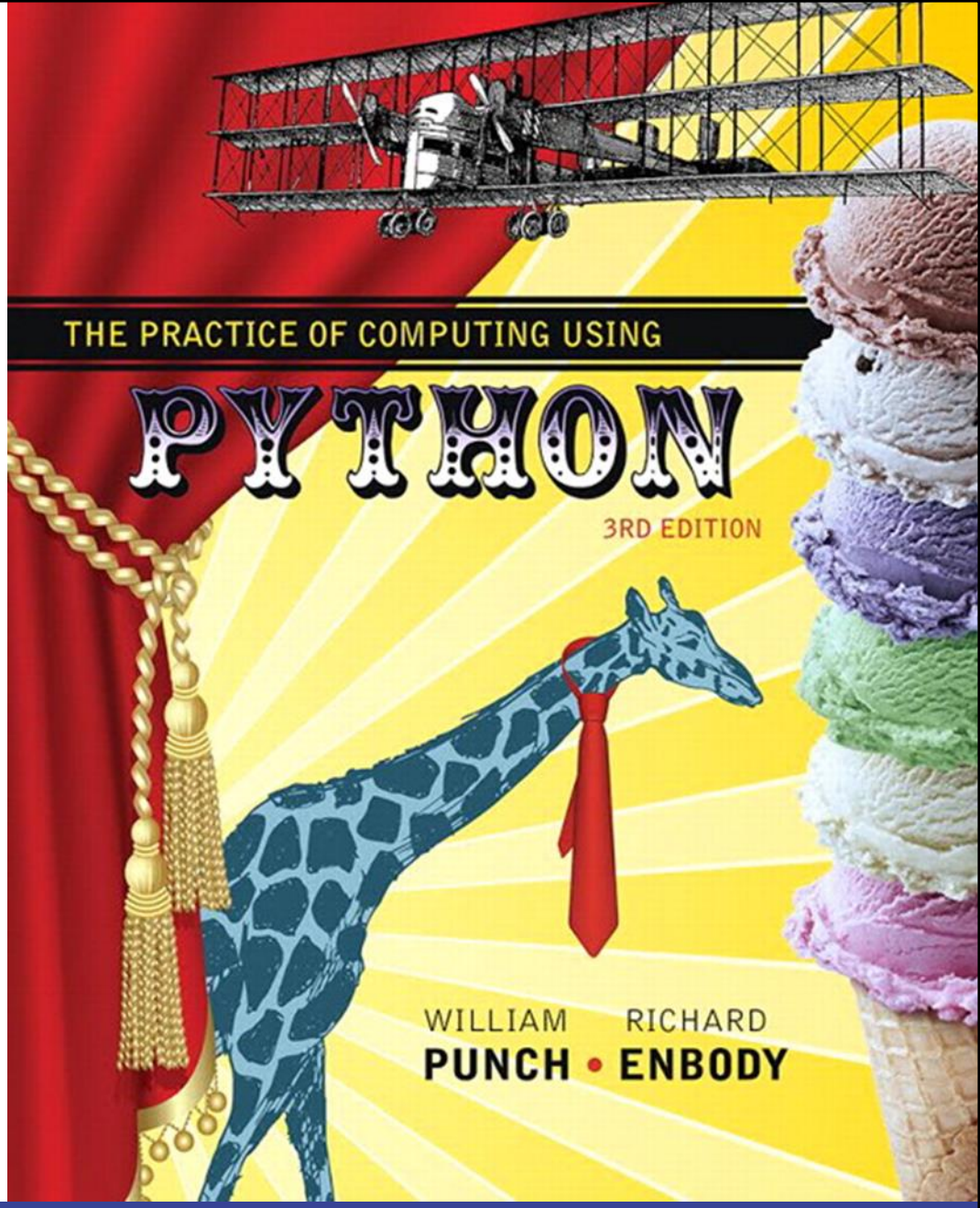
chapter 15

Recursion: Another Control Mechanism



Recursive Function

- a recursive function is a ***function that calls itself***
- leads to some funny definitions
  - def: recursion. see recursion
- when you first see it, it looks odd

```
def factorial(n):
    """Recursive factorial."""
    if n == 1:
        return 1                          # base case
    else:
        return n * factorial(n-1)    # recursive case
```

It Doesn’t Do Anything!

