CHAPTER 12 Recursion

TOPICS

12.1 Introduction to Recursion

CONCEPT: A recursive function is a function that calls itself.

You have seen instances of functions calling other functions. In a program, the `main` function might call function `A`, which then might call function `B`. It's also possible for a function to call itself. A function that calls itself is known as a *recursive function*. For example, look at the `message` function shown in Program 12-1.

Program 12-1 (endless_recursion.py)

```
# This program has a recursive function.

def main():
    message()

def message():
    print('This is a recursive function.')
    message()

# Call the main function.
main()
```

(program output continues)

Program Output *(continued)*

```
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
```

. . . and this output repeats forever!

The `message` function displays the string 'This is a recursive function' and then calls itself. Each time it calls itself, the cycle is repeated. Can you see a problem with the function? There's no way to stop the recursive calls. This function is like an infinite loop because there is no code to stop it from repeating. If you run this program, you will have to press Ctrl+C on the keyboard to interrupt its execution.

Like a loop, a recursive function must have some way to control the number of times it repeats. The code in Program 12-2 shows a modified version of the `message` function. In this program, the `message` function receives an argument that specifies the number of times the function should display the message.

Program 12-2 **(`recursive.py`)**

```
 1  # This program has a recursive function.
 2
 3  def main():
 4      # By passing the argument 5 to the message
 5      # function we are telling it to display the
 6      # message five times.
 7      message(5)
 8
 9  def message(times):
10      if times > 0:
11          print('This is a recursive function.')
12          message(times - 1)
13
14  # Call the main function.
15  main()
```

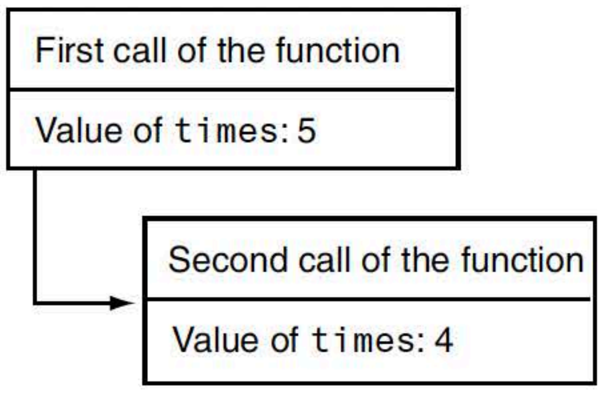
Program Output

```
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
```

The `message` function in this program contains an `if` statement in line 10 that controls the repetition. As long as the `times` parameter is greater than zero, the message 'This is a recursive function' is displayed, and then the function calls itself again, but with a smaller argument.

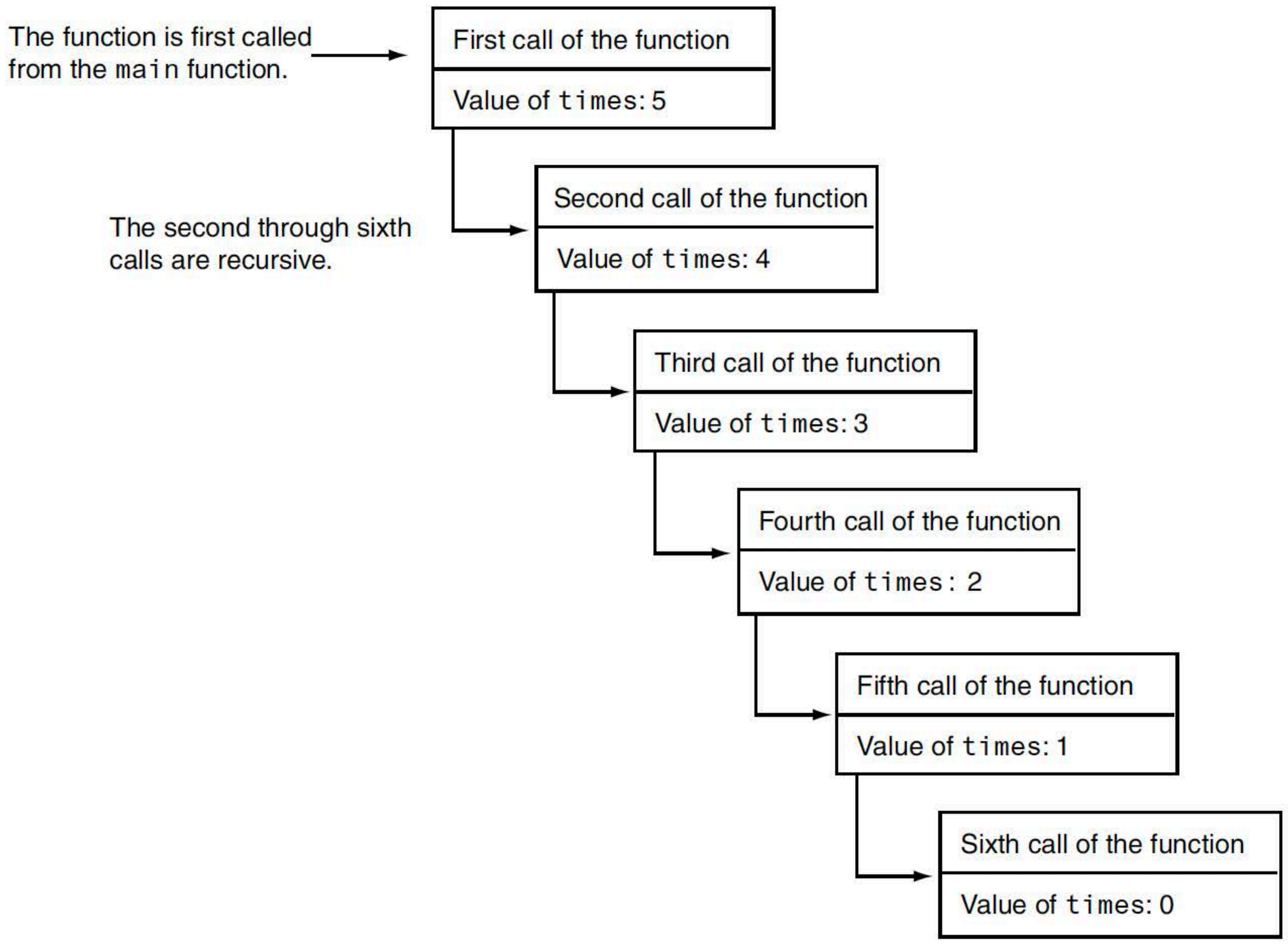
In line 7 the `main` function calls the `message` function passing the argument 5. The first time the function is called the `if` statement displays the message and then calls itself with 4 as the argument. Figure 12-1 illustrates this.

