CHAPTER

10

Recursion

10.1 Introduction to Recursion 330

10.2 Examples of Recursion 336

10.3 Run Time Analysis 347

10.4 Searching 354

Case Study: Tower of Hanoi 359

Chapter Summary 360

Solutions to Practice Problems 360

Exercises 362

Problems 363

IN THIS CHAPTER, we learn about recursion, a powerful problem-solving technique, and run time analysis.

Recursion is a problem-solving technique that expresses the solution to a problem in terms of solutions to subproblems of the original problem. Recursion can be used to solve problems that might otherwise be quite challenging. The functions developed by solving a problem recursively will naturally call themselves, and we refer to them as recursive functions. We also show how namespaces and the program stack support the execution of recursive functions.

We demonstrate the wide use of recursion in number patterns, fractals, virus scanners, and searching. We differentiate between linear and nonlinear recursion and illustrate the close relationship between iteration and linear recursion.

As we discuss when recursion should and should not be used, the issue of program run time comes up. So far we have not worried much about the efficiency of our programs. We now rectify this situation and use the opportunity to analyze several fundamental search tasks. We develop a tool that can be used to analyze experimentally the running time of functions with respect to the size of the input.

10.1 Introduction to Recursion

A *recursive* function is a function that calls itself. In this section we explain what this means and how recursive functions get executed. We also introduce *recursive thinking* as an approach to problem solving. In the next section, we apply recursive thinking and how to develop recursive functions.

Functions that Call Themselves

Here is an example that illustrates what we mean by a function that calls itself:

Module: ch10.py

```
def countdown(n):
        print(n)
        countdown(n-1)
```

In the implementation of function `countdown()`, the function `countdown()` is called. So, function `countdown()` calls itself. When a function calls itself, we say that it makes a *recursive call*.

Let's understand the behavior of this function by tracing the execution of function call `countdown(3)`:

- When we execute `countdown(3)`, the input 3 is printed and then `countdown()` is called on the input decremented by 1—that is, $3 - 1 = 2$. We have 3 printed on the screen, and we continue tracing the execution of `countdown(2)`.
- When we execute `countdown(2)`, the input 2 is printed and then `countdown()` is called on the input decremented by 1—that is, $2 - 1 = 1$. We now have 3 and 2 printed on the screen, and we continue tracing the execution of `countdown(1)`.
- When we execute `countdown(1)`, the input 1 is printed and then `countdown()` is called on the input decremented by 1—that is, $1 - 1 = 0$. We now have 3, 2, and 1 printed on the screen, and we continue tracing the execution of `countdown(0)`.
- When we execute `countdown(0)`, the input 0 is printed and then `countdown()` is called on the input, 0, decremented by 1—that is, $0 - 1 = -1$. We now have 3, 2, 1, and 0 printed on the screen, and we continue tracing the execution of `countdown(-1)`.
- When we execute `countdown(-1)`, ...

It seems that the execution will never end. Let's check:

```
>>> countdown(3)
3
2
1
0
-1
-2
-3
...
```

The behavior of the function is to count down, starting with the original input number. If we let the function call `countdown(3)` execute for a while, we get:

```
...
```

```
-973
-974
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    countdown(3)
  File "/Users/me/ch10.py"...
    countdown(n-1)
...
```

And after getting many lines of error messages, we end up with:

```
RuntimeError: maximum recursion depth exceeded
```

OK, so the execution was going to go on forever, but the Python interpreter stopped it. We will explain why the Python VM does this soon. The main point to understand right now is that a recursive function will call itself forever unless we modify the function so there is a *stopping condition*.

Stopping Condition

To show this, suppose that the behavior we wanted to achieve with the `countdown()` function is really:

```
>>> countdown(3)
3
2
1
Blastoff!!!
```


or

```
>>> countdown(0)
Blastoff!!!
```

Function `countdown()` is supposed to count down to 0, starting from a given input n; when 0 is reached, `Blastoff!!!` should be printed.

To implement this version of `countdown()`, we consider two cases that depend on the value of the input n. When the input n is 0 or negative, all we need to do is print `'Blastoff!!!'`:

```
def countdown(n):
    'counts down to 0'
    if n <= 0:                # base case
        print('Blastoff!!!')
    else:
... # remainder of function
```

We call this case the *base case* of the recursion; it is the condition that will ensure that the recursive function is not going to call itself forever.

The second case is when the input n is positive. In that case we do the same thing we did before:

```
        print(n)
        countdown(n-1)
```

How does this code implement the function `countdown()` for input value $n > 0$? The insight used in the code is this: *Counting down from (positive number) n can be done by printing n first and then counting down from $n-1$.* This fragment of code is called *the recursive step*.

With the two cases resolved, we obtain the recursive function:

Module: ch10.py

```
def countdown(n):
    'counts down from n to 0'
    if n <= 0:                  # base case
        print('Blastoff!!!')
    else:                       # n > 0: recursive step
        print(n)                    # print n first and then
        countdown(n-1)              # count down from n-1 to 0
                                    # recursively
```

Properties of Recursive Functions

A recursive function that terminates will always have:

1. One or more base cases, which provide the stopping condition for the recursion. In function `countdown()`, the base case is the condition $n \leq 0$, where n is the input.
2. One or more recursive calls, which must be on arguments that are "closer" to the base case than the function input. In function `countdown()`, the sole recursive call is made on $n-1$, which is "closer" to the base case than input n.

What is meant by "closer" depends on the problem solved by the recursive function. The idea is that each recursive call should be made on problem inputs that are closer to the base case; this will ensure that the recursive calls eventually will get to the base case that will stop the execution.

In the remainder of this section and the next, we present many more examples of recursion. The goal is to learn how to develop recursive functions. To do this, we need to learn how to think recursively—that is, to describe the solution to a problem in terms of solutions of its subproblems. Why do we need to bother? After all, function `countdown()` could have been implemented easily using iteration. (Do it!) The thing is that recursive functions provide us with an approach that is an alternative to the iterative approach we used in Chapter 5. For some problems, this alternative approach actually is the easier, and sometimes, much easier approach. When you start writing programs that search the Web, for example, you will appreciate having mastered recursion.

Recursive Thinking

We use recursive thinking to develop recursive function `vertical()` that takes a nonnegative integer as input and prints its digits stacked vertically. For example:

```
>>> vertical(3124)
3
1
2
4
```

To develop `vertical()` as a recursive function, the first thing we need to do is decide the base case of the recursion. This is typically done by answering the question: When is the problem of printing vertically easy? For what kind of nonnegative number?

The problem is certainly easy if the input n has only one digit. In that case, we just output n itself:

```
>>> vertical(6)
6
```

So we make the decision that the base case is when $n < 10$. Let's start the implementation of the function `vertical()`:

```
def vertical(n):
    'prints digits of n vertically'
    if n < 10:          # base case: n has 1 digit
        print(n)            # just print n
    else:               # recursive step: n has 2 or more digits
        # remainder of function
```

Function `vertical()` prints `n` if `n` is less than 10 (i.e., `n` is a single digit number).

Now that we have a base case done, we consider the case when the input n has two or more digits. In that case, we would like to break up the problem of printing vertically number n into "easier" subproblems, involving the vertical printing of numbers "smaller" than n. In this problem, "smaller" should get us closer to the base case, a single-digit number. This suggests that our recursive call should be on a number that has fewer digits than n.

This insight leads to the following algorithm: Since n has at least two digits, we break the problem:

a. Print vertically the number obtained by removing the last digit of n; this number is "smaller" because it has one less digit. For $n = 3124$, this would mean calling function `vertical()` on 312.

b. Print the last digit. For $n = 3124$, this would mean printing 4.

The last thing to figure out is the math formulas for (1) the last digit of n and (2) the number obtained by removing the last digit. The last digit is obtained using the modulus (`%`) operator:

```
>>> n = 3124
>>> n%10
4
```

We can "remove" the last digit of n using the integer division operator (`//`):

```
>>> n//10
312
```

With all the pieces we have come up with, we can write the recursive function:

Module: ch10.py

```
def vertical(n):
    'prints digits of n vertically'
    if n < 10:          # base case: n has 1 digit
        print(n)            # just print n
    else:               # recursive step: n has 2 or more digits
        vertical(n//10)     # recursively print all but last digit
```

Practice Problem 10.1

Implement recursive method `reverse()` that takes a nonnegative integer as input and prints its digits vertically, starting with the low-order digit.

```
>>> reverse(3124)
4
2
1
3
```

Let's summarize the process of solving a problem recursively:

1. First decide on the base case or cases of the problem that can be solved directly, without recursion.
2. Figure out how to break the problem into one or more subproblems that are closer to the base case; the subproblems are to be solved recursively. The solutions to the subproblems are used to construct the solution to the original problem.

Practice Problem 10.2

Use recursive thinking to implement recursive function `cheers()` that, on integer input n, outputs n strings `'Hip '` followed by `'Hurray!!! '`.

```
>>> cheers(0)
Hurray!!!
>>> cheers(1)
Hip Hurray!!!
>>> cheers(4)
Hip Hip Hip Hip Hurray!!!
```

The base case of the recursion should be when n is 0; your function should then print `Hurrah`. When $n > 1$, your function should print `'Hip '` and then recursively call itself on integer input $n - 1$.

Practice Problem 10.3

In Chapter 5, we implemented function `factorial()` iteratively. The factorial function $n!$ has a natural recursive definition:

$$n! = \begin{array}{rl} 1 & \text{if n = 0} \\ n \cdot (n-1)! & \text{if n > 0} \end{array}$$

Reimplement function `factorial()` function using recursion. Also, estimate how many calls to `factorial()` are made for some input value $n > 0$.

Recursive Function Calls and the Program Stack

Before we practice solving problems using recursion, we take a step back and take a closer look at what happens when a recursive function gets executed. Doing so should help us recognize that recursion does work.

We consider what happens when function `vertical()` is executed on input $n = 3124$. In Chapter 7, we saw how namespaces and the program stack support function calls and the normal execution control flow of a program. Figure 10.1 illustrates the sequence of recursive function calls, the associated namespaces, and the state of the program stack during the execution of `vertical(3124)`.

Module: ch10.py

```
1  def vertical(n):
2      'prints digits of n vertically'
3      if n < 10:              # base case: n has 1 digit
4          print(n)                # just print n
5      else:                   # recursive step: n has 2 or more digits
6          vertical(n//10)         # recursively print all but last digit
7          print(n%10)             # print last digit of n
```

The difference between the execution shown in Figure 10.1 and Figure 7.5 in Chapter 7 is that in Figure 10.1, the same function gets called: function `vertical()` calls `vertical()`, which calls `vertical()`, which calls `vertical()`. In Figure 7.5, function `f()` calls `g()`, which calls `h()`. Figure 10.1 thus underlines that a namespace is associated with every function call rather than with the function itself.

