Clouds are not spheres, mountains are not cones, coastlines are not circles, and bark is not smooth, nor does lightning travel in a straight line.

Benoît Mandelbrot The Fractal Geometry of Nature (1983)

Though this be madness, yet there is method in't.

William Shakespeare Hamlet (Act II, Scene II)

H AVE you ever noticed, while laying on your back under a tree, that each branch of the tree resembles the tree itself? If you could take any branch, from the smallest to the largest, and place it upright in the ground, it could probably be mistaken for a smaller tree. This phenomenon, called *self-similarity*, is widespread in nature. There are also computational problems that, in a more abstract way, exhibit self-similarity. In this chapter, we will discuss a computational technique, called *recursion*, that we can use to elegantly solve problems that exhibit this property. As the second quotation above suggests, recursion may seem foreign at first, but it really is quite natural and just takes some practice to master.

# 9.1 FRACTALS

Nature is not geometric, at least not in a traditional sense. Instead, natural structures are complex and not easily described. But many natural phenomena do share a common characteristic: if you zoom in on any part, that part resembles the whole. For example, consider the images in Figure 9.1. In the bottom two images, we can



Figure 9.1 Fractal patterns in nature. Clockwise from top left: a nautilus shell [69], the coastline of Norway [70], a closeup of a leaf [71], branches of a tree, a rock outcropping, and lightning [72]. The insets in the bottom two images show how smaller parts resemble the whole.



Figure 9.2 A tree produced by tree(george, 100, 4). The center figure illustrates what is drawn by each numbered step of the function. The figure on the right illustrates the self-similarity in the tree.

see that if we zoom in on parts of the rock face and tree, these parts resemble the whole. (In nature, the resemblance is not always exact, of course.) These kinds of structures are called *fractals*, a term coined by mathematician Benoît Mandelbrot, who developed the first theory of fractal geometry.

Trees

An algorithm for creating a fractal shape is *recursive*, meaning that it invokes itself on smaller and smaller scales. Let's consider the example of the simple tree shown on the left side of Figure 9.2. Notice that each of the two main branches of the tree is a smaller tree with the same structure as the whole. As illustrated in the center of Figure 9.2, to create this fractal tree, we first draw a trunk and then, for each branch, we draw two smaller trees at 30-degree angles using the same algorithm. Each of these smaller trees is composed of a trunk and two yet smaller trees, again drawn with the same tree-drawing algorithm. This process could theoretically continue forever, producing a tree with infinite complexity. In reality, however, the process eventually stops by invoking a non-recursive **base case**. The base case in Figure 9.2 is a "tree" that consists of only a single line segment.

This recursive structure is shown more precisely on the right side of Figure 9.2. The *depth* of the tree is a measure of its distance from the base case. The overall tree in Figure 9.2 has depth 4 and each of its two main branches is a tree with depth 3. Each of the two depth 3 trees is composed of two depth 2 trees. Finally, each of the four depth 2 trees is composed of two depth 1 trees, each of which is only a line segment.

The following tree function uses turtle graphics to draw this tree.

```
import turtle
  def tree(tortoise, length, depth):
      """Recursively draw a tree.
      Parameters:
          tortoise: a Turtle object
          length: the length of the trunk
          depth:
                     the desired depth of recursion
      Return value: None
      .....
      if depth <= 1:
                                         # the base case
          tortoise.forward(length)
          tortoise.backward(length)
      else:
                                         # the recursive case
          tortoise.forward(length)
1
           tortoise.left(30)
2
           tree(tortoise, length * (2 / 3), depth - 1)
3
           tortoise.right(60)
4
           tree(tortoise, length * (2 / 3), depth - 1)
5
6
           tortoise.left(30)
           tortoise.backward(length)
7
  def main():
      george = turtle.Turtle()
      george.left(90)
                                   # point north
      tree(george, 100, 4)
                                   # draw the tree
  main()
```

The initial statements in the main function initialize the turtle and orient it to the north. Then the tree function is called with tree(george, 100, 4). On lines 1–2, the turtle moves forward length units to draw the trunk, and then turns 30 degrees to the left. This is illustrated in the center of Figure 9.2; the numbers correspond to the line numbers in the function. Next, to draw the smaller tree, we call the tree function *recursively* on line 3 with two-thirds of the length, and a value of depth that is one less than what was passed in. The depth parameter controls how long we continue to draw smaller trees recursively. After the call to tree returns, the turtle turns 60 degrees to the right on line 4 to orient itself to draw the right tree. On line 5, we recursively call the tree function again with arguments that are identical to those on line 3. When that call returns, the turtle retraces its steps in lines 6–7 to return to the origin.

### 9.1 FRACTALS **369**



Figure 9.3 An illustration of the recursive calls in tree(george, 100, 4).

The case when depth is at most 1 is called the *base case* because it does not make a recursive call to the function; this is where the recursion stops.

**Reflection 9.1** Try running the tree-growing function with a variety of parameters. Also, try changing the turn angles and the amount length is shortened. Do you understand the results you observe?

Figure 9.3 illustrates the recursive function calls that are made by the tree function when length is 100 and depth is 3. The top box represents a call to the tree function with parameters tortoise = george, length = 100, and depth = 3. This function calls two instances of tree with length = 100 \* (2 / 3) = 66.67 and depth = 2. Then, each of these instances of tree calls two instances of tree with length = 66.67 \* (2 / 3) = 44.44 and depth = 1. When depth is 1, tree simply draws a line segment and returns.

**Reflection 9.2** The numbers on the lines in Figure 9.3 represent the order in which the recursive calls are made. Can you see why that is?

**Reflection 9.3** What would happen if we removed the base case from the algorithm by deleting the first four statements, so that the line numbered 1 was always the first statement executed?

Without the base case, the function would continue to make recursive calls forever! Therefore, the base case is extremely important to the correctness of the algorithm.

## Snowflakes

One of the most famous fractal shapes is the *Koch curve*, named after Swedish mathematician Helge von Koch. A Koch curve begins with a single line segment with length  $\ell$ . Then, as shown in Figure 9.4(b), that line segment is divided into three equal parts, each with length  $\ell/3$ . The middle part of this divided segment is replaced by two sides of an equilateral triangle (with side length  $\ell/3$ ). Next, each of the four line segments of length  $\ell/3$  is divided in the same way, with the middle



Figure 9.4 The Koch curve at depths 0, 1, 2, and 3.

segment again replaced by two sides of an equilateral triangle with side length  $\ell/9$ , etc., as shown in Figures 9.4(c)–(d). As with the tree above, this process could theoretically go on forever, producing an infinitely intricate pattern.

Notice that, like the tree, this shape exhibits self-similarity; each "side" of the Koch curve is itself a Koch curve with smaller depth. The Koch curve in Figure 9.4(b) with depth 1 consists of four smaller Koch curves with depth 0 and length  $\ell/3$ . Likewise, the Koch curve with depth 2 in Figure 9.4(c) consists of four smaller Koch curves with depth 1, and the Koch curve with depth 3 in Figure 9.4(d) consists of four smaller Koch curves with depth 2.

We can use our understanding of this self-similarity to write an algorithm to produce a Koch curve with any desired overall length and depth:

Algorithm Koch Curve					
Input: overall <i>length</i> and the <i>depth</i>					
1	if <i>depth</i> is 0:				
2	draw a line with length equal to <i>length</i>				
3	else:				
4	draw a Koch curve with length $=$ <i>length</i> / 3 and depth $=$ <i>depth</i> $-$ 1				
5	turn left 60 degrees				
6	draw another Koch curve with length $= length / 3$ and depth $= depth - 1$				
7	turn right 120 degrees				
8	draw another Koch curve with length = <i>length</i> / 3 and depth = <i>depth</i> - 1				
9	turn left 60 degrees				
10	draw another Koch curve with length $= length / 3$ and depth $= depth - 1$				

The base case occurs when the depth is zero, in which case we simply draw a line. Otherwise, the algorithm runs through the steps we followed above. But notice that steps 4, 6, 8, and 10 are actually recursively calling upon the algorithm itself to draw smaller Koch curves. So the algorithm is more correctly written as follows.

## 9.1 FRACTALS **371**



**Reflection 9.4** Follow the algorithm above to draw (on paper) a Koch curve with depth 1. Then follow the algorithm again to draw one with depth 2.