Figure 12-1 First two calls of the function



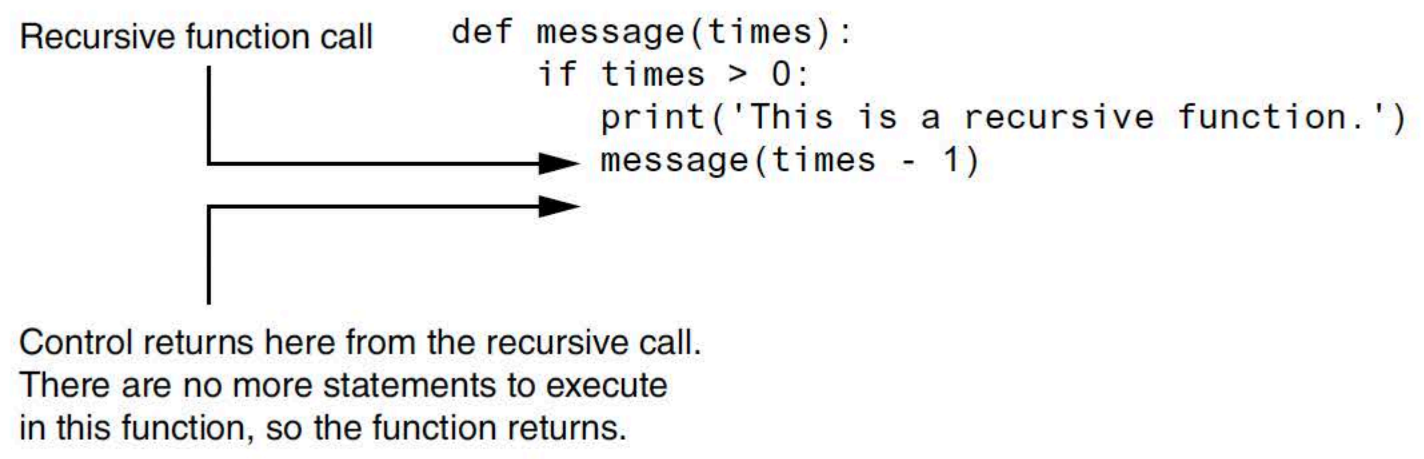
The diagram shown in Figure 12-1 illustrates two separate calls of the `message` function. Each time the function is called, a new instance of the `times` parameter is created in memory. The first time the function is called, the `times` parameter is set to 5. When the function calls itself, a new instance of the `times` parameter is created, and the value 4 is passed into it. This cycle repeats until finally, zero is passed as an argument to the function. This is illustrated in Figure 12-2.