- this is a common complaint when one first sees a recursive function
  - what exactly is it doing?
  - it doesn’t seem to do anything!
- our goal is to understand what it means to write a recursive function from a programmer’s and computer’s view

Defining a Recursive Function

Divide and Conquer

- recursion is a natural outcome of a divide and conquer approach to problem solving
- a recursive function defines how to break a problem down (divide) and how to reassemble (conquer) the sub-solutions into an overall solution

Base Case

- recursion is a process not unlike loop iteration
  - you must define how long (how many iterations) recursion will proceed through until it stops
- the base case defines this limit
- without the base case, recursion will continue infinitely (just like an infinite loop)

Simple Example

Lao-Tzu: “A journey of 1000 miles begins with a single step.”

def journey (steps):

- the first step is easy (base case)
- the n^{th} step is easy having complete the previous n-1 steps (divide and conquer)

```
def take_step(n):
    if n == 1:  # base case
        return "Easy"
    else:
        this_step = "step(" + str(n) + ")"
        previous_steps = take_step(n-1)      # recursive call
        return this_step + " + " + previous_steps
```

Factorial

- factorial(4) = 4! = 4 * 3 * 2 * 1
- the result of the last step, multiply by 1, is defined based on all the previous results that have been calculated


Code Listing 15-2

```
def factorial(n):
    """Recursive factorial."""
    if n == 1:
        return 1                          # base case
    else:
        return n * factorial(n-1)    # recursive case
```

Trace the Recursive Calls

4! = 4 * 3!	start first function invocation
3! = 3 * 2!	start second function invocation
2! = 2 * 1!	start third function invocation
1! = 1	fourth invocation, base case
2! = 2 * 1 = 2	third invocation finishes
3! = 3 * 2 = 6	second invocation finishes
4! = 4 * 6 = 24	first invocation finishes

Fibonacci Sequence

- each Fibonacci number depends on the two previous Fibonacci results in the sequence

  0 1 1 2 3 5 8 13 21 …

- the base values are

  fibo(0) == 0

  fibo(1) == 1

- in general,

  fibo (x) = fibo(x-1) + fibo(x-2)


Code Listing 15-3

```
def fibonacci(n):
    """Recursive Fibonacci sequence."""
    if n == 0 or n == 1:      # base cases
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)  # recursive case
```

Trace

- fibo(4) = fibo(3) + fibo(2)
- fibo(3) = fibo(2) + fibo(1)
- fibo(2) = fibo(1) + fibo(0) = 2 # base case
- fibo(3) = 2 + fibo(1) = 3 # base case
- fibo(4) = 3 + 2 = 5

Reverse String

- we know Python has a very simple way to reverse a string, but let's see if we can write a recursive function that reverses a string without using slicing


Code Listing 15-4
Template for reversal

```
# recursive function to reverse a string

def reverser (a_str):
    # base case
    # recursive step
       # divide into parts
       # conquer/reassemble

the_str = input("Reverse what string:")
result = reverser(the_str)
print("The reverse of {} is {}".format(the_str,result))
```

Base Case

- what string do we know how to trivially reverse?
  - a string with one character, when reversed, gives back exactly the same string
- we use this as our base case


Code Listing 15-5

```
def reverser (aStr):
    if len(aStr) == 1:      # base case
        return aStr
    # recursive step
       # divide into parts
       # conquer/reassemble

theStr = raw_input("Reverse what string: ")
result = reverser(theStr)
print ("Reverse of %s is %s\n" % (theStr,result))
```

Recursive Step

- we must base the recursive step on what came before, plus the extra step we are presently in
  - thus, the reverse of a string is the reverse of all but the first character of the string, which is placed at the end
  - we assume the rev function will reverse the rest

```
rev(s) = rev(s[1:]) + s[0]
```


Code Listing 15-6

```
def reverse (a_str):
    """Recursive function to reverse a string."""
    print("Got as an argument:",a_str)
    # base case
    if len(a_str) == 1:
        print("Base Case!")
        return a_str
    # recursive step
    else:
        new_str =reverse(a_str[1:]) + a_str[0]
        print("Reassembling {} and {} into {}".\
          format(a_str[1:],a_str[0],new_str))

the_str = input("What string: ")
print()
result_str = reverse(the_str)
print("The reverse of {} is {}".format(the_str,result_str))
```

```
>python reverser.py
What string:abcde

Got as an argument: abcde
Got as an argument: bcde
Got as an argument: cde
Got as an argument: de
Got as an argument: e
Base Case!
Reassembling e and d into ed
Reassembling de and c into edc
Reassembling cde and b into edcb
Reassembling bcde and a into edcba
```

How Does Python Keep Track?