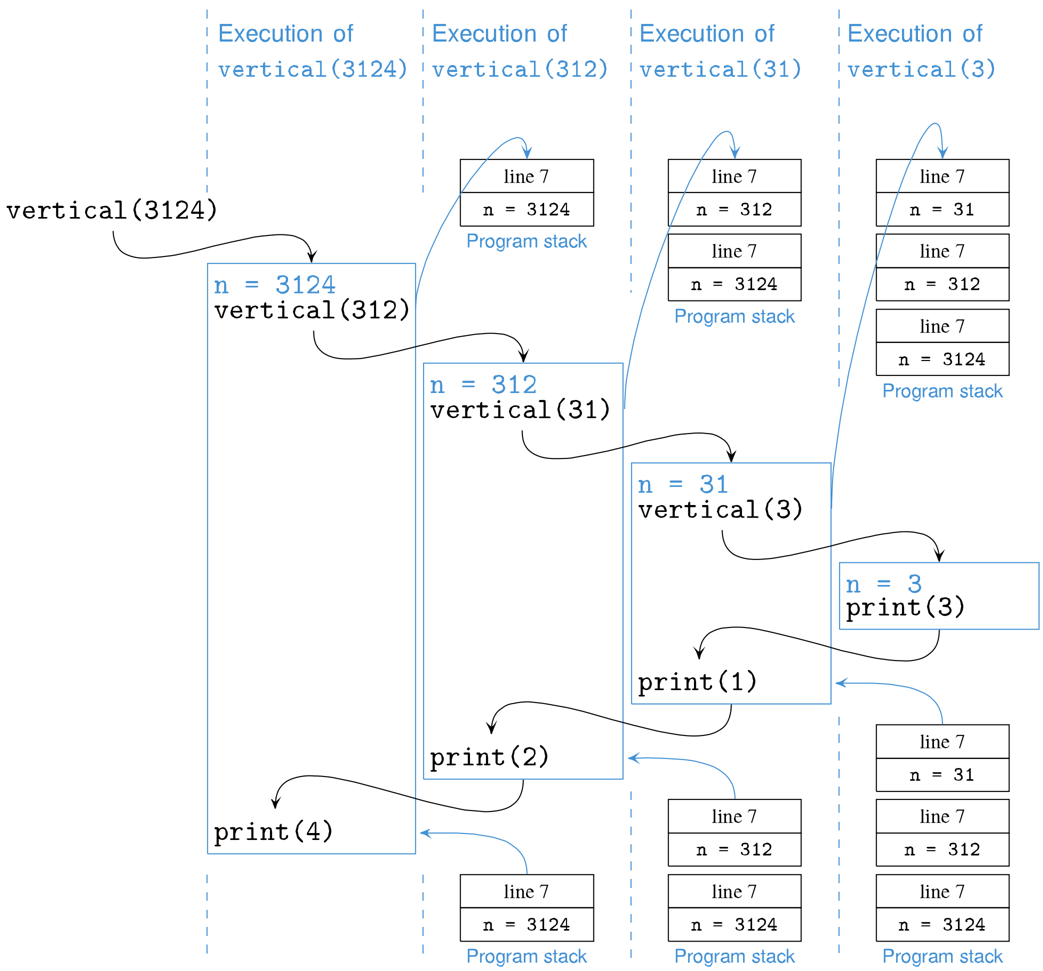


Figure 10.1 Recursive function execution. `vertical(3124)` executes in a namespace in which `n` is 3124. Just before call `vertical(312)` is made, values in the namespace (3124) and the next line to execute (line 7) are stored in the program stack. Then `vertical(312)` executes in a new namespace in which `n` is 312. Stack frames are similarly added just before recursive calls `vertical(31)` and `vertical(3)`. Call `vertical(3)` executes in a new namespace in which `n` is 3 and 3 is printed. When `vertical(3)` terminates, the namespace of `vertical(31)` is restored: `n` is 31, and the statement in line 7, `print(n%10)`, prints 1. Similarly, namespaces of `vertical(312)` and `vertical(3124)` are restored as well.

10.2 Examples of Recursion

In the previous section, we introduced recursion and how to solve problems using recursive thinking. The problems we used did not really showcase the power of recursion: Each problem could have been solved as easily using iteration. In this section, we consider problems that are far easier to solve with recursion.

Recursive Number Sequence Pattern

We start by implementing function `pattern()` that takes a nonnegative integer n and prints a number pattern:

```
>>> pattern(0)
0
>>> pattern(1)
0 1 0
>>> pattern(2)
0 1 0 2 0 1 0
>>> pattern(3)
0 1 0 2 0 1 0 3 0 1 0 2 0 1 0
>>> pattern(4)
0 1 0 2 0 1 0 3 0 1 0 2 0 1 0 4 0 1 0 2 0 1 0 3 0 1 0 2 0 1 0
```

How do we even know that this problem should be solved recursively? A priori, we do not, and we need to just try it and see whether it works. Let's first identify the base case. Based on the examples shown, we can decide that the base case is input 0 for which the function `pattern()` should just print 0. We start the implementation of the function:

```
def pattern(n):
    'prints the nth pattern'
    if n == 0:
        print(0)
    else:
        # remainder of function
```

We now need to describe what the function `pattern()` does for positive input n. Let's look at the output of `pattern(3)`, for example

```
>>> pattern(3)
0 1 0 2 0 1 0 3 0 1 0 2 0 1 0
```

and compare it to the output of `pattern(2)`

```
>>> pattern(2)
0 1 0 2 0 1 0
```

As Figure 10.2 illustrates, the output of `pattern(2)` appears in the output of `pattern(3)`, not once but twice:

Figure 10.2 Output of `pattern(3)`**.** The output of

```
pattern(3)    | 0 1 0 2 0 1 0 |  3  | 0 1 0 2 0 1 0 |
                  pattern(2)            pattern(2)
```

It seems that the correct output of `pattern(3)` can be obtained by calling the function `pattern(2)`, then printing 3, and then calling `pattern(2)` again. In Figure 10.3, we illustrate the similar behavior for the outputs of `pattern(2)` and `pattern(1)`:

`pattern(2)`	$\boxed{0\ 1\ 0}$	2	$\boxed{0\ 1\ 0}$
	`pattern(1)`		`pattern(1)`
`pattern(1)`	$\boxed{0}$	1	$\boxed{0}$
	`pattern(0)`		`pattern(0)`

Figure 10.3 Outputs of `pattern(2)` and `pattern(1)`. The output of `pattern(2)` can be obtained from the output of `pattern(1)`. The output of `pattern(1)` can be obtained from the output of `pattern(0)`.

In general, the output for `pattern(n)` is obtained by executing `pattern(n-1)`, then printing the value of `n`, and then executing `pattern(n-1)` again:

```
        ... # base case of function
    else
        pattern(n-1)
        print(n)
        pattern(n-1)
```

Let's try the function as implemented so far:

```
>>> pattern(1)
0
1
0
```

Almost done. In order to get the output in one line, we need to remain in the same line after each print statement. So the final solution is:

Module: ch10.py

```
def pattern(n):
    'prints the nth pattern'
    if n == 0:              # base case
        print(0, end=' ')
    else:                   # recursive step: n > 0
        pattern(n-1)            # print n-1st pattern
        print(n, end=' ')       # print n
        pattern(n-1)            # print n-1st pattern
```

Practice Problem 10.4

Implement recursive method `pattern2()` that takes a nonnegative integer as input and prints the pattern shown next. The patterns for inputs 0 and 1 are nothing and one star, respectively:

```
>>> pattern2(0)
>>> pattern2(1)
*
```

The patterns for inputs 2 and 3 are shown next.

```
>>> pattern2(2)
*
**
*
>>> pattern2(3)
*
**
*
***
*
**
*
```

Fractals

In our next example of recursion, we will also print a pattern, but this time it will be a graphical pattern drawn by a `Turtle` graphics object. For every nonnegative integer n, the printed pattern will be a curve called the *Koch curve* K_n. For example, Figure 10.4 shows Koch curve K_5.

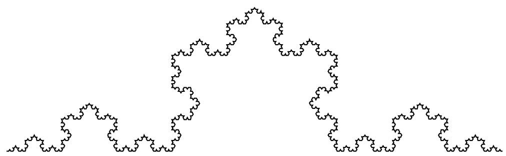


Figure 10.4 Koch curve K_5**.** A fractal curve often resembles a snowflake.

We will use recursion to draw Koch curves such as K_5. To develop the function that is used to draw this and other Koch curves, we look at the first few Koch curves. Koch curves K_0, K_1, K_2, and K_3 are shown on the left of Figure 10.5.

If you look carefully at the patterns, you might notice that each Koch curve K_i, for $i > 0$, contains within itself several copies of Koch curve K_{i-1}. For example, curve K_2 contains four copies of (smaller versions of) curve K_1.

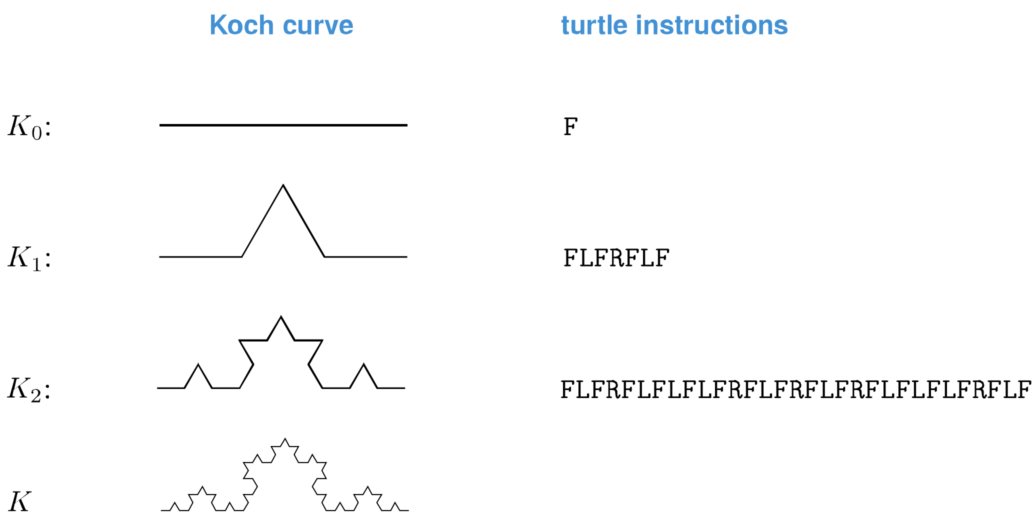


Figure 10.5 Koch curves with drawing instructions. On the left, from top to bottom, are Koch curves K_0, K_1, K_2, and K_3. The drawing instructions for Koch curves K_0, K_1, and K_2 are shown as well. The instructions are encoded using letters `F`, `L`, and `R` corresponding to "move forward," "rotate left 60 degrees," and "rotate right 120 degrees."

More precisely, to draw Koch curve K_2, a `Turtle` object should follow these instructions:

1. Draw Koch curve K_1.
2. Rotate left 60 degrees.
3. Draw Koch curve K_1.
4. Rotate right 120 degrees.
5. Draw Koch curve K_1.
6. Rotate left 60 degrees.
7. Draw Koch curve K_1.