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


The function is first called from the main function.
The second through sixth calls are recursive.
First call of the function
Value of times: 5
Second call of the function
Value of times: 4
Third call of the function
Value of times: 3
Fourth call of the function
Value of times: 2
Fifth call of the function
Value of times: 1
Sixth call of the function
Value of times: 0

As you can see in the figure, the function is called six times. The first time it is called from the `main` function, and the other five times it calls itself. The number of times that a function calls itself is known as the *depth of recursion*. In this example, the depth of recursion is five. When the function reaches its sixth call, the `times` parameter is set to 0. At that point, the `if` statement's conditional expression is false, so the function returns. Control of the program returns from the sixth instance of the function to the point in the fifth instance directly after the recursive function call. This is illustrated in Figure 12-3.

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



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

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

Because there are no more statements to be executed after the function call, the fifth instance of the function returns control of the program back to the fourth instance. This repeats until all instances of the function return.

12.2 Problem Solving with Recursion

CONCEPT: A problem can be solved with recursion if it can be broken down into smaller problems that are identical in structure to the overall problem.

The code shown in Program 12-2 demonstrates the mechanics of a recursive function. Recursion can be a powerful tool for solving repetitive problems, and is commonly studied in upper-level computer science courses. It may not yet be clear to you how to use recursion to solve a problem.

First, note recursion is never required to solve a problem. Any problem that can be solved recursively can also be solved with a loop. In fact, recursive algorithms are usually less efficient than iterative algorithms. This is because the process of calling a function requires several actions to be performed by the computer. These actions include allocating memory for parameters and local variables, and storing the address of the program location where control returns after the function terminates. These actions, which are sometimes referred to as *overhead*, take place with each function call. Such overhead is not necessary with a loop.

Some repetitive problems, however, are more easily solved with recursion than with a loop. Where a loop might result in faster execution time, the programmer might be able to design a recursive algorithm faster. In general, a recursive function works as follows:

- If the problem can be solved now, without recursion, then the function solves it and returns
- If the problem cannot be solved now, then the function reduces it to a smaller but similar problem and calls itself to solve the smaller problem

In order to apply this approach, first, we identify at least one case in which the problem can be solved without recursion. This is known as the *base case*. Second, we determine a way to solve the problem in all other circumstances using recursion. This is called the *recursive case*. In the recursive case, we must always reduce the problem to a smaller version of the original problem. By reducing the problem with each recursive call, the base case will eventually be reached and the recursion will stop.

Using Recursion to Calculate the Factorial of a Number

Let's take an example from mathematics to examine an application of recursive functions. In mathematics, the notation $n!$ represents the factorial of the number n. The factorial of a nonnegative number can be defined by the following rules:

If $n = 0$ then $n! = 1$
If $n > 0$ then $n! = 1 \times 2 \times 3 \times \ldots \times n$

Let's replace the notation $n!$ with factorial(n), which looks a bit more like computer code, and rewrite these rules as follows:

If $n = 0$ then factorial(n) $= 1$
If $n > 0$ then factorial(n) $= 1 \times 2 \times 3 \times \ldots \times n$

These rules state that when n is 0, its factorial is 1. When n is greater than 0, its factorial is the product of all the positive integers from 1 up to n. For instance, factorial(6) is calculated as $1 \times 2 \times 3 \times 4 \times 5 \times 6$.

When designing a recursive algorithm to calculate the factorial of any number, first we identify the base case, which is the part of the calculation that we can solve without recursion. That is the case where n is equal to 0 as follows:

If $n = 0$ then factorial(n) $= 1$

This tells us how to solve the problem when n is equal to 0, but what do we do when n is greater than 0? That is the recursive case, or the part of the problem that we use recursion to solve. This is how we express it:

If $n > 0$ then factorial(n) $= n \times$ factorial($n - 1$)

This states that if n is greater than 0, the factorial of n is n times the factorial of $n - 1$. Notice how the recursive call works on a reduced version of the problem, $n - 1$. So, our recursive rule for calculating the factorial of a number might look like this:

If $n = 0$ then factorial(n) $= 1$
If $n > 0$ then factorial(n) $= n \times$ factorial($n - 1$)

The code in Program 12-3 shows how we might design a factorial function in a program.

Program 12-3

```python
# This program uses recursion to calculate
# the factorial of a number.

def main():
    # Get a number from the user.
    number = int(input('Enter a nonnegative integer: '))

    # Get the factorial of the number.
    fact = factorial(number)

    # Display the factorial.
    print('The factorial of', number, 'is', fact)

# 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)

# Call the main function.
main()
```

Program Output (with input shown in bold)