The Stack

- a ***stack*** is a data structure, like a list or a dictionary, but with a few different characteristics
- a stack is a sequence
- a stack only allows access to one end of its data, the ***top*** of the stack

Figure 15.1

Operations

- **`pop`**
  - remove top of stack
  - stack is one element smaller
- **`push(val)`**
  - add val to the stack
  - val is now the top
  - stack is one element larger
- **`top`**
  - reveals the top of the stack
  - no modification to stack

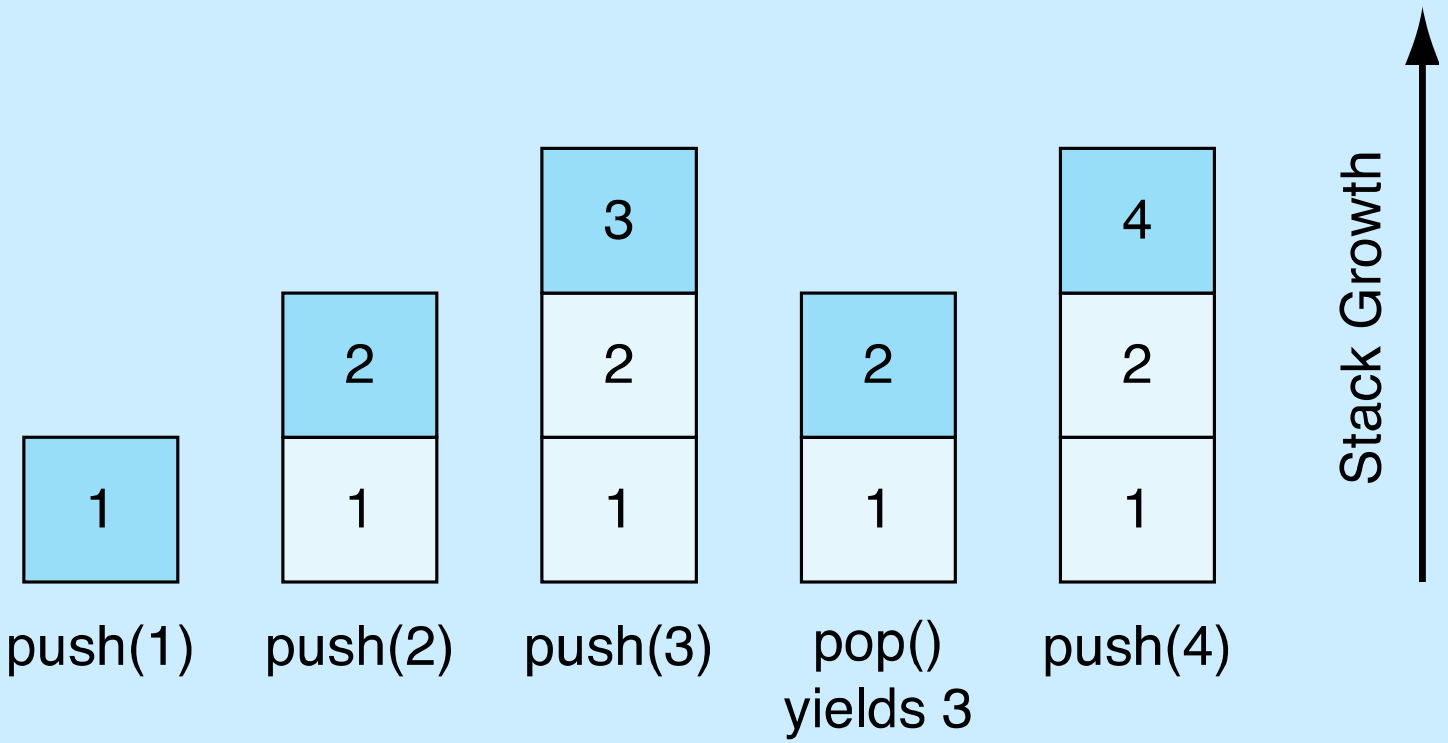


Figure 15.2 The operation of a stack data structure

Stack of Function Calls

- Python maintains a stack of function calls
- if a function calls another function, or itself recursively, the new function is pushed onto the calling stack and the previous function waits
- the top is always the active function
- when a pop occurs, the function below becomes active

```
def factorial(n):
    """recursive factorial with print to show operation."""
    indent = 4*(6-n)*" "  # more indent on deeper recursion
    print(indent + "Enter factorial n = ", n)
    if n == 1:                # base case
        print(indent + "Base case.")
        return 1
    else:                     # recursive case
        print(indent + "Before recursive call f(" + str(n-1) + ")")
        # separate recursive call allows print after call
        rest = factorial(n-1)
        print(indent + "After recursive call f(" + str(n-1) + ") = ", rest)
        return n * rest
```

```
>>> factorial(4)
        Enter factorial n =  4
        Before recursive call f(3)
            Enter factorial n =  3
            Before recursive call f(2)
                Enter factorial n =  2
                Before recursive call f(1)
                    Enter factorial n =  1
                    Base case.
                After recursive call f(1) =  1
            After recursive call f(2) =  2
        After recursive call f(3) =  6

24
```