This algorithm can be directly translated into Python. The only additional thing we need is a turtle to do the drawing.

```
def koch(tortoise, length, depth):
    """Recursively draw a Koch curve.
    Parameters:
        tortoise: a Turtle object
        length: the length of a line segment
        depth:
                  the desired depth of recursion
    Return value: None
    11 11 11
    if depth == 0:
                                                # base case
        tortoise.forward(length)
    else:
                                                # recursive case
        koch(tortoise, length / 3, depth - 1)
        tortoise.left(60)
        koch(tortoise, length / 3, depth - 1)
        tortoise.right(120)
        koch(tortoise, length / 3, depth - 1)
        tortoise.left(60)
        koch(tortoise, length / 3, depth - 1)
```

**Reflection 9.5** Write a main function that creates a Turtle object and calls this function. Experiment by calling koch with different values of length and depth.

We can attach three Koch curves at 120-degree angles to produce an intricate snowflake shape like that in Figure 9.5.



Figure 9.5 A three-sided Koch snowflake.

**Reflection 9.6** Look carefully at Figure 9.5. Can you see where the three individual Koch curves are connected?

The following function draws this Koch snowflake.

```
def kochSnowFlake(tortoise, length, depth):
    """Recursively draw a Koch snowflake.

Parameters:
    tortoise: a Turtle object
    length: the length of a line segment
    depth: the desired depth of recursion

Return value: None
"""
for side in range(3):
    koch(tortoise, length, depth)
    tortoise.right(120)
```

**Reflection 9.7** Insert this function into the previous program and call it from main. Try making different snowflakes by increasing the number of sides (and decreasing the right turn angle).

Imagine a Koch snowflake made from *infinitely* recursive Koch curves. Paradoxically, while the area inside any Koch snowflake is clearly finite (because it is bounded), the length of its border is infinite! In fact, the distance between any two points on its border is infinite! To see this, notice that, at every stage in its construction, each line segment is replaced with four line segments that are one-third the length of the original. Therefore, the total length of that "side" increases by one-third. Since this happens infinitely often, the perimeter of the snowflake continues to grow forever.



Figure 9.6 Depths 0, 1, 2, and 3 of a quadratic Koch curve.



Figure 9.7 Depths 0, 1, 2, and 3 of a Sierpinski triangle.

## **Exercises**

- 9.1.1\* Modify the recursive tree-growing function so that it branches at random angles between 10 and 60 degrees (instead of 30 degrees) and it shrinks the trunk/branch length by a random fraction between 0.5 and 0.75. Do your new trees now look more "natural"?
- 9.1.2. The *quadratic Koch curve* is similar to the Koch curve, but it replaces the middle segment of each side with three sides of a square instead, as illustrated in Figure 9.6. Write a recursive function

```
quadkoch(tortoise, length, depth)
```

that draws the quadratic Koch curve with the given segment length and depth.

- 9.1.3. The following activities are recursive in the sense that each step can be considered a smaller version of the original activity. Describe how this is the case and how the "input" gets smaller each time. What is the base case of each operation below?
  - (a) evaluating an arithmetic expression like 7 + (15 3)/4
  - (b) the chain rule in calculus (if you have taken calculus)
  - (c) one hole of golf
  - (d) driving directions to some destination
- 9.1.4<sup>\*</sup> Generalize the Koch snowflake function with an additional parameter so that it can be used to draw a snowflake with any number of sides.
- 9.1.5. The Sierpinski triangle, depicted in Figure 9.7, is another famous fractal. The fractal at depth 0 is simply an equilateral triangle. The triangle at depth 1 is composed of three smaller triangles, as shown in Figure 9.7(b). (The larger outer triangle and the inner "upside down" triangle are indirect effects of the



Figure 9.8 Hilbert space-filling curves with depths 0, 1, 2, and 3.

positions of these three triangles.) At depth 2, each of these three triangles is replaced by three smaller triangles. And so on. Write a recursive function

sierpinski(tortoise, p1, p2, p3, depth)

that draws a Sierpinski triangle with the given depth. The triangle's three corners should be at coordinates p1, p2, and p3 (all tuples). It will be helpful to also write two smaller functions that you can call from sierpinski: one to draw a simple triangle, given the coordinates of its three corners, and one to compute the midpoint of a line segment.

9.1.6. The Hilbert space-filling curve, shown in Figure 9.8, is a fractal path that visits all of the cells in a square grid in such a way that close cells are visited close together in time. For example, the figure to the right shows how a depth 2 Hilbert curve visits the cells in an 8 × 8 grid.

	4

The following high-level algorithm draws a Hilbert curve.

Algorithm Hilbert Curve					
Input:	depth and mode				
1	if $depth > 0$ :				
2	turn 90 degrees to the right [left]				
3	HILBERT CURVE ( <i>depth</i> – 1, opposite <i>mode</i> )				
4	draw a line segment				
5	turn 90 degrees to the left [right]				
6	HILBERT CURVE (depth – 1, same mode)				
7	draw a line segment				
8	HILBERT CURVE (depth – 1, same mode)				
9	turn 90 degrees to the left [right]				
10	draw a line segment				
11	HILBERT CURVE (depth – 1, opposite mode)				
12	turn 90 degrees to the right [left]				
'					



Figure 9.9 Sierpinski carpets with depths 0, 1, and 2. (The gray bounding box shows the extent of the drawing area; it is not actually part of the fractal.)

In the base case, when the depth is 0, the algorithm draws nothing. In the recursive case, the algorithm can be in one of two different modes. In the first mode, lines 2 and 12 turn right, and lines 5 and 9 turn left. In the other mode, these directions are reversed (indicated in square brackets). Lines 3 and 11 make recursive calls that switch the mode. The algorithm assumes the turtle is initially pointing north. Each of the line segments is the same length (say 10).

Write a recursive function

#### hilbert(tortoise, mode, depth)

that draws a Hilbert space-filling curve with the given depth. The Boolean parameter mode indicates which mode the algorithm should draw in. (Think about how you can accommodate both drawing modes by changing the angle of the turns.)

9.1.7. A fractal pattern called the Sierpinski carpet is shown in Figure 9.9. At depth 0, it is simply a filled square one-third the width of the overall square space containing the fractal. At depth 1, this center square is surrounded by eight one-third size Sierpinski carpets with depth 0. At depth 2, the center square is surrounded by eight one-third size Sierpinski carpets with depth 1. Write a function

```
carpet(tortoise, upperLeft, width, depth)
```

that draws a Sierpinski carpet with the given depth. The parameter upperLeft refers to the coordinates of the upper left corner of the fractal and width refers to the overall width of the fractal.

9.1.8. Modify your Sierpinski carpet function from the last exercise so that it displays the color pattern shown in Figure 9.10.

# 9.2 RECURSION AND ITERATION

We typically only solve problems recursively when they obviously exhibit selfsimilarity or seem "naturally recursive," as with fractals. But recursion is not some obscure problem-solving technique. Although recursive algorithms may seem quite



Figure 9.10 Colorful Sierpinski carpets with depths 0, 1, and 2.

different from iterative algorithms, recursion and iteration are actually just two sides of the same coin. Every iterative algorithm can be written recursively, and vice versa. This realization may help take some of the mystery out of the technique.

Consider the problem of summing the numbers in a list. Of course, this is easily achieved iteratively with a for loop:

```
def sumList(data):
    """Compute the sum of the values in a list.
    Parameter:
        data: a list of numbers
    Return value: the sum of the values in the list
    """
    total = 0
    for value in data:
        total = total + value
    return total
```

Let's think about how we could achieve the same thing recursively. To solve a problem recursively, we need to think about how we could solve it if we had a solution to a smaller *subproblem*. A subproblem is the same as the original problem, but with only part of the original input.

In the case of summing the numbers in a list named data, a subproblem would be summing the numbers in a *slice* of data. Consider the following example:

data 
$$\longrightarrow [1, \underbrace{7, 3, 6}_{\text{data}[1:]}]$$

If we had the sum of the numbers in data[1:], i.e., sumList(data[1:]), then we could compute the value of sumList(data) by simply adding this sum to data[0]. In other words, sumList(data) is equal to data[0] plus the solution to the subproblem sumList(data[1:]). In terms of the example above, if

## 9.2 RECURSION AND ITERATION **377**



Figure 9.11 A representation of the function calls in the recursive sumList function. The red numbers indicate the order in which the events occur. The black numbers next to the dashed arrows are return values.

we knew that sumList([7, 3, 6]) returned 16, then we could easily find that sumList([1, 7, 3, 6]) is 1 + 16 = 17.

