

CHAPTER 11

Recursion

11.1 A Bullseye Class

11.2 Our Own List Implementation

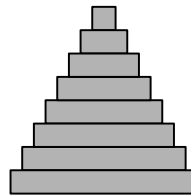
11.3 Functional Recursion

11.4 Binary Search

11.5 Case Study: Solving a Puzzle

11.6 Chapter Review

Computers get much of their power from repetition. For example, in Section 4.3 we used a loop to draw successive levels of the following pyramid:



In this chapter, we introduce another form of repetition known as *recursion* that can be used to design classes and functions. We begin by demonstrating *structural recursion* as a natural way to define objects that have one or more (smaller) instances of the same class as attributes. For example the 8-level pyramid can be viewed as a single bottom level with a 7-level pyramid built on top. That 7-level pyramid is itself a bottom level with a 6-level pyramid on top of it, and so on. Eventually we reach a so-called *base case* in which recursion is no longer necessary; for this example, a 1-level pyramid is a single block. In this chapter we consider two detailed implementations using structural recursion. First we augment our graphics package to include a class that represents a bullseye pattern. Second, we provide our very own (recursive) implementation of a list class.

A second form of recursion, known as *functional recursion*, occurs when a behavior is expressed using a “smaller” version of that same behavior. For example a common childhood game involves one player guessing a number from 1 to 100 while the other gives hints as to whether the true answer is higher or lower than the guess. A wise strategy for a player is to guess the middle value, in this case 50. If that guess is not correct, the hint will narrow the remaining search to either the range 1 to 49 or the range 51 to 100. The strategy at this point can be described as a repetition of the original strategy, although adapted to consider the currently known lower and upper bounds. Much as a function can call another function, it is allowed to (recursively) call itself. We will explore several examples of functional recursion in the latter part of the chapter.

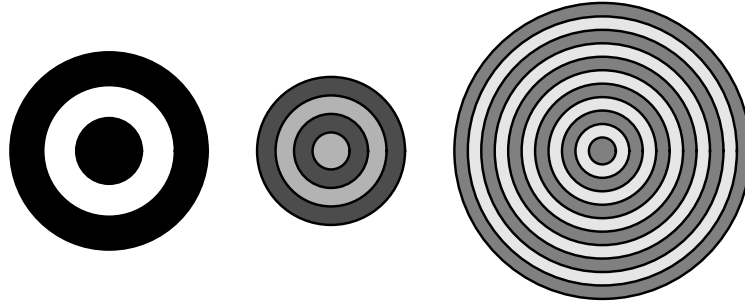
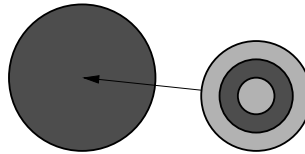


FIGURE 11.1: Bullseye instances with 3, 4, and 11 bands respectively.

11.1 A Bullseye Class

We begin by considering the development of a bullseye class within the framework of our graphics package. We envision a bullseye as a sequence of concentric circles with alternating colors, as shown in Figure 11.1. Although we could create this image using a loop, this is a natural opportunity to demonstrate recursion. A bullseye can be viewed as a single outer circle with a smaller bullseye drawn on top. Of course the smaller bullseye should be positioned so that its center is exactly the same as the outer circle.



We use structural recursion to develop a `Bullseye` class. Internally, a bullseye instance maintains two attributes: `_outer`, which is the outermost circle, and `_rest`, which is a reference to another bullseye providing the interior structure. For the public interface, we allow the user to specify the total number of bands, the overall radius of the bullseye, and the choice of two colors. Complete code for the class is given in Figure 11.2. Our class formally inherits from the `Drawable` class of the `cs1graphics` module so that it can be incorporated as part of that graphics package (see Section 9.4 for similar use of inheritance).