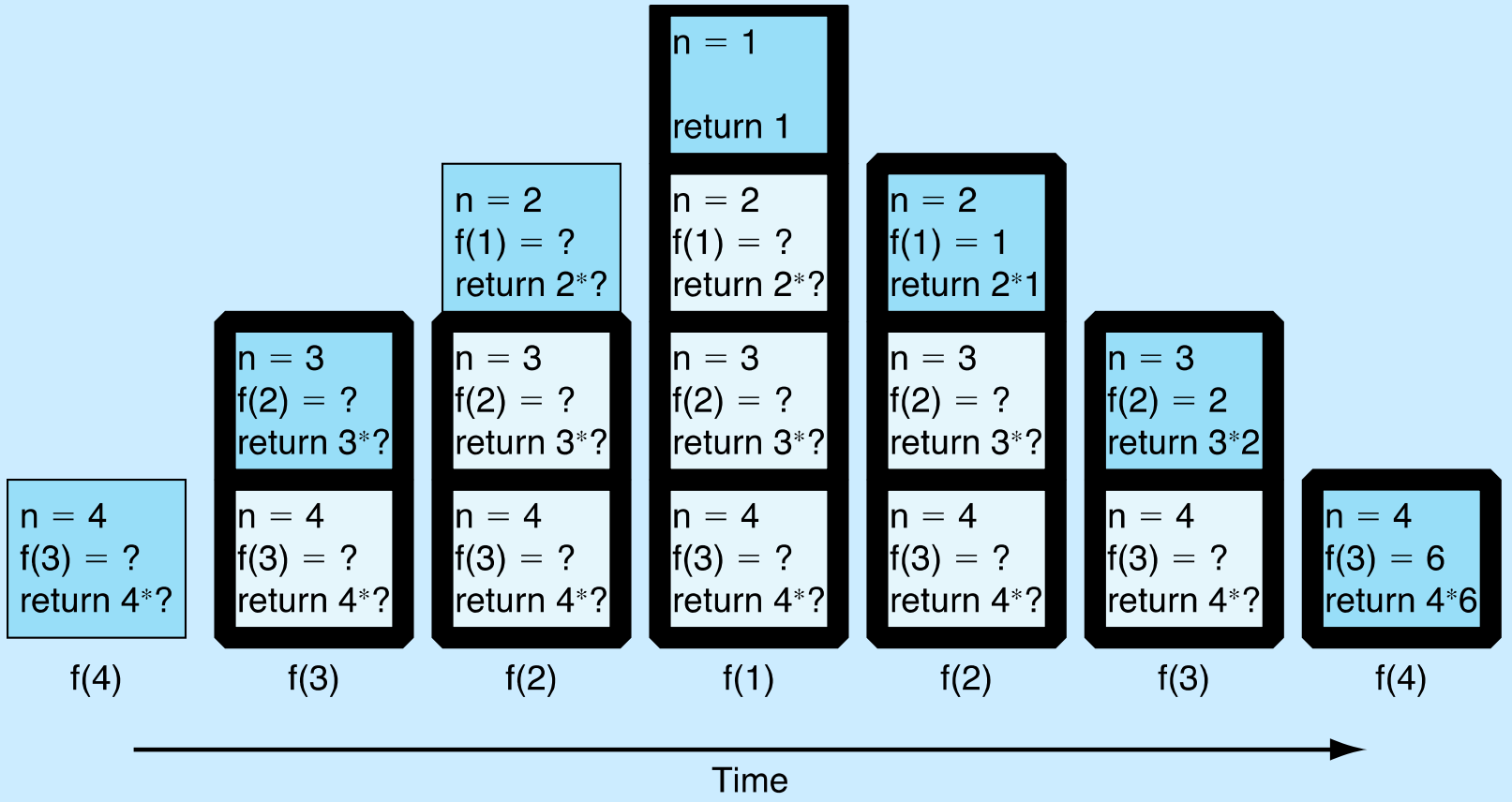