Note that these instructions are described recursively. This suggests that what we need to do is develop a recursive function `koch(n)` that takes as input a nonnegative integer n and returns instructions that a `Turtle` object can use to draw Koch curve K_n. The instructions can be encoded as a string of letters `F`, `L`, and `R` corresponding to instructions "move forward," "rotate left 60 degrees," and "rotate right 120 degrees," respectively. For example, instructions for drawing Koch curves K_0, K_1, and K_2 are shown on the right of Figure 10.5. The function `koch()` should have this behavior:

```
>>> koch(0)
'F'
>>> koch(1)
'FLFRFLF'
>>> koch(2)
'FLFRFLFLFLFRFLFRFLFRFLFLFLFRFLF'
```

Now let's use the insight we developed about drawing curve K_2 in terms of drawing K_1 to understand how the instructions to draw K_2 (computed by function call `koch(2)`) are obtained using instructions to draw K_1 (computed by function call `koch(1)`). As Figure 10.6 illustrates, the instructions for curve K_1 appear in the instructions of curve K_2 four times:

`koch(2)`	$\boxed{\texttt{FLFRFLF}}$	L	$\boxed{\texttt{FLFRFLF}}$	R	$\boxed{\texttt{FLFRFLF}}$	L	$\boxed{\texttt{FLFRFLF}}$
	`koch(1)`		`koch(1)`		`koch(1)`		`koch(1)`

Figure 10.6 Output of `Koch(2)`**.** `Koch(1)` can be used to construct the output of `Koch(2)`.

Similarly, the instructions to draw K_1, output by `koch(1)`, contain the instructions to draw K_0, output by `koch(0)`, as shown in Figure 10.7:

`koch(1)`	$\boxed{\texttt{F}}$	L	$\boxed{\texttt{F}}$	R	$\boxed{\texttt{F}}$	L	$\boxed{\texttt{F}}$
	`koch(0)`		`koch(0)`		`koch(0)`		`koch(0)`

Figure 10.7 Output of `Koch(1)`**.** `Koch(0)` can be used to construct the output of `Koch(1)`.

Now we can implement function `koch()` recursively. The base case corresponds to input 0. In that case, the function should just return instruction `'F'`:

```
def koch(n):
    if n == 0:
        return 'F'
    # remainder of function
```

For input $n > 0$, we generalize the insight illustrated in Figures 10.6 and 10.7. The instructions output by `koch(n)` should be the concatenation:

```
koch(n-1) + 'L' + koch(n-1) + 'R' + koch(n-1) + 'L' + koch(n-1)
```

and the function `koch()` is then

```
def koch(n):
    if n == 0:
        return 'F'
    return koch(n-1) + 'L' + koch(n-1) + 'R' + koch(n-1) + 'L' + \
           koch(n-1)
```

If you test this function, you will see that it works. There is an efficiency issue with this implementation, however. In the last line, we call function `koch()` on the *same input* four times. Of course, each time the returned value (the instructions) is the same. Our implementation is very wasteful.

CAUTION

Avoid Repeating the Same Recursive Calls

Often, a recursive solution is most naturally described using several identical recursive calls. We just saw this with the recursive function `koch()`. Instead of repeatedly calling the same function on the same input, we can call it just once and reuse its output multiple times.

The better implementation of function `koch()` is then:

Module: ch10.py

```
def koch(n):
    'returns turtle directions for drawing curve Koch(n)'

    if n == 0:        # base case
        return 'F'

    tmp = koch(n-1) # recursive step: get directions for Koch(n-1)
                    # use them to construct directions for Koch(n)

    return tmp + 'L' + tmp + 'R' + tmp + 'L' + tmp
```

The last thing we have to do is develop a function that uses the instructions returned by function `koch()` and draws the corresponding Koch curve using a `Turtle` graphics object. Here it is:

Module: ch10.py

```
from turtle import Screen, Turtle
def drawKoch(n):
    'draws nth Koch curve using instructions from function koch()'

    s = Screen()              # create screen
    t = Turtle()              # create turtle
    directions = koch(n)      # obtain directions to draw Koch(n)
```

```
9      for move in directions: # follow the specified moves
10         if move == 'F':
11             t.forward(300/3**n) # move forward, length normalized
12         if move == 'L':
13             t.lt(60)            # rotate left 60 degrees
14         if move == 'R':
15             t.rt(120)           # rotate right 60 degrees
16     s.bye()
```

Line 11 requires some explanation. The value `300/3**n` is the length of a forward turtle move. It depends on the value of `n` so that, no matter what the value of `n` is, the Koch curve has width 300 pixels and fits in the screen. Check this for `n` equal to 0 and 1.

DETOUR

Koch Curves and Other Fractals

The Koch curves K_n were first described in a 1904 paper by the Swedish mathematician Helge von Koch. He was particularly interested in the curve K_∞ that is obtained by pushing n to ∞.

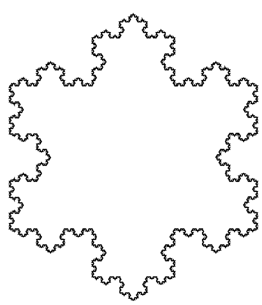
The Koch curve is an example of a *fractal*. The term *fractal* was coined by French mathematician Benoît Mandelbrot in 1975 and refers to curves that:

- Appear “fractured” rather than smooth
- Are *self-similar* (i.e., they look the same at different levels of magnification)
- Are naturally described recursively

Physical fractals, developed through recursive physical processes, appear in nature as snowflakes and frost crystals on cold glass, lightning and clouds, shorelines and river systems, cauliflower and broccoli, trees and ferns, and blood and pulmonary vessels.

Practice Problem 10.5

Implement function `snowflake()` that takes a nonnegative integer n as input and prints a snowflake pattern by combining three Koch curves K_n in this way: When the turtle is finished drawing the first and the second Koch curve, the turtle should rotate right 120 degrees and start drawing a new Koch curve. Shown here is the output of `snowflake(4)`.



Virus Scanner

We now use recursion to develop a virus scanner, that is, a program that systematically looks at every file in the filesystem and prints the names of the files that contain a known *computer virus signature*. The signature is a specific string that is evidence of the presence of the virus in the file.

DETOUR

Viruses and Virus Scanners

A *computer virus* is a small program that, usually without the user's knowledge, is attached to or incorporated in a file hosted on the user's computer and does nefarious things to the host computer when executed. A computer virus may corrupt or delete data on a computer, for example.

A virus is an executable program, stored in a file as a sequence of bytes just like any other program. If the computer virus is identified by a computer security expert and the sequence of bytes is known, all that needs to be done to check whether a file contains the virus is to check whether that sequence of bytes appears in the file. In fact, finding the *entire* sequence of bytes is not really necessary; searching for a carefully chosen fragment of this sequence is enough to identify the virus with high probability. This fragment is called the *signature* of the virus: It is a sequence of bytes that appears in the virus code but is unlikely to appear in an uninfected file.

A *virus scanner* is a program that periodically and *systematically* scans every file in the computer filesystem and checks each for viruses. The scanner application will have a list of virus signatures that is updated regularly and automatically. Each file is checked for the presence of some signature in the list and flagged if it contains that signature.

We use a dictionary to store the various virus signatures. It maps virus names to virus signatures:

```
>>> signatures = {'Creeper':'ye8009g2h1azzx33',
                  'Code Red':'99dh1cz963bsscs3',
                  'Blaster':'fdp1102k1ks6hgbc'}
```

(While the names in this dictionary are names of real viruses, the signatures are completely fake.)

The virus scanner function takes, as input, the dictionary of virus signatures and the pathname (a string) of the top folder or file. It then visits every file contained in the top folder, its subfolders, subfolders of its subfolders, and so on. An example folder `'test'` is shown in Figure 10.8 together with all the files and folders that are contained in it, directly or indirectly. The virus scanner would visit every file shown in Figure 10.8 and could produce, for example, this output:

File: test.zip

```
>>> scan('test', signatures)
test/fileA.txt, found virus Creeper
test/folder1/fileB.txt, found virus Creeper
test/folder1/fileC.txt, found virus Code Red
test/folder1/folder11/fileD.txt, found virus Code Red
test/folder2/fileD.txt, found virus Blaster
test/folder2/fileE.txt, found virus Blaster
```

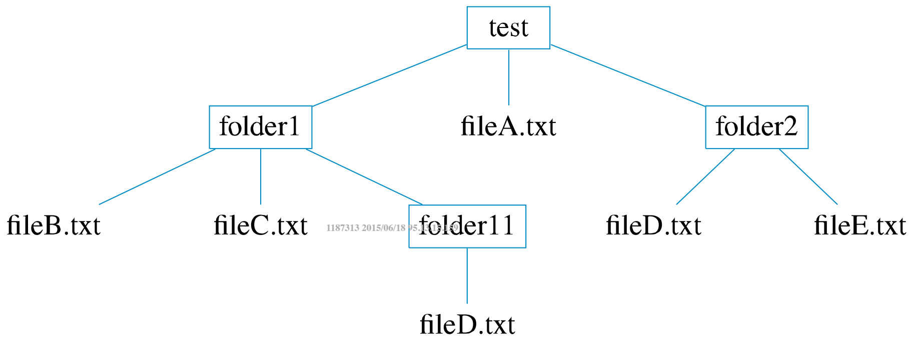


Figure 10.8 Filesystem fragment. Illustrated is folder `'test'` and all its descendant folders and files.

Because of the recursive structure of a filesystem (a *folder* contains files and other *folders*), we use recursion to develop the virus scanner function `scan()`. When the input pathname is the pathname of a file, the function should open, read, and search the file for virus signatures; this is the base case. When the input pathname is the pathname of a folder, `scan()` should recursively call itself on every file and subfolder of the input folder; this is the recursive step. The complete implementation is:

Module: ch10.py

```
import os
def scan(pathname, signatures):
    '''scans pathname or, if pathname is a folder, scans all files
       contained, directly or indirectly, in the folder pathname'''
    if os.path.isfile(pathname): # base case, scan pathname
        infile = open(pathname)
        content = infile.read()
        infile.close()

        for virus in signatures:
            # check whether virus signature appears in content
            if content.find(signatures[virus]) >= 0:
                print('{}, found virus {}'.format(pathname, virus))
        return

    # pathname is a folder so recursively scan every item in it
    for item in os.listdir(pathname):

        # create pathname for item relative
        # to current working directory
        # fullpath = pathname + '/' + item          # Mac only
        # fullpath = pathname + '\' + item          # Windows only
        fullpath = os.path.join(pathname, item)     # any OS

        scan(fullpath, signatures)
```

This program uses functions from the Standard Library module `os`. The module `os` contains functions that provide access to operating system resources such as the filesystem. The three `os` module functions we are using are:

a. `listdir()`. Takes, as input, an absolute or relative pathname (as a string) of a folder and returns the list of all files and subfolders contained in the input folder.

b. `path.isfile()`. Takes, as input, an absolute or relative pathname (as a string) and returns `True` if the pathname refers to a regular file, `False` otherwise.

c. `path.join()`. Takes as input two pathnames, joins them into a new pathname, inserting \ or / as needed, and returns it.

