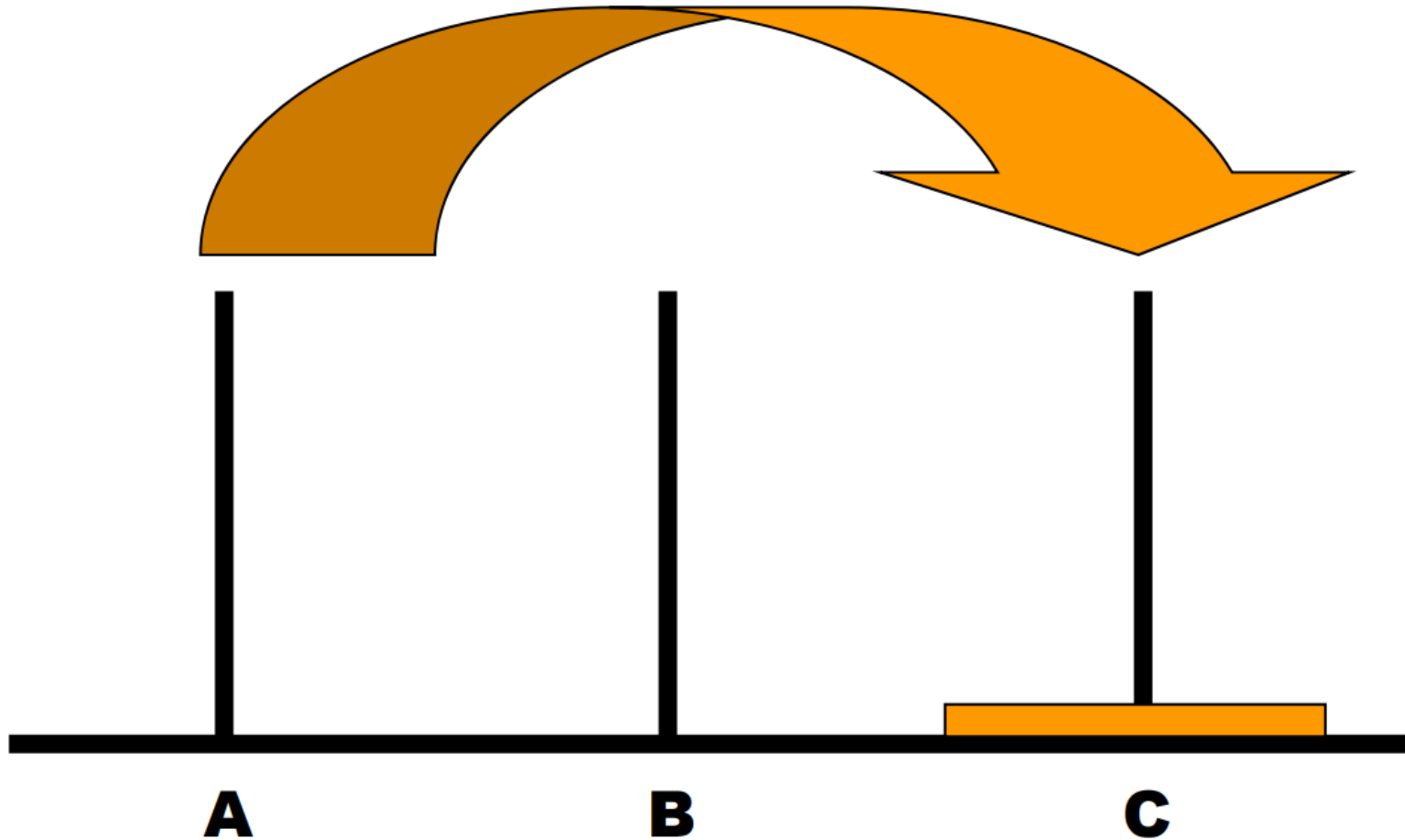
# The Solution

- 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.