**Reflection 9.8** Will this work if data is empty?

Since there is no data[0] or data[1:] when data is empty, the method above will not work. But we can easily check for this case and return 0; this is the base case of the function. Putting these two parts together, we have the following recursive version of the function:

But does this actually work? It sure does. To see why, let's look at Figure 9.11. At the top of the diagram, in box (a), is a representation of a main function that calls sumList with the argument [1, 7, 3, 6]. The steps that follow are marked in red in the figure.

- Calling sumList creates an instance of the sumList function, represented in box (b), with the parameter data assigned the list [1, 7, 3, 6]. Since data is not an empty list, the function will return 1 + sumList([7, 3, 6]), the value enclosed in the gray box.
- 2. To evaluate the return value in box (b), we must call sumList again with the argument [7, 3, 6], resulting in another instance of the sumList function, represented in box (c). The instance of sumList in box (b) must wait to return its value until it receives the return value from box (c). The instance of sumList in box (c) will return 7 + sumList([3, 6]).
- Evaluating this return value requires that sumList be called again, resulting in the instance of sumList in box (d). This instance will return the result of 3 + sumList([6]).
- Calling sumList([6]) creates yet another instance of the sumList function, shown in box (e). This instance will return 6 + sumList([]).
- 5. The instance of sumList in box (e) calls sumList([]), creating the instance of sumList in box (f).
- 6. The value of the data parameter in box (f) is an empty list, so the if condition in sumList is True, invoking the base case. The base case immediately returns 0 to the instance of sumList that called it, in box (e).
- 7. Once the instance of sumList in box (e) receives 0 from sumList([]), it returns 6 + 0 = 6 to the instance of sumList in box (d).
- 8. Since the instance of sumList in box (d) now has a value for sumList([6]), it can return 3 + 6 = 9 to the instance of sumList in box (c).
- 9. Now that the instance of sumList in box (c) has a value for sumList([3, 6]), it returns 7 + 9 = 16 to the instance of sumList in box (b).
- 10. Finally, the instance of sumList in box (b) can return 1 + 16 = 17 to main.

Notice that the sequence of function calls, moving down the figure from (a) to (f), only ended because we eventually reached the *base case* in step 6, which caused the function to return without making another recursive call. This step initiated a process of moving back up through the previous function calls, allowing each one in turn to return their value to the instance of the function that called it.

Every recursive function must have a non-recursive base case, and each recursive call must get one step closer to the base case. This may sound familiar; it is very similar to the way we must think about while loops. Each iteration of a while loop must move one step closer to the loop condition becoming false.

## Solving a problem recursively

The following five questions generalize the process we followed to design the recursive sumList function. We will illustrate each question by summarizing how we answered it in the design of the sumList.

1. What does a subproblem look like?

A subproblem is to compute the sum of a slice of the list.

2. Suppose you could ask an all-knowing oracle for the solution to any subproblem (but not for the problem itself). Which subproblem solution would be the most useful for solving the original problem?

The most useful subproblem solution would be the solution for the slice of the list that contains all but one element of the original, e.g., sumList(data[1:]).

3. How do you find the solution to the original problem using this subproblem solution? Implement this as the recursive step of your recursive function.

The solution to sumList(data) is data[0] + sumList(data[1:]). Therefore, the recursive step in the function should be

```
return data[0] + sumList(data[1:])
```

4. What are the simplest subproblems that you can solve non-recursively, and what are their solutions? Implement your answer as the base case of the recursive function.

The simplest subproblem would be to compute the sum of an empty list, which is 0, of course. So the base case should be

if len(data) == 0:
 return 0

5. For any possible parameter value, will the recursive calls eventually reach the base case?

Yes, since an empty list will obviously reach the base case and passing any other list as an argument will result in a sequence of recursive calls, each of which involves a list that is one element shorter than in the previous call.

**Reflection 9.9** How could we have answered question 2 differently? What is another subproblem that involves all but one element of the original list? Using this subproblem instead, answer the rest of the questions to write an alternative recursive sumList function.

An alternative subproblem would be sumList(data[:-1]) (all but the last element). In this version of the function, the base case is the same, but the recursive case would be

return sumList(data[:-1]) + data[-1] # recursive case

# Palindromes

Let's look at another example. A *palindrome* is any sequence of characters that reads the same forward and backward. For example, radar, star rats, and now I won are all palindromes. An iterative function that determines whether a string is a palindrome is shown below.

```
def palindrome(text):
    """Determine whether a string is a palindrome.
    Parameter: a string text
    Return value: a Boolean value indicating whether text is a palindrome
    """
    for index in range(len(text) // 2):
        if text[index] != text[-(index + 1)]:
            return False
    return True
```

Let's answer the five questions above to develop an equivalent recursive algorithm for this problem.

**Reflection 9.10** First, what does a subproblem look like?

A subproblem would be to determine whether a slice of the string (i.e., a substring) is a palindrome.

**Reflection 9.11** Second, if you could know whether any slice is a palindrome, which would be the most helpful?

It is often helpful to look at an example. Consider the following:

$$text \longrightarrow 'n \underbrace{ow \ I \ wo}_{text[1:-1]} n$$

If we begin by looking at the first and last characters and determine that they are *not* the same, then we know that the string is not a palindrome. But if they *are* the same, then the question of whether the string is a palindrome is decided by whether the slice that omits the first and last characters, i.e., text[1:-1], is a palindrome. So it would be helpful to know the result of palindrome(text[1:-1]).

**Reflection 9.12** Third, how could we use this information to determine whether the whole string is a palindrome?

If the first and last characters are the same and text[1:-1] is a palindrome, then text is a palindrome. Otherwise, text is not a palindrome. In other words, our desired return value is the value of the following Boolean expression.

return (text[0] == text[-1]) and palindrome(text[1:-1])

If the first part is true, then the answer depends on whether the slice is a palindrome (palindrome(text[1:-1])). Otherwise, if the first part is false, then the entire



Figure 9.12 A representation of the function calls in the recursive palindrome function. The red numbers indicate the order in which the events happen. On the left is an instance in which the function reaches the base case and returns True. On the right is an instance in which the function returns False.

and expression is false. Furthermore, due to the short circuit evaluation of the and operator, the recursive call to palindrome will be skipped.

**Reflection 9.13** What are the simplest subproblems that we can solve non-recursively, and what are their solutions? Implement your answer as the base case.

The simplest string is, of course, the empty string, which we can consider a palindrome. But strings containing a single character are also palindromes, since they read the same forward and backward. So we know that any string with length at most one is a palindrome. But we also need to think about strings that are obviously *not* palindromes. Our discussion above already touched on this; when the first and last characters are different, we know that the string cannot be a palindrome. Since this

situation is already handled by the Boolean expression above, we do not need a separate base case for it.

Putting this all together, we have the following elegant recursive function:

Let's look more closely at how this recursive function works. On the left side of Figure 9.12 is a representation of the recursive calls that are made when the palindrome function is called with the argument 'now I won'. From the main function in box (a), palindrome is called with 'now I won', creating the instance of palindrome in box (b). Since the first and last characters of the parameter are equal (the (text[0] == text[-1]) part of the return statement is not shown to make the pictures less cluttered), the function will return the value of palindrome('ow I wo'). But, in order to get this value, it needs to call palindrome again, creating the instance in box (c). These recursive calls continue until we reach the base case in box (f), where the length of the parameter is one. The instance of palindrome in box (f) returns True to the previous instance of palindrome in box (e) returns to (d) the value True that it just received from (f). The value of True is propagated in this way all the way up the chain until it eventually reaches main, where it is assigned to the variable named pal.

To see how the function returns False, let's consider the example on the right side of Figure 9.12. In this example, the recursive palindrome function is called from main in box (a) with the non-palindromic argument 'now I win', which creates the instance of palindrome in box (b). As before, since the first and last characters of the parameter are equal, the function will return the value of palindrome('ow I wi'). Calling palindrome with this parameter creates the instance in box (c). But now, since (text[0] == text[-1]) is False, the instance of palindrome in box (c) returns False, and this value is propagated up to the main function.

#### Guessing passwords

One technique that hackers use to compromise computer systems is to rapidly try all possible passwords up to some given length.

**Reflection 9.14** How many possible passwords are there with length n, if there are c possible characters to choose from?

The number of different passwords with length n is

$$\underbrace{c \cdot c \cdot c \cdots c}_{n \text{ times}} = c^n.$$

For example, there are

# $26^8 = 208,827,064,576 \approx 208$ billion

different eight-character passwords that use only lowercase letters. But there are

# $67^{12} = 8,182,718,904,632,857,144,561 \approx 8$ sextillion

different twelve-character passwords that draw from the lower and uppercase letters, digits, and the five special characters , &, #, ?, and , which is why websites make you use long passwords with all of these characters! When you use a long enough password and enough different characters, the "guess and check" method is useless.