We explain further why we need the third function. The function `listdir()` *does not* return a list of *pathnames* but just a list of file and folder *names*. For example, when we start executing `scan('test')` (we ignore the second argument of `scan()` in this discussion), the function `listdir()` will get called in this way:

```
>>> os.listdir('test')
['fileA.txt', 'folder1', 'folder2']
```

If we were to make the recursive call `scan('folder1')`, then, when this function call starts executing, the function `listdir()` would get called on pathname `'folder1'`, with this result:

```
>>> os.listdir('folder1')
Traceback (most recent call last):
  File "<pyshell#387>", line 1, in <module>
    os.listdir('folder1')
OSError: [Errno 2] No such file or directory: 'folder1'
```

The problem is that the *current working directory* during the execution of `scan('test')` is the folder that contains the folder `test`; the folder `'folder1'` is not in there, thus the error.

Instead of making the call `scan('folder1')`, we need to make the call on a pathname that is either absolute or relative with respect to the current working directory. The pathname of `'folder1'` can be be obtained by concatenating `'test'` and `'folder1'` as follows

```
'test' + '\' + 'folder1'
```

(on a Windows box) or, more generally, concatenating `pathname` and `item` as follows

```
path = pathname + '\' + item
```

This works on Windows machines but not on UNIX, Linux, or MAC OS X machines because pathnames use the forward slashes (/) in those operating systems. A better, portable solution is to use the `path.join()` function from module `os`. It will work for all operating systems and thus be system independent. For example, on a Mac:

```
>>> pathname = 'test'
>>> item = 'folder1'
>>> os.path.join(pathname, item)
'test/folder1'
```

Here is a similar example executed on a Windows box:

```
>>> pathname = 'C://Test/virus'
>>> item = 'folder1'
>>> os.path.join(pathname, item)
```

Linear recursion

The three problems we have considered in this section—printing the number sequence pattern, drawing the Koch curve, and scanning the filesystem for viruses—could all have been solved without recursion. Iterative solutions for these problems really do exist. The iterative solutions, however, require algorithms that are more complex than recursion and that are beyond the scope of an introductory computer science textbook.

The problems we considered in Section 10.1, on the other hand, have simple iterative solutions. Recursive functions `vertical()`, `reverse()`, `cheers()`, and `factorial()` from Section 10.1 could have as easily been developed using iteration. In fact, the recursive and iterative solutions are closely related. The two implementations of function `factorial()` from Practice Problem 10.3 and Practice Problem 5.4 can be used to illustrate this. While one implementation is recursive and the other is iterative, both functions use a similar process to compute $n!$: they both compute a sequence of intermediate results $i!$, for $i = 1, \ldots, n$, obtained by multiplying the previous intermediate result $(i-1)!$ with i. The recursive function can thus be viewed as a recursive implementation of this idea.

When the recursive step of a function is implemented using a single recursive call that computes the "previous" intermediate result and a "basic," nonrecursive (problem specific) operation that computes the "next" intermediate result, the function is said to use *linear recursion*. In function `vertical()`, for example, the recursive step consists of a single recursive call `vertical(n//10)` that prints all but the last digit of `n` and statement `print(n%10)` that prints the last digit.

Linear recursion is a particularly useful technique for implementing fundamental functions on lists. For example, a function that adds the numbers in a list of numbers can be implemented using linear recursion as follows:

Module: ch10.py

```
def recSum(lst):
    'returns the sum of items in list lst'
    if len(lst) == 0:
        return 0
    return recSum(lst[:-1]) + lst[-1]
```

Note that the recursive step consists of a single recursive call that sums all the numbers in the list but the last and a "basic" operation that adds the last number to this sum.

Practice Problem 10.6

Using linear recursion, implement function `recNeg()` that takes a list of numbers as input and returns `True` if some number in the list is negative, and `False` otherwise.

```
>>> recNeg([3, 1, -1, 5])
True
>>> recNeg([3, 1, 0, 5])
False
```

In the next example, we implement function `recIncr()` that takes a list of numbers as input and returns a copy of the list with every number in the list incremented by one:

```
>>> lst = [1, 4, 9, 16, 25]
>>> recIncr(lst)
[2, 5, 10, 17, 26]
```

We choose to implement the function using linear recursion instead of iteration:

Module: ch10.py

```
def recIncr(lst):
    'returns list [lst[0]+1, lst[1]+1, ..., lst[n-1]+1]'
    if len(lst) == 0:
        return []
    return recIncr(lst[:-1]) + [lst[-1]+1]
```

The recursive step consists of concatenating the list obtained by the recursive call and the list containing the last number in the list incremented by one.

The function `recIncr()` is an example of a function that takes a list and returns a copy of it in which the same operation was performed on every list item. Incrementing every number in the list by one is just one of the many operations one may wish to perform on items of a list. It would thus be useful to implement a more abstract function `recMap()` that takes, as input, the *operation* as well as the list and then applies the operation to every item in the list. What "operation" really means, of course, is a function. For example, if we wanted to use function `recMap()` to increment every number in a list of numbers, we would first have to define the function that we want to apply to every number:

```
>>> def f(i):
        return i + 1
```

Then we would use `recMap()` to apply function `f` to every number in the list:

```
>>> recMap(lst, f)
[2, 5, 10, 17, 26]
```

If, instead, we wanted to obtain a list containing the square roots of the numbers in list `lst`, we would apply the `math.sqrt` function instead:

```
>>> from math import sqrt
>>> recMap(lst, sqrt)
[1.0, 2.0, 3.0, 4.0, 5.0]
```

Note that the input argument of `recMap()` is `f`, not `f()`, or `sqrt`, not `sqrt()`. This is because we are simply passing a reference to the function object, not making a function call.

We can implement `recMap()` using linear recursion:

Module: ch10.py

```
def recMap(lst, f):
    'returns list [f(lst[0]), f(lst[1]), ..., f(lst[n-1])]'
    if len(lst) == 0:
        return []
    return recMap(lst[:-1], f) + [f(lst[-1])]
```

DETOUR

Higher-Order Functions

In function `recMap()`, the second input argument is a function. A function that takes another function as input or that returns a function is called a *higher-order function*. Treating a function like a value is a style of programming that is used extensively in the *functional programming* paradigm which we introduce in Section 12.3.

Python supports higher-order functions because the name of a function is treated no differently from the name of any other object, so it can be treated as a value. Not all languages support higher-order functions. A few other ones that do are LISP, Perl, Ruby, and JavaScript.

Practice Problem 10.7

Using function `recMap()`, write a short statement that evaluates to a list containing the sums of the rows of a two-dimensional table of numbers called `table`.

10.3 Run Time Analysis

The correctness of a program is of course our main concern. However, it is also important that the program is usable or even efficient. In this section, we continue the use of recursion to solve problems, but this time with an eye on efficiency. In our first example, we apply recursion to a problem that does not seem to need it and get a surprising gain in efficiency. In the second example, we take a problem that seems tailored for recursion and obtain an extremely inefficient recursive program.

The Exponent Function

We consider next the implementation of the exponent function a^n. As we have seen already, Python provides the exponentiation operator `**`:

```
>>> 2**4
16
```

But how is the operator `**` implemented? How would we implement it if it was not available? The straightforward approach is to just multiply the value of `a` n times. The accumulator pattern can be used to implement this idea:

Module: ch10.py

```
def power(a, n):
    'returns a to the nth power'
    res = 1
    for i in range(n):
        res *= a
    return res
```

You should convince yourself that the function `power()` works correctly. But is this the best way to implement the function `power()`? Is there an implementation that would run faster? It is clear that the function `power()` will perform n multiplications to compute a^n. If n is 10,000, then 10,000 multiplications are done. Can we implement `power()` so significantly fewer multiplications are done, say about 20 instead of 10,000?

Let's see what the recursive approach will give us. We are going to develop a recursive function `rpower()` that takes inputs a and nonnegative integer n and returns a

The natural base case is when the input n is 0. Then $a^n = 1$ and so 1 must be returned:

```
def rpower(a, n):
    'returns a to the nth power'
    if n == 0:                 # base case: n == 0
        return 1
    # remainder of function
```

Now let's handle the recursive step. To do this, we need to express a^n, for $n > 0$, recursively in terms of smaller powers of a (i.e., "closer" to the base case). That is actually not hard, and there are many ways to do it:

$$\begin{aligned} a^n &= a^{n-1} \times a \\ a^n &= a^{n-2} \times a^2 \\ a^n &= a^{n-3} \times a^3 \\ &\ldots \\ a^n &= a^{n/2} \times a^{n/2} \end{aligned}$$

The appealing thing about the last expression is that the two terms, $a^{n/2}$ and $a^{n/2}$, are the same; therefore, we can compute a^n by making only one recursive call to compute $a^{n/2}$. The only problem is that $n/2$ is not an integer when n is odd. So we consider two cases.

As we just discovered, when the value of n is even, we can compute `rpower(a, n)` using the result of `rpower(a, n//2)` as shown in Figure 10.9:

Figure 10.9 Computing a^n recursively. When n is even, $a^n = a^{n/2} \times a^{n/2}$.

$$\texttt{rpower(2, n)} \;=\; \underbrace{\boxed{2 \times 2 \times \ldots \times 2}}_{\texttt{power(2, n//2)}} \times \underbrace{\boxed{2 \times 2 \times \ldots \times 2}}_{\texttt{power(2, n//2)}}$$

When the value of n is odd, we still can use the result of recursive call `rpower(a, n//2)` to compute `rpower(a, n)`, albeit with an additional factor a, as illustrated in Figure 10.10:

Figure 10.10 Computing a^n recursively. When n is odd, $a^n = a^{\lfloor n/2 \rfloor} \times a^{\lfloor n/2 \rfloor} \times a$.

$$\texttt{rpower(2, n)} \;=\; \underbrace{\boxed{2 \times 2 \times \ldots \times 2}}_{\texttt{power(2, n//2)}} \times \underbrace{\boxed{2 \times 2 \times \ldots \times 2}}_{\texttt{power(2, n//2)}} \times \boxed{2}$$

These insights lead us to the recursive implementation of `rpower()` shown next. Note that only one recursive call `rpower(a, n//2)` is made.

Module: ch10.py

```
def rpower(a, n):
    'returns a to the nth power'
    if n == 0:                 # base case: n == 0
        return 1

    tmp = rpower(a, n//2)    # recursive step: n > 0

    if n % 2 == 0:
        return tmp*tmp           # a**n = a**(n//2) * a**a(n//2)
    else: # n % 2 == 1
        return a*tmp*tmp         # a**n = a**(n//2) * a**a(n//2) * a
```

We now have two implementations of the exponentiation function, `power()` and `rpower()`. How can we tell which is more efficient?