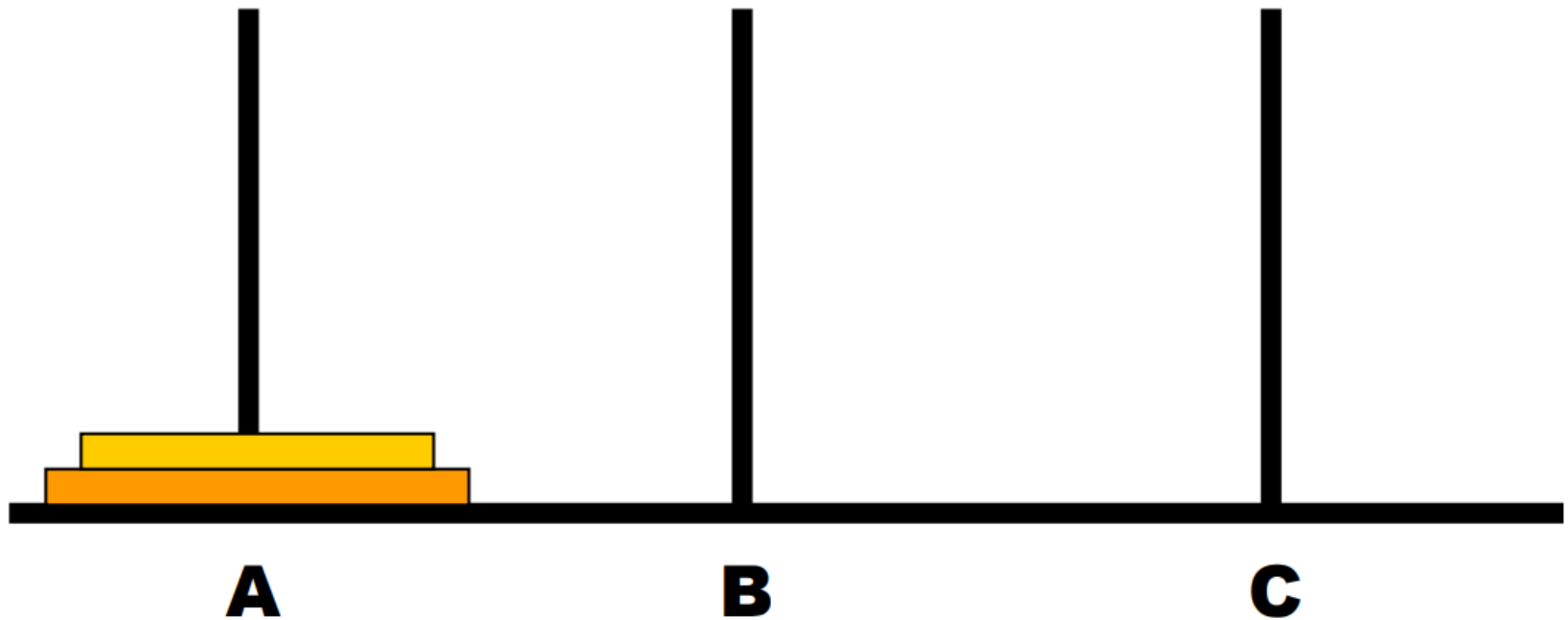
# A Single Disk Tower



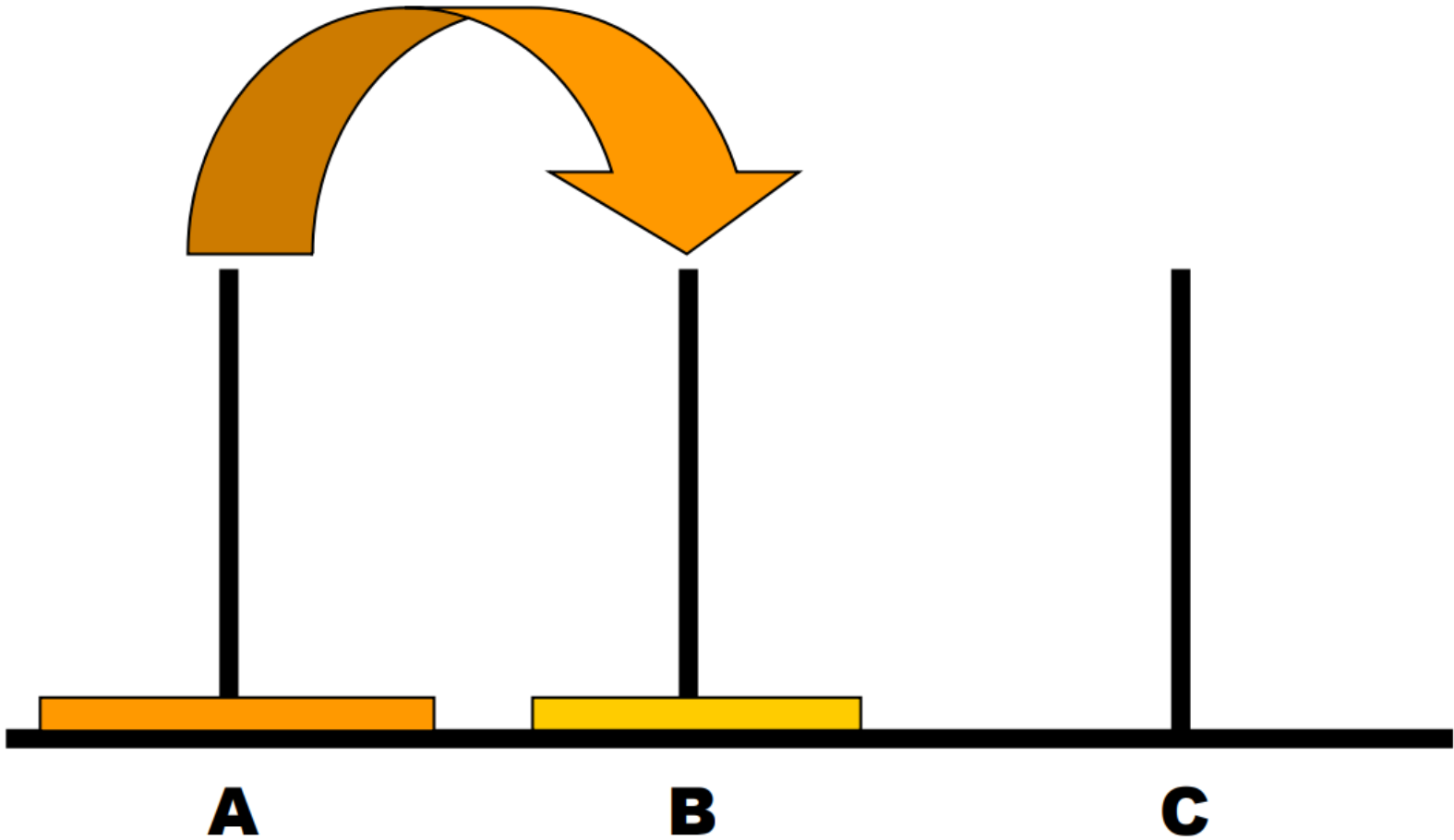
# A Single Disk Tower



# A 2-Disk Tower

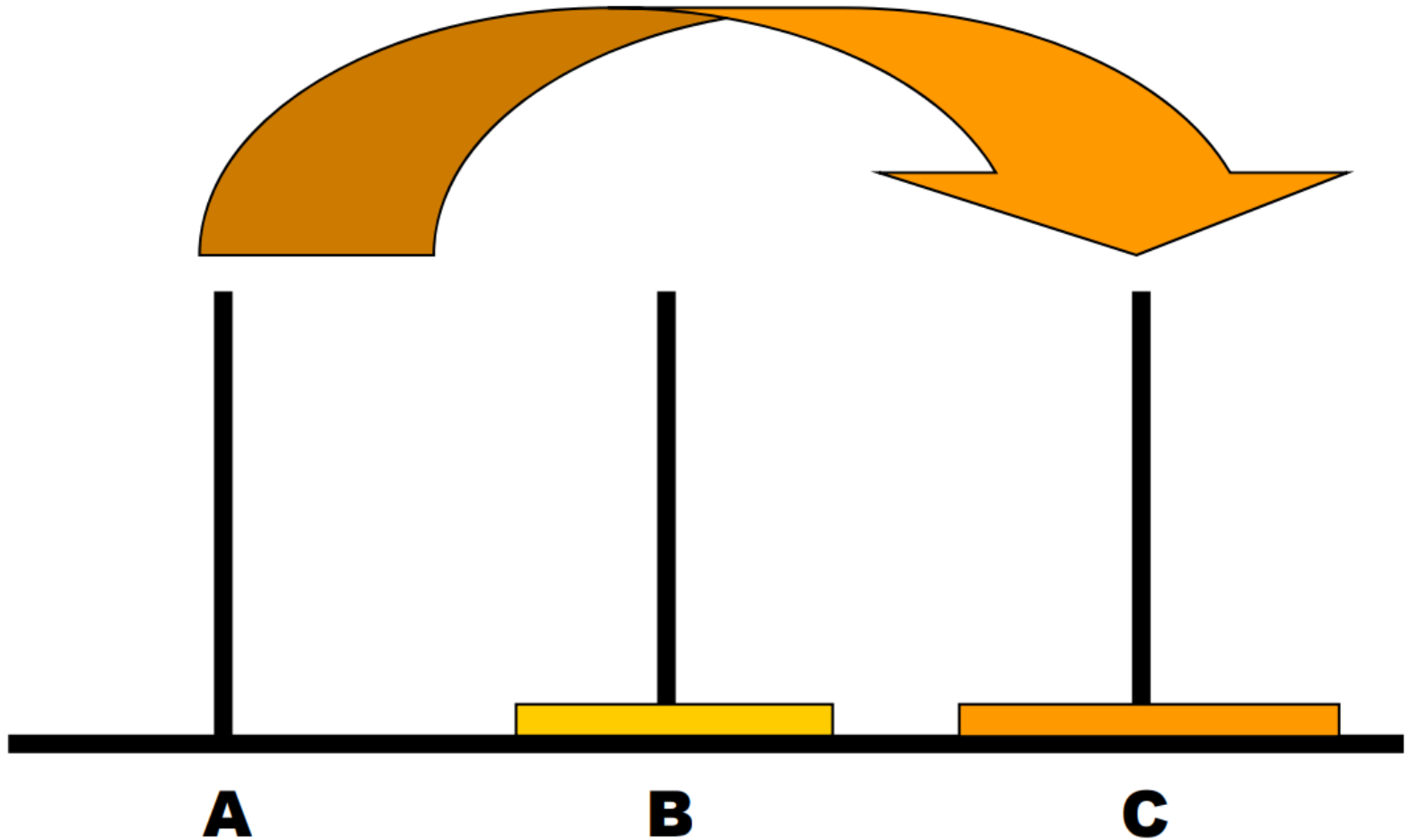


# Move 1

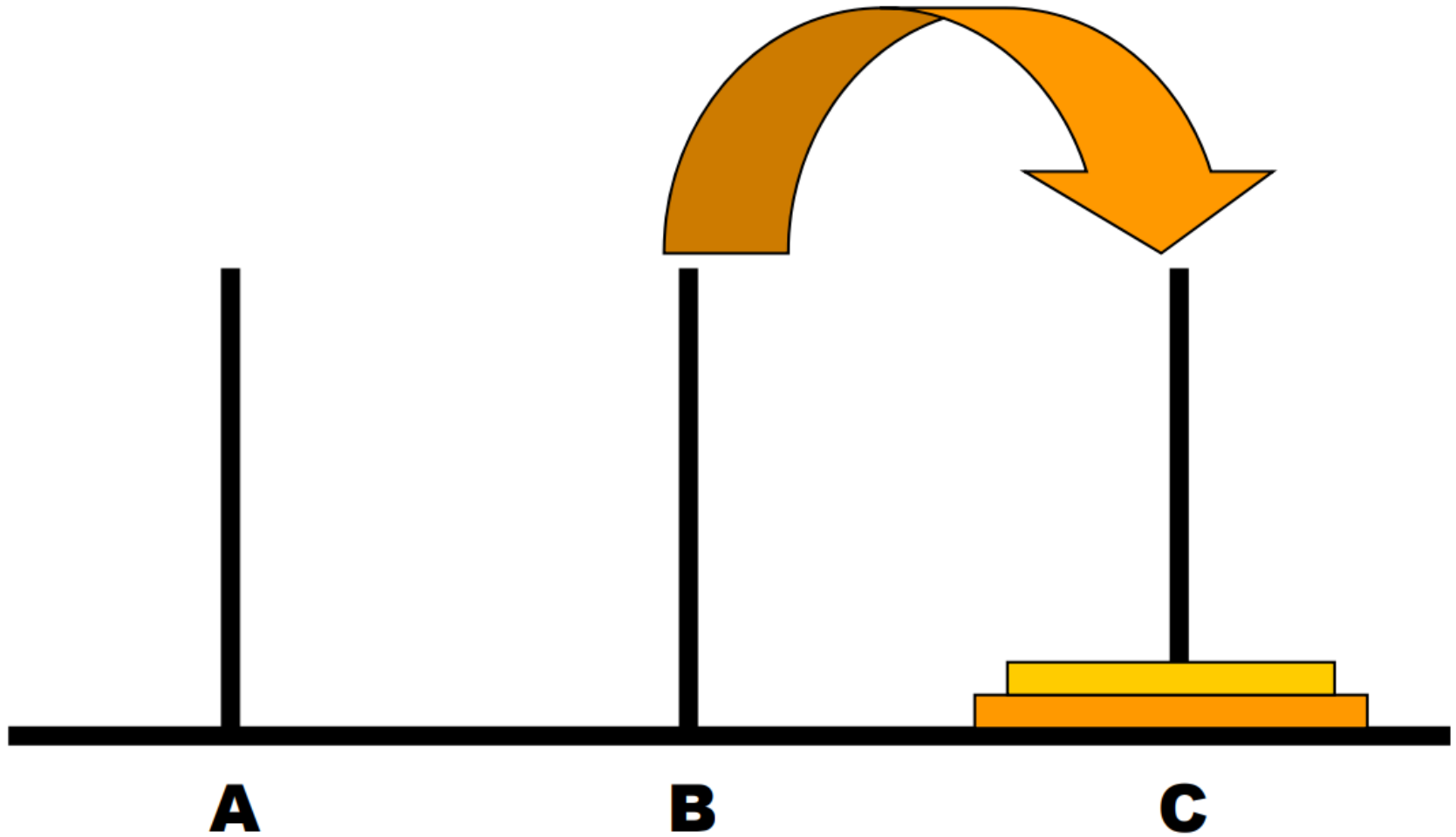




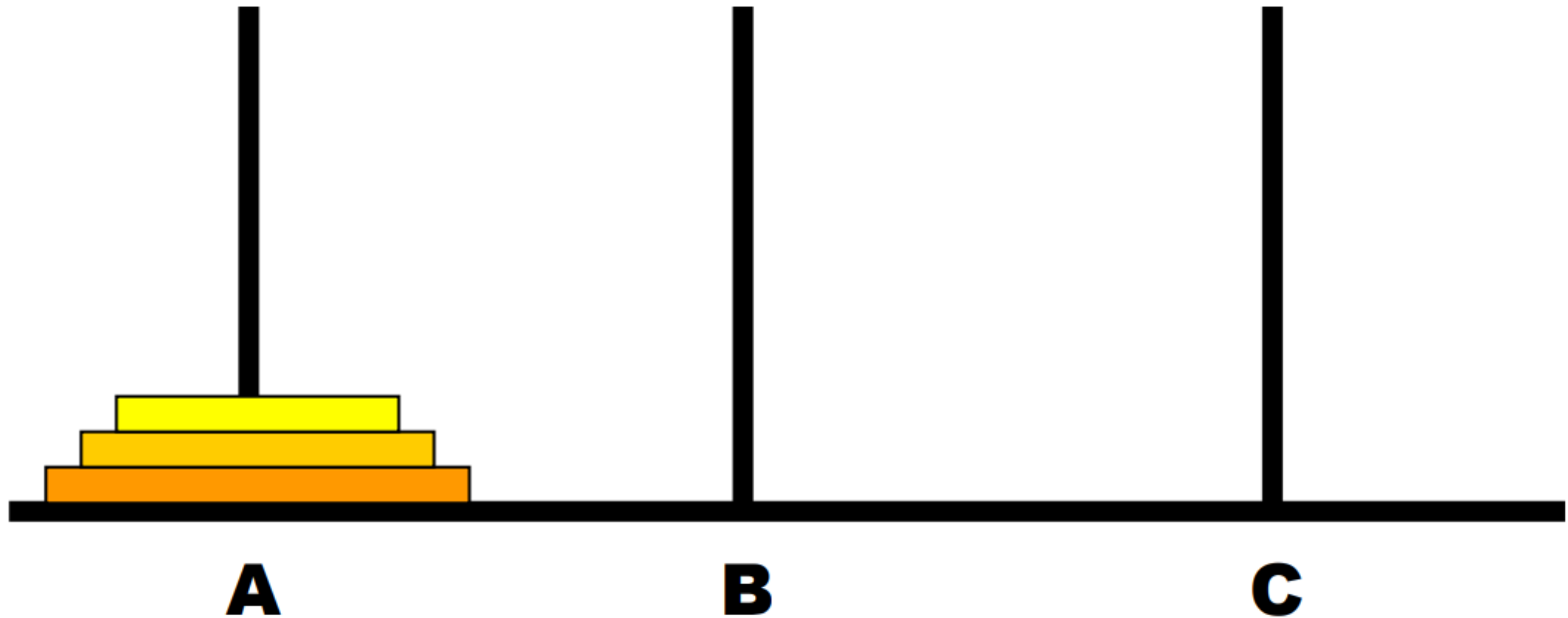
# Move 2



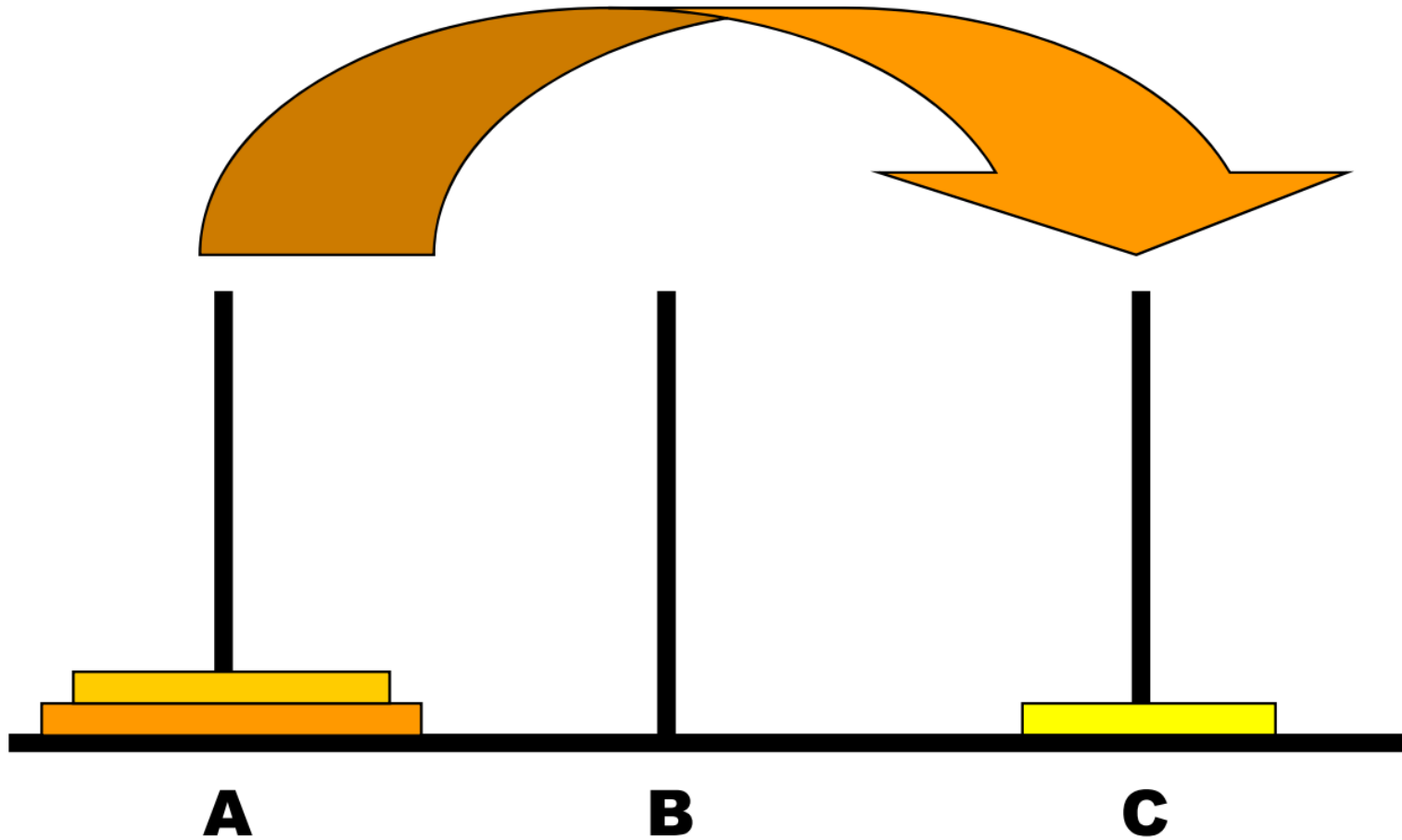
# Move 3



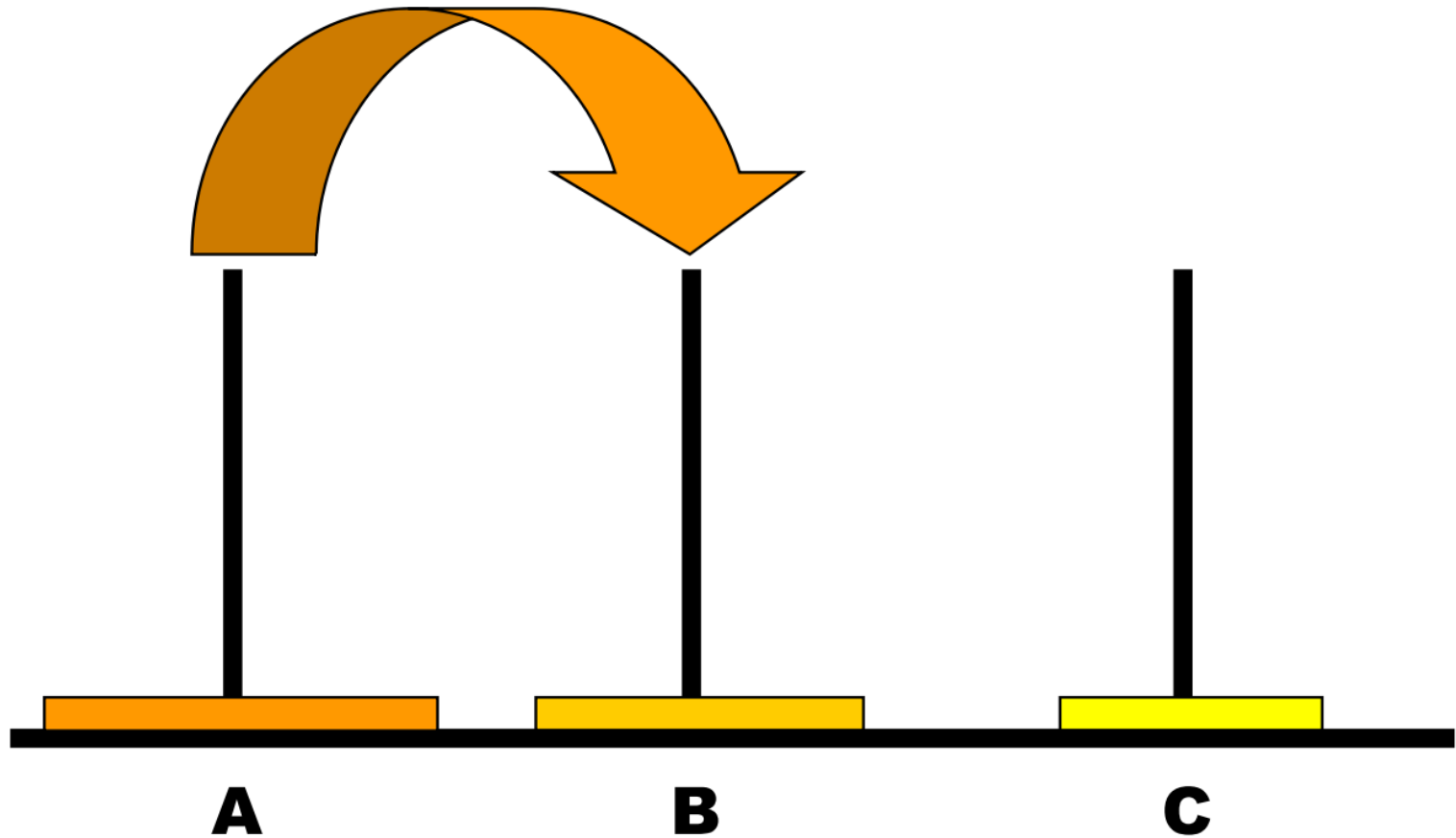
# A 3-Disk Tower



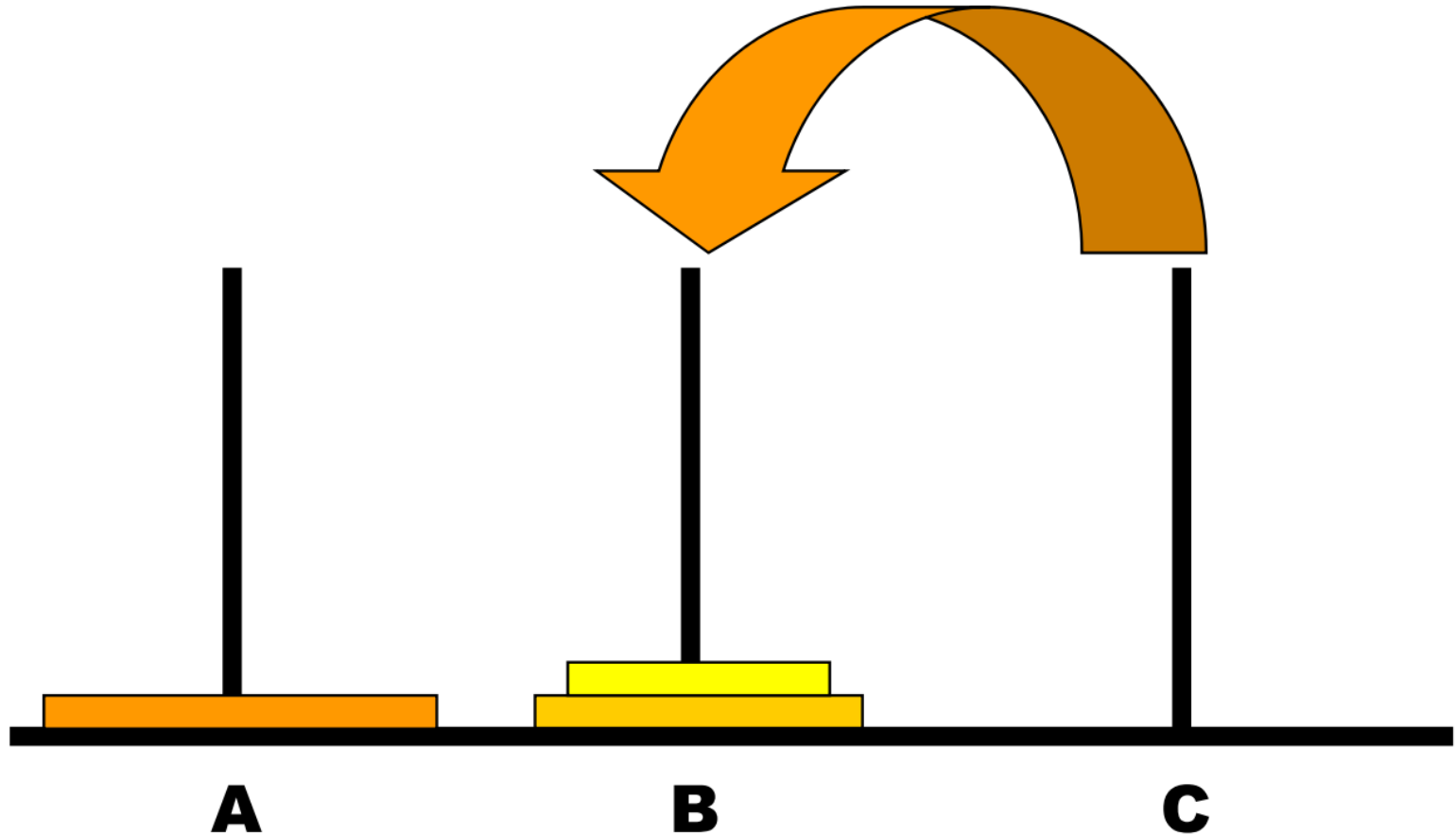
# Move 1



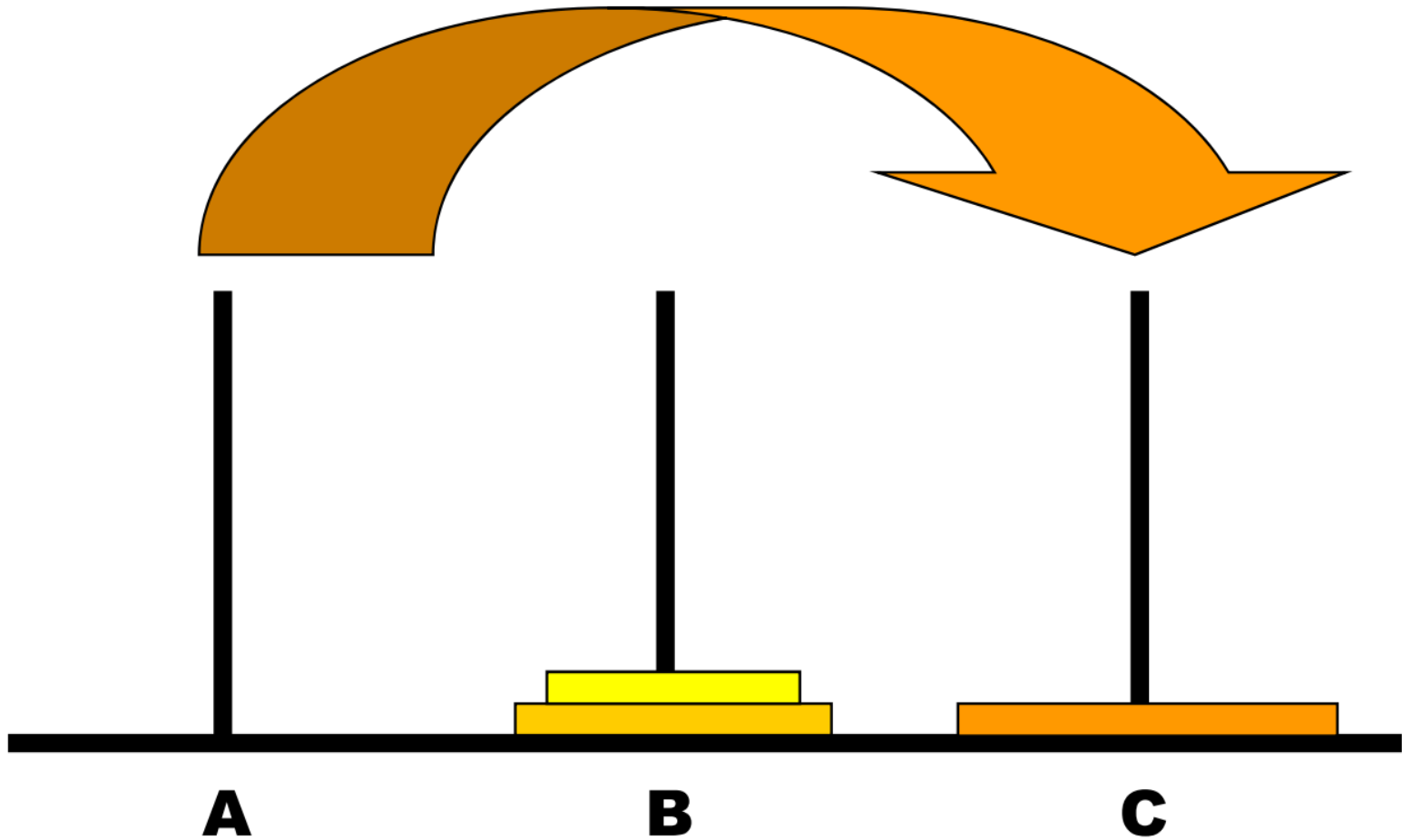
# Move 2



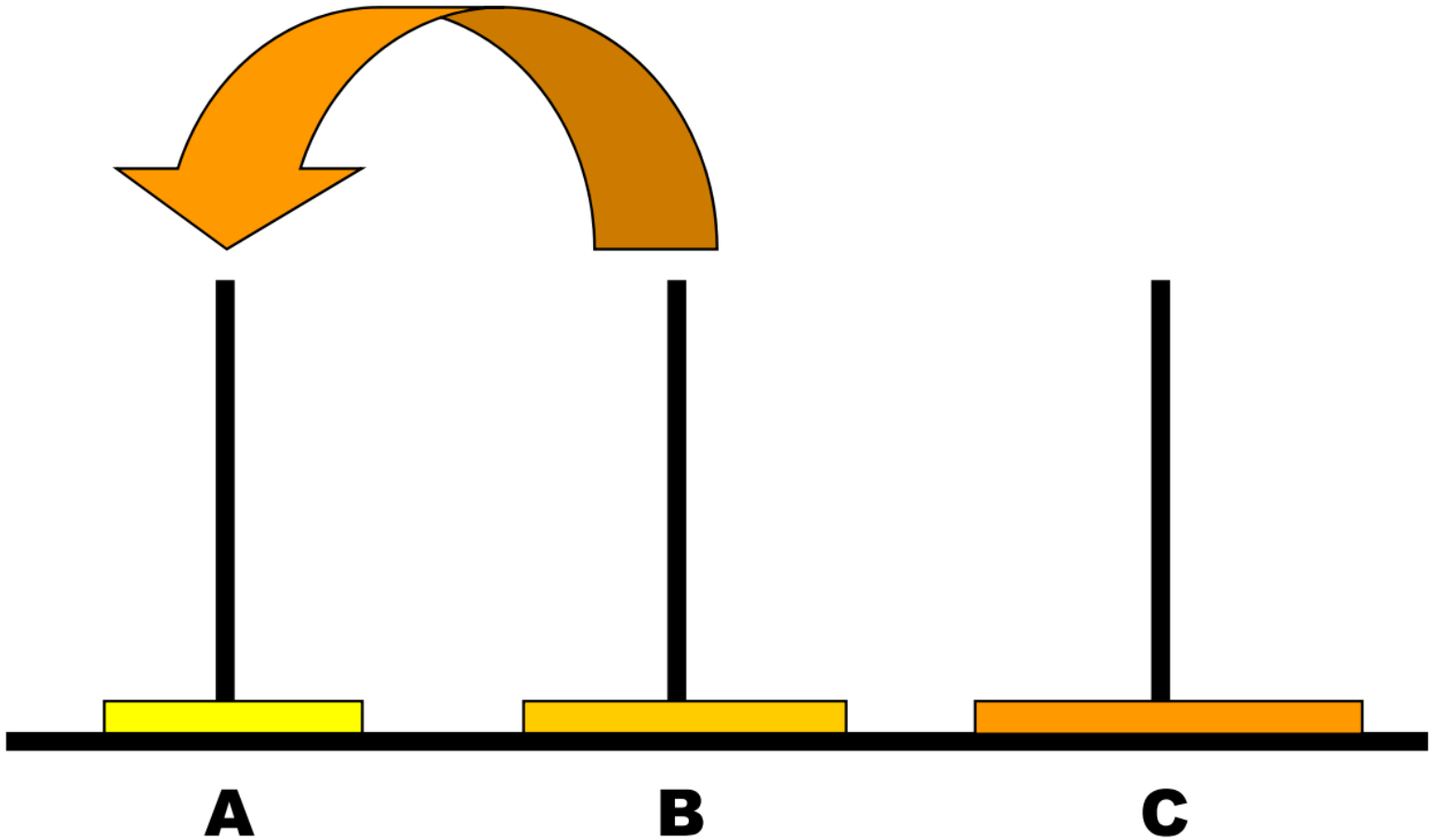
# Move 3



# Move 4

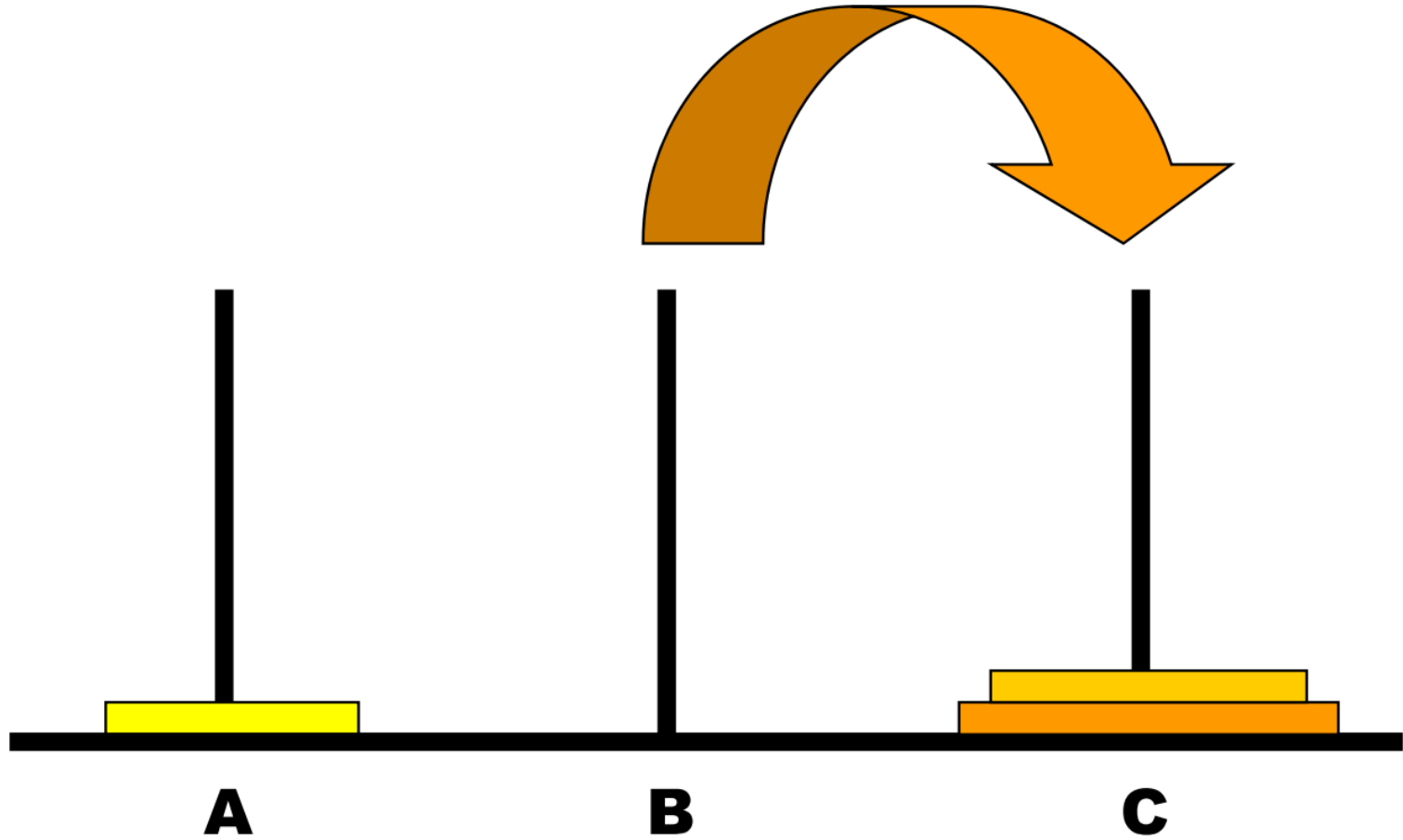


# Move 5

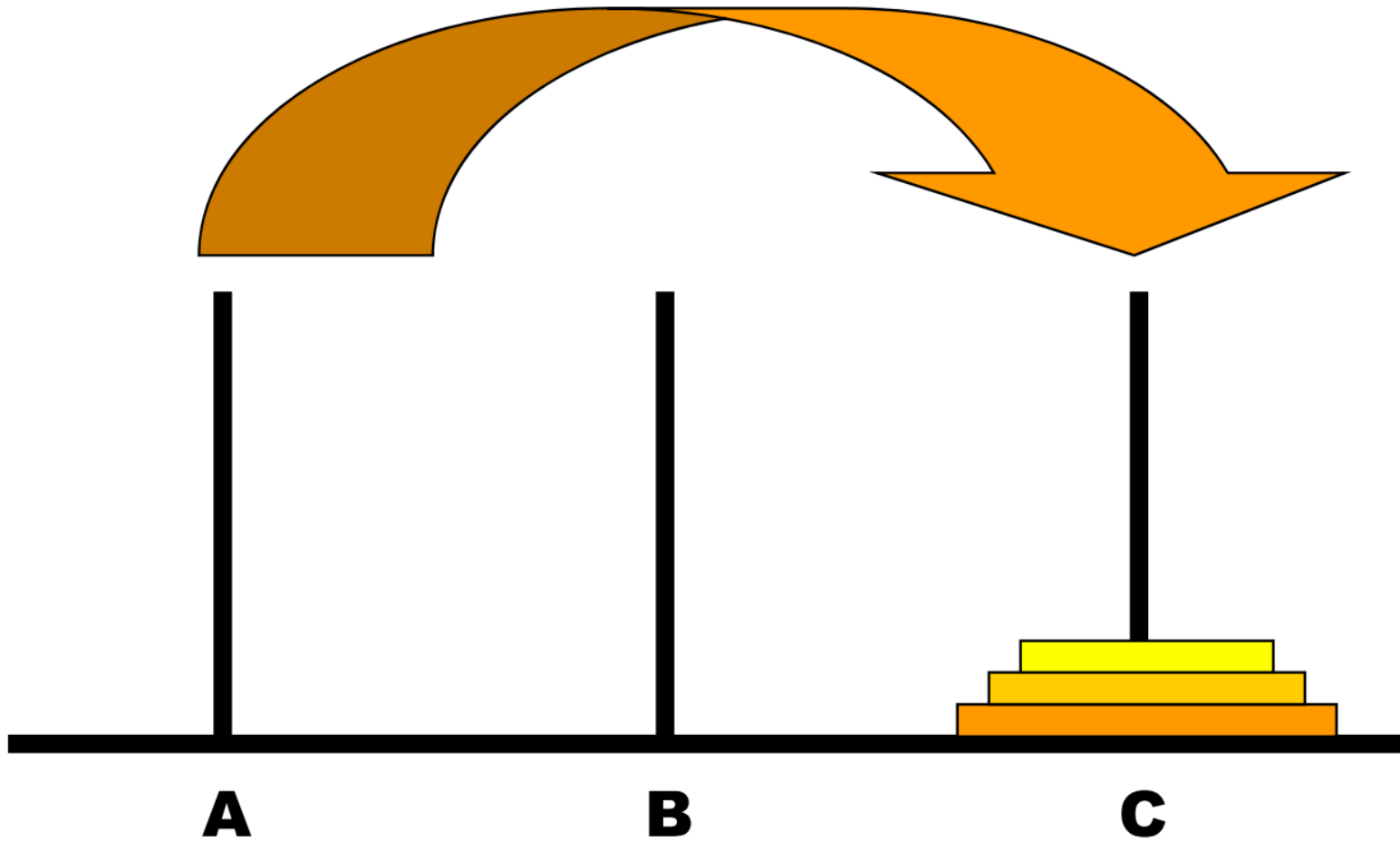




# Move 6



# 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

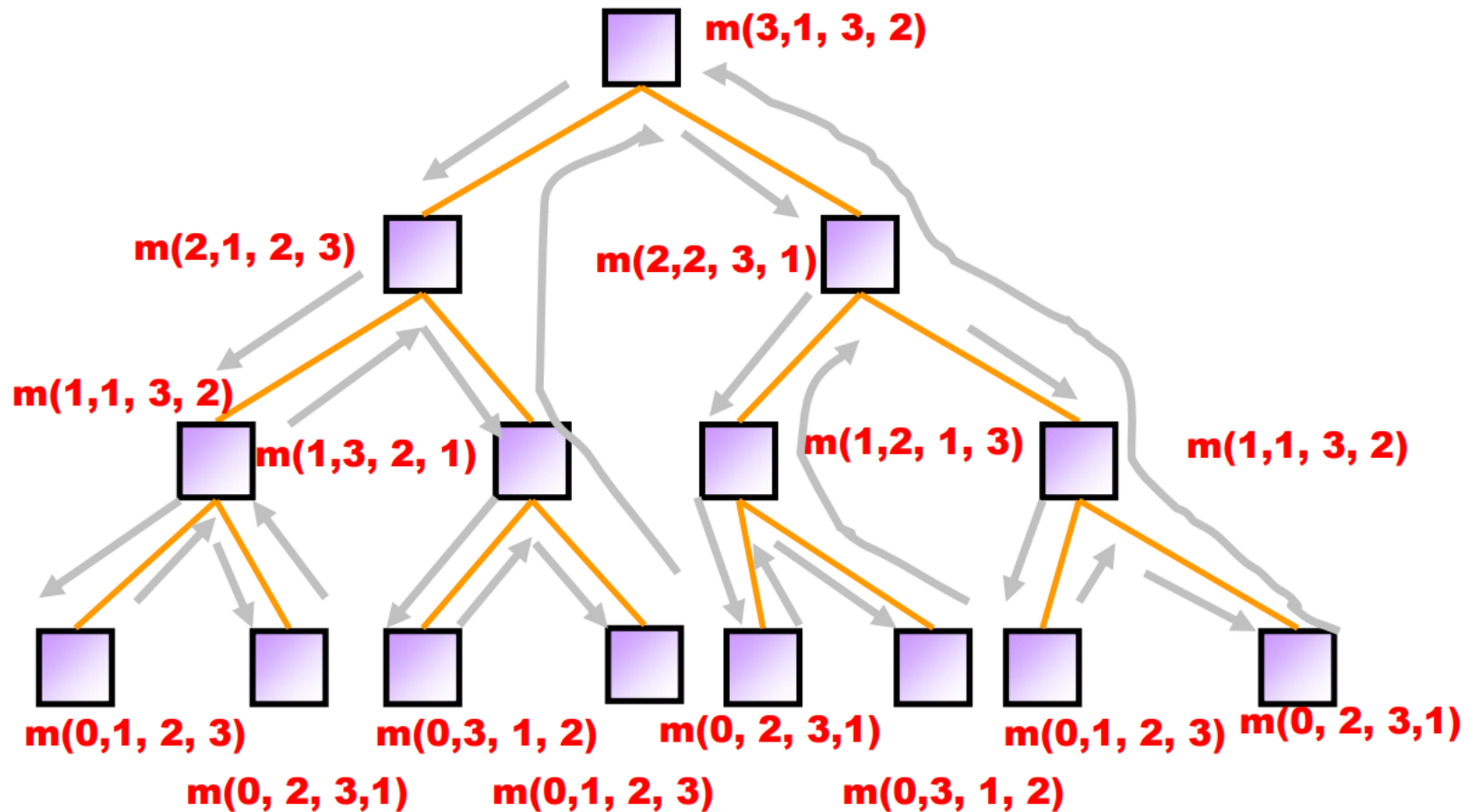
Move disk 3 from tower 1 to tower 3

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



# 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**.

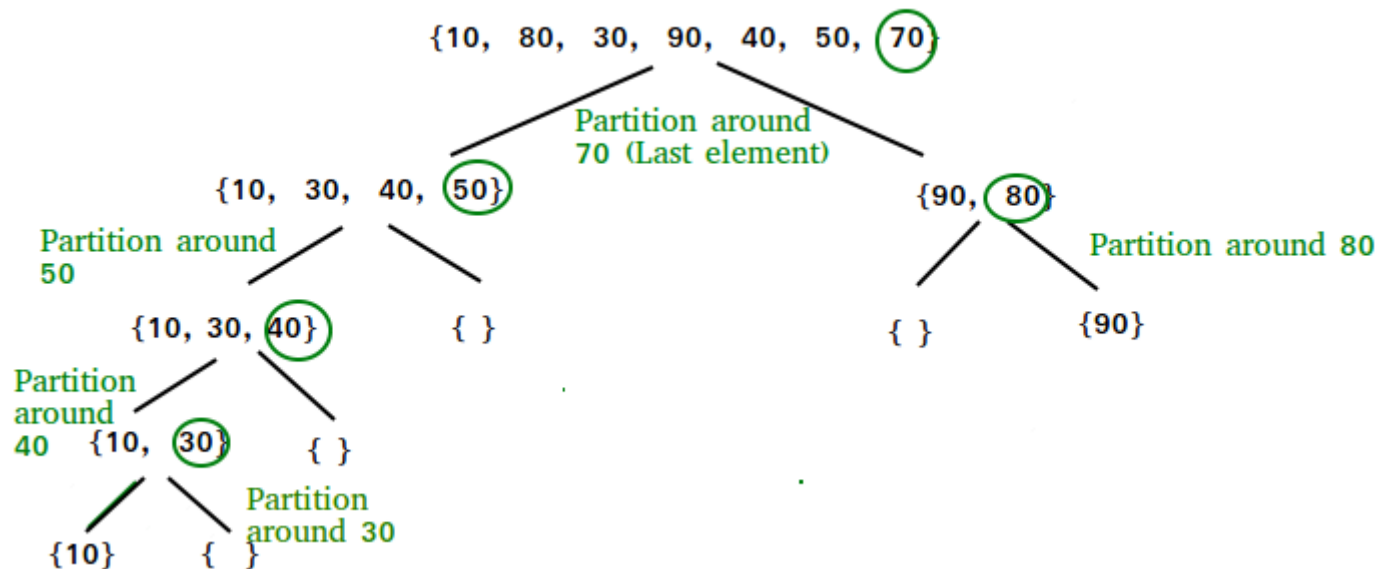


# Sorting a Sequence of Data

- **Quick sort**
  - a divide-and-conquer algorithm
  - pick an element as pivot; partition the data sequence into two sequences around the pivot
    - The left sequence contains element smaller than pivot
    - The right sequence contains element larger than pivot
  - This partition is recursively applied to the sequence until no partition can be done

# Sorting a Sequence of Data

- An example of quick sort



# Recursive Quick Sort

- The algorithm

- Key part: partition of the data sequence

```
partition(arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1) // Index of smaller element

    for (j = low; j <= high - 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++; // increment index of smaller element
            if(i != j) swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

# Recursive Quick Sort

- The algorithm

- Key part: partition of the data sequence

```
arr[] = {10, 80, 30, 90, 40, 50, 70}
```

```
Indexes:  0   1   2   3   4   5   6
```

```
low = 0, high = 6, pivot = arr[h] = 70
```

```
Initialize index of smaller element, i = -1
```

```
Traverse elements from j = low to high-1
```

```
j = 0 : Since arr[j] <= pivot, do i++
```

```
i = 0
```

```
arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j  
                                         // are same
```

```
j = 1 : Since arr[j] > pivot, do nothing
```

```
// No change in i and arr[]
```

# Recursive Quick Sort

- The algorithm

- Key part: partition of the data sequence

**j = 2** : Since  $\text{arr}[j] \leq \text{pivot}$ , do  $i++$  and  $\text{swap}(\text{arr}[i], \text{arr}[j])$

**i = 1**

$\text{arr}[] = \{10, 30, 80, 90, 40, 50, 70\}$  // We swap 80 and 30

**j = 3** : Since  $\text{arr}[j] > \text{pivot}$ , do nothing

// No change in  $i$  and  $\text{arr}[]$

**j = 4** : Since  $\text{arr}[j] \leq \text{pivot}$ , do  $i++$  and  $\text{swap}(\text{arr}[i], \text{arr}[j])$

**i = 2**

$\text{arr}[] = \{10, 30, 40, 90, 80, 50, 70\}$  // 80 and 40 Swapped

**j = 5** : Since  $\text{arr}[j] \leq \text{pivot}$ , do  $i++$  and swap  $\text{arr}[i]$  with  $\text{arr}[j]$

**i = 3**

$\text{arr}[] = \{10, 30, 40, 50, 80, 90, 70\}$  // 90 and 50 Swapped

We come out of loop because  $j$  is now equal to  $\text{high}-1$ .

**Finally we place pivot at correct position by swapping  $\text{arr}[i+1]$  and  $\text{arr}[\text{high}]$  (or pivot)**

$\text{arr}[] = \{10, 30, 40, 50, 70, 90, 80\}$  // 80 and 70 Swapped

# Recursive Quick Sort

- The algorithm
  - The recursive part

```
quickSort(arr[], low, high)
```

```
{
```

```
    if (low < high)
```

```
    {
```

```
        /* pi is partitioning index, arr[pi] is now  
           at right place */
```

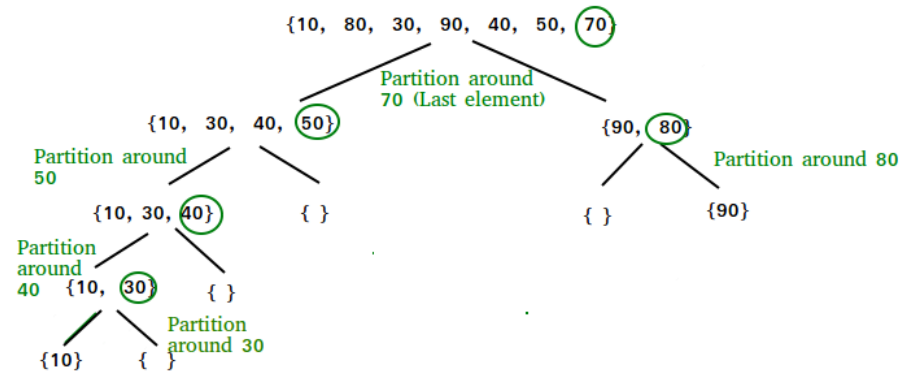
```
        pi = partition(arr, low, high);
```

```
        quickSort(arr, low, pi - 1); // Before pi
```

```
        quickSort(arr, pi + 1, high); // After pi
```

```
    }
```

```
}
```



# Recursive Quick Sort for Array

- **Constructing a dynamic array**
  - Declaring dynamic array structure

```
struct ARRAY
{
    int size;
    int count;
    float* data;
};
```

Note that the size of the array (for storage) could be larger than the real number (count) of the array

# Recursive Quick Sort for Array

- **Constructing a dynamic array**
  - What functions we need (by design)?

```
bool create_array(int, ARRAY*);  
void destroy_array(ARRAY*);
```

```
bool array_add_element(ARRAY*, float);  
bool array_remove_element(ARRAY*, int);
```

```
int get_array_count(const ARRAY*);  
int get_array_size(const ARRAY*);
```

```
float& get_array_element(int, const ARRAY*);  
void set_array_element(int, float, ARRAY*);
```

```
void print_array(const ARRAY*);
```



# Recursive Quick Sort for Array

- **Constructing a dynamic array**
  - Create the array

```
bool create_array(int size, ARRAY* p_array_out)
{
    if (size <= 0)
        return false;

    if (p_array_out != NULL)
    {
        p_array_out->data = (float*)malloc(sizeof(float) * size);
        if (p_array_out->data == NULL)
            return false;

        p_array_out->size = size;
        p_array_out->count = 0;

        return true;
    }
    else
        return true;
}
```

# Recursive Quick Sort for Array

- **Constructing a dynamic array**
  - Destroy the array

```
void destroy_array(ARRAY* p_array)
{
    if (p_array != NULL)
    {
        if (p_array->data != NULL)
            free(p_array->data);

        p_array->size = 0;
        p_array->count = 0;
    }
}
```

# Recursive Quick Sort for Array

- **Constructing a dynamic array**

- Add array element (dynamically)

```
bool array_add_element(ARRAY* p_array, float element_value)
{
    if (p_array == NULL || p_array->data == NULL
        || p_array->count < 0 || p_array->size <= 0)
        return false;

    if (p_array->count + 1 <= p_array->size)
    {
        p_array->count++;
        p_array->data[p_array->count - 1] = element_value;
        return true;
    }
    else
    {
        ...
    }

    return true;
}
```

# Recursive Quick Sort for Array

- **Constructing a dynamic array**
  - Add array element (dynamically)

```
bool array_add_element(ARRAY* p_array, float element_value)
{
    ...
    else
    {
        float* data_prev = p_array->data;
        p_array->data = (float*)realloc(data_prev, p_array->size
                                         + ARRAY\_INCREMENT\_SIZE);

        if (p_array->data == NULL)
            return false;

        p_array->size += ARRAY\_INCREMENT\_SIZE;

        p_array->count++;
        p_array->data[p_array->count - 1] = element_value;
        return true;
    }

    return true;
}
```

# Recursive Quick Sort for Array

- **Constructing a dynamic array**
  - Remove array element

```
bool array_remove_element(ARRAY* p_array, int i)
{
    if (i < 0 || i >= p_array->count)
        return false;

    int rest_count = p_array->count - i - 1;
    int copy_size = sizeof(float) * rest_count;

    float* p_temp_data = (float*)malloc(copy_size);
    if (p_temp_data == NULL)
        return false;
    memcpy(p_temp_data, &p_array->data[i + 1], copy_size);

    memcpy(&p_array->data[i], p_temp_data, copy_size);
    p_array->data[p_array->count - 1] = 0.0f;

    p_array->count--;

    ...
}
```

# Recursive Quick Sort for Array

- **Constructing a dynamic array**
  - Remove array element

```
bool array_remove_element(ARRAY* p_array, int i)
{
    ...

    if (p_array->count < p_array->size - 2 * ARRAY_INCREMENT_SIZE)
    {
        float* p_temp_data = p_array->data;
        p_array->data = (float*)realloc(p_temp_data, p_array->size
                                         - ARRAY_INCREMENT_SIZE);

        if (p_array->data == NULL)
            return false;
    }

    return true;
}
```

# Recursive Quick Sort for Array

- Constructing a dynamic array

- Remove array element

```
bool array_remove_element2(ARRAY* p_array, int i)
{
    if (i < 0 || i >= p_array->count)
        return false;

    for (int j = i; j < p_array->count - 1; j++)
        p_array->data[j] = p_array->data[j + 1];
    p_array->data[p_array->count - 1] = 0.0f;

    p_array->count--;

    if (p_array->count < p_array->size - 2 * ARRAY_INCREMENT_SIZE)
    {
        float* p_temp_data = p_array->data;
        p_array->data = (float*)realloc(
            p_temp_data, p_array->size - ARRAY_INCREMENT_SIZE);
        if (p_array->data == NULL)
            return false;
    }
    return true;
}
```

# Recursive Quick Sort for Array

- **Constructing a dynamic array**

- Other array functions

```
int get_array_count(const ARRAY* p_array)
{
    return p_array->count;
}
```

```
int get_array_size(const ARRAY* p_array)
{
    return p_array->size;
}
```

```
float& get_array_element(int i, const ARRAY* p_array)
{
    return p_array->data[i];
}
```

```
void set_array_element(int i, float data_value, ARRAY* p_array)
{
    p_array->data[i] = data_value;
}
```



# Recursive Quick Sort for Array

- **Constructing a dynamic array**
  - Print(display) the array on the screen

```
void print_array(const ARRAY* p_array)
{
    for (int i = 0; i < get_array_count(p_array) - 1; i++)
        printf("%f, ", get_array_element(i, p_array));

    printf("%f\n", get_array_element(
        get_array_count(p_array) - 1, p_array));
}
```

# Recursive Quick Sort for Array

- Implementing quicksort with dynamic array
  - Partition function

```
int quick_sort_array_partition
    (ARRAY*p_array , int low, int high)
{
    float pivot = get_array_element(high,p_array);
    int i = low-1;

    for (int j = low; j < high; j++)
    {
        if (get_array_element(j, p_array) < pivot)
        {
            i++;
            if (i != j)
                swap_data(&get_array_element(i, p_array),
                        &get_array_element(j, p_array));
        }
    }
    swap_data(&get_array_element(i+1, p_array),
            &get_array_element(high, p_array));

    return i+1;
}
```

```
void swap_data(float* a, float* b)
{
    float t = *a;
    *a = *b;
    *b = t;
}
```

```
partition(arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1) // Index of smaller element

    for (j = low; j <= high - 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++;
            if(i != j) swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high]
    return (i + 1)
}
```

# Recursive Quick Sort for Array

- **Implementing quicksort with dynamic array**
  - Recursive function call for partitioning

```
bool quick_sort_array(ARRAY* p_array_in_out, int low, int high)
{
    if (low < high)
    {
        int partition_index =
            quick_sort_array_partition(p_array_in_out, low, high);

        quick_sort_array(p_array_in_out, low, partition_index - 1);
        quick_sort_array(p_array_in_out, partition_index + 1, high);

        return true;
    }
    else
        return false;
}
```

# Recursive Quick Sort for Array

- How to use the quicksort for dynamic array?

```
void test_array_sort()
{
    ARRAY a;
    if (!create_array(20, &a))
    {
        printf("unable to create the array!\n");
        return;
    }

    printf("Please input the array elements:\n");

    while (1)
    {
        char str[32];
        scanf("%s", str);
        if (strcmp(str, "quit") == 0)
            break;
        array_add_element(&a, (float)atof(str));
    }
    ...
}
```

# Recursive Quick Sort for Array

- How to use the quicksort for dynamic array?

```
void test_array_sort()
{
    ...

    printf("Your input ARRAY is:\n");
    print_array(&a);

    printf("\nSorting the whole array...\n");
    quick_sort_array(&a, 0, get_array_count(&a) - 1);
    printf("Your sorted array is:\n");
    print_array(&a);

    destroy_array(&a); //don't forget to destroy the array
}
```

# 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.