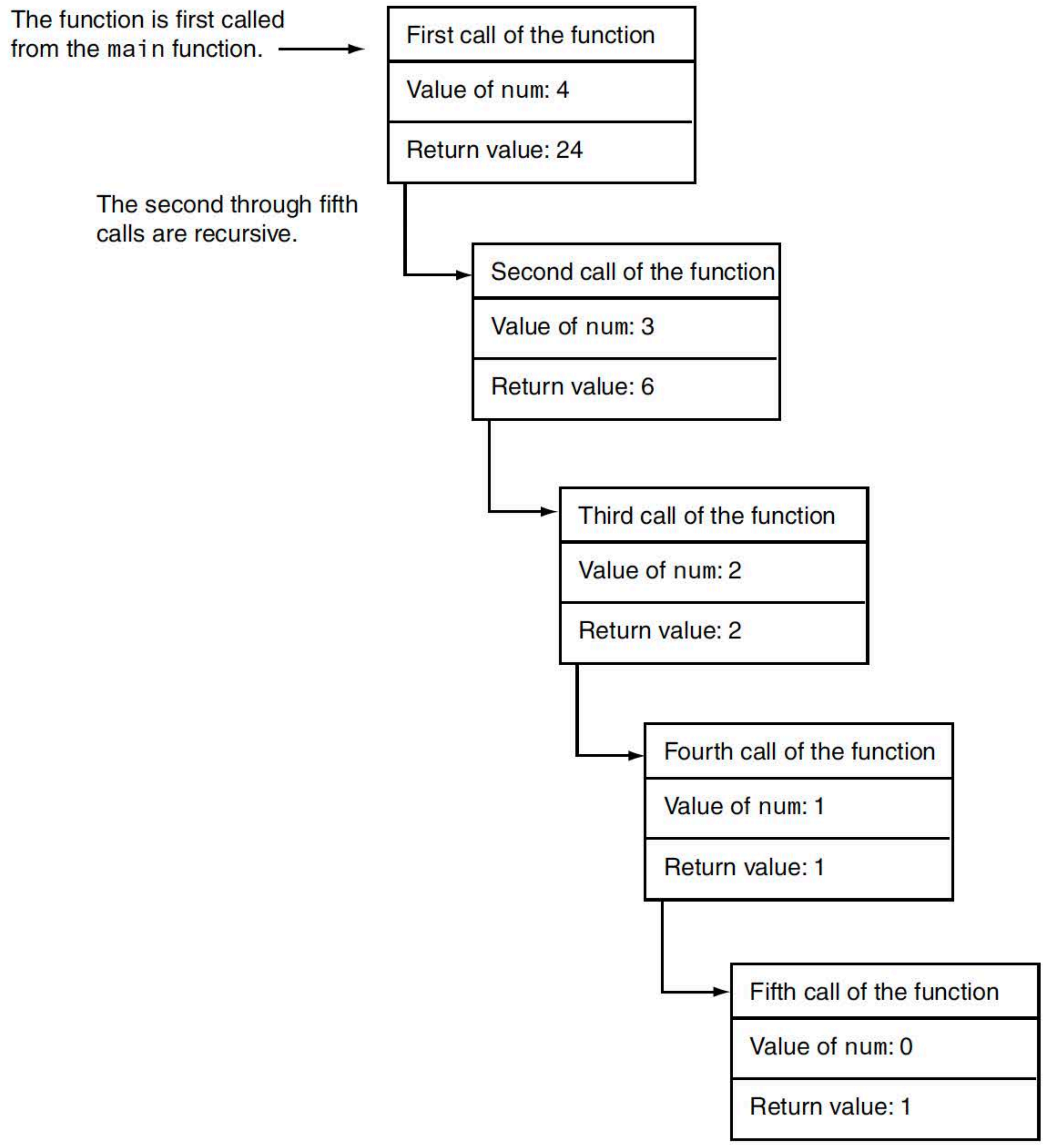
```
Enter a nonnegative integer: 4 Enter
The factorial of 4 is 24
```

In the sample run of the program, the `factorial` function is called with the argument 4 passed to `num`. Because `num` is not equal to 0, the `if` statement's `else` clause executes the following statement:

```
return num * factorial(num - 1)
```

Although this is a `return` statement, it does not immediately return. Before the return value can be determined, the value of `factorial(num – 1)` must be determined. The `factorial` function is called recursively until the fifth call, in which the `num` parameter will be set to zero. Figure 12-4 illustrates the value of `num` and the return value during each call of the function.

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


The function is first called from the main function.
First call of the function
Value of num: 4
Return value: 24
The second through fifth calls are recursive.
Second call of the function
Value of num: 3
Return value: 6
Third call of the function
Value of num: 2
Return value: 2
Fourth call of the function
Value of num: 1
Return value: 1
Fifth call of the function
Value of num: 0
Return value: 1

The figure illustrates why a recursive algorithm must reduce the problem with each recursive call. Eventually, the recursion has to stop in order for a solution to be reached.

If each recursive call works on a smaller version of the problem, then the recursive calls work toward the base case. The base case does not require recursion, so it stops the chain of recursive calls.

Usually, a problem is reduced by making the value of one or more parameters smaller with each recursive call. In our `factorial` function, the value of the parameter `num` gets closer to 0 with each recursive call. When the parameter reaches 0, the function returns a value without making another recursive call.