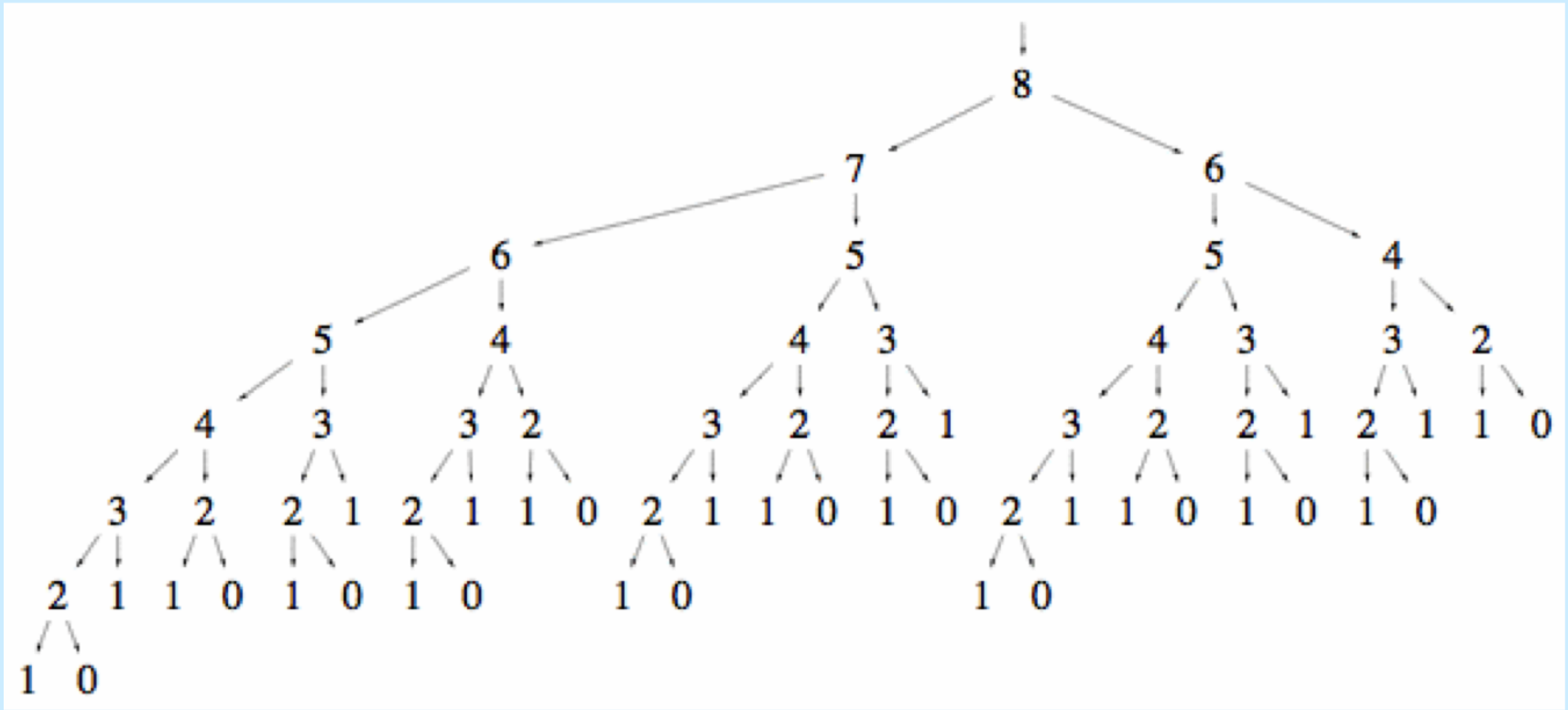