Let's think about how we could generate a list of possible passwords by first considering the simpler problem of generating all binary strings (or "bit strings") of a given length. This is the same problem, but using only two characters: '0' and '1'. For example, the list of all binary strings with length three is ['000', '001', '010', '011', '100', '101', '110', '111'].

Thinking about this problem iteratively can be daunting. However, it becomes *easier* if we think about it recursively, in terms of smaller versions of itself. As shown below, a list of binary strings with a particular length can be created easily if we already have a list of binary strings that are one bit shorter. We simply make two copies of the list of shorter binary strings, and then precede all of the strings in the first copy with a 0 and all of the strings in the second copy with a 1.



In the illustration above, the list of binary strings with length 2 is created from two copies of the list of binary strings with length 1. Then the list of binary strings with length 3 is created from two copies of the list of binary strings with length 2. In general, the list of all binary strings with a given length is the concatenation of

- (a) the list of all binary strings that are one bit shorter and preceded by zero and
- (b) the list of all binary strings that are one bit shorter and preceded by one.

**Reflection 9.15** What is the base case of this algorithm?

The base case occurs when the length is 0, and there are no binary strings. However, the problem says that the return value should be a list of strings, so we will return a

list containing an empty string in this case. The following function implements this recursive algorithm.

```
def bitStrings(length):
    """Return a list of all binary strings with the given length.
    Parameter:
        length: the length of the binary strings
    Return value: the list of binary strings
    11.11.11
    if length == 0:
                                                   # base case
        return ['']
    shorterList = bitStrings(length - 1) # recursively get shorter
                                          # bit strings
   newBitStrings = []
                                                  # create a list
                                                      of bit strings
   for shorterString in shorterList:
                                                  #
        newBitStrings.append('0' + shorterString) #
                                                     with prefix 0
    for shorterString in shorterList:
                                                  # append bit strings
        newBitStrings.append('1' + shorterString) # with prefix 1
   return newBitStrings
                                                 # return all bit strings
```

In the recursive step, we call bitStrings(length - 1) to get a list of all bit strings with length equal to length - 1, and then assign this list to shorterList. Next we create a new list named newBitStrings that will hold all of the bit strings with the desired length. In the first for loop, we append to this list all of the bit strings in shorterList, preceded by '0'. Then, in the second for loop, we append all of the bit strings in shorterList, preceded by '1'.

**Reflection 9.16** Why will this algorithm not work if we return an empty list in the base case instead of a list containing an empty string? What would be returned if we did return an empty list instead?

If we return an empty list in the base case instead of a list containing an empty string, the function will return an empty list. To see why, consider what would happen if we called bitStrings(1). Then shorterList = bitStrings(0) will be assigned an empty list. This would mean that there is nothing to iterate over in the two for loops, and nothing is appended to newBitStrings. Since bitStrings(2) calls bitStrings(1), this means that bitStrings(2) will also return the empty list, and so on for any value of length!

**Reflection 9.17** The bitStrings function above contains two nearly identical for loops, one for the '0' prefix and one for the '1' prefix. How can we combine these two loops?

We can combine the two loops by repeating a more generic version of the loop for each of the characters '0' and '1':

```
def bitStrings(length):
    """ (docstring omitted) """

    if length == 0:
        return ['']

    shorterList = bitStrings(length - 1)

    newBitStrings = []
    for character in ['0', '1']:
        for shorterString in shorterList:
            newBitStrings.append(character + shorterString)

    return newBitStrings
```

We can use a very similar algorithm to generate a list of possible passwords. The only difference is that, instead of preceding each shorter string with 0 and 1, we need to precede each shorter string with every character in the set of allowable characters. The following function, with a string of allowable characters assigned to an additional parameter, is a simple generalization of our **bitStrings** function.

```
def passwords(length, characters):
    """Return a list of all possible passwords with the given length,
       using the given characters.
    Parameters:
        length:
                    the length of the passwords
        characters: a string containing the characters to use
   Return value: the list of passwords
    ......
    if length == 0:
        return ['']
    shorterList = passwords(length - 1, characters)
    passwordList = []
    for character in characters:
        for shorterPassword in shorterList:
            passwordList.append(character + shorterPassword)
   return passwordList
```

**Reflection 9.18** How would we call the passwords function to generate a list of all bit strings with length 5? What about all passwords with length 4 containing the characters 'abc123'? What about all passwords with length 8 containing lowercase letters? (Do not actually try this last one!)

# Exercises

Write a **recursive** function for each of the following problems. Test your functions with both common and boundary case arguments, and document your test cases.

9.2.1*	Write a recursive function sumIt(n)				
	that returns the sum of the integers from 1 to n.				
9.2.2.	Write a recursive function				
	factorial(n)				
	that returns the value of $n! = 1 \cdot 2 \cdot 3 \cdots n$ .				
9.2.3.	Write a recursive function				
	power(a, n)				
	that returns the value of $a^n$ without using the <b>**</b> operator.				
9.2.4.	Euclid's greatest common divisor (GCD) algorithm uses the fact that the GCD of $m$ and $n$ is the same as the GCD of $n$ and $m \mod n$ . Write a recursive function $gcd(m, n)$				
	that implements Euclid's algorithm. In the base case, gcd(m, 0) is m.				
9.2.5*	Write a recursive function				
	length(data)				
	that returns the length of the list data without using the len function.				
9.2.6*	Write a recursive function				
	minList(data)				
	that returns the minimum of the items in the list of numbers named data. You may use the built-in min function for finding the minimum of (only) two numbers.				
9.2.7*	Write a recursive function				
	reverse(text)				
	that returns the reverse of the string named text.				
9.2.8.	Write a recursive function underscore(text)				
	that returns a version of the string named ${\tt text}$ with all spaces replaced by underscore (_) characters.				
9.2.9.	Write a recursive function				
	noSpaces(text)				
	that returns a version of the string named text with all spaces removed.				
9.2.10.	Write a recursive function				
	increment(data)				
	that returns a new list containing all of the values in the list of numbers named data incremented by one.				

9.2.11. Write a recursive function

#### int2string(n)

that converts an integer value n to its string equivalent, *without* using the str function. For example, int2string(1234) should return the string '1234'.

9.2.12<sup>\*</sup> Write a recursive function

countUpper(text)

that returns the number of uppercase letters in the string text.

9.2.13. Write a recursive function

count(text, target)

that returns the number of occurrences of the string target in the string text.

9.2.14. Write a recursive function

evenLength(data)

that returns True if data contains an even number of items, and False otherwise. Do not use the len function.

9.2.15. Write a recursive function

search(data, target)

that returns True if target is contained in the list data, and False otherwise.

9.2.16. Write a recursive function

#### equal(list1, list2)

that returns a Boolean value indicating whether the two lists are equal without testing whether list1 == list2. Only compare the lengths of the lists and test whether individual list elements are equal.

9.2.17. Write a recursive function

#### powerSet(n)

that returns a list of all subsets of the integers 1, 2, ..., n. A subset is a list of zero or more unique items from a set. The set of all subsets of a set is also called the *power set*. For example, subsets(n) should return the list [[], [1], [2], [2, 1], [3], [3, 1], [3, 2], [3, 2, 1]]. (Hint: this is similar to the bitStrings function.)

9.2.18. Suppose you work for a state in which all vehicle license plates consist of a string of letters followed by a string of numbers, such as 'ABC 123'. Write a recursive function

licensePlates(length, letters, numbers)

that returns a list of strings representing all possible license plates of this form, with length letters and length numbers chosen from the given strings. For example, licensePlates(2, 'XY', '12') should return the following list of 16 possible license plates consisting of two letters drawn from 'XY' followed by two digits drawn from '12':

['XX 11', 'XX 21', 'XY 11', 'XY 21', 'XX 12', 'XX 22', 'XY 12', 'XY 22', 'YX 11', 'YX 21', 'YY 11', 'YY 21', 'YX 12', 'YX 22', 'YY 12', 'YY 22']

(Hint: this is similar to the passwords function.)





Figure 9.13 Illustration of the recursive algorithm for Tower of Hanoi with three disks.