Counting Operations

One way to compare the efficiency of two functions is to count the number of operations executed by each function on the same input. In the case of `power()` and `rpower()`, we limit ourselves to counting just the number of multiplications

Clearly, `power(2, 10000)` will need 10,000 multiplications. What about `rpower(2, 10000)`? To answer this question, we modify `rpower()` so it *counts* the number of multiplications performed. We do this by incrementing a `global` variable `counter`, defined outside the function, each time a multiplication is done:

Module: ch10.py

```
def rpower(a, n):
    'returns a to the nth power'
    global counter      # counts number of multiplications

    if n==0:
        return 1
    # if n > 0:
    tmp = rpower(a, n//2)

    if n % 2 == 0:
        counter += 1
        return tmp*tmp      # 1 multiplication

    else: # n % 2 == 1
        counter += 2
        return a*tmp*tmp    # 2 multiplications
```


Now we can do the counting:

```
>>> counter = 0
>>> rpower(2, 10000)
199506311688...792596709376
>>> counter
19
```

Thus, recursion led us to a way to do exponentiation that reduced the number of multiplications from 10,000 to 23.

Fibonacci Sequence

We introduced the Fibonacci sequence of integers in Chapter 5:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ \ldots$$

We also described a method to construct the Fibonacci sequence: A number in the sequence is the sum of the previous two numbers in the sequence (except for the first two 1s). This rule is recursive in nature. So, if we are to implement a function `rfib()` that takes a nonnegative integer n as input and returns the nth Fibonacci number, a recursive implementation seems natural. Let's do it.

Since the recursive rule applies to the numbers after the 0th and 1st Fibonacci number, it makes sense that the base case is when $n \leq 1$ (i.e., $n = 0$ or $n = 1$). In that case, `rfib()` should return 1:

```
def rfib(n):
    'returns nth Fibonacci number'
    if n < 2:                    # base case
        return 1
    # remainder of function
```

The recursive step applies to input $n > 1$. In that case, the nth Fibonacci number is the sum of the $n-1$st and $n-2$nd:

Module: ch10.py

```
def rfib(n):
    'returns nth Fibonacci number'
    if n < 2:                      # base case
        return 1

    return rfib(n-1) + rfib(n-2)  # recursive step
```

Let's check that function `rfib()` works:

```
>>> rfib(0)
1
>>> rfib(1)
1
>>> rfib(4)
5
>>> rfib(8)
34
```

The function seems correct. Let's try to compute a larger Fibonacci number:

```
>>> rfib(35)
14930352
```

Hmmm. It's correct, but it took a while to compute. (Try it.) If you try

```
>>> rfib(100)
...
```

you will be waiting for a very long time. (Remember that you can always stop the program execution by hitting Ctrl-c simultaneously.)

Is computing the 36th Fibonacci number really that time consuming? Recall that we already implemented a function in Chapter 5 that returns the nth Fibonacci number:

Module: ch10.py

```
def fib(n):
    'returns nth Fibonacci number'
    previous = 1     # 0th Fibonacci number
    current = 1      # 1st Fibonacci number
    i = 1            # index of current Fibonacci number

    while i < n:     # while current is not nth Fibonacci number
        previous, current = current, previous+current
        i += 1

```

Let's see how it does:

```
>>> fib(35)
14930352
>>> fib(100)
573147844013817084101
>>> fib(10000)
54438373113565...
```

Instantaneous in all cases. Let's investigate what is wrong with `rfib()`.

Experimental Analysis of Run Time

One way to precisely compare functions `fib()` and `rfib()`—or other functions for that matter—is to run them on the same input and compare their run times. As good (lazy) programmers, we like to automate this process, so we develop an application that can be used to analyze the run time of a function. We will make this application generic in the sense that it can be used on functions other than just `fib()` and `rfib()`.

Our application consists of several functions. The key one that measures the run time on one input is `timing()`: It is a higher-order function that takes as input (1) a function `func` and (2) an "input size" (as an integer), runs function `func` on an input of the given size, and returns the execution time.

Module: ch10.py

```
import time
def timing(func, n):
    'runs func on input returned by buildInput'
    funcInput = buildInput(n)  # obtain input for func
    start = time.time()         # take start time
    func(funcInput)             # run func on funcInput
    end = time.time()           # take end time
    return end - start          # return execution time
```

Function `timing()` uses the `time()` function from the `time` module to obtain the current time before and after the execution of the function `func`; the difference between the two will be the execution time. (*Note:* The timing can be affected by other tasks the computer may be doing, but we avoid dealing with this issue.)

The function `buildInput()` takes an input size and returns an object that is an appropriate input for function `func()` and has the right input size. This function is dependent on the function `func()` we are analyzing. In the case of the Fibonacci functions `fib()` and `rfib()`, the input corresponding to input size n is just n:

Module: ch10.py

```
def buildInput(n):
    'returns input for Fibonacci functions'
    return n
```

Comparing the run times of two functions on the same input does not tell us much about which function is better (i.e., faster). It is more useful to compare the run times of the two functions on *several* different inputs. In this way, we can attempt to understand the behavior of the two functions as the input size (i.e., the problem size) becomes larger. We develop, for that purpose, function `timingAnalysis` that runs an arbitrary function on a series of inputs of increasing size and report run times.

Module: ch10.py

```
def timingAnalysis(func, start, stop, inc, runs):
    '''prints average run times of function func on inputs of
       size start, start+inc, start+2*inc, ..., up to stop'''
    for n in range(start, stop, inc):  # for every input size n
        acc = 0.0                      # initialize accumulator

        for i in range(runs):     # repeat runs times:
            acc += timing(func, n)    # run func on input of size n
                                      # and accumulates run times
        # print average run times for input size n
        formatStr = 'Run time of {}({}) is {:.7f} seconds.'
        print(formatStr.format(func.__name__, n, acc/runs))
```


Function `timingAnalysis` takes, as input, function `func` and numbers `start`, `stop`, `inc`, and `runs`. It first runs `func` on several inputs of size `start` and prints the average run time. Then it repeats that for input sizes `start+inc`, `start+2*inc`, . . . up to input size `stop`.

When we run `timinAnalysis()` on function `fib()` with input sizes 24, 26, 28, 30, 32, 34, we get:

```
>>> timingAnalysis(fib, 24, 35, 2, 10)
Run time of fib(24) is 0.0000173 seconds.
Run time of fib(26) is 0.0000119 seconds.
Run time of fib(28) is 0.0000127 seconds.
Run time of fib(30) is 0.0000136 seconds.
Run time of fib(32) is 0.0000144 seconds.
Run time of fib(34) is 0.0000151 seconds.
```

When we do the same on function `rfib()`, we get:

```
>>> timingAnalysis(rfib, 24, 35, 2, 10)
Run time of fibonacci(24) is 0.0797332 seconds.
Run time of fibonacci(26) is 0.2037848 seconds.
Run time of fibonacci(28) is 0.5337492 seconds.
Run time of fibonacci(30) is 1.4083670 seconds.
Run time of fibonacci(32) is 3.6589111 seconds.
Run time of fibonacci(34) is 9.5540136 seconds.
```

We graph the results of the two experiments in Figure 10.11.

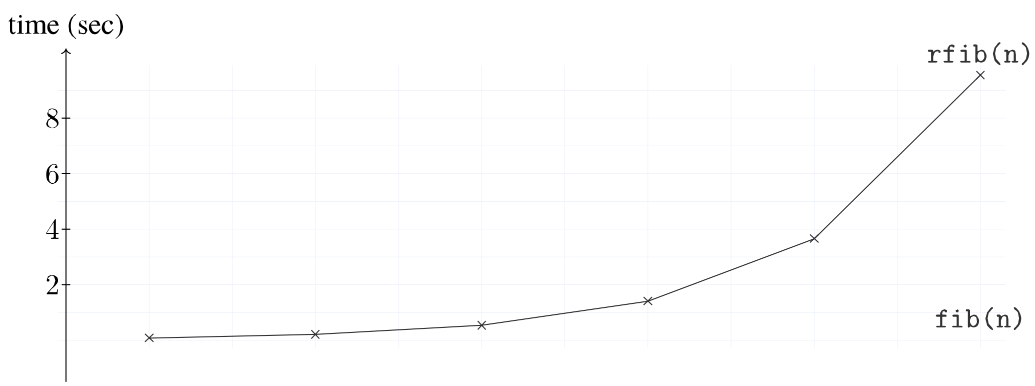


Figure 10.11 Run time graph. Shown are the average run times, in seconds, of `fib()` and `rfib()` for inputs $n = 24$, 26, 28, 32, and 34.

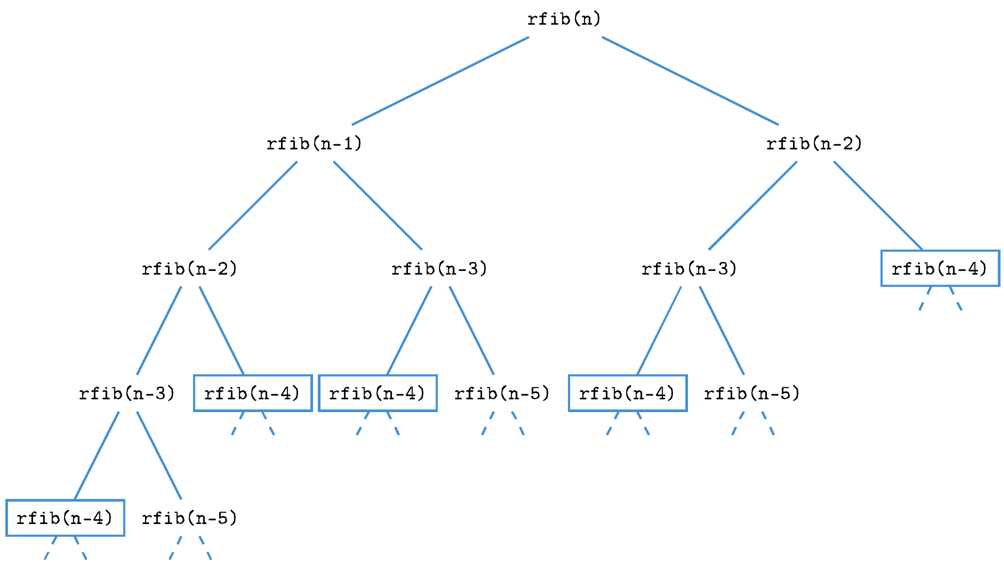


Figure 10.12 Tree of recursive calls. Computing `rfib(n)` requires making two recursive calls: `rfib(n-1)` and `rfib(b-2)`. Computing `rfib(n-1)` requires making recursive calls `rfib(n-2)` and `rfib(n-3)`; computing `rfib(n-2)` requires recursive calls `rfib(n-3)` and `rfib(n-4)`. The same recursive calls will be made multiple times. For example, `rfib(n-4)` will be recomputed five times.

The run times of `fib()` are negligible. However, the run times of `rfib()` are increasing rapidly as the input size increases. In fact, the run time more than doubles between successive input sizes. This means that the run time increases exponentially with respect to the input size. In order to understand the reason behind the poor performance of the recursive function `rfib()`, we illustrate its execution in Figure 10.12.

Figure 10.12 shows some of the recursive calls made when computing `rfib(n)`. To compute `rfib(n)`, recursive calls `rfib(n-1)` and `rfib(n-2)` must be made; to compute `rfib(n-1)` and `rfib(n-2)`, separate recursive calls `rfib(n-2)` and `rfib(n-3)`, and `rfib(n-2)` and `rfib(n-3)`, respectively, must be made. And so on.