Direct and Indirect Recursion

The examples we have discussed so far show recursive functions or functions that directly call themselves. This is known as *direct recursion*. There is also the possibility of creating indirect recursion in a program. This occurs when function `A` calls function `B`, which in turn calls function `A`. There can even be several functions involved in the recursion. For example, function `A` could call function `B`, which could call function `C`, which calls function `A`.

Checkpoint

12.1 It is said that a recursive algorithm has more overhead than an iterative algorithm. What does this mean?

12.2 What is a base case?

12.3 What is a recursive case?

12.4 What causes a recursive algorithm to stop calling itself?

12.5 What is direct recursion? What is indirect recursion?

12.3 Examples of Recursive Algorithms

Summing a Range of List Elements with Recursion

In this example, we look at a function named `range_sum` that uses recursion to sum a range of items in a list. The function takes the following arguments: a list that contains the range of elements to be summed, an integer specifying the index of the starting item in the range, and an integer specifying the index of the ending item in the range. Here is an example of how the function might be used:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
my_sum = range_sum(numbers, 3, 7)
print(my_sum)
```

The second statement in this code specifies that the `range_sum` function should return the sum of the items at indexes 3 through 7 in the `numbers` list. The return value, which in this case would be 30, is assigned to the `my_sum` variable. Here is the definition of the `range_sum` function:

```
def range_sum(num_list, start, end):
    if start > end:
        return 0
    else:
        return num_list[start] + range_sum(num_list, start + 1, end)
```

This function's base case is when the `start` parameter is greater than the `end` parameter. If this is true, the function returns the value 0. Otherwise, the function executes the following statement:

```
return num_list[start] + range_sum(num_list, start + 1, end)
```

This statement returns the sum of num_list[start] plus the return value of a recursive call. Notice in the recursive call, the starting item in the range is start + 1. In essence, this statement says "return the value of the first item in the range plus the sum of the rest of the items in the range." Program 12-4 demonstrates the function.

Program 12-4

```
# This program demonstrates the range_sum function.

def main():
    # Create a list of numbers.
    numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

    # Get the sum of the items at indexes 2
    # through 5.
    my_sum = range_sum(numbers, 2, 5)

    # Display the sum.
    print('The sum of items 2 through 5 is', my_sum)

# The range_sum function returns the sum of a specified
# range of items in num_list. The start parameter
# specifies the index of the starting item. The end
# parameter specifies the index of the ending item.
def range_sum(num_list, start, end):
    if start > end:
        return 0
    else:
        return num_list[start] + range_sum(num_list, start + 1, end)

# Call the main function.
main()
```

Program Output

```
The sum of elements 2 through 5 is 18
```

The Fibonacci Series

Some mathematical problems are designed to be solved recursively. One well-known example is the calculation of Fibonacci numbers. The Fibonacci numbers, named after the Italian mathematician Leonardo Fibonacci (born circa 1170), are the following sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, . . .

Notice after the second number, each number in the series is the sum of the two previous numbers. The Fibonacci series can be defined as follows:

If $n = 0$ then $\quad \text{Fib}(n) = 0$
If $n = 1$ then $\quad \text{Fib}(n) = 1$
If $n > 1$ then $\quad \text{Fib}(n) = \text{Fib}(n - 1) + \text{Fib}(n - 2)$

A recursive function to calculate the *n*th number in the Fibonacci series is shown here:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

Notice this function actually has two base cases: when *n* is equal to 0, and when *n* is equal to 1. In either case, the function returns a value without making a recursive call. The code in Program 12-5 demonstrates this function by displaying the first 10 numbers in the Fibonacci series.

Program 12-5 **(fibonacci.py)**

```
# This program uses recursion to print numbers
# from the Fibonacci series.

def main():
    print('The first 10 numbers in the')
    print('Fibonacci series are:')

    for number in range(1, 11):
        print(fib(number))

# The fib function returns the nth number
# in the Fibonacci series.
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)

# Call the main function.
main()
```

Program Output

```
The first 10 numbers in the
Fibonacci series are:
1
```

```
1
2
3
5
8
13
21
34
55
```

Finding the Greatest Common Divisor

Our next example of recursion is the calculation of the greatest common divisor (GCD) of two numbers. The GCD of two positive integers *x* and *y* is determined as follows:

If *x* can be evenly divided by *y*, then gcd(*x*, *y*) = *y*
Otherwise, gcd(*x*, *y*) = gcd(*y*, remainder of *x*/*y*)

This definition states that the GCD of *x* and *y* is *y* if *x*/*y* has no remainder. This is the base case. Otherwise, the answer is the GCD of *y* and the remainder of *x*/*y*. The code in Program 12-6 shows a recursive method for calculating the GCD.

Program 12-6 (gcd.py)