# 9.3 THE MYTHICAL TOWER OF HANOI

The **Tower of Hanoi** is a game that was first marketed in 1883 by French mathematician Édouard Lucas. As illustrated in Figure 9.13(a), the game is played on a board with three pegs. One peg holds some number of disks with unique diameters, ordered smallest to largest. The objective of the game is to move this "tower" of disks from their original peg to another peg, one at a time, without ever placing a larger disk on top of a smaller one. The game was purported to have originated in an ancient legend. In part, the game's instruction sheet reads:

According to an old Indian legend, the Brahmins have been following each other for a very long time on the steps of the altar in the Temple of Benares, carrying out the moving of the Sacred Tower of Brahma with sixty-four levels in fine gold, trimmed with diamonds from Golconde. When all is finished, the Tower and the Brahmins will fall, and that will be the end of the world!<sup>1</sup>

This game is interesting because it is naturally solved using the following recursive insight. To move n disks from the first peg to the third peg, we must first be able to move the bottom (largest) disk on the first peg to the bottom position on the third peg. The only way to do this is to somehow move the top n - 1 disks from the first peg to the second peg, to get them out of the way, as illustrated in Figure 9.13(b). But notice that moving n - 1 disks is a subproblem of moving n disks because it is the same problem but with only part of the input. The source and destination pegs are different in the original problem and the subproblem, but this can be handled by making the source, destination, and intermediate pegs additional inputs to the problem. Because this step is a subproblem, we can perform it recursively! Once this is accomplished, we are free to move the largest disk from the first peg to the third peg, as in Figure 9.13(c). Finally, we can once again recursively move the n - 1 disks from the second peg to the third peg, shown in Figure 9.13(d). In summary, we have the following recursive algorithm:

<sup>&</sup>lt;sup>1</sup>http://www.cs.wm.edu/~pkstoc/toh.html

- 1. Recursively move the top n-1 disks from the source peg to the intermediate peg, as in Figure 9.13(b).
- 2. Move one disk from the source peg to the destination peg, as in Figure 9.13(c).
- 3. Recursively move the n-1 disks from the intermediate peg to the destination peg, as in Figure 9.13(d).

**Reflection 9.19** What is the base case in this recursive algorithm? In other words, what is the simplest subproblem that will be reached by these recursive calls?

The simplest case would involve having no disks at all, in which case, we do nothing.

We cannot actually write a Python function to move the disks for us, but we can write a function that gives us instructions on how to do so. The following hanoi function does this, following exactly the algorithm described above.

```
def move(source, destination): # just print an instruction
    print('Move a disk from peg ' + str(source) + ' to peg '
                                  + str(destination) + '.')
def hanoi(n, source, destination, intermediate):
    ""Print instructions for solving the Tower of Hanoi puzzle.
    Parameters:
        n:
                      the number of disks
        source:
                     the source peg
        destination: the destination peg
        intermediate: the other peg
    Return value: None
    .....
    if n >= 1:
        hanoi(n - 1, source, intermediate, destination)
        move(source, destination)
        hanoi(n - 1, intermediate, destination, source)
```

The parameters n, source, destination, and intermediate represent the number of disks, the source peg, the destination peg, and the remaining peg that can be used as a temporary resting place for disks that we need to get out of the way. In our examples, the left peg was the source, the right peg was the destination, and the middle peg was the intermediate. Notice that the base case is implicit: if n is less than one, the function simply returns.

When we execute this function, we can name our pegs anything we want. For example, if we name our pegs A, B, and C, then

hanoi(8, 'A', 'C', 'B')

will print instructions for moving eight disks from A to C, using B as the intermediate.

**Reflection 9.20** Execute the function with three disks. Does it work? How many steps are necessary? What about with four and five disks? Do you see a pattern?

## \*Is the end of the world nigh?

The original game's instructions claimed that the world would end when the monks finished moving 64 disks. So how long does this take? To derive a general answer, let's start by looking at small numbers of disks. When there is one disk, only one move is necessary: move the disk from the source peg to the destination peg. When there are two disks, the algorithm moves the smaller disk to the intermediate peg, then the larger disk to the destination peg, and finally the smaller disk to the destination peg, for a total of three moves.

When n is three, we need to first move the top two disks to the intermediate peg which, we just deduced, requires three moves. Then we move the largest disk to the destination peg, for a total of four moves so far. Finally, we move the two disks from the intermediate peg to the destination peg, which requires another three moves, for a total of seven moves.

In general, notice that the number of moves required for n disks is the number of moves required for n-1 disks, plus one move for the bottom disk, plus the number of moves required for n-1 disks again. In other words, if the function M(n) represents the number of moves required for n disks, then

$$M(n) = M(n-1) + 1 + M(n-1) = 2M(n-1) + 1.$$

Does this look familiar? This is a *difference equation*, just like those in Chapter 4. In this context, a function that is defined in terms of itself is also called a *recurrence relation*. The sequence produced by this recurrence relation is shown in the following table.

n	M(n)
1	1
2	3
3	7
4	15
5	31
÷	÷

**Reflection 9.21** Do you see the pattern in the table? What is the formula for M(n) in terms of n?

M(n) is always one less than  $2^n$ . In other words, the algorithm requires

$$M(n) = 2^n - 1$$

moves to solve the game when there are n disks. This expression is called a *closed* 

form for the recurrence relation because it is defined only in terms of n, not M(n-1). According to this formula, moving 64 disks would require

$$2^{64} - 1 = 18,446,744,073,709,551,615$$

moves. The end of the world is apparently not coming any time soon!

### Exercises

- 9.3.1. Design a recursive algorithm for a version of the Tower of Hanoi puzzle in which you can only move disks between adjacent pegs, in either direction. In other words, you cannot move a disk directly from A to C or from C to A. Implement your algorithm as a recursive function in Python.
- 9.3.2. How many moves does your algorithm from Exercise 9.3.1 make when there are n disks? (Think about n = 1, 2, and 3 disks first. Then derive a general formula.)
- 9.3.3\* In another variation of the Tower of Hanoi puzzle, there are four pegs, labeled A, B, C, and D, and two stacks of disks. A white stack of n disks starts on peg A and a black stack starts on peg C. The goal is to exchange the two stacks, i.e., move the white stack to peg C and the black stack to peg A, without ever placing a larger disk on top of a smaller one. This problem can be solved by using the classical Tower of Hanoi algorithm three times.
  - (a) Explain how with a pseudocode algorithm.
  - (b) Implement your algorithm in Python. You will first need to create slightly modified versions of the move and hanoi functions that incorporate the color of the disk(s) that are being moved. Then write a new function solveTower4 that calls your modified hanoi function appropriately to solve this problem.
  - (c) How many moves does your algorithm make with n disks?
- 9.3.4. The four-peg problem in Exercise 9.3.3 can be solved in fewer moves with the following algorithm:
  - 1. Use the classical Tower of Hanoi algorithm to move the top n-1 white disks from peg A to peg B, using peg D as the intermediate peg.
  - 2. Move the largest white disk from peg A to peg D.
  - 3. Use the classical algorithm to move all *n* black disks from peg C to peg A, using peg D as the intermediate.
  - 4. Move the largest white disk from peg D to peg C.
  - 5. Use the classical algorithm to move the top n-1 white disks from peg B to peg C, using peg D as the intermediate.
  - (a) When all *n* black disks are moved in step 3 of the algorithm, using peg D as the intermediate peg, the largest white disk is still on peg D. Why does this not violate the rules of the puzzle?
  - (b) Implement this algorithm in Python.
  - (c) How many moves does this algorithm make with n disks?

# 9.4 RECURSIVE LINEAR SEARCH

As we have already seen, linear search is a workhorse of an algorithm, used in a variety of settings. In this section, we will develop a recursive version of this algorithm and then show that it has the same time complexity as the iterative version. Later, in Section 10.1, we will develop a more efficient algorithm for searching in a sorted list.

**Reflection 9.22** What does a subproblem look like in the search problem? What would be the most useful subproblem to have an answer for?

In the search problem, a subproblem is to search for the target item in a smaller list. Since we can only "look at" one item at a time, the most useful subproblem will be to search for the target item in a sublist that contains all but one item, say the first one. This way, we can break the original problem into two parts: (a) determine if the target is the first item and (b) determine if the target is in the rest of the list. Of course, if the first item is the one we are searching for, we can avoid the recursive call altogether. Otherwise, we return the index that is returned by the search of the rest of the list.

#### **Reflection 9.23** What is the base case for this problem?

We have already discussed one base case: if the target item is the first item in the list, we simply return its index. Another base case would be when the list is empty. In this case, the item for which we are searching is not in the list, so we return -1. The following function (almost) implements the recursive algorithm we have described.

However, as indicated by the red question marks above, we have a problem.

**Reflection 9.24** What is the problem indicated by the red question marks? Why can we not just return 0 in that case?

When we find the target at the beginning of the list and are ready to return its index in the original list, we do not know what it was! We know that it is at index 0 in the current sublist being searched, but we have no way of knowing where this sublist starts in the original list. Therefore, we need to add a third parameter to the function that keeps track of the original index of the first item in the sublist data. In each recursive call, we add one to this argument since the index of the new front item in the list will be one more than that of the current front item.

Creating a new slice of the list in each recursive call, which is a relatively costly operation, can also now be eliminated. We can pass in the entire list instead, and use the value of first to identify the "first" item we are considering in the second base case.

As shown above, this change also necessitates a change in our first base case because the length of the list is no longer decreasing to zero. Since the intent of the function is to search in the list between indices first and len(data) - 1, we will consider the list under consideration to be empty if the value of first is greater than the last index in the list. Just to be safe, we also make sure that first is at least zero.

```
Reflection 9.25 Use the recursive linearSearch function to search for 'keys' in the list ['sock', 'keys', 'phone', 'remote']. What do you pass in for the third parameter?
```

When calling the recursive linearSearch function, we need to initially pass 0 in as the third parameter since that is the index of the first item that we wish to search.

position = linearSearch(['sock', 'keys', 'phone', 'remote'], 'keys', 0)

# \*Efficiency of recursive linear search

Like the iterative linear search, this algorithm has linear time complexity. But we cannot derive this result in the same way we did for the iterative version. Instead, we need to use a recurrence relation, as we did to find the number of moves required in the Tower of Hanoi algorithm.

Let T(n) represent the worst case number of comparisons required by a linear search when the length of the list is n. When we look at the algorithm above, we see that there are two comparisons that the algorithm must make before reaching a recursive function call. But since it only matters asymptotically that this number is a constant, we will simply represent the number of comparisons before the recursive call as a constant value c. Therefore, the number of comparisons necessary in the base case in which the list is empty (n = 0) is T(0) = c. In recursive cases, there are additional comparisons to be made in the recursive call.

**Reflection 9.26** How many more comparisons are made in the recursive call to linearSearch?

The size of the sublist yet to be considered in each recursive call is n-1, one less than in the current instance of the function. Therefore, the number of comparisons in each recursive call must be the number of comparisons required by a linear search on a list with length n-1, which is T(n-1). So the total number of comparisons is

$$T(n) = T(n-1) + c.$$

But this is not very helpful in determining what the time complexity of the linear search is. To get this recurrence relation into a more useful form, we can think about the recurrence relation as saying that the value of T(n) is equal to, or can be replaced by, the value of T(n-1) + c, as illustrated below:

$$T(n) \downarrow T(n-1) + c$$

**Reflection 9.27** Now what can we replace T(n-1) with?

T(n-1) is just T(n) with n-1 substituted for n. Therefore, using the definition of T(n) above,

$$T(n-1) = T(n-1-1) + c = T(n-2) + c.$$

So we can also substitute T(n-2) + c for T(n-1), as shown in the third row of Figure 9.14. Similarly, Figure 9.14 uses the equivalences

$$T(n-2) = T(n-2-1) + c = T(n-3) + c$$

and

$$T(n-3) = T(n-3-1) + c = T(n-4) + c$$

to continue these substitutions until we reach T(0).

The right side of the figure illustrates the accumulation of c's (which are not substituted) as we proceed downward. Since c is the number of comparisons in each recursive call, these values on the right represent the total number of comparisons made so far. Notice that the number subtracted from n in the argument of T at



Figure 9.14 An illustration of how to derive a closed form for the recurrence relation T(n) = T(n-1) + c.

each step is equal to the multiplier in front of the accumulated c's at that step. In other words, to the right of each T(n-i), the accumulated value of c's is  $i \cdot c$ . When we finally reach T(0), which is the same as T(n-n), the total on the right must be nc. Finally, we showed above that T(0) = c, so the total number of comparisons is (n+1)c. This expression is called the *closed form* of the recurrence relation because it does not involve any values of T(n). Since (n+1)c is  $\mathcal{O}(n)$  asymptotically, recursive linear search is a linear-time algorithm, just like the iterative linear search. Intuitively, this should make sense because the two algorithms essentially do the same thing: they both look at every item in the list until the target is found.

# **Exercises**

- 9.4.1\* The linearSearch\_Draft1 function, without the first parameter, can work if we only need to know whether the target is in the list. Write a working version of this function that returns True if the target item is contained in the list and False otherwise.
- 9.4.2\* Unlike our final version of the linearSearch function, the function you wrote in the previous exercise uses slicing. Is this still a linear-time algorithm?
- 9.4.3. Write a new version of recursive linear search that instead looks at the last item in the list, and recursively calls the function with the sublist not containing the last item.
- 9.4.4. Write a new version of recursive linear search that only looks at the items in the list with even indices. For example, linearSearch([1, 2, 3, 4, 2], 2, 0) should return the index 4 because it would not see the target, 2, at index 1.
- 9.4.5. Write a recursive function

sumSearch(data, total, first)

that returns the first index in data, greater than or equal to first, for which the sum of the values in data[:index + 1] is greater than or equal to total. If the sum of all of the values in the list is less than total, the function should return -1. For example, sumSearch([2, 1, 4, 3], 4) returns index 2 because  $2+1+4 \ge 4$  but 2+1 < 4.

# 9.5 DIVIDE AND CONQUER

The algorithm for the Tower of Hanoi game elegantly used recursion to divide the problem into three simpler subproblems: recursively move n - 1 disks, move one disk, and then recursively move n - 1 disks again. Such algorithms came to be called **divide and conquer algorithms** because they "divide" a hard problem into two or more subproblems, and then "conquer" each subproblem recursively. The divide and conquer technique has been found to yield similarly elegant, and often quite efficient, algorithms for a wide variety of problems.

It is actually useful to think of divide and conquer algorithms as comprising three steps instead of two:

- 1. Divide the problem into two or more subproblems.
- 2. Conquer each subproblem recursively.
- 3. *Combine* the solutions to the subproblems into a solution for the original problem.

In the Tower of Hanoi algorithm, the "combine" step was essentially non-existent. Once the subproblems had been "conquered," we were done. But other problems do require this step at the end.

In this section, we will design elegant divide and conquer algorithms that choose optimal buy and sell dates on the stock market and navigate a rover through a maze.

# Buy low, sell high

Suppose that you have created a model to predict the future daily closing prices of a particular stock. With this list of prices, you would like to determine when to buy and sell the stock to maximize your profit. For example, when should you buy and sell if your model predicts the following prices for the next ten days?

Day 1	2	3	4	5	6	7	8	9	10
Price 3.90	3.60	3.65	3.71	3.78	4.95	3.21	4.50	3.18	3.53

It is tempting to look for the minimum price (\$3.18) and then look for the maximum price after that day. But this clearly does not work with this example. Even choosing the second smallest price (\$3.21) does not give the optimal answer. The most profitable choice is to buy on day 2 at \$3.60 and sell on day 6 at \$4.95, for a profit of \$1.35 per share.

One way to find this answer is to look at all possible pairs of buy and sell dates, and pick the pair with the maximum profit. (You may actually have already done this in Exercise 7.5.7.) Since there are n(n-1)/2 such pairs, this yields an algorithm with time complexity  $\mathcal{O}(n^2)$ .

However, divide and conquer yields a more efficient algorithm. Notice that if we divide the list of prices in half, then the optimal pair of dates must either be in the left half, the right half, or straddle the two halves, with the buy date in the left half and sell date in the right half. This observation can be used to design the following divide and conquer algorithm:

- 1. *Divide* the problem into two subproblems: (a) finding the optimal buy and sell dates in the left half of the list and (b) finding the optimal buy and sell dates in the right half of the list.
- 2. *Conquer* the two subproblems by executing the algorithm recursively on these two smaller lists of prices.
- 3. Combine the solutions by choosing the most profitable buy and sell dates from among (a) the best dates in the left half, (b) the best dates in the right half, and (c) the best dates that straddle the two halves.

**Reflection 9.28** Is there an easy way to find the best buy and sell dates that straddle the two halves, with the buy date in the left half and sell date in the right half?

At first glance, it might look like the "combine" step would require another recursive call to the algorithm. But finding the optimal buy and sell dates with this particular restriction is actually quite easy. The best buy date in the left half must be the one with the minimum price, and the best sell date in the right half must be the one with the maximum price. So finding these buy and sell dates simply amounts to finding the minimum price in the left half of the list and the maximum price in the right half, which we already know how to do.

Before we implement this algorithm, let's apply it to the example above:

[3.90, 3.60, 3.65, 3.71, 3.78, 4.95, 3.21, 4.50, 3.18, 3.53]

- First, we divide the list into two halves: [3.90, 3.60, 3.65, 3.71, 3.78] and [4.95, 3.21, 4.50, 3.18, 3.53].
- 2. Next, we recursively find the maximum profits in the left half and the right half. In the left half, the maximum profit is attained by buying on day 2 for 3.60 and selling on day 5 at 3.78, for a profit of 3.78 3.60 = 0.18. In the right half, we maximize our profit by buying on day 7 for 3.21 and selling on day 8 at 4.50, for a profit of 4.50 3.21 = 1.29. In the actual algorithm, these results will be derived from a sequence of several more recursive calls, but we will pretend for now that this has already happened.
- 3. Finally, we find the maximum profit possible by holding the stock from the first half to the second half. Since the minimum price in the first half is 3.60 on day 2 and the maximum price in the second half is 4.95 on day 6, this profit is 4.95 3.60 = 1.35. We return the maximum of 0.18, 1.29, and 1.35, which is 1.35. We achieved this by buying on day 2 and selling on day 6.

**Reflection 9.29** Since this is a recursive algorithm, we also need a base case. What is the simplest list in which to find the optimal buy and sell dates?

The easiest case would be a list with less than two prices; then we never buy at all (or, equivalently, buy and sell on the same day), for zero profit. The following function implements the divide and conquer algorithm in a very straightforward way, but it just finds the optimal profit, not the actual buy and sell days. Finding these days requires just a little more work, which we leave for you as an exercise.

```
1 def profit(prices):
      """Find the maximum profit from a list of daily stock prices.
2
      Parameter:
3
          prices: a list of daily stock prices
4
      Return value: the maximum profit
5
      11.11.11
6
      if len(prices) <= 1:</pre>
                                                  # base case
7
          return 0
8
      midIndex = len(prices) // 2
                                                  # divide in half
9
      leftProfit = profit(prices[:midIndex])
                                                  # conquer 2 halves
10
      rightProfit = profit(prices[midIndex:])
11
      buy = min(prices[:midIndex])
                                                  # min price on left
12
      sell = max(prices[midIndex:])
                                                  # max price on right
13
      midProfit = sell - buy
                                                  # max left -> right profit
14
      return max(leftProfit, rightProfit, midProfit) # combine 3 cases
15
```

#### 9.5 DIVIDE AND CONQUER **399**



Figure 9.15 A representation of the function calls in the recursive profit function. The red numbers indicate the order in which the events happen.

**Reflection 9.30** Call this function with the list of prices that we used in the example above.

That this algorithm actually works may seem like magic at this point. But rest assured that, like all recursive algorithms, there is a perfectly good reason why it works. The process is sketched out in Figure 9.15 for a small list containing just the first four prices in our example. Each bold rectangle represents an instance of a function call. The smaller gray boxes represent how the return value is computed in each one. In all but the base cases, the return value is the maximum of leftProfit (represented by left), rightProfit (represented by right), and midProfit (represented by mid).

At the top of Figure 9.15, the **profit** function is called with a list of four prices. The **profit** function then recursively calls itself with the first two prices and the last two prices (lines 10–11 in the function). The red numbers on the arrows in the figure show the order of the subsequent events.

- 1. The first recursive call (on line 10) to profit([3.90, 3.60]) is represented by box (b). This call to profit results in two more recursive calls, labeled (c) and (d).
- 2-3. The call to profit([3.90]) in box (c) is a base case which returns 0. This value is assigned to leftProfit (left) in box (b).
- 4-5. The second recursive call from box (b) to profit([3.60]), shown in box (d), is also a base case which returns 0. This value is assigned to rightProfit (right) in box (b).
  - 6. Back in box (b), after the two recursive calls return, the profit from holding the

stock in the combine step, -0.30, is assigned in line 14 to midProfit (mid). The maximum of the three values, which in this case is 0, is returned back to box (a) and assigned to leftProfit (left).

- 7. Now the second recursive call (on line 11) to profit([3.65, 3.71]) is made from box (a). This instance of the function is shown in box (e).
- 8-11. This recursive call results in a sequence of function calls that is very similar to steps 2–5, as illustrated in boxes (e)–(g).
  - 12. The recursive call to profit([3.65, 3.71]) returns 0.06, which is assigned to rightProfit (right) in box (a).
  - 13. Finally, the maximum profit across the two halves in (a) is found to be 3.71
     3.60 = 0.11. Then the maximum of leftProfit, which is 0, rightProfit, which is 0.06, and midProfit, which is 0.11, is returned by the original function call.

## Navigating a maze

Suppose we want to navigate a robotic rover through an unknown, obstacle-filled terrain. For simplicity, we will assume that the landscape is laid out on a grid and the rover is only able to "see" and move to the four grid cells to its east, south, west, and north in each step, as long as they do not contain obstacles.

To navigate the rover to its destination on the grid (or determine that the destination cannot be reached), we can use a technique called *depth-first search*. The depth-first search algorithm explores a grid by first exploring in one direction as far as it can from the source. Then it *backtracks* to follow paths that branch off in each of the other three directions.

Put another way, a depth-first search divides the problem of searching for a path to the destination into four subproblems: searching for a path starting from the cell to the east, from the cell to the south, from the cell to the west, and from the cell to the north. To solve each of these subproblems, the algorithm follows this identical procedure again, just from a different starting point. In terms of the three divide and conquer steps, the depth-first search algorithm looks like this:

- 1. *Divide* the problem into four subproblems. Each subproblem searches for a path that starts from one of the four neighboring cells.
- 2. *Conquer* the subproblems by recursively executing the algorithm from each of the neighboring cells.
- 3. *Combine* the solutions to the subproblems by returning success if any of the subproblems were successful. Otherwise, return failure.

To illustrate, consider the grid in Figure 9.16(a). In this example, we are attempting to find a path from the green cell in position (1,1) to the red cell in position (3,0).

9.5 DIVIDE AND CONQUER **401** 



Figure 9.16 An illustration of a depth-first search on a grid.

The black cells represent obstacles that the rover cannot move through. The depthfirst search algorithm will visit neighboring cells in clockwise order: east, south, west, north. The algorithm starts at cell (1,1) and looks east to (1,2), but cannot move in that direction due to an obstacle. Therefore, it next explores the cell to the south in position (2,1), colored blue in Figure 9.16(b). From this cell, it recursively executes the same algorithm, first looking east to (2,2). Since this cell is not blocked, it is the next one visited, as represented in Figure 9.16(c). The depth-first search algorithm is recursively called again from cell (2,2), but the cells to the east, south, and north are blocked; and the cell to the west has already been visited. Therefore, the depth-first search algorithm returns failure to the cell at (2,1). We color cell (2,2) light blue to indicate that it has been visited, but is no longer on the path to the destination. From cell (2,1), the algorithm has already looked east, so it now moves south to (3,1), as shown in Figure 9.16(d). In the next step, shown in Figure 9.16(e), the algorithm moves south again to (4,1) because the cell to the east is blocked.

**Reflection 9.31** When it is at cell (3,1), why does the algorithm not "see" the destination in cell (3,0)?

It does not yet "see" the destination because it looks east and south before it looks west. Since there is an open cell to the south, the algorithm will follow that

possibility first. In the next steps, shown in Figure 9.16(f)–(g), the algorithm is able to move east, and in Figure 9.16(h), it is only able to move north. At this point, the algorithm backtracks to (4,1) over three steps, as shown in Figure 9.16(i)–(k), because all possible directions have already been attempted from cells (3,3), (4,3), and (4,2). From cell (4,1), the algorithm next moves west to cell (4,0), as shown in Figure 9.16(l), because it has already moved east and there is no cell to the south. Finally, from cell (4,0), it cannot move east, south, or west; so it moves north where it finally finds the destination.

The final path shown in blue illustrates the path that the algorithm from the source to destination.

**Reflection 9.32** Did the depth-first search algorithm find the shortest path?

A depth-first search is not guaranteed to find the shortest, or even a short, path. But it will find *a* path if one exists. Another algorithm, called a *breadth-first search*, can be used to find the shortest path.

**Reflection 9.33** What is the base case of the depth-first search algorithm? For what types of source cells can the algorithm finish without making any recursive calls?

There are two kinds of base cases in depth-first search, corresponding to the two possible outcomes. One base case occurs when the source cell is not a "legal" cell from which to start a new search. These source cells are outside the grid, blocked, or already visited. In these cases, we simply return failure. The other base case occurs when the source cell is the same as the destination cell. In this case, we return success.

The depth-first search algorithm is implemented by the following function. The function returns True (success) if the destination was reached by a path and False (failure) if the destination could not be found.

```
BLOCKED = 0 # site is blocked
OPEN = 1
             # site is open and not visited
VISITED = 2 # site is open and already visited
def dfs(grid, source, dest):
    """Perform a depth-first search on a grid to determine if there
       is a path between a source and destination.
    Parameters:
                a two-dimensional grid (list of lists)
        grid:
        source: a (row, column) tuple to start from
        dest: a (row, column) tuple to reach
   Return value: Boolean indicating whether destination was reached
    .....
    (row, col) = source
   rows = len(grid)
    columns = len(grid[0])
```

```
if (row < 0) or (row >= rows) or (col < 0) or (col >= columns) \setminus
  or (grid[row][col] == BLOCKED) or (grid[row][col] == VISITED):
    return False
                                      # dead end (base case 1)
if source == dest:
                                      # dest found (base case 2)
    return True
grid[row][col] = VISITED
                                      # visit this cell
if dfs(grid, (row, col + 1), dest):
                                     # search east
    return True
                                      # and return if dest found
if dfs(grid, (row + 1, col), dest):
                                      # else search south
    return True
                                      # and return if dest found
if dfs(grid, (row, col - 1), dest):
                                      # else search west
    return True
                                      #
                                        and return if dest found
if dfs(grid, (row - 1, col), dest):
                                      # else search north
                                         and return if dest found
    return True
                                      #
return False
                                      # destination was not found
```

The variable names BLOCKED, VISITED, and OPEN represent the possible status of each cell. For example, the grid in Figure 9.16 is represented by

grid = [[BLOCKED, OPEN, BLOCKED, OPEN, OPEN], [OPEN, OPEN, BLOCKED, OPEN, OPEN], [BLOCKED, OPEN, OPEN, BLOCKED, OPEN], [OPEN, OPEN, BLOCKED, OPEN, BLOCKED], [OPEN, OPEN, OPEN, OPEN, BLOCKED]]

When a cell is visited, its value is changed from OPEN to VISITED by the dfs function. There is a program available on the book website that includes additional turtle graphics code to visualize how the cells are visited in this depth-first search. Download this program and run it on several random grids.

**Reflection 9.34** Our dfs function returns a Boolean value indicating whether the destination was reached, but it does not actually give us the path (as marked in blue in Figure 9.16). How can we modify the function to do this?

This modification is actually quite simple, although it may take some time to understand how it works. The idea is to add another parameter, a list named path, to which we append each cell after we mark it as visited. The values in this list contain the sequence of cells visited in the recursive calls. However, we remove the cell from path if we get to the end of the function where we return False because getting this far means that this cell is not part of a successful path after all. In our example in Figure 9.16, initially coloring a cell blue is analogous to appending that cell to the path, while recoloring a cell light blue when backtracking is analogous to removing the cell from the path. We leave an implementation of this change as an exercise. Two projects at the end of this chapter demonstrate how depth-first search can be used to solve other problems as well.

## Exercises

Write a **recursive** divide and conquer function for each of the following problems. Each of your functions should contain at least two recursive calls.

9.5.1\* Write a recursive divide and conquer function

power(a, n)

that returns the value of  $a^n$ , utilizing the fact that  $a^n = (a^{n/2})^2$  when n is even and  $a^n = (a^{(n-1)/2})^2 \cdot a$  when n is odd. Assume that n is a non-negative integer.

9.5.2. The Fibonacci sequence is a sequence of integers in which each number is the sum of the previous two. The first two Fibonacci numbers are 1,1, so the sequence begins 1, 1, 2, 3, 5, 8, 13, .... Write a function

fibonacci(n)

that returns the  $n^{th}$  Fibonacci number.

- 9.5.3. In the profit function, we defined the left half as ending at index midIndex 1 and the right half starting at index midIndex. Would it also work to have the left half end at index midIndex and the right half start at index midIndex + 1? Why or why not?
- 9.5.4\* The profit function in the text takes a single list as the parameter and calls the function recursively with slices of this list. In this exercise, you will write a more efficient version of this function

profit(prices, first, last)

that does not use slicing in the arguments to the recursive calls. Instead, the function will pass in the entire list in each recursive call, with the two additional parameters assigned the first and last indices of the sublist that we want to consider. In the divide step, the function will need to assign midIndex the index that is midway between first and last (which is usually not last // 2). To find the maximum profit achievable with a list of prices, the function must initially be called with profit(prices, 0, len(prices) - 1).

- 9.5.5. Modify the version of the function that you wrote in Exercise 9.5.4 so that it returns the most profitable buy and sell days instead of the maximum profit.
- 9.5.6. Write a divide and conquer version of the recursive linear search from Section 9.4 that checks if the middle item is equal to the target in each recursive call and then recursively calls the function with the first half and second half of the list, as needed. Your function should return the index in the list where the target value was found, or −1 if it was not found. If there are multiple instances of the target in the list, your function will not necessarily return the minimum index at which the target can be found. (This function is quite similar to Exercise 9.5.4.)
- 9.5.7. Write a new version of the depth-first search function

dfs(grid, source, dest, path)

in which the parameter **path** contains the sequence of cell coordinates that comprise a path from **source** to **dest** in the grid when the function returns. The initial value of **path** will be an empty list. In other words, to find the path in Figure 9.16, your function will be called like this:

```
path = []
if dfs(grid, (1, 1), (3, 0), path):
    print('A path was found: ' + str(path))
else:
    print('A path was not found.')
```

In this example, the final value of path should be

[(1, 1), (2, 1), (3, 1), (4, 1), (4, 0)]

 $9.5.8^*$  Write a recursive function

numPaths(n, row, column)

that returns the number of distinct paths in an empty  $n \times n$  grid from the cell in the given row and column to the cell in position (n-1, n-1). For example, if n = 3, then the number of paths from (0,0) to (2,2) is six, as illustrated to the right.



- 9.5.9. In Section 9.2, we developed a recursive function named **bitStrings** that returned a list of all binary strings with a given length. We can design an alternative divide and conquer algorithm for the same problem by using the following insight. The list of *n*-bit binary strings with the common prefix p (with length less than *n*) is the concatenation of
  - (a) the list of *n*-bit binary strings with the common prefix p + '0' and
  - (b) the list of *n*-bit binary strings with the common prefix p + '1'.

For example, the list of all 4-bit binary strings with the common prefix 01 is the list of 4-bit binary strings with the common prefix 010 (namely, 0100 and 0101) plus the list of 4-bit binary strings with the common prefix 011 (namely, 0110 and 0111).

Write a recursive divide and conquer function

binary(prefix, n)

that uses this insight to return a list of all *n*-bit binary strings with the given prefix. To compute the list of 4-bit binary strings, you would call the function initially with binary('', 4).

## \*9.6 LINDENMAYER SYSTEMS

This section is available on the book website.

# 9.7 SUMMARY AND FURTHER DISCOVERY

Some problems, like many natural objects, "naturally" exhibit self-similarity. In other words, a problem solution is simply stated in terms of solutions to smaller versions of itself. It is often easier to see how to solve such problems recursively than it is iteratively. An algorithm that utilizes this technique is called *recursive*.

We suggested answering five questions to solve a problem recursively:

- 1. What does a subproblem look like?
- 2. Which subproblem solution would be the most useful for solving the original problem?
- 3. How do we find the solution to the original problem using this subproblem solution? Implement this as the recursive step of our recursive function.
- 4. What are the simplest subproblems that we can solve non-recursively, and what are their solutions? Implement your answer as the base case of the recursive function.
- 5. For any possible parameter value, will the recursive calls eventually reach the base case?

We designed recursive algorithms for about a dozen different problems in this chapter to illustrate how widely recursion can be applied. But learning how to solve problems recursively definitely takes time and practice. The more problems you solve, the more comfortable you will become!

## Notes for further discovery

The first epigraph at the beginning of this chapter is from the first page of Benoît Mandelbrot's *The Fractal Geometry of Nature* [38]. The second is from Shakespeare's *Hamlet*, Act II, Scene II.

Aristid Lindenmayer's work was described in a now freely available book titled, *The* Algorithmic Beauty of Plants [52], published in 1990.

# \*9.8 PROJECTS

This section is available on the book website.