The computation of `rfib()` includes two separate computations of `rfib(n-2)` and should therefore take more than twice as long as `rfib(n-2)`. This explains the exponential growth in run time. It also shows the problem with the recursive solution `rfib()`: It keeps making and executing the same function calls, over and over. The function call `rfib(n-4)`, for example, is made and executed five times, even though the result is always the same.

Practice Problem 10.8

Using the run time analysis application developed in this section, analyze the run time of functions `power()` and `rpower()` as well as built-in operator `**`. You will do this by running `timingAnalysis()` on functions `power2()`, `rpower2()`, and `pow2()` defined next and using input sizes 20,000 through 80,000 with a step size of 20,000.

```
def power2(n):
    return power(2,n)
def rpower2(n):
    return rpower(2,n)
def pow2(n):
    return 2**n
```

When done, argue which approach the built-in operator `**` likely uses.

10.4 Searching

In the last section, we learned that the way we design an algorithm and implement a program can have a significant effect on the program's run time and ultimately its usefulness with large data sets. In this section, we consider how reorganizing the input data set and adding structure to it can dramatically improve the run time, and usefulness, of a program. We focus on several fundamental search tasks and usually use sorting to give structure to the data set. We start with the fundamental problem of checking whether a value is contained in a list.

Linear Search

Both the `in` operator and the `index()` method of the `list` class search a list for a given item. Because we have been (and will be) using them *a lot*, it is important to understand how fast they execute.

Recall that the `in` operator is used to check whether an item is in the list or not:

```
>>> lst = random.sample(range(1,100), 17)
>>> lst
[28, 72, 2, 73, 89, 90, 99, 13, 24, 5, 57, 41, 16, 43, 45, 42, 11]
>>> 45 in lst
True
>>> 75 in lst
False
```

The `index()` method is similar: Instead of returning `True` or `False`, it returns the index of the first occurrence of the item (or raises an exception if the item is not in the list).

If the data in the list is not structured in some way, there is really only one way to implement `in` and `index()`: a systematic search through the items in the list, whether from index 0 and up, from index -1 and down, or something equivalent. This type of search is called *linear search*. Assuming the search is done from index 0 and up, linear search would look at 15 elements in the list to find 45 and *all of them* to find that 75 is not in the list.

A linear search may need to look at every item in the list. Its run time, in the worst case, is thus proportional to the size of the list. If the data set is not structured and the data items cannot be compared, linear search is really the only way search can be done on a list.

Binary Search

If the data in the list is comparable, we can improve the search run time by sorting the list first. To illustrate this, we use the same list `lst` as used in linear search, but now sorted:

```
>>> lst.sort()
>>> lst
[2, 5, 11, 13, 16, 24, 28, 41, 42, 43, 45, 57, 72, 73, 89, 90, 99]
```

Suppose we are searching for the value of `target` in list `lst`. Linear search compares `target` with the item at index 0 of `lst`, then with the item at index 1, 2, 3, and so on. Suppose, instead, we start the search by comparing `target` with the item at index i, for some arbitrary index i of `lst`. Well, there are three possible outcomes:

- We are lucky: `lst[i] == target` is true, or
- `target < lst[i]` is true, or
- `target > lst[i]` is true.

Let's do an example. Suppose the value of `target` is 45 and we compare it with the item at index 5 (i.e., 24). It is clear that the third outcome, `target > lst[i]`, applies in this case. Because list `lst` is sorted, this tells us that `target` cannot possibly be to the left of 24, that is, in sublist `lst[0:5]`. Therefore, we should continue our search for `target` to the right of 24 (i.e., in sublist `lst[6:17]`), as illustrated in Figure 10.13.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	5	11	13	16	$\boxed{24}$	28	41	42	43	45	57	72	73	89	90	99
						28	41	42	43	45	57	72	73	89	90	99

Figure 10.13 Binary search. By comparing 45, the value of `target`, with the item at index 5 of `lst`, we have reduced the search space to the sublist `lst[6:]`.

The main insight we just made is this: With just one comparison, between `target` and `list[5]`, we have reduced our search space from 17 list items to 11. (In linear search, a comparison reduces the search space by just 1.) Now we should ask ourselves whether a different comparison would reduce the search space even further.

In a sense, the outcome `target > lst[5]` was unlucky: `target` turns out to be in the larger of `lst[0:5]` (with 5 items) and `lst[6:17]` (with 11 items). To reduce the role of luck, we could ensure that both sublists are about the same size. We can achieve that by comparing `target` to 42—that is, the item in the middle of the list (also called the *median*).

The insights we just developed are the basis of a search technique called *binary search*. Given a list and a target, binary search returns the index of the target in the list, or -1 if the target is not in the list.

Binary search is easy to implement recursively. The base case is when the list `lst` is empty: `target` cannot possibly be in it, and we return -1. Otherwise, we compare `target` with the list median. Depending on the outcome of the comparison, we are either done or continue the search, recursively, on a sublist of `lst`.

We implement binary search as the recursive function `search()`. Because recursive calls will be made on sublists `lst[i:j]` of the original list `lst`, the function `search()` should take, as input, not just `lst` and `target` but also indices `i` and `j`:

Module: ch10.py

```
def search(lst, target, i, j):
    '''attempts to find target in sorted sublist lst[i:j];
       index of target is returned if found, -1 otherwise'''
    if i == j:                       # base case: empty list
        return -1                    # target cannot be in list

    mid = (i+j)//2                   # index of median of l[i:j]

    if lst[mid] == target:           # target is the median
        return mid
    if target < lst[mid]:            # search left of median
        return search(lst, target, i, mid)
    else:                            # search right of median
        return search(lst, target, mid+1, j)
```

To start the search for `target` in `lst`, indices 0 and `len(lst)` should be given:

```
>>> target = 45
>>> search(lst, target, 0, len(lst))
10
```

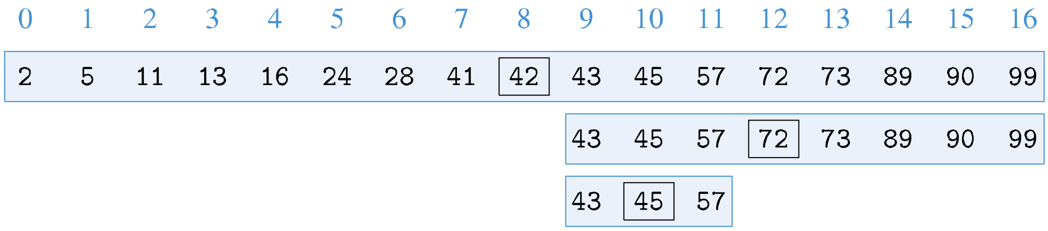


Figure 10.14 Binary search. The search for 45 starts in list `lst[0:17]`. After 45 is compared to the list median (42), the search continues in sublist `lst[9:17]`. After 45 is compared to this list's median (72), the search continues in `lst[9:12]`. Since 45 is the median of `lst[9:12]`, the search ends.

Figure 10.14 illustrates the execution of this search.

Linear versus Binary Search

To convince ourselves that binary search is, on average, much faster than linear search, we perform an experiment. Using the `timingAnalysis()` application we developed in the last section, we compare the performance of our function `search()` and the built-in list method `index()`. To do this, we develop functions `binary()` and `linear()` that pick a random item in the input list and call `search()` or invoke method `index()`, respectively, to find the item:

Module: ch10.py

```
def binary(lst):
    'chooses item in list lst at random and runs search() on it'
    target=random.choice(lst)
    return search(lst, target, 0, len(lst))

def linear(lst):
    'choose item in list lst at random and runs index() on it'
    target=random.choice(lst)
    return lst.index(target)
```

The list `lst` of size `n` we will use is a random sample of `n` numbers in the range from 0 to $2n - 1$.

Module: ch10.py

```
def buildInput(n):
    'returns a random sample of n numbers in range [0, 2n)'
    lst = random.sample(range(2*n), n)
    lst.sort()
    return lst
```

Here are the results:

```
>>> timingAnalysis(linear, 200000, 1000000, 200000, 20)
Run time of linear(200000) is 0.0046095
Run time of linear(400000) is 0.0091411
Run time of linear(600000) is 0.0145864
Run time of linear(800000) is 0.0184283
>>> timingAnalysis(binary, 200000, 1000000, 200000, 20)
Run time of binary(200000) is 0.0000681
Run time of binary(400000) is 0.0000762
Run time of binary(600000) is 0.0000943
Run time of binary(800000) is 0.0000933
```

It is clear that binary search is much faster and the run time of linear search grows proportionally with the list size. The interesting thing about the run time of binary search is that it does not seem to be increasing much. Why is that?

Whereas linear search may end up looking at every item in the list, binary search will look at far fewer list items. To see this, recall our insight that with every binary search comparison, the search space decreases by more than a half. Of course, when the search space becomes of size 1 or less, the search is over. The number of binary search comparisons in a list of size n is bounded by this value: the number of times we can halve n division before it becomes 1. In equation form, it is the value of x in

$$\frac{n}{2^x} = 1$$

The solution to this equation is $x = \log_2 n$, the logarithm base two of n. This function does indeed grow very slowly as n increases.

In the remainder of this section we look at several other fundamental search-like problems and analyze different approaches to solving them.

Uniqueness Testing

We consider this problem: Given a list, is every item in it unique? One natural way to solve this problem is to iterate over the list and for each list item check whether the item appears more than once in the list. Function `dup1` implements this idea:

Module: ch10.py

```
def dup1(lst):
    'returns True if list lst has duplicates, False otherwise'
    for item in lst:
        if lst.count(item) > 1:
            return True
    return False
```

The list method `count()`, just like the `in` operator and the `index` method, must perform a linear search through the list to count all occurrences of a target item. So, in `duplicates1()`, linear search is performed for every list item. Can we do better?

What if we sorted the list first? The benefit of doing this is that duplicate items will be next to each other in the sorted list. Therefore, to find out whether there are duplicates, all we need to do is compare every item with the item before it:

Module: ch10.py

```
def dup2(lst):
    'returns True if list lst has duplicates, False otherwise'
    lst.sort()
    for index in range(1, len(lst)):
        if lst[index] == lst[index-1]:
            return True
    return False
```

The advantage of this approach is that it does only one pass through the list. Of course, there is a cost to this approach: We have to sort the list first.

In Chapter 6, we saw that dictionaries and sets can be useful to check whether a list contains duplicates. Functions `dup3()` and `dup4()` use a dictionary or a set, respectively, to check whether the input list contains duplicates:

Module: ch10.py

```
def dup3(lst):
    'returns True if list lst has duplicates, False otherwise'
    s = set()
    for item in lst:
        if item in s:
            return False
        else:
            s.add(item)
    return True

def dup4(lst):
    'returns True if list lst has duplicates, False otherwise'
    return len(lst) != len(set(lst))
```


We leave the analysis of these four functions as an exercise.