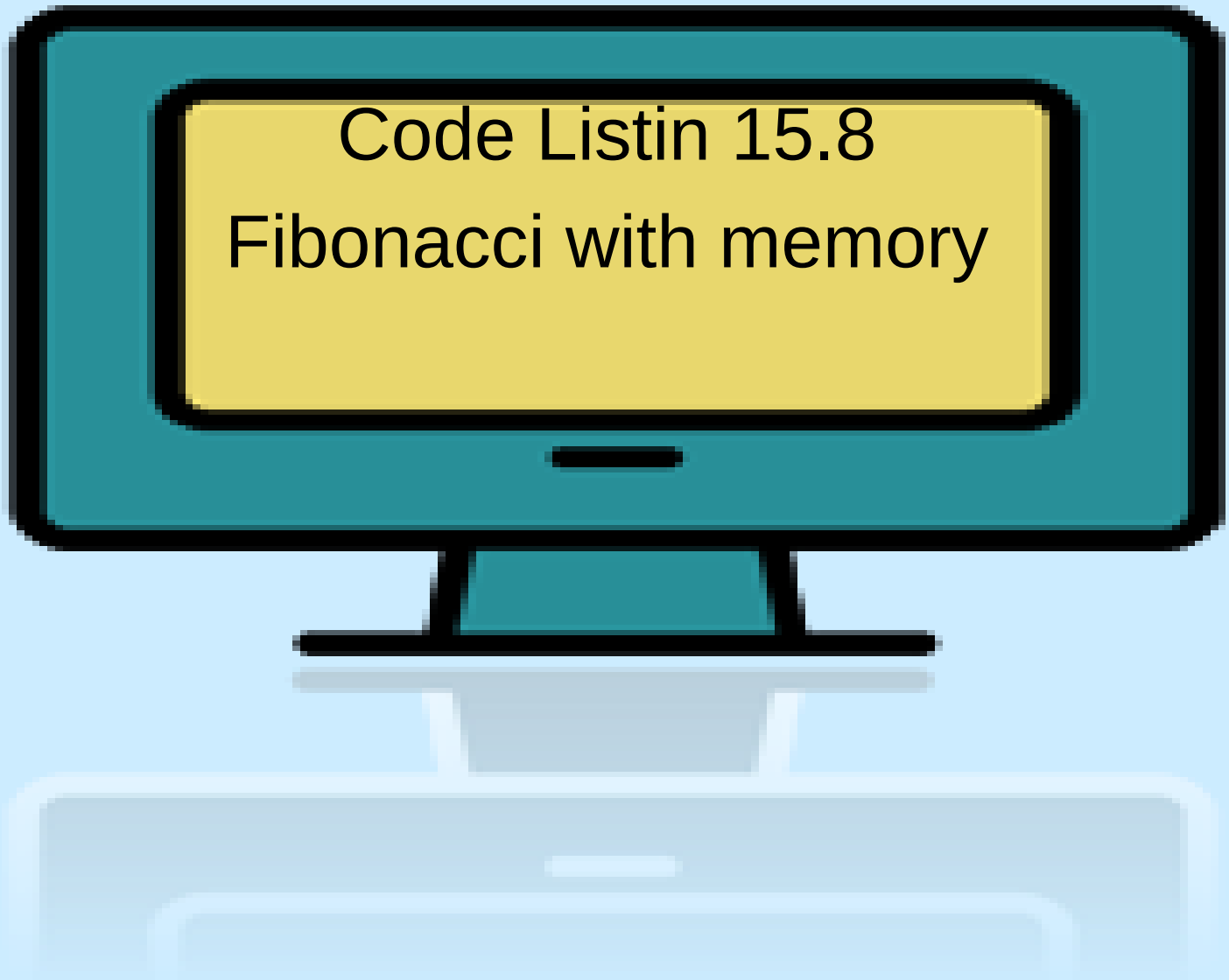


Figure 15.3 Call stack for factorial(4). Note the question marks.

A Better Fibonacci

- the recursive function that we have written previously is very wasteful
  - it calls the function with the same argument many times
  - never "remembers" the previous result




Code Listin 15.8
Fibonacci with memory

```
def fibonacci(n):
    """Recursive fibonacci that remembers previous values"""
    if n not in fibo_dict:
        # recursive case, store in the dict
        fibo_dict[n] = fibonacci(n-1) + fibonacci(n-2)
    return fibo_dict[n]

# global fibonacci dictionary.
fibo_dict = {}

# enter the base cases
fibo_dict[0] = 1
fibo_dict[1] = 1

fibo_val = input("Calculate what Fibonacci value:")
print("Fibonnaci value of",fibo_val,"is",
      fibonacci(int(fibo_val)))
```

Recursive Figures

Fractal Objects

- you can use recursion to draw many real world figures
- fractals are a class of drawing that has a couple of interesting properties
  - upon magnification, the “shape” of the figure remains the same
  - the resulting figure has a floating point dimensionality value (2.35 D for example)

Algorithm to Draw a Tree

1. draw an edge
2. turn left
3. draw edge and left branch # recurse
4. turn right
5. draw edge and right branch # recurse
6. turn left
7. backup


Code Listing 15-9

```
from turtle import *

def branch(length, level):
    if level <= 0:                  # base case
        return
    forward(length)
    left(45)
    branch(0.6*length, level-1) # recursive case: left branch
    right(90)
    branch(0.6*length, level-1) # recursive case: right branch
    left(45)
    backward(length)
    return

# turn to get started
left(90)
branch(100,5)
```

Recursion Notes

- note that length is reduced as we recurse down (making for shorter branches)
- the numbers on the right of the following picture show the order in which the branches are drawn

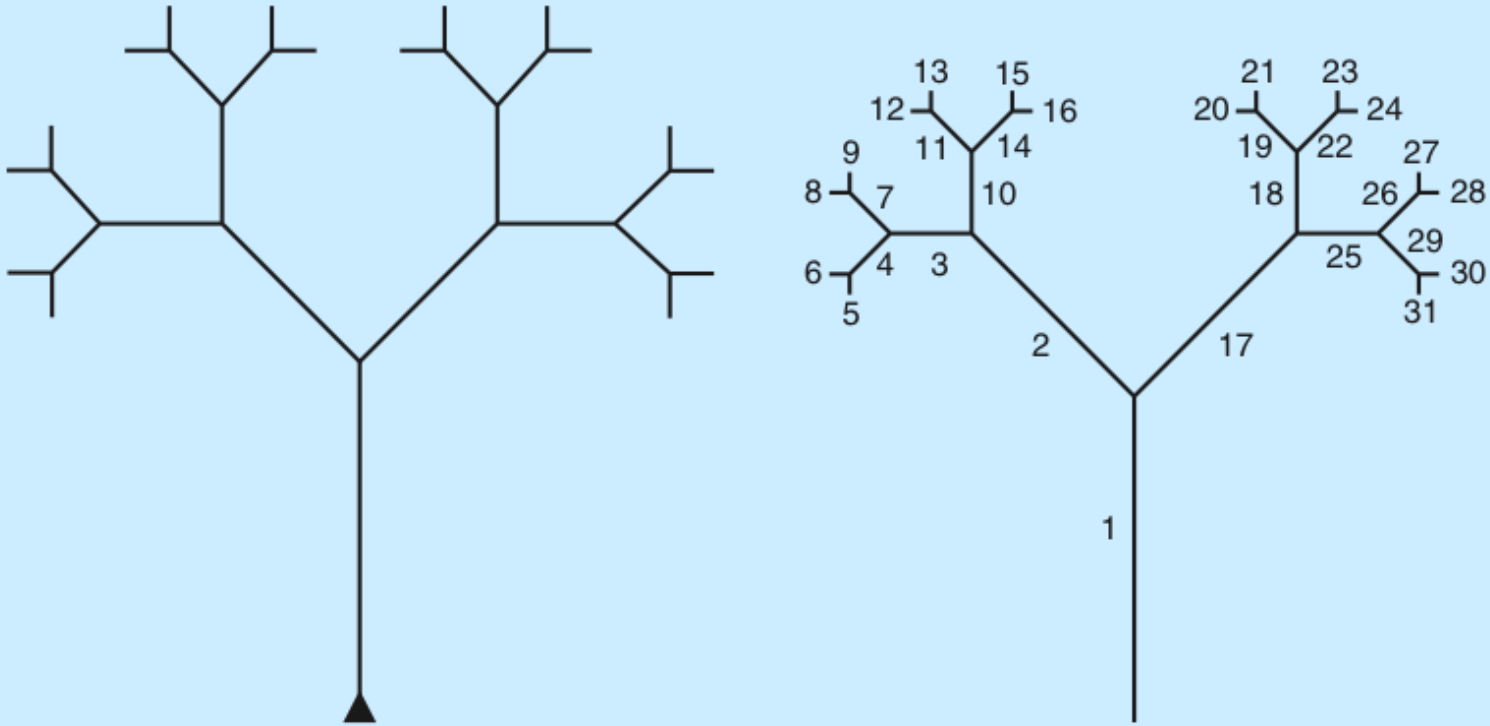


Figure 15.5 Recursive tree; (a) Python-drawn (left); (b) order-of-drawing on right.

Sierpinski Triangles

Little simpler than the tree:

1. draw edge
2. recurse
3. backward
4. turn 120 degrees


Code Listing 15.10
Sierpinski Triangles

```
# Draw Sierpinski figure

from turtle import *

def sierpinski(length, depth):
    if depth > 1: dot()  # mark position to better see recursion
    if depth == 0:       # base case
        stamp() # stamp a triangular shape
    else:
        forward(length)
        sierpinski(length/2, depth-1)  # recursive call
        backward(length)
        left(120)
        forward(length)
        sierpinski(length/2, depth-1)   # recursive call
        backward(length)
        left(120)
        forward(length)
        sierpinski(length/2, depth-1)  # recursive call
        backward(length)
        left(120)

sierpinski(200,6)
```

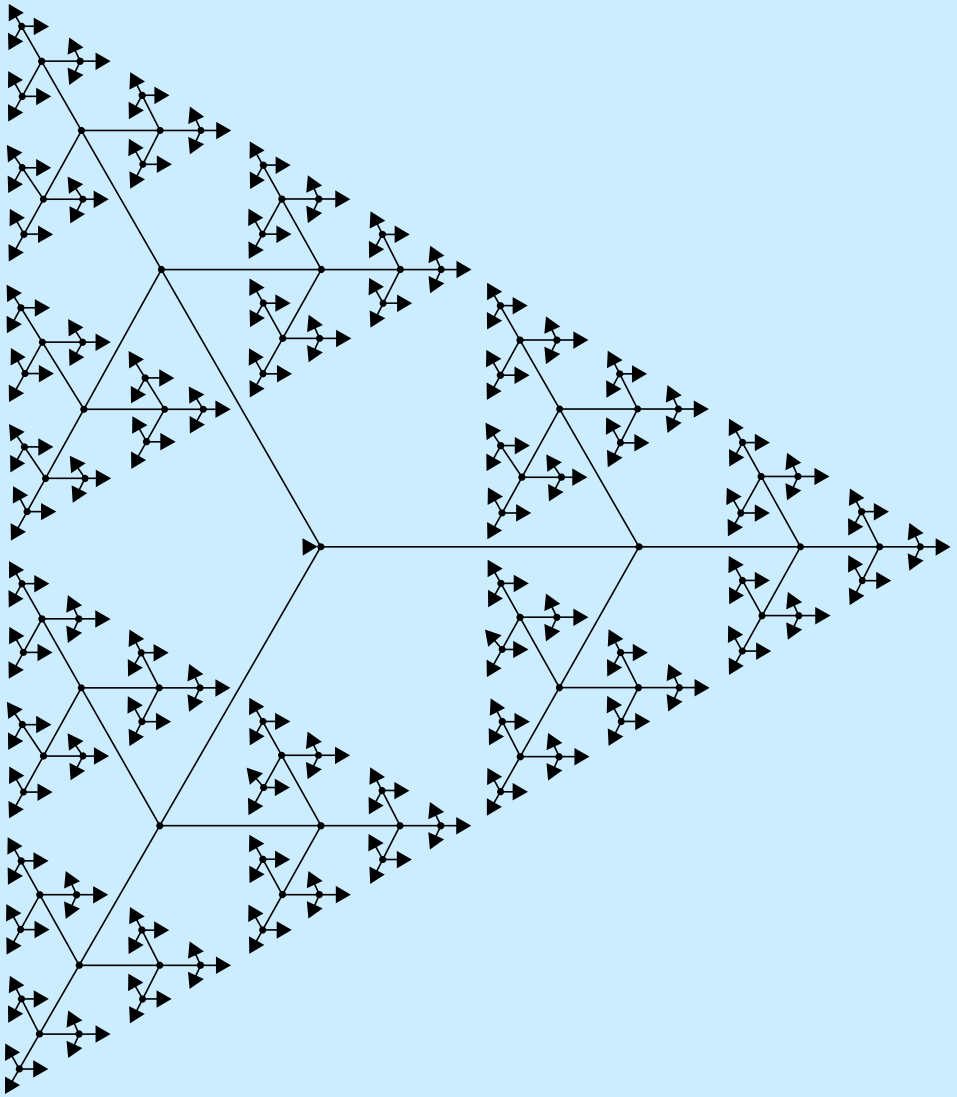


Figure 15.6 Sierpinski triangle

Some Recursive Details

- recursive functions are easy to write and lend themselves to divide and conquer
- they can be slow (all the pushing and popping on the stack)
- can be converted from recursive to iterative, but that can be hard depending on the problem