```python
# This program uses recursion to find the GCD
# of two numbers.

def main():
    # Get two numbers.
    num1 = int(input('Enter an integer: '))
    num2 = int(input('Enter another integer: '))

    # Display the GCD.
    print('The greatest common divisor of')
    print('the two numbers is', gcd(num1, num2))

# 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)

# Call the main function.
main()
```

(program output continues)

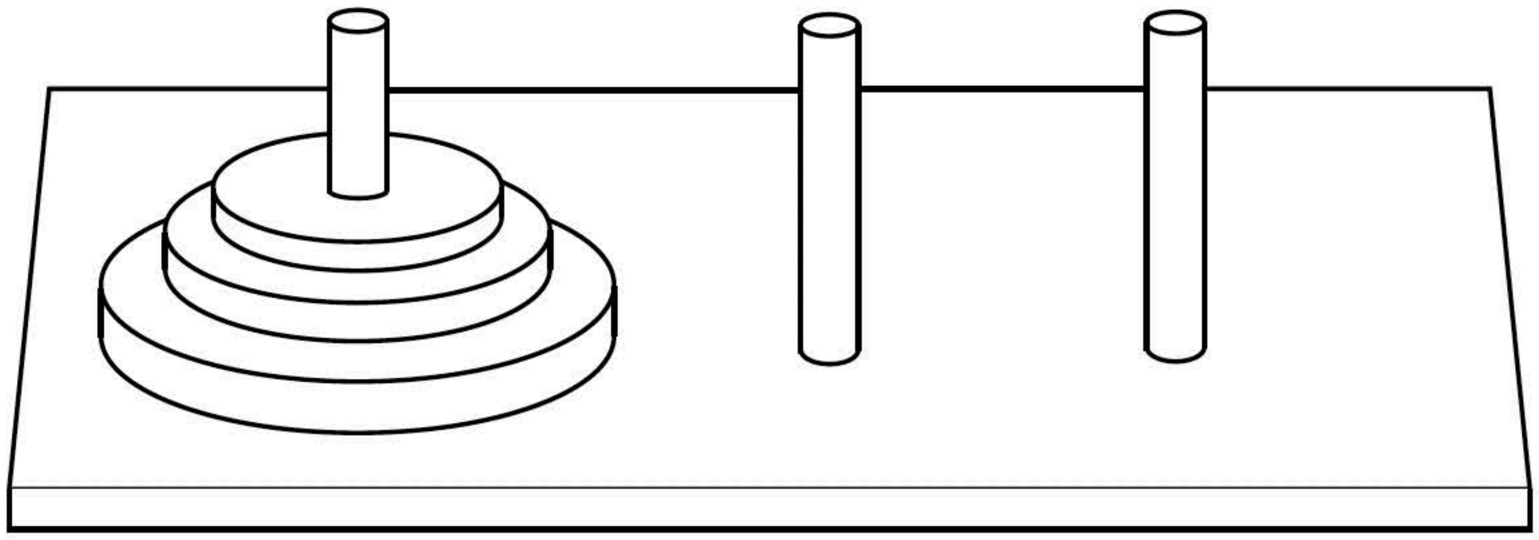
Program Output (with input shown in bold)

```
Enter an integer: 49 [Enter]
Enter another integer: 28 [Enter]
The greatest common divisor of
these two numbers is 7
```

The Towers of Hanoi

The Towers of Hanoi is a mathematical game that is often used in computer science to illustrate the power of recursion. The game uses three pegs and a set of discs with holes through their centers. The discs are stacked on one of the pegs as shown in Figure 12-5.

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



Notice the discs are stacked on the leftmost peg, in order of size with the largest disc at the bottom. The game is based on a legend where a group of monks in a temple in Hanoi have a similar set of pegs with 64 discs. The job of the monks is to move the discs from the first peg to the third peg. The middle peg can be used as a temporary holder. Furthermore, the monks must follow these rules while moving the discs:

- Only one disk may be moved at a time
- A disk cannot be placed on top of a smaller disc
- All discs must be stored on a peg except while being moved

According to the legend, when the monks have moved all of the discs from the first peg to the last peg, the world will come to an end.[1]

To play the game, you must move all of the discs from the first peg to the third peg, following the same rules as the monks. Let's look at some example solutions to this game, for different numbers of discs. If you only have one disc, the solution to the game is

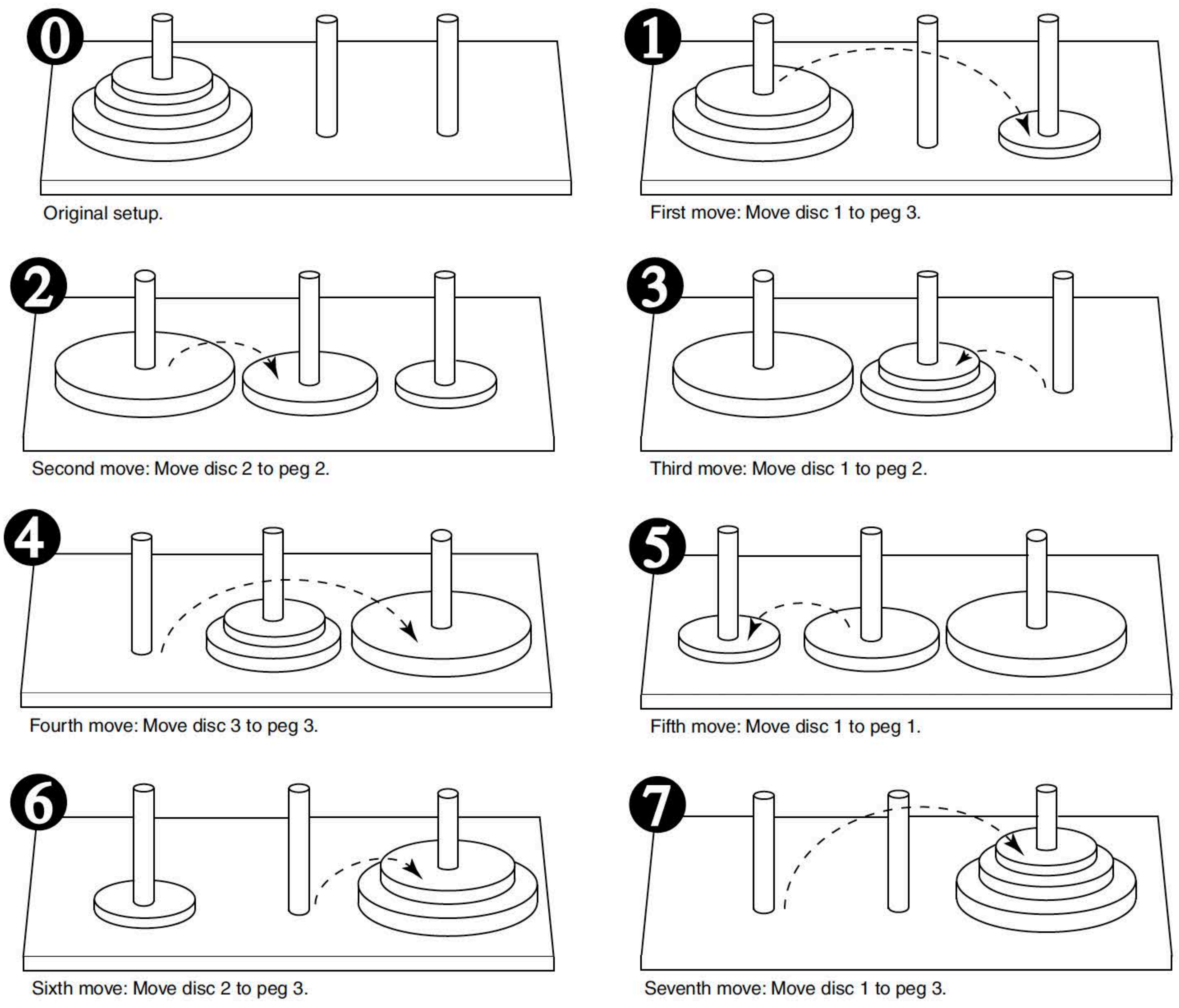
[1]In case you're worried about the monks finishing their job and causing the world to end anytime soon, you can relax. If the monks move the discs at a rate of 1 per second, it will take them approximately 585 billion years to move all 64 discs!

simple: move the disc from peg 1 to peg 3. If you have two discs, the solution requires three moves:

- Move disc 1 to peg 2
- Move disc 2 to peg 3
- Move disc 1 to peg 3

Notice this approach uses peg 2 as a temporary location. The complexity of the moves continues to increase as the number of discs increases. To move three discs requires the seven moves shown in Figure 12-6.

Figure 12-6 Steps for moving three pegs


0
Original setup.
1
First move: Move disc 1 to peg 3.
2
Second move: Move disc 2 to peg 2.
3
Third move: Move disc 1 to peg 2.
4
Fourth move: Move disc 3 to peg 3.
5
Fifth move: Move disc 1 to peg 1.
6
Sixth move: Move disc 2 to peg 3.
7
Seventh move: Move disc 1 to peg 3.

The following statement describes the overall solution to the problem:

Move n discs from peg 1 to peg 3 using peg 2 as a temporary peg.

The following summary describes a recursive algorithm that simulates the solution to the game. Notice in this algorithm, we use the variables A, B, and C to hold peg numbers.