Practice Problem 10.9

Using an experiment, analyze the run time of functions `dup1()`, `dup2()`, `dup3()`, and `dup4()`. You should test each function on 10 lists of size 2000, 4000, 6000, and 8000 obtained from:

```
import random
def buildInput(n):
    'returns a list of n random integers in range [0, n**2)'
    res = []
    for i in range(n):
        res.append(random.choice(range(n**2)))
    return res
```

Note that the list returned by this function is obtained by repeatedly choosing n numbers in the range from 0 to $n^2 - 1$ and may or may not contain duplicates. When done, comment on the results.

Selecting the *k*th Largest (Smallest) Item

Finding the largest (or smallest) item in an unsorted list is best done with a linear search. Finding the second, or third, largest (or smallest) kth smallest can be also done with a linear search, though not as simply. Finding the kth largest (or smallest) item for large k can easily be done by sorting the list first. (There are more efficient ways to do this, but they are beyond the scope of this text.) Here is a function that returns the kth smallest value in a list:

Module: ch10.py

```
def kthsmallest(lst, k):
    'returns kth smallest item in list lst'
    lst.sort()
    return lst[k-1]
```

Computing the Most Frequently Occurring Item

The problem we consider next is searching for the most frequently occurring item in a list. We actually know how to do this, and more: In Chapter 6, we saw how dictionaries can be used to compute the frequency of *all* items in a sequence. However, if all we want is to find the most frequent item, using a dictionary is overkill and a waste of memory space.

We have seen that by sorting a list, all the duplicate items will be next to each other. If we iterate through the sorted list, we can count the length of each sequence of duplicates and keep track of the longest. Here is the implementation of this idea:

Module: ch10.py

```
def frequent(lst):
    '''returns most frequently occurring item
       in non-empty list lst'''
    lst.sort()                 # first sort list

    currentLen = 1             # length of current sequence
    longestLen = 1             # length of longest sequence
    mostFreq   = lst[0]        # item with longest sequence

    for i in range(1, len(lst)):
        # compare current item with previous
        if lst[i] == lst[i-1]: # if equal
            # current sequence continues
            currentLen+=1

        else:                  # if not equal
            # update longest sequence if necessary
            if currentLen > longestLen: # if sequence that ended
                                        # is longest so far
                longestLen = currentLen # store its length
                mostFreq   = lst[i-1]   # and the item
            # new sequence starts
            currentLen = 1

    return mostFreq
```

Practice Problem 10.10

Implement function `frequent2()` that uses a dictionary to compute the frequency of every item in the input list and returns the item that occurs the most frequently. Then perform an experiment and compare the run times of `frequent()` and `frequent2()` on a list obtained using the `buildInput()` function defined in Practice Problem 10.9.

Case Study: Tower of Hanoi

In Case Study CS.10, we consider the Tower of Hanoi problem, the classic example of a problem easily solved using recursion. We also use the opportunity to develop a visual application by developing new classes and using object-oriented programming techniques.

Chapter Summary

The focus of this chapter is recursion and the process of developing a recursive function that solves a problem. The chapter also introduces formal run time analysis of programs and applies it to various search problems.

Recursion is a fundamental problem-solving technique that can be applied to problems whose solution can be constructed from solutions of "easier" versions of the problem. Recursive functions are often far simpler to describe (i.e., implement) than nonrecursive solutions for the same problem because they leverage operating system resources, in particular the program stack.

In this chapter, we devolop recursive functions for a variety of problems, such as the visual display of fractals and the search for viruses in the files of a filesystem. The main goal of the exposition, however, is to make explicit how to do recursive thinking, a way to approach problems that leads to recursive solutions.

In some instances, recursive thinking offers insights that lead to solutions that are more efficient than the obvious or original solution. In other instances, it will lead to a solution that is far worse. We introduce run time analysis of programs as a way to quantify and compare the execution times of various programs. Run time analysis is not limited to recursive functions, of course, and we use it to analyze various search problems as well.

Solutions to Practice Problems

10.1 The function `reverse()` is obtained by modifying function `vertical()` (and renaming it, of course). Note that function `vertical()` prints the last digit after printing all but the last digit. Function `reverse()` should just do the opposite:

```
def reverse(n):
    'prints digits of n vertically starting with low-order digit'
    if n < 10:          # base case: one-digit number
        print(n)
    else:               # n has at least 2 digits
       print(n%10)      # print last digit of n
       reverse(n//10)   # recursively print in reverse all but
                        # the last digit
```

10.2 In the base case, when $n = 0$, just `'Hurray!!!'` should be printed. When $n > 0$, we know that at least one `'Hip'` should be printed, which we do. That means that $n - 1$ strings `'Hip'` and then `'Hurray!!!'` remain to be printed. That is exactly what recursive call `cheers(n-1)` will achieve.

```
def cheers(n):
    'prints cheer'
    if n == 0:
        print('Hurray!!!')
    else: # n > 0
        print('Hip', end=' ')
        cheers(n-1)
```

10.3 By the definition of the factorial function $n!$, the base case of the recursion is $n = 0$ or $n = 1$. In those cases, the function `factorial()` should return 1. For $n > 1$, the recursive definition of $n!$ suggests that function `factorial()` should return `n * factorial(n-1)`:

```
def factorial(n):
    'returns n!'
    if n == 0:              # base case
        return 1
    return factorial(n-1) * n # recursive step when n > 0
```

10.4 In the base case, when $n = 0$, nothing is printed. If $n > 0$, note that the output of `pattern2(n)` consists of the output of `pattern2(n-1)`, followed by a row of `n` stars, followed by the output of `pattern2(n-1)`:

```
def pattern2(n):
    'prints the nth pattern'
    if n > 0:
        pattern2(n-1)  # prints pattern2(n-1)
        print(n * '*')   # print n stars
        pattern2(n-1)  # prints pattern2(n-1)
```

10.5 As Figure 10.15 of `snowflake(4)` illustrates, a snowflake pattern consists of three patterns `koch(3)` drawn along the sides of an equilateral triangle.

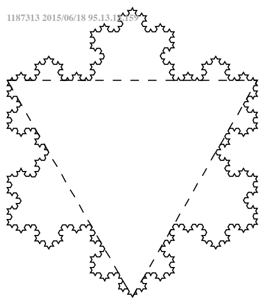


Figure 10.15 The pattern `snowflake(4)`.

To draw the pattern `snowflake(n)`, all we need to do is draw pattern `koch(n)`, turn right 120 degrees, draw `koch(n)` again, turn right 120 degrees, and draw `koch(n)` one last time.

```
def drawSnowflake(n):
    'draws nth snowflake curve using function koch() 3 times'
    s = Screen()
    t = Turtle()
    directions = koch(n)

    for i in range(3):
        for move in directions: # draw koch(n)
            if move == 'F':
                t.fd(300/3**n)
            if move == 'L':
                t.lt(60)
            if move == 'R':
                t.rt(120)
        t.rt(120)               # turn right 120 degrees
```

10.6 If the list is empty, the returned value should be `False`; otherwise, `True` should be returned if and only if `lst[:-1]` contains a negative number or `lst[-1]` is negative:

```
def recNeg(lst):
    '''returns True if some number in list lst is negative,
       False otherwise'''
    if len(lst) == 0:
        return False
    return recNeg(lst[:-1]) or lst[-1] < 0
```

10.7 The buil-in function `sum()` should be applied to every item (row) of `table`:

```
>>> table = [[1,2,3], [4,5,6]]
>>> recMap(table, sum)
[6, 15]
```

10.8 After running the tests, you will note that the run times of `power2()` are significantly worse than the run times of `pow2()` and `rpow2()` which are very, very close. It seems that the built-in operator `**` uses an approach that is equivalent to our recursive solution.

10.9 Even though `dup2()` has the additional sorting step, you will note that `dup1()` is much slower. This means that the multiple linear searches approach of `dup1()` is very inefficient. The dictionary and set approaches in `dup3` and `dup4()` did best, with the set approach winning overall. The one issue with these last two approaches is that they both use an extra container, so they take up more memory space.

10.10 You can use the function `frequency` from Chapter 6 to implement `freqent2()`.

Exercises

10.11 Using Figure 10.1 as a model, draw all the steps that occur during the execution of `countdown(3)`, including the state of the program stack at the beginning and end of every recursive call.

10.12 Swap statements in lines 6 and 7 of function `countdown()` to create function `countdown2()`. Explain how it differs from `countdown()`.

10.13 Using Figure 10.1 as a model, draw all the steps that occur during the execution of `countdown2(3)`, where `countdown2()` is the function from Exercise 10.12.

10.14 Modify the function `countdown()` so it exhibits this behavior:

```
>>> countdown3(5)
5
4
3
    BOOOM!!!
    Scared you...
2
1
Blastoff!!!
```

10.15 Using Figure 10.1 as a model, draw all the steps that occur during the execution of `pattern(2)`, including the state of the program stack at the beginning and end of every recursive call.

10.16 The recursive formula for computing the number of ways of choosing k items out of a set of n items, denoted $C(n, k)$, is:

$$C(n,k) = \begin{cases} 1 & \text{if } k = 0 \\ 0 & \text{if } n < k \\ C(n-1, k-1) + C(n-1, k) & \text{otherwise} \end{cases}$$

The first case says there is one way to choose no item; the second says that there is no way to choose more items than available in the set. The last case separates the counting of sets of k items containing the last set item and the counting of sets of k items *not* containing the last set item. Write a recursive function `combinations()` that computes $C(n, k)$ using this recursive formula.

```
>>> combinations(2, 1)
0
>>> combinations(1, 2)
2
>>> combinations(2, 5)
10
```

10.17 Just as we did for the function `rpower()`, modify function `rfib()` so that it counts the number of recursive calls made. Then use this function to count the number of calls made for $n = 10, 20, 30$.

Problems

10.18 Write a recursive method `silly()` that takes one nonnegative integer `n` as input and then prints `n` question marks, followed by `n` exclamation points. Your program should use no loops.

```
>>> silly(0)
>>> silly(1)
* !
>>> silly(10)
* * * * * * * * * * ! ! ! ! ! ! ! ! ! !
```

10.19 Write a recursive method `numOnes()` that takes a nonnegative integer n as input and returns the number of 1s in the binary representation of n. Use the fact that this is equal to the number of 1s in the representation of $n//2$ (integer division), plus 1 if n is odd.

```
>>> numOnes(0)
0
>>> numOnes(1)
1
>>> numOnes(14)
3
```

10.20 In Chapter 5 we developed Euclid's Greatest Common Divisor (GCD) algorithm using iteration. Euclid's algorithm is naturally described recursively:

$$gcd(a,b) = \begin{cases} a & \text{if } b = 0 \\ gcd(b, a\%b) & \text{otherwise} \end{cases}$$

Using this recursive definition, implement recursive function `rgcd()` that takes two non-negative numbers a and b, with $a > b$, and returns the GCD of a and b:

```
>>> rgcd(3,0)
3
>>> rgcd(18,12)
6
```

10.21 Write a method `rem()` that takes as input a list containing, possibly, duplicate values and returns a copy of the list in which one copy of every duplicate value was removed.

```
>>> rem([4])
[]
>>> rem([4, 4])
[4]
>>> rem([4, 1, 3, 2])
[]
>>> rem([2, 4, 2, 4, 4])
[2, 4, 4]
```

10.22 You're visiting your hometown and are planning to stay at a friend's house. It just happens that all your friends live on the same street. In order to be efficient, you would like to stay at the house of a friend who is in a central location in the following sense: the same number of friends, within 1, live in either direction. If two friends' houses satisfy this criterion, choose the friend with the smaller street address.