The constructor accepts four parameters (`numBands`, `radius`, `primary`, `secondary`), where the bands alternate in color starting with `primary` as the outermost band. At line 10, we invoke the parent constructor to establish the relevant state of a `Drawable` object (as is standard when using inheritance). Lines 11 and 12 create a single circle to represent the outermost band of our bullseye. As such, its radius is set to the desired radius of the overall bullseye and its color is set to `primary`. Lines 14–18 are used to establish the rest of the bullseye. If the caller requests a bullseye with a single band, then the outer circle trivially suffices. In this case, we set `self._rest` to `None` at line 15 to signify that there is no rest of the bullseye. This serves as our base case. More generally, we establish the rest of the bullseye as a smaller version of a bullseye. The inner bullseye is constructed at line 18 with different parameters than the original, having a smaller radius and inverted use of colors.

```
1 from cs1graphics import *
2
3 class Bullseye(Drawable):
4     def __init__(self, numBands, radius, primary='black', secondary='white'):
5         if numBands <= 0:
6             raise ValueError('Number of bands must be positive')
7         if radius <= 0:
8             raise ValueError('radius must be positive')
9
10        Drawable.__init__(self)           # must call parent constructor
11        self._outer = Circle(radius)
12        self._outer.setFillColor(primary)
13
14        if numBands == 1:
15            self._rest = None
16        else: # create new bullseye with one less band, reduced radius, and inverted colors
17            innerR = float(radius) * (numBands-1) / numBands
18            self._rest = Bullseye(numBands-1, innerR, secondary, primary)
19
20        def getNumBands(self):
21            bandcount = 1                 # outer is always there
22            if self._rest:                 # still more
23                bandcount += self._rest.getNumBands()
24            return bandcount
25
26        def getRadius(self):
27            return self._outer.getRadius() # ask the circle
28
29        def setColors(self, primary, secondary):
30            self._outer.setFillColor(primary)
31            if self._rest:
32                self._rest.setColors(secondary, primary) # color inversion
33
34        def _draw(self):
35            self._beginDraw()             # required protocol for Drawable
36            self._outer._draw()           # draw the circle
37            if self._rest:
38                self._rest._draw()        # recursively draw the rest
39            self._completeDraw()          # required protocol for Drawable
```

FIGURE 11.2: Complete code for our Bullseye implementation (documentation excluded).

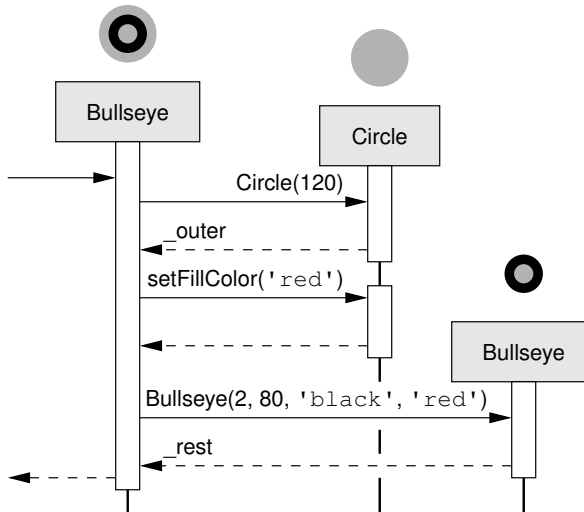


FIGURE 11.3: Top-level trace of Bullseye(3, 120, 'red', 'black')

Unfolding a recursion

There are two ways to envision the recursive process. When we discussed the flow of control in Section 5.2.1, we drew an analogy between calling a function and hiring a contractor to do some work. The original caller interacts with the contractor, yet does not follow the precise details of how that work is accomplished. As an example consider the construction of a three-banded bullseye, `Bullseye(3, 120, 'red', 'black')`. Figure 11.3 portrays a top-level trace of the process. We first see the creation and configuration of the outer circle, and then the creation of an inner bullseye, instantiated as `Bullseye(2, 80, 'black', 'red')`. In some sense, it is just that easy. But there is more happening behind the scene. Our code defining the creation of a bullseye relies upon the presumption that we have means for creating another bullseye (at line 18). How is that other bullseye created? Using the same algorithm!

To truly understand recursion, it helps to more carefully trace the complete execution through a process called *unfolding a recursion*. Figure 11.4 provides such a diagram for our sample bullseye. If you take the perspective of the original bullseye, portrayed on the far left of the diagram, its interactions are identical to that shown in Figure 11.3. It initiates the construction of the outer circle, the coloring of that circle, and the construction of an inner bullseye. What is new in Figure 11.4 is the detailed portrayal of the subsequent construction `Bullseye(2, 80, 'black', 'red')`. It progresses by creating a black outer circle with radius 80 and then its own inner bullseye, with parameters `Bullseye(1, 40, 'red', 'black')`. The last case is traced as well, but the construction algorithm proceeds differently because the desired number of bands is one. An appropriate outer circle is constructed and colored, but the rest of this bullseye is set to **None**.

