Algorithms and Data Structures

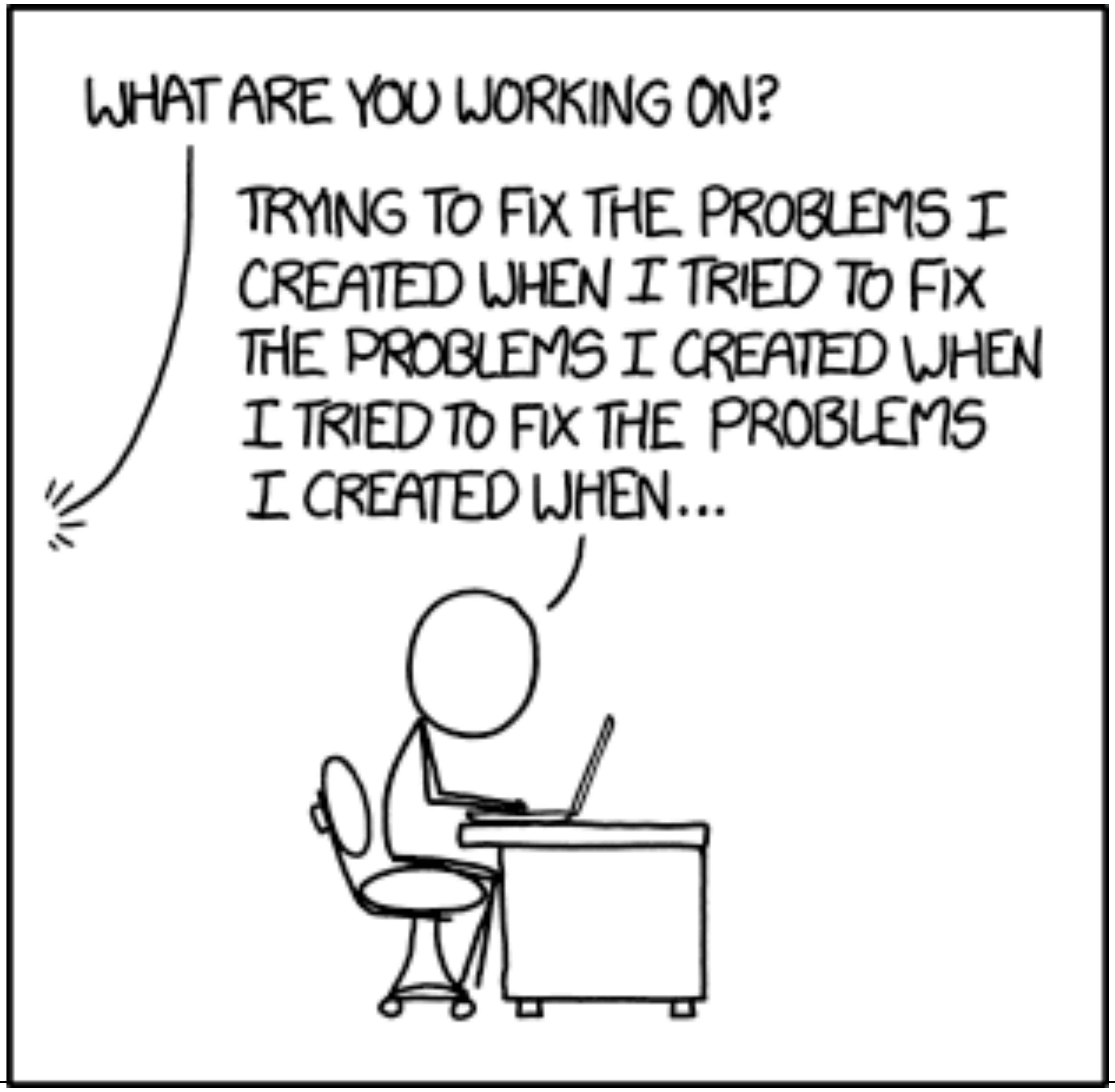
Recursion

Agenda

- What is recursion?
- Why we need recursion?
- How to def a recursive function?
- Practices

Introduction to Recursion

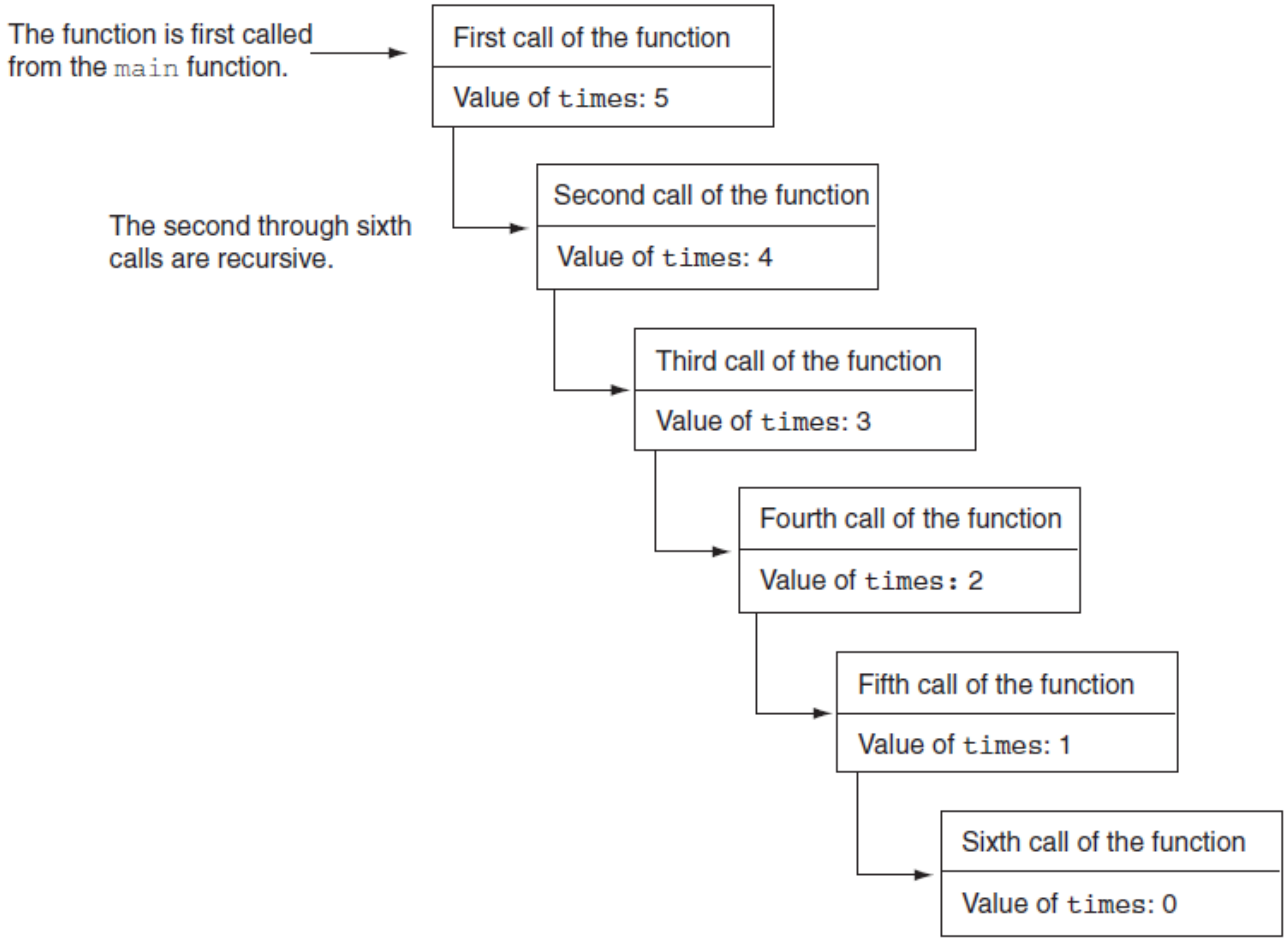
- Recursive function: a function that calls itself



Introduction to Recursion

- Recursive function must have a way to control the number of times it repeats
    - Usually involves an `if-else` statement which defines when the function should return a value and when it should call itself
- Depth of recursion: the number of times a function calls itself

Figure 12-2 Six calls to the `message` function



Introduction to Recursion (cont'd.)

Figure 12-3 Control returns to the point after the recursive function call

Recursive function call

```
def message(times):
    if times > 0:
        print('This is a recursive function.')
        message(times - 1)
```

Control returns here from the recursive call.
There are no more statements to execute
in this function, so the function returns.

Problem Solving with Recursion

- Recursion is a powerful tool for solving repetitive problems
- Recursion is never required to solve a problem
  - Any problem that can be solved recursively can be solved with a loop
  - Recursive algorithms usually less efficient than iterative ones
    - Due to *overhead* of each function call

Problem Solving with Recursion (cont'd.)

- Some repetitive problems are more easily solved with recursion
- General outline of recursive function:
  - If the problem can be solved now without recursion, solve and return
    - Known as the *base case*
  - Otherwise, reduce problem to smaller problem of the same structure and call the function again to solve the smaller problem
    - Known as the *recursive case*

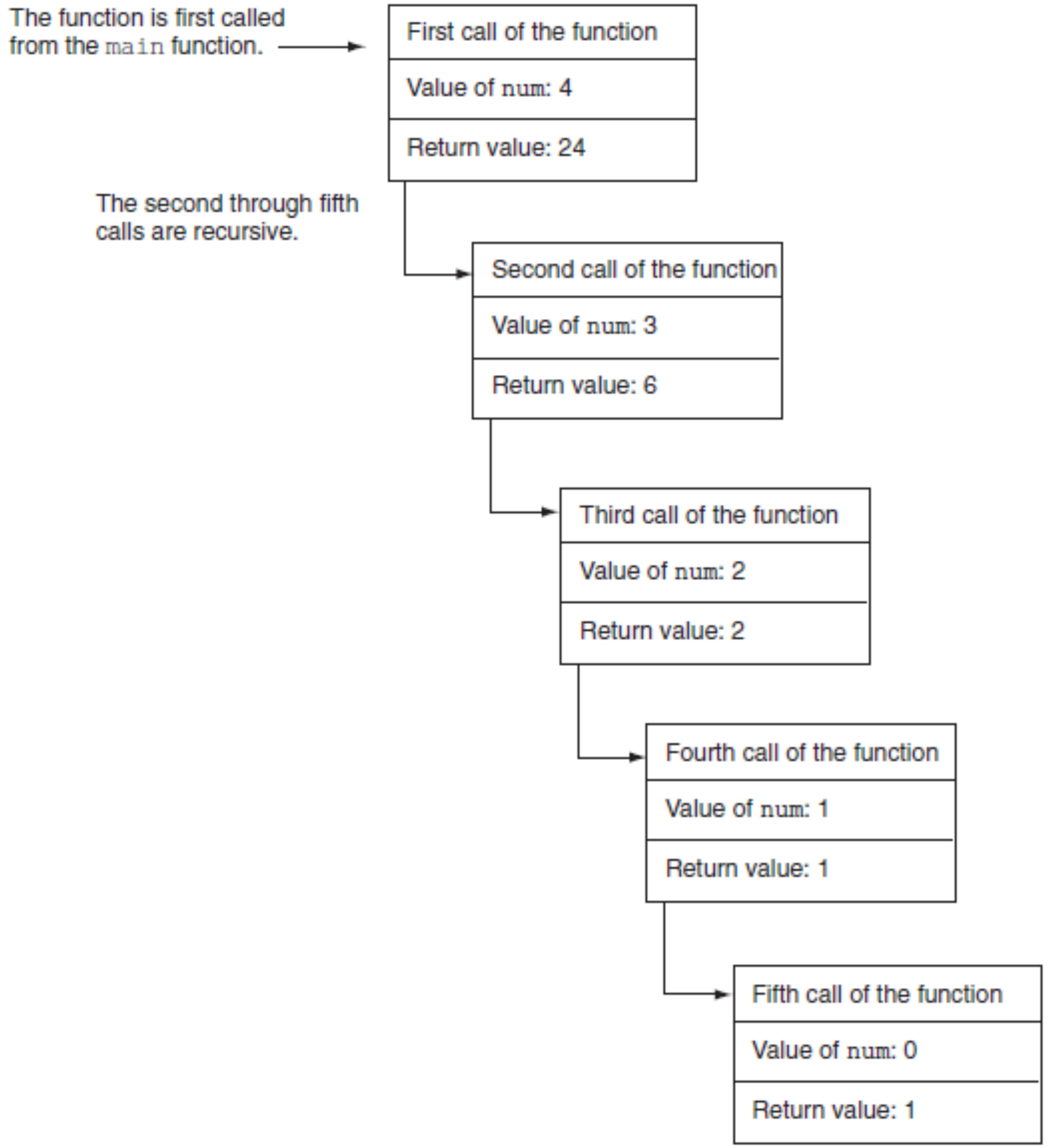
Using Recursion to Calculate the Factorial of a Number

- In mathematics, the $n!$ notation represents the factorial of a number n
  - For $n = 0$, $n! = 1$
  - For $n > 0$, $n! = 1 \times 2 \times 3 \times \ldots \times n$
- The above definition lends itself to recursive programming
  - n = 0 is the base case
  - n > 0 is the recursive case
    - factorial(n) $= n \times$ factorial(n-1)

Using Recursion (cont'd.)

```
# The factorial function uses recursion to
# calculate the factorial of its argument,
# which is assumed to be nonnegative.
def factorial(num):
    if num == 0:
        return 1
    else:
        return num * factorial(num - 1)
```

Figure 12-4 The value of `num` and the return value during each call of the function



Using Recursion (cont'd.)

- Since each call to the recursive function reduces the problem:
  - Eventually, it will get to the base case which does not require recursion, and the recursion will stop
- Usually the problem is reduced by making one or more parameters smaller at each function call

Direct and Indirect Recursion

- Direct recursion: when a function directly calls itself
  - All the examples shown so far were of direct recursion
- Indirect recursion: when function A calls function B, which in turn calls function A

Finding the Greatest Common Divisor

- Calculation of the greatest common divisor (GCD) of two positive integers
  - If x can be evenly divided by y, then
  - gcd(x,y) = y
  - Otherwise, gcd(x,y) = gcd(y, remainder of x/y)
- Corresponding function code:

```
# The gcd function returns the greatest common
# divisor of two numbers.
def gcd(x, y):
    if x % y == 0:
        return y
    else:
        return gcd(x, x % y)
```

The Towers of Hanoi

- Mathematical game commonly used to illustrate the power of recursion
  - Uses three pegs and a set of discs in decreasing sizes
  - Goal of the game: move the discs from leftmost peg to rightmost peg
    - Only one disc can be moved at a time
    - A disc cannot be placed on top of a smaller disc
    - All discs must be on a peg except while being moved

The Towers of Hanoi (cont'd.)

Figure 12-5 The pegs and discs in the Tower of Hanoi game

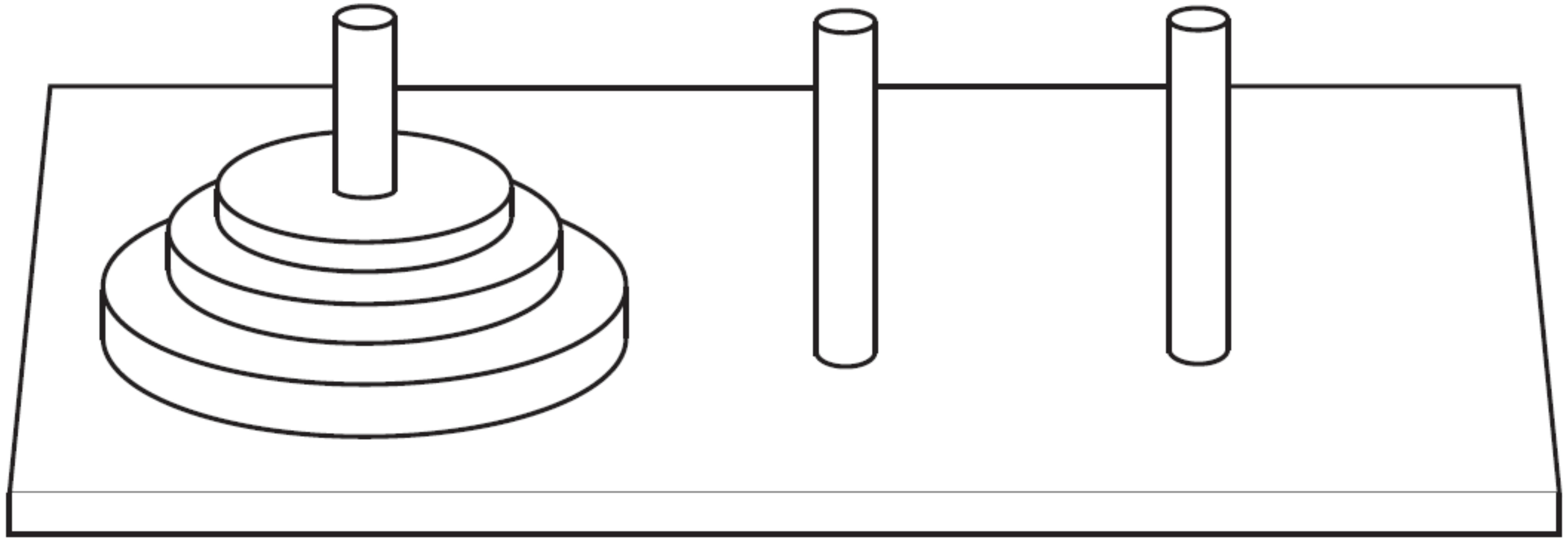
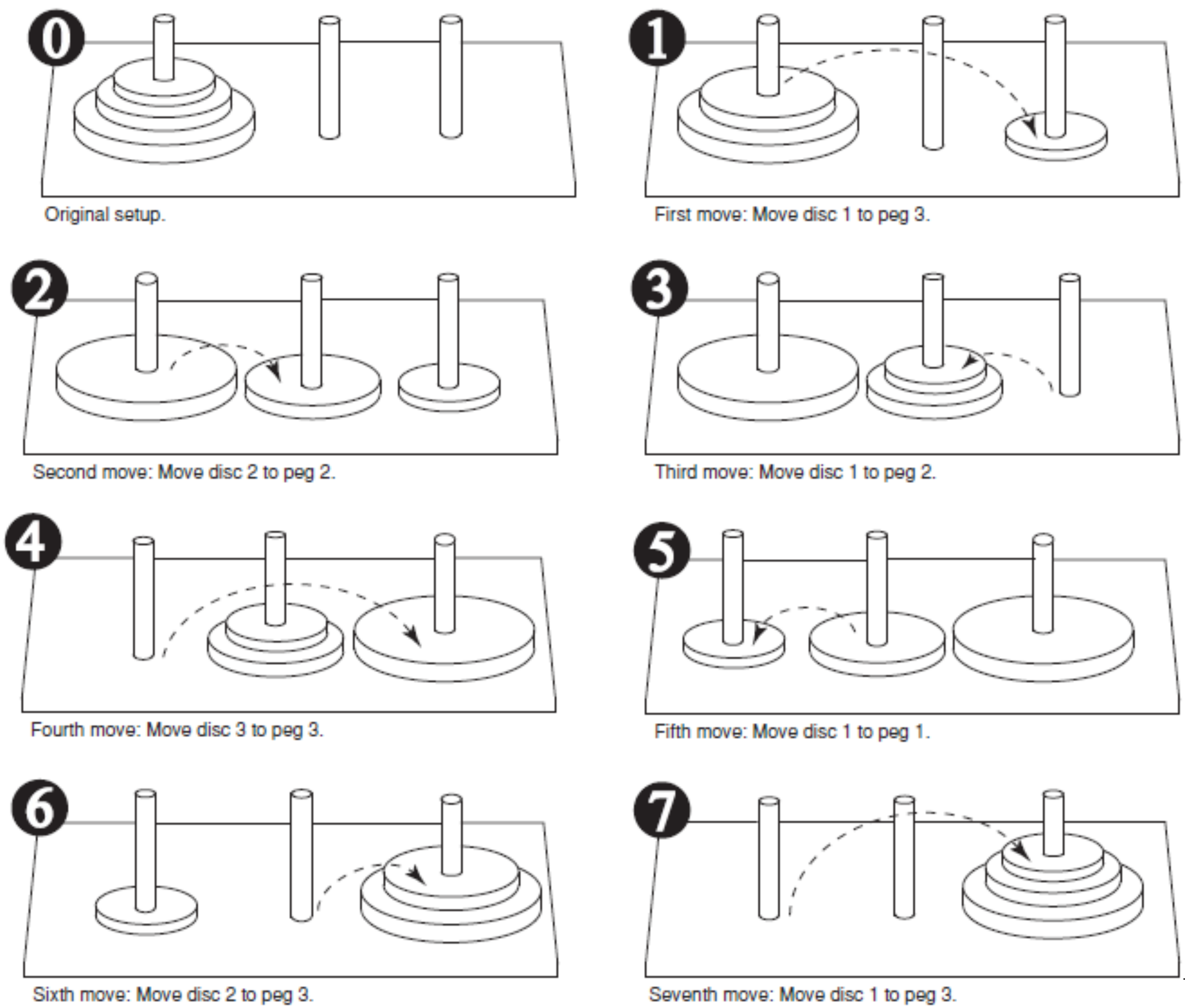


Figure 12-6 Steps for moving three pegs



The Towers of Hanoi (cont'd)

- Problem statement: move n discs from peg 1 to peg 3 using peg 2 as a temporary peg
- Recursive solution:
  - If n == 1: Move disc from peg 1 to peg 3
  - Otherwise:
    - Move n-1 discs from peg 1 to peg 2, using peg 3
    - Move remaining disc from peg 1 to peg 3
    - Move n-1 discs from peg 2 to peg 3, using peg 1

The Towers of Hanoi (cont'd.)

```
# The moveDiscs function displays a disc move in
# the Towers of Hanoi game.
# The parameters are:
#     num:        The number of discs to move.
#     from_peg:   The peg to move from.
#     to_peg:     The peg to move to.
#     temp_peg:   The temporary peg.
def move_discs(num, from_peg, to_peg, temp_peg):
    if num > 0:
        move_discs(num - 1, from_peg, temp_peg, to_peg)
        print('Move a disc from peg', from_peg, 'to peg', to_peg)
        move_discs(num - 1, temp_peg, to_peg, from_peg)
```

Recursion versus Looping

- Reasons not to use recursion:
  - Less efficient: entails function calling overhead that is not necessary with a loop
  - Usually a solution using a loop is more evident than a recursive solution
- Some problems are more easily solved with recursion than with a loop
  - Example: Fibonacci, where the mathematical definition lends itself to recursion

Practice

- def recur_sum(low, high) to print the summary of a range of integers in [low, high] using recursion.
  - hint: a + b + c = a + (b + c)
  - base case: sum = low if high = low
  - recursive case: sum = high + recur_sum(low, high - 1)

Practice

- def num_digits(num) to print how many digits a positive integer has. For example 123456 -> 6, 7654321 ->7, 1 -> 1
  - hint: num//10 will reduce the digit by 1
  - base case: num < 10, return 1
  - recursive case: return 1 + num_digits(num//10)

Challenges

- Find all possible permutations of a String: For example,
  - Given a string "sky" , our function should print
    output : "ysk","ksy","yks","kys","syk" and “sky”
  - Given a string "ooo" , our function should print output : print six times "ooo"

Challenges

- Implement Merge Sort on an integer array
- Implement Quick sort on an integer array

Challenges

- Maximum Subarray: Given an integer array nums, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.
  - Example:
    - **Input:** [-2,1,-3,4,-1,2,1,-5,4],
    - Output: 6
    - **Explanation:** [4,-1,2,1] has the largest sum = 6.

Thank you!