```
To move n discs from peg A to peg C, using peg B as a temporary peg, do the following:
If n > 0:
    Move n − 1 discs from peg A to peg B, using peg C as a temporary peg.
    Move the remaining disc from peg A to peg C.
    Move n − 1 discs from peg B to peg C, using peg A as a temporary peg.
```

The base case for the algorithm is reached when there are no more discs to move. The following code is for a function that implements this algorithm. Note the function does not actually move anything, but displays instructions indicating all of the disc moves to make.

```
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)
```

This function accepts arguments into the following parameters:

`num`	The number of discs to move.
`from_peg`	The peg to move the discs from.
`to_peg`	The peg to move the discs to.
`temp_peg`	The peg to use as a temporary peg.

If num is greater than 0, then there are discs to move. The first recursive call is as follows:

```
move_discs(num - 1, from_peg, temp_peg, to_peg)
```

This statement is an instruction to move all but one disc from `from_peg` to `temp_peg`, using `to_peg` as a temporary peg. The next statement is as follows:

```
print('Move a disc from peg', from_peg, 'to peg', to_peg)
```

This simply displays a message indicating that a disc should be moved from `from_peg` to `to_peg`. Next, another recursive call is executed as follows:

```
move-discs(num - 1, temp_peg, to_peg, from_peg)
```

This statement is an instruction to move all but one disc from `temp_peg` to `to_peg`, using `from_peg` as a temporary peg. The code in Program 12-7 demonstrates the function by displaying a solution for the Tower of Hanoi game.

Program 12-7 **(towers_of_hanoi.py)**

```
# This program simulates the Towers of Hanoi game.

def main():
    # Set up some initial values.
    num_discs = 3
    from_peg = 1
    to_peg = 3
    temp_peg = 2

    # Play the game.
    move_discs(num_discs, from_peg, to_peg, temp_peg)
    print('All the pegs are moved!')

# 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)

# Call the main function.
main()
```

Program Output

```
Move a disc from peg 1 to peg 3
Move a disc from peg 1 to peg 2
Move a disc from peg 3 to peg 2
Move a disc from peg 1 to peg 3
Move a disc from peg 2 to peg 1
Move a disc from peg 2 to peg 3
Move a disc from peg 1 to peg 3
All the pegs are moved!
```

Recursion versus Looping

Any algorithm that can be coded with recursion can also be coded with a loop. Both approaches achieve repetition, but which is best to use?

There are several reasons not to use recursion. Recursive function calls are certainly less efficient than loops. Each time a function is called, the system incurs overhead that is not necessary with a loop. Also, in many cases, a solution using a loop is more evident than a recursive solution. In fact, the majority of repetitive programming tasks are best done with loops.

Some problems, however, are more easily solved with recursion than with a loop. For example, the mathematical definition of the GCD formula is well suited to a recursive approach. If a recursive solution is evident for a particular problem, and the recursive algorithm does not slow system performance an intolerable amount, then recursion would be a good design choice. If a problem is more easily solved with a loop, however, you should take that approach.

Review Questions

Multiple Choice

1. A recursive function _______________.
   a. calls a different function
   b. abnormally halts the program
   c. calls itself
   d. can only be called once
2. A function is called once from a program's `main` function, then it calls itself four times. The depth of recursion is _______________.
   a. one
   b. four
   c. five
   d. nine
3. The part of a problem that can be solved without recursion is the _______________ case.
   a. base
   b. solvable
   c. known
   d. iterative
4. The part of a problem that is solved with recursion is the _______________ case.
   a. base
   b. iterative
   c. unknown
   d. recursion
5. When a function explicitly calls itself, it is called _______________ recursion.
   a. explicit
   b. modal
   c. direct
   d. indirect
6. When function A calls function B, which calls function A, it is called _______________ recursion.
   a. implicit
   b. modal
   c. direct
   d. indirect
7. Any problem that can be solved recursively can also be solved with a _______________.
   a. decision structure
   b. loop
   c. sequence structure
   d. case structure
8. Actions taken by the computer when a function is called, such as allocating memory for parameters and local variables, are referred to as _______________.
   a. overhead
   b. set up
   c. clean up
   d. synchronization

9. A recursive algorithm must _______________ in the recursive case.
    a. solve the problem without recursion
    b. reduce the problem to a smaller version of the original problem
    c. acknowledge that an error has occurred and abort the program
    d. enlarge the problem to a larger version of the original problem
10. A recursive algorithm must _______________ in the base case.
    a. solve the problem without recursion
    b. reduce the problem to a smaller version of the original problem
    c. acknowledge that an error has occurred and abort the program
    d. enlarge the problem to a larger version of the original problem

True or False

1. An algorithm that uses a loop will usually run faster than an equivalent recursive algorithm.
2. Some problems can be solved through recursion only.
3. It is not necessary to have a base case in all recursive algorithms.
4. In the base case, a recursive method calls itself with a smaller version of the original problem.

Short Answer