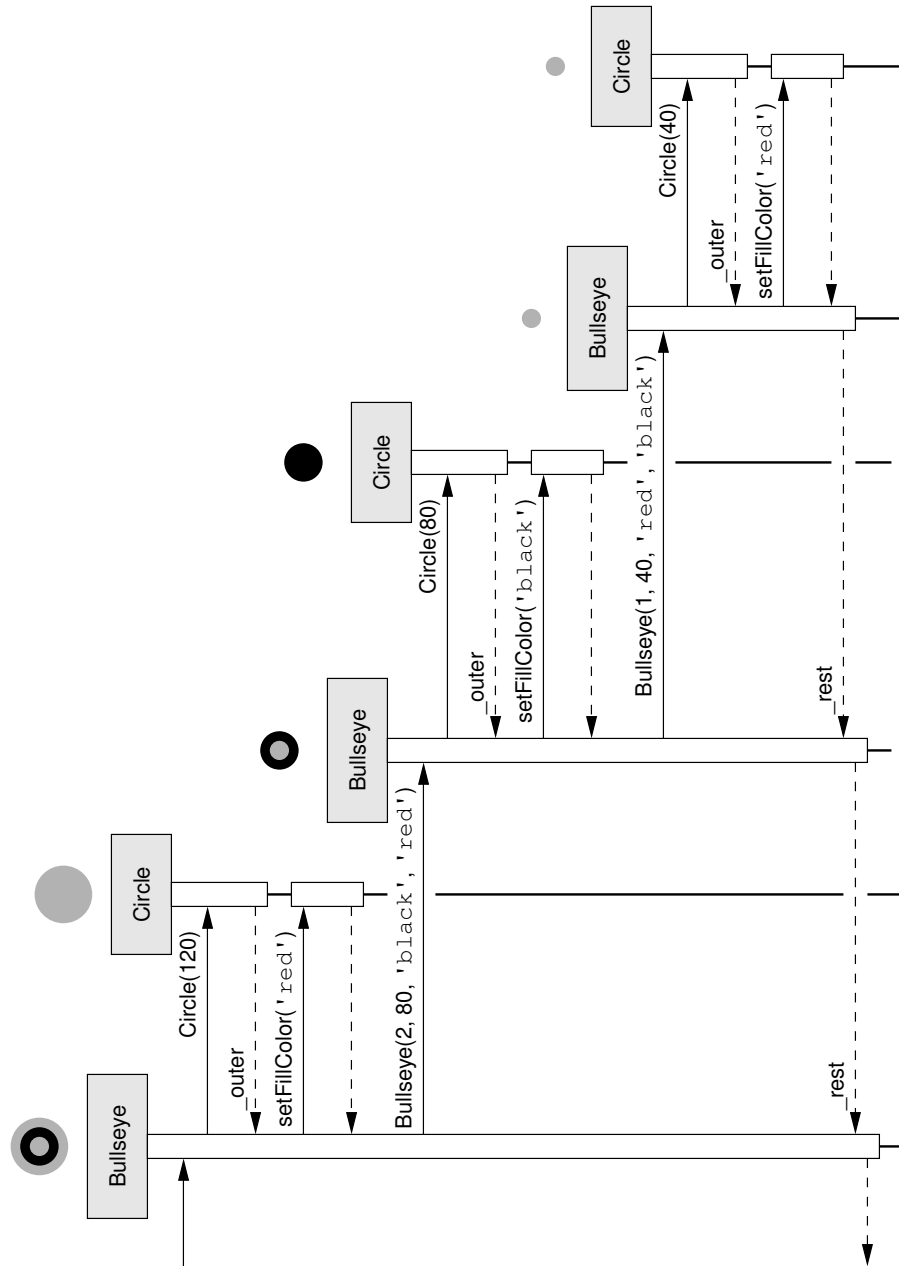


FIGURE 11.4: Unfolding the recursion Bullseye(3, 120, 'red', 'black')

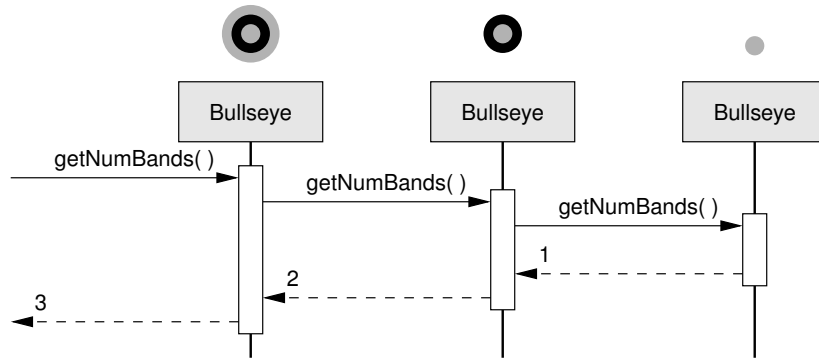


FIGURE 11.5: Sequence diagram for the call `getNumBands()`.

Additional Bullseye methods

Our class supports several additional methods. The `getNumBands` method, at lines 20–24, returns the total number of bands to the caller. Although we did not explicitly record this value it is easily recalculated with the following recursion:

```

20 def getNumBands(self):
21     bandcount = 1                # outer is always there
22     if self._rest:             # still more
23         bandcount += self._rest.getNumBands()
24     return bandcount
    
```

Each bullseye has