Write function `address()` that takes a list of street numbers and returns the street number you should stay at.

```
>>> address([2, 1, 8, 5, 9])
5
>>> address([2, 1, 8, 5])
2
>>> address([1, 1, 1, 2, 3, 3, 4, 4, 4, 5])
3
```

10.23 Develop a recursive function `tough()` that takes two nonnegative integer arguments and outputs a pattern as shown below. *Hint:* The first argument represents the indentation of the pattern, whereas the second argument—always a power of 2—indicates the number of "*"s in the longest line of "*"s in the pattern.

```
>>> f(0, 0)
>>> f(0, 1)
 *
>>> f(0, 2)
 *
 **
  *
```

```
>>> f(0, 4)
 *
 **
  *
 ****
   *
   **
    *
```

10.24 Write a recursive method `base()` that takes a nonnegative integer n and a positive integer $1 < b < 10$ and *prints* the base-b representation of integer n.

```
>>> base(0, 2)
0
>>> base(1, 2)
1
>>> base(10, 2)
1010
>>> base(10, 3)
1 0 1
```

10.25 Implement function `permutations()` that takes a list `lst` as input and returns a list of all permutations of `lst` (so the returned value is a list of lists). Do this recursively as follows: If the input list `lst` is of size 1 or 0, just return a list *containing* list `lst`. Otherwise, make a recursive call on the sublist `lst[1:]` to obtain the list of all permutations of all items of `lst` except `lst[0]`. Then, for each such permutation (i.e., list) `perm`, generate permutations of `lst` by inserting `lst[0]` into all possible positions of `perm`.

```
>>> permutations([1, 2])
[[1, 2], [2, 1]]
>>> permutations([1, 2, 3])
[[1, 2, 3], [2, 1, 3], [2, 3, 1], [1, 3, 2], [3, 1, 2], [3, 2, 1]]
>>> permutations([1, 2, 3, 4])
[[1, 2, 3, 4], [2, 1, 3, 4], [2, 3, 1, 4], [2, 3, 4, 1],
[1, 3, 2, 4], [3, 1, 2, 4], [3, 2, 1, 4], [3, 2, 4, 1],
[1, 3, 4, 2], [3, 1, 4, 2], [3, 4, 1, 2], [3, 4, 2, 1],
[1, 2, 4, 3], [2, 1, 4, 3], [2, 4, 1, 3], [2, 4, 3, 1],
[1, 4, 2, 3], [4, 1, 2, 3], [4, 2, 1, 3], [4, 2, 3, 1],
[1, 4, 3, 2], [4, 1, 3, 2], [4, 3, 1, 2], [4, 3, 2, 1]]
```

10.26 Implement function `anagrams()` that computes anagrams of a given word. An anagram of word A is a word B that can be formed by rearranging the letters of A. For example, the word pot is an anagram of the word top. Your function will take as input the name of a file of words and as well as a word, and print all the words in the file that are anagrams of the input word. In the next examples, use file `words.txt` as your file of words.

File: words.txt

```
>>> anagrams('words.txt', 'trace')
crate
cater
react
```

10.27 Write a function `pairs1()` that takes as inputs a list of integers and an integer target value and returns `True` if there are two numbers in the list that add up to the target and `False` otherwise. Your implementation should use the nested loop pattern and check all pairs of numbers in the list.

```
>>> pairs1([4, 1, 9, 3, 5], 13)
True
>>> pairs1([4, 1, 9, 3, 5], 11)
False
```

When done, reimplement the function so that it sorts the list first and then *efficiently* searches for the pair. Analyze the run time of both implementations using the `timingAnalysis()` app. (Function `buildInput()` should generate a tuple containing the list and the target.)

10.28 In this problem, you will develop a function that crawls through "linked" files. Every file visited by the crawler will contain zero or more links, one per line, to other files and nothing else. A link to a file is just the name of the file. For example, the content of file `file0.txt` is:

```
file1.txt
file2.txt
```

The first line represents the link o file `file1.txt` and the second is a link to `file2.txt`.

Implement recursive method `crawl()` that takes as input a file name (as a string), prints a message saying the file is being visited, opens the file, reads each link, and recursively continues the crawl on each link. The below example uses a set of files packaged in archive `files.zip`.

File: files.zip

```
>>> crawl('file0.txt')
Visiting  file0.txt
Visiting  file1.txt
Visiting  file3.txt
Visiting  file4.txt
Visiting  file8.txt
Visiting  file9.txt
Visiting  file2.txt
Visiting  file5.txt
Visiting  file6.txt
Visiting  file7.txt
```

10.29 Pascal's triangle is an infinite two-dimensional pattern of numbers whose first five lines are illustrated in Figure 10.16. The first line, line 0, contains just 1. All other lines start and end with a 1 too. The other numbers in those lines are obtained using this rule: The number at position i is the sum of the numbers in position $i - 1$ and i in the previous line.

Figure 10.16 Pascal's triangle. Only the first five lines of Pascal's triangle are shown.

```
            1
         1     1
      1     2     1
   1     3     3     1
1     4     6     4     1
```

Implement recursive function `pascalLine()` that takes a nonnegative integer n as input and returns a list containing the sequence of numbers appearing in the nth line of Pascal's triangle.

```
>>> pascalLine(0)
[1]
>>> pascalLine(2)
[1, 2, 1]
>>> pascalLine(3)
[1, 3, 3, 1]
>>> pascalLine(4)
[1, 4, 6, 4, 1]
```

10.30 Implement recursive function `traverse()` that takes as input a pathname of a folder (as a string) and an integer `d` and prints on the screen the pathname of every file and subfolder contained in the folder, directly or indirectly. The file and subfolder pathnames should be output with an indentation that is proportional to their depth with respect to the topmost folder. The next example illustrates the execution of `traverse()` on folder `'test'` shown in Figure 10.8.

File: test.zip

```
>>> traverse('test', 0)
test/fileA.txt
test/folder1
  test/folder1/fileB.txt
  test/folder1/fileC.txt
  test/folder1/folder11
    test/folder1/folder11/fileD.txt
test/folder2
  test/folder2/fileD.txt
  test/folder2/fileE.txt
```


10.31 Implement function `search()` that takes as input the name of a file and the pathname of a folder and searches for the file in the folder and any folder contained in it, directly or indirectly. The function should return the pathname of the file, if found; otherwise, `None` should be returned. The below example illustrates the execution of `search('file.txt', 'test')` from the parent folder of folder `'test'` shown in Figure 10.8.

File: test.zip

```
>>> search('fileE.txt', 'test')
  test/folder2/fileE.txt
```

10.32 The Lévy curves are fractal graphical patterns that can be defined recursively. Like the Koch curves, for every nonnegative integer $n > 0$, the Lévy curve L_n can be defined in terms of Lévy curve L_{n-1}; Lévy curve L_0 is just a straight line. Figure 10.17 shows the Lévy curve L_8.

(a) Find more information about the Lévy curve online and use it to implement recursive function `levy()` that takes a nonnegative integer n and returns turtle instructions encoded with letters L, R and, F, where L means "rotate left 45 degrees," R means "rotate right 90 degrees," and F means "go forward."

```
>>> levy(0)
```

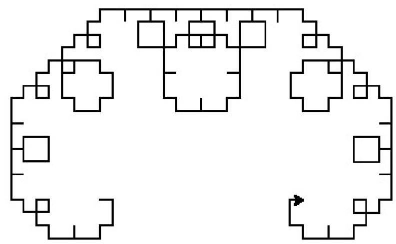


Figure 10.17 Lévy curve L_8.

```
>>> levy(1)
'LFRFL'
>>> levy(2)
'LLFRFLRLFRFLL'
```

(b) Implement function `drawLevy()`) so that it takes nonnegative integer n as input and draws the Lévy curve L_n using instructions obtained from function `levy()`.

10.33 In the simple coin game you are given an initial number of coins and then, in every iteration of the game, you are required to get rid of a certain number of coins using one of the following rules. If n is the number of coins you have then:

- If n is divisible by 10, then you may give back 9 coins.
- If n is even, then you may give back exactly $n/2 - 1$ coins.
- If n is divisible by 3, then you may give back 7 coins.
- If n is divisible by 4, then you may give back 6 coins.

If none of the rules can be applied, you lose. The goal of the game is to end up with exactly 8 coins.

Note that more than one rule may be applied for some values of n. If n is 20, for example, rule 1 could be applied to end up with 11 coins. Since no rule can be applied to 11 coins, you would lose the game. Alternatively, rule 4 could be applied to end up with 14 coins, and then rule 2 could be applied to end up with 8 coins and win the game.

Implement a function `coins()` that takes as input the initial number of coins and returns `True` if there is some way to play the game and end up with 8 coins. The function should return `False` only if there is no way to win.

```
>>> coins(7)
False
>>> coins(8)
True
>>> coins(20)
True
>>> coins(66)
False
>>> coins(99)
True
```

10.34 Using linear recursion, implement function `recDup()` that takes a list as input and returns a copy of it in which every list item has been duplicated.

```
>>> recDup(['ant', 'bat', 'cat', 'dog'])
[
```

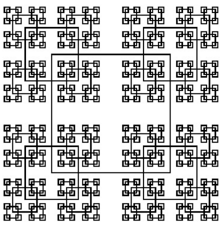
10.35 Using linear recursion, implement function `recReverse()` that takes a list as input and returns a reversed copy of the list.

```
>>> lst = [1, 3, 5, 7, 9]
>>> recReverse(lst)
[9, 7, 5, 3, 1]
```

10.36 Using linear recursion, implement function `recSplit()` that takes, as input, a list `lst` and a nonnegative integer `i` no greater than the size of `lst`. The function should split the list into two parts so that the second part contains exactly the last `i` items of the list. The function should return a list containing the two parts.

```
>>> recSplit([1, 2, 3, 4, 5, 6, 7], 3)
[[1, 2, 3, 4], [5, 6, 7]]
```

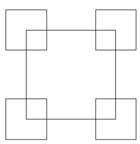
10.37 Implement a function that draws patterns of squares like this:



(a) To get started, first implement function `square()` that takes as input a `Turtle` object and three integers x, y, and s and makes the `Turtle` object trace a square of side length s centered at coordinates (x, y).

```
>>> from turtle import Screen, Turtle
>>> s = Screen()
>>> t = Turtle()
>>> t.pensize(2)
>>> square(t, 0, 0, 200)    # draws the square
```

(b) Now implement recursive function `squares()` that takes the same inputs as function `square` plus an integer n and draws a pattern of squares. When $n = 0$, nothing is drawn. When $n = 1$, the same square drawn by `square(t, 0, 0, 200)` is drawn. When $n = 2$ the pattern is:



Each of the four small squares is centered at an endpoint of the large square and has length $1/2.2$ of the original square. When $n = 3$, the pattern is: