UNIVERSITY of LOUISVILLE
dare to be great

CPP: Recursion

CECS130
Introduction to Programming Languages
Dr. Roman V. Yampolskiy

Recursive Definitions

- Recursion: solving a problem by reducing it to smaller versions of itself
- $0! = 1$ (1)

  $n! = n \times (n-1)!$ if $n > 0$ (2)
- The definition of factorial in equations (1) and (2) is called a recursive definition
- Equation (1) is called the base case
- Equation (2) is called the general case

Recursive Definitions (continued)

- Recursive definition: defining a problem in terms of a smaller version of itself
  - Every recursive definition must have one (or more) base cases
  - The general case must eventually reduce to a base case
  - The base case stops the recursion

Recursive Algorithms

- Recursive algorithm: finds a solution by reducing problem to smaller versions of itself
  - Must have one (or more) base cases
- General solution must eventually reduce to a base case
- Recursive function: a function that calls itself
- Recursive algorithms are implemented using recursive functions

Recursive Functions

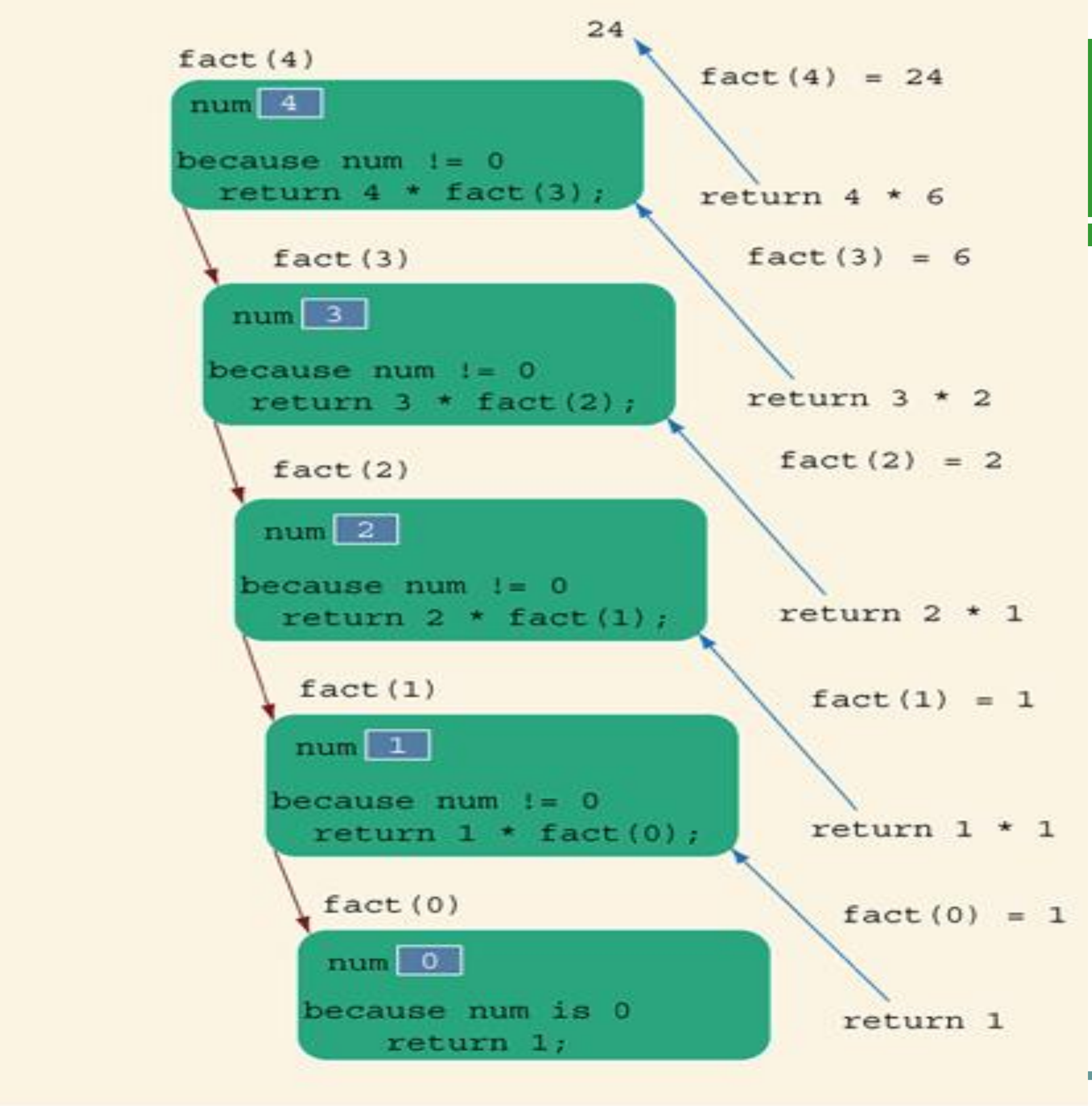
- Think of a recursive function as having infinitely many copies of itself
- Every call to a recursive function has
  - Its own code
  - Its own set of parameters and local variables
- After completing a particular recursive call
  - Control goes back to the calling environment, which is the previous call

Recursive Functions (continued)

- The current (recursive) call must execute completely before control goes back to the previous call
- Execution in the previous call begins from the point immediately following the recursive call
- <u>Tail recursive function</u>: A recursive function in which the last statement executed is the recursive call
  - Example: the function fact

Recursive Functions (continued)

```
int fact(int num)
{
     if (num == 0)
          return 1;
     else
          return num * fact(num - 1);
}
```



Execution of fact(4)

Direct and Indirect Recursion

- Directly recursive: a function that calls itself
- Indirectly recursive: a function that calls another function and eventually results in the original function call

Infinite Recursion

- Infinite recursion: every recursive call results in another recursive call
  - In theory, infinite recursion executes forever
- Because computer memory is finite:
  - Function executes until the system runs out of memory
  - Results in an abnormal program termination

Infinite Recursion (continued)

- To design a recursive function:
  - Understand problem requirements
  - Determine limiting conditions
  - Identify base cases and provide a direct solution to each base case
  - Identify general cases and provide a solution to each general case in terms of smaller versions of itself

Problem Solving Using Recursion

- General case: List size is greater than 1
- To find the largest element in `list[a]...list[b]`
  - Find largest element in `list[a + 1]...list[b]` and call it `max`
  - Compare the elements `list[a]` and `max`

    if (`list[a] >= max`)

      the largest element in `list[a]...list[b]` is

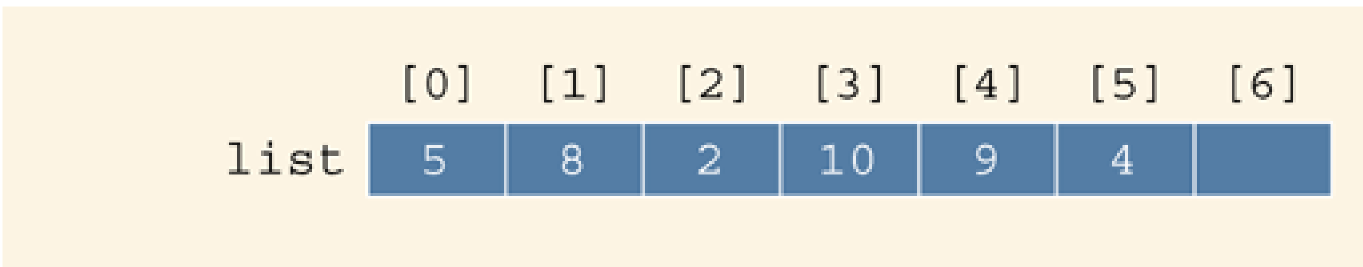
      `list[a]`

    otherwise

      the largest element in `list[a]...list[b]` is `max`

Problem Solving Using Recursion

Example: Largest Element in an Array



`list` with six elements

Problem Solving Using Recursion

Example: Largest Element in an Array

- `list[a]...list[b]` stands for the array elements `list[a]`, `list[a + 1], ..., list[b]`.
- `list[0]...list[5]` represents the array elements `list[0]`, `list[1], list[2], list[3], list[4], and list[5]`.
- If `list` is of length `1`, then `list` has only one element, which is the largest element.
- Suppose the length of `list` is greater than `1`.
  - To find the largest element in `list[a]...list[b]`, we first find the largest element in `list[a + 1]...list[b]` and then compare this largest element with `list[a]`.
  - The largest element in `list[a]...list[b]` is given by:

```
maximum(list[a], largest(list[a + 1]...list[b]))
```

Problem Solving Using Recursion

Base Case: The size of the list is 1
The only element in the list is the largest element

General Case: The size of the list is greater than 1
To find the largest element in list[a]...list[b]

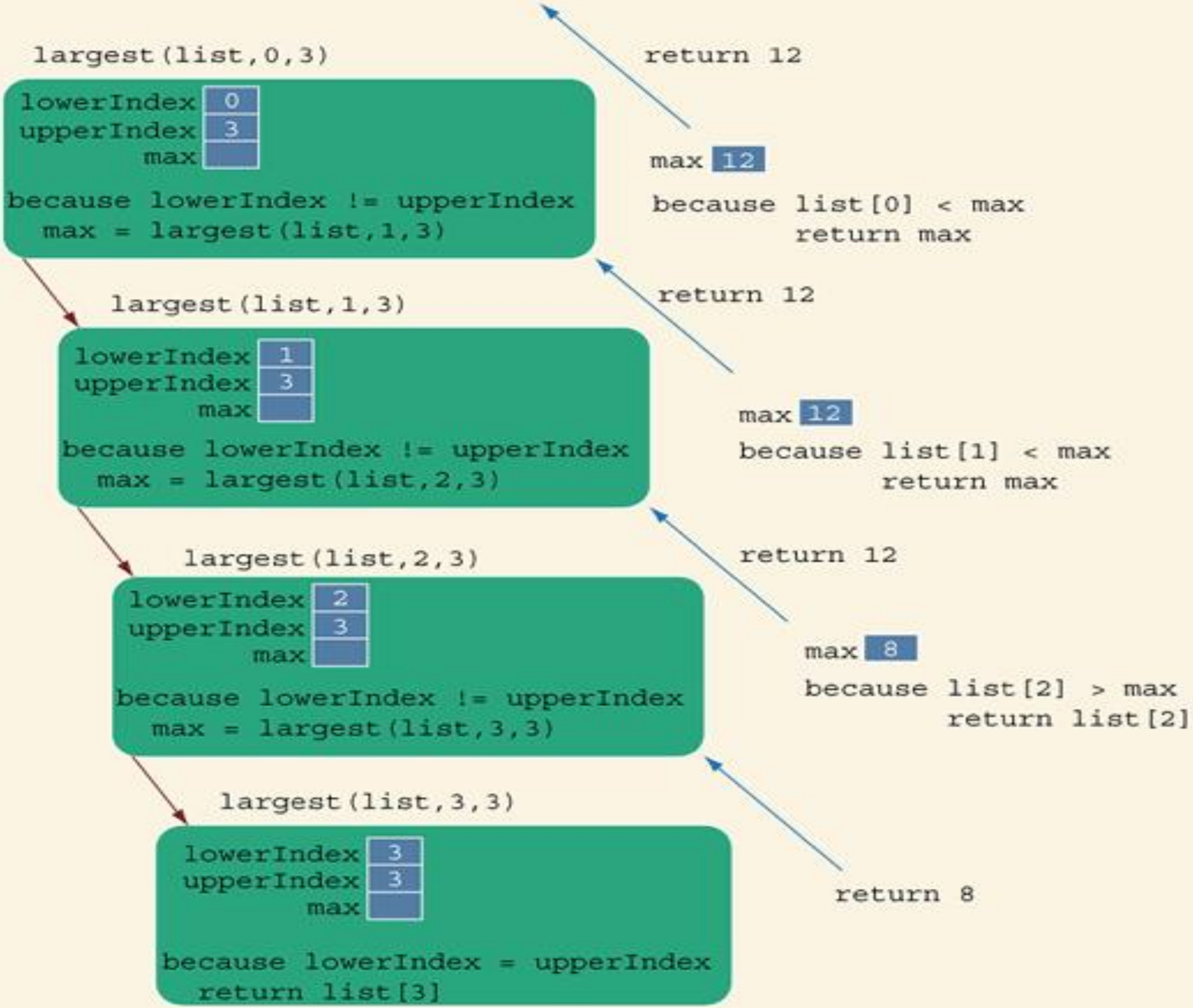
```
a. Find the largest element in list[a + 1]...list[b]
   and call it max
b. Compare the elements list[a] and max
   if (list[a] >= max)
      the largest element in list[a]...list[b] is list[a]
   otherwise
      the largest element in list[a]...list[b] is max
```

```
int largest(const int list[], int lowerIndex, int upperIndex)
{
    int max;

    if (lowerIndex == upperIndex) //size of the sublist is one
        return list[lowerIndex];
    else
    {
        max = largest(list, lowerIndex + 1, upperIndex);

        if (list[lowerIndex] >= max)
            return list[lowerIndex];
        else
            return max;
    }
}
```

	[0]	[1]	[2]	[3]
list	5	10	12	8

list with four elements

```
cout << largest(list, 0, 3) << endl;
```



Execution of largest(list, 0, 3)

Example: Fibonacci Number

$$rFibNum(a,b,n) = \begin{cases} a & \text{if } n = 1 \\ b & \text{if } n = 2 \\ rFibNum(a,b,n-1) + rFibNum(a,b,n-2) & \text{if } n > 2. \end{cases} \quad (16\text{-}3)$$

```
rFibNum(2, 5, 4)
```

1. `rFibNum(2, 5, 4) = rFibNum(2, 5, 3) + rFibNum(2, 5, 2)`

Next, we determine `rFibNum(2, 5, 3)` and `rFibNum(2, 5, 2)`. Let us first determine `rFibNum(2, 5, 3)`. Here, $a = 2$, $b = 5$, and n is 3. Because n is 3,

1.a. `rFibNum(2, 5, 3) = rFibNum(2, 5, 2) + rFibNum(2, 5, 1)`

This statement requires us to determine `rFibNum(2, 5, 2)` and `rFibNum(2, 5, 1)`. In `rFibNum(2, 5, 2)`, $a = 2$, $b = 5$, and $n = 2$. Therefore, from the definition given in Equation 16-3, it follows that:

1.a.1. `rFibNum(2, 5, 2) = 5`

To find `rFibNum(2, 5, 1)`, note that $a = 2$, $b = 5$, and $n = 1$. Therefore, by the definition given in Equation 16-3,

1.a.2. `rFibNum(2, 5, 1) = 2`

We substitute the values of `rFibNum(2, 5, 2)` and `rFibNum(2, 5, 1)` into (1.a) to get:

`rFibNum(2, 5, 3) = 5 + 2 = 7`

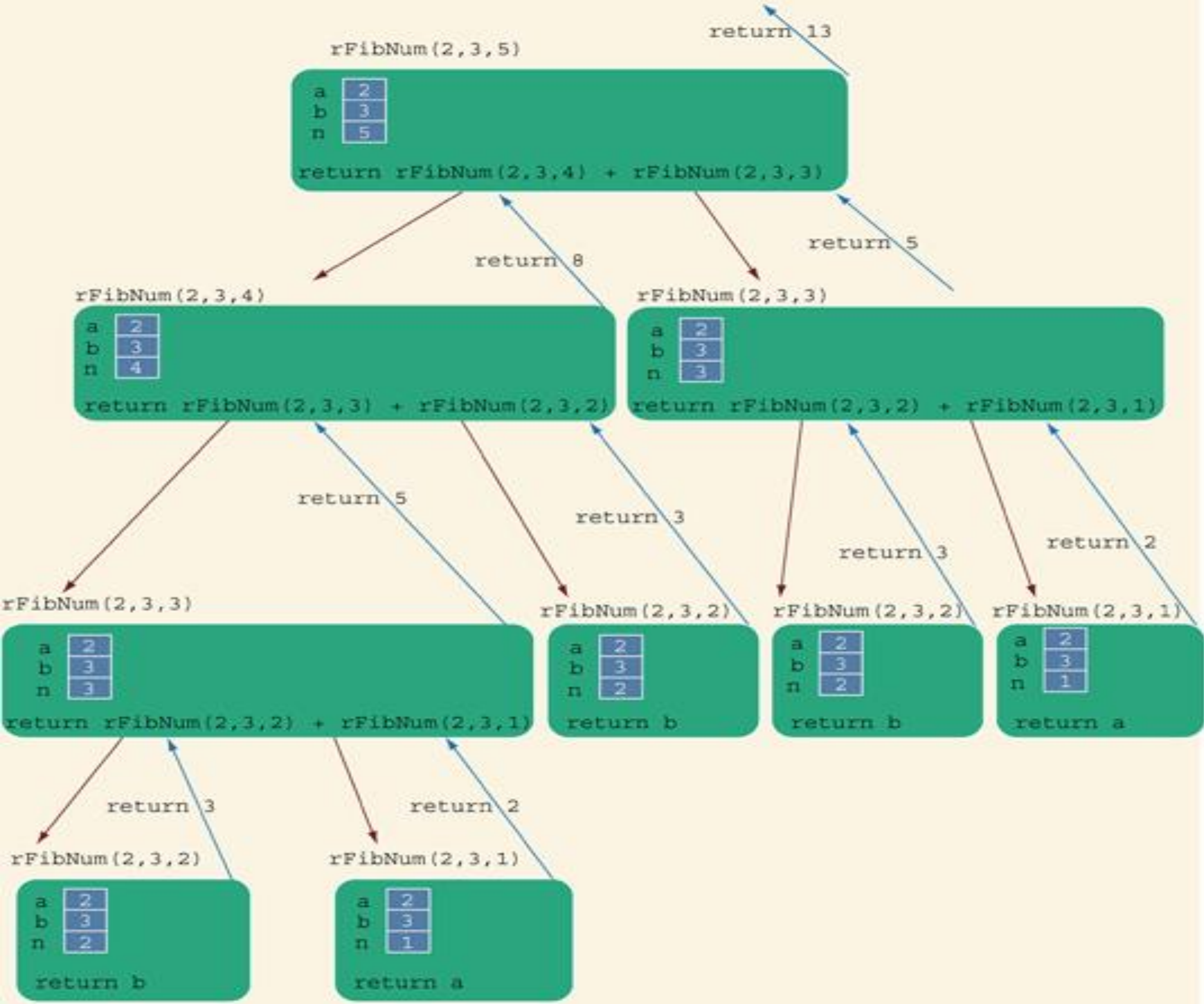
Next, we determine `rFibNum(2, 5, 2)`. As in (1.a.1), `rFibNum(2, 5, 2) = 5`. We can substitute the values of `rFibNum(2, 5, 3)` and `rFibNum(2, 5, 2)` into (1) to get:

`rFibNum(2, 5, 4) = 7 + 5 = 12`

Example: Fibonacci Number

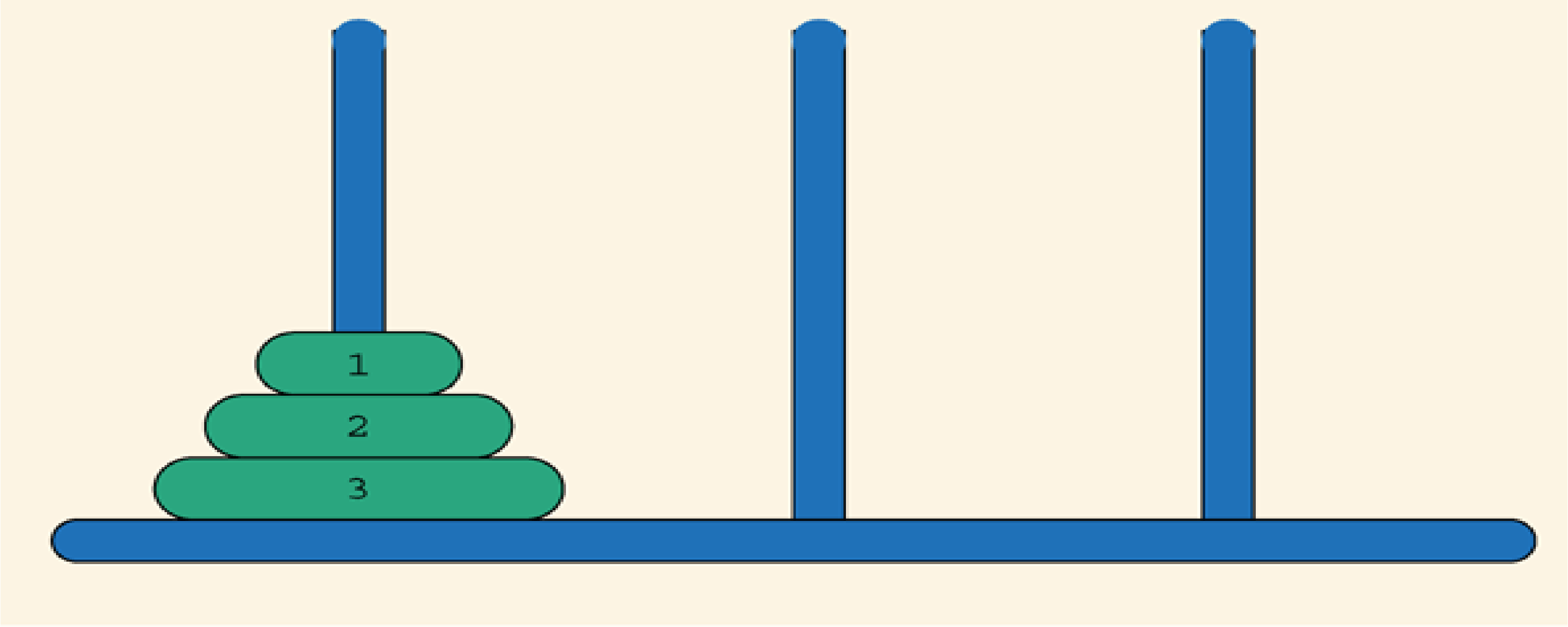
```
int rFibNum(int a, int b, int n)
{
    if (n == 1)
        return a;
    else if (n == 2)
        return b;
    else
        return rFibNum(a, b, n - 1) + rFibNum(a, b, n - 2);
}
```

```
cout << rFibNum(2, 3, 5) << endl;
```



Execution of rFibNum(2, 3, 5)

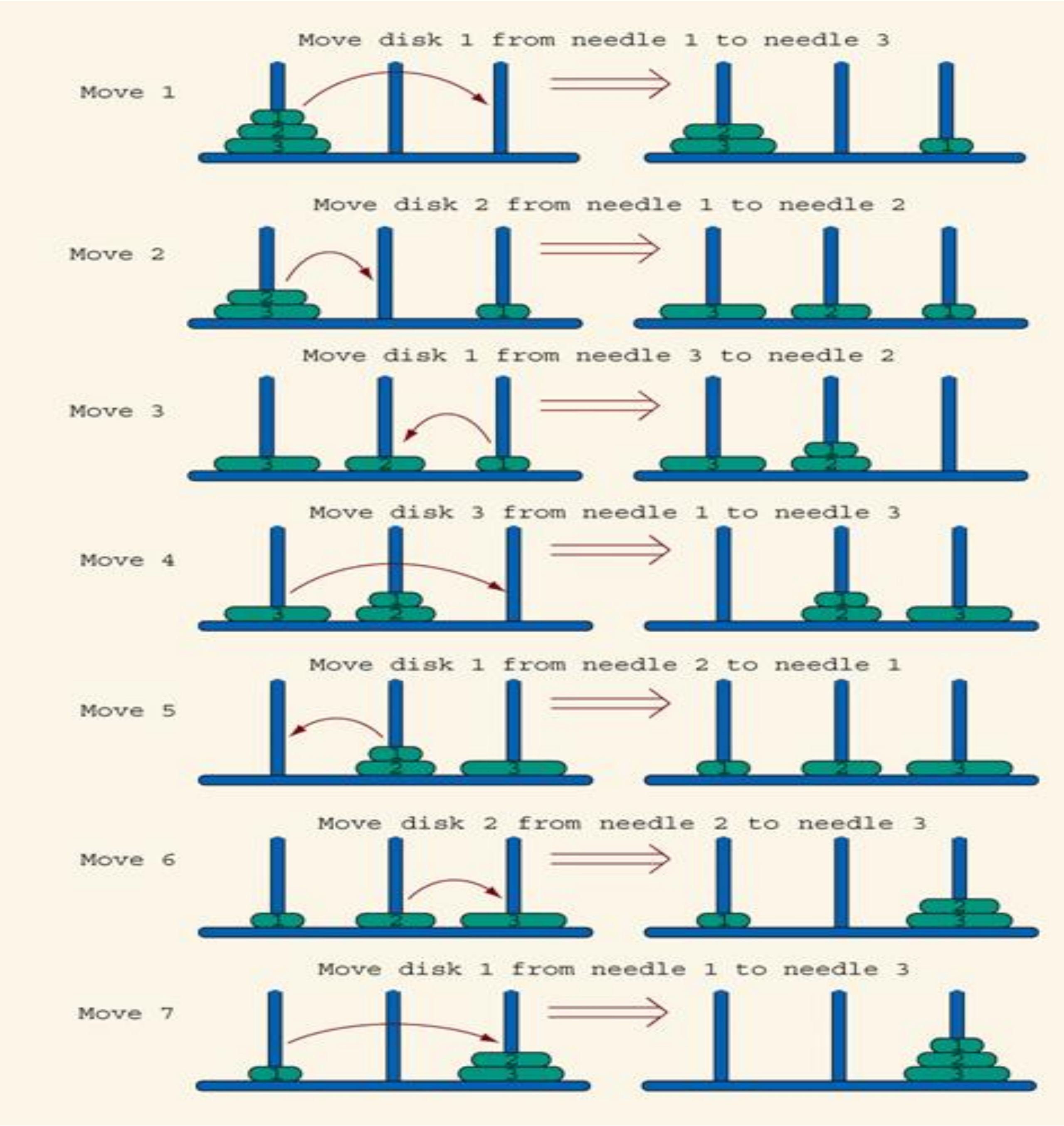
Example: Tower of Hanoi



Tower of Hanoi problem with three disks

The rules for moving the disks are as follows:

1. Only one disk can be moved at a time.
2. The removed disk must be placed on one of the needles.
3. A larger disk cannot be placed on top of a smaller disk.



Solution to Tower of Hanoi problem with three disks

1. Move the top $n - 1$ disks from needle 1 to needle 2, using needle 3 as the intermediate needle.
2. Move disk number n from needle 1 to needle 3.
3. Move the top $n - 1$ disks from needle 2 to needle 3, using needle 1 as the intermediate needle.

This recursive algorithm translates into the following C++ function:

```
void moveDisks(int count, int needle1, int needle3, int needle2)
{
    if (count > 0)
    {
        moveDisks(count - 1, needle1, needle2, needle3);

        cout << "Move disk " << count << " from " << needle1
             << " to " << needle3 << "." << endl;

        moveDisks(count - 1, needle2, needle3, needle1);
    }
}
```

Tower of Hanoi: Analysis

- Let us determine how long it would take to move 64 disks from needle 1 to needle 3.
- If needle 1 contains 3 disks, then the number of moves required to move all 3 disks from needle 1 to needle 3 is $2^3 - 1 = 7$.
- If needle 1 contains 64 disks, then the number of moves required to move all 64 disks from needle 1 to needle 3 is $2^{64} - 1$.
- Because $2^{10} = 1024 \approx 1000 = 10^3$, we have

  $$2^{64} = 2^4 \times 2^{60} \approx 2^4 \times 10^{18} = 1.6 \times 10^{19}$$

- The number of seconds in one year is approximately 3.2×10^7.
- Suppose we move one disk per second and we do not rest.
- Now:

$$\begin{aligned} 1.6 \times 10^{19} &= 5 \times 3.2 \times 10^{18} = 5 \times (3.2 \times 10^7) \times 10^{11} \\ &= (3.2 \times 10^7) \times (5 \times 10^{11}) \end{aligned}$$

- The time required to move all 64 disks from needle 1 to needle 3 is roughly 5×10^{11} years.

Recursion or Iteration?

- There are usually two ways to solve a particular problem:
  - Iteration (looping)
  - Recursion
- Which method is better—iteration or recursion?
- In addition to the nature of the problem, the other key factor in determining the best solution method is efficiency

Memory allocation

- Whenever a function is called
  - Memory space for its formal parameters and (automatic) local variables is allocated
- When the function terminates
  - That memory space is then deallocated
- Every (recursive) call has its own set of parameters and (automatic) local variables

Efficiency

- Overhead associated with executing a (recursive) function in terms of
  - Memory space
  - Computer time
- A recursive function executes more slowly than its iterative counterpart

Efficiency (continued)

- On slower computers, especially those with limited memory space
  - The slow execution of a recursive function would be visible
- Today’s computers are fast and have inexpensive memory
  - Execution of a recursion function is not noticeable

Efficiency (continued)

- The choice between the two alternatives depends on the nature of the problem
- For problems such as mission control systems
  - Efficiency is absolutely critical and dictates the solution method

Efficiency (continued)

- An iterative solution is more obvious and easier to understand than a recursive solution
- If the definition of a problem is inherently recursive
  - Consider a recursive solution

Programming Example

- Use recursion to convert a non-negative integer in decimal format (base 10) into the equivalent binary number (base 2)
- Define some terms:
  - Let x be an integer
  - The remainder of x after division by 2 is the rightmost bit of x
  - The rightmost bit of 33 is 1 because 33 % 2 is 1
  - The rightmost bit of 28 is 0 because 28 % 2 is 0

Programming Example (continued)

- To find the binary representation of 35
  - Divide 35 by 2
  - The quotient is 17 and the remainder is 1
  - Divide 17 by 2
  - The quotient is 8 and the remainder is 1
  - Divide 8 by 2
  - The quotient is 4 and the remainder is 0
  - Continue this process until the quotient becomes 0

Programming Example (continued)

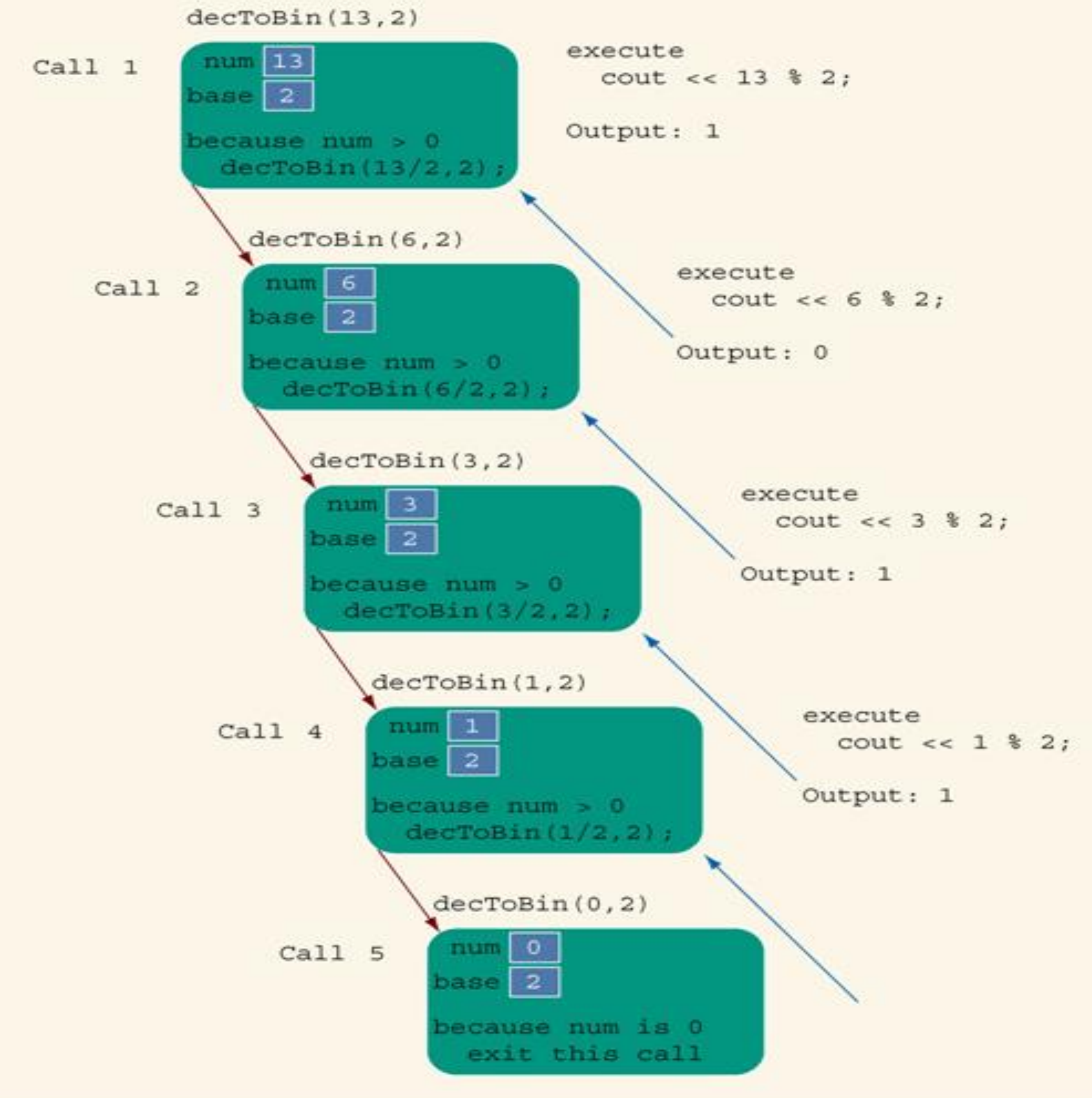
- The rightmost bit of 35 cannot be printed until we have printed the rightmost bit of 17
- The rightmost bit of 17 cannot be printed until we have printed the rightmost bit of 8, and so on
- The binary representation of 35 is the binary representation of 17 (the quotient of 35 after division by 2) followed by the rightmost bit of 35

1. binary(num) = num if num = 0.
2. binary(num) = binary(num / 2) followed by num % 2 if num > 0.

Programming Example (continued)

```
void decToBin(int num, int base)
{
    if (num > 0)
    {
        decToBin(num / base, base);
        cout << num % base;
    }
}
```

decToBin(13, 2);



Execution of decToBin(13, 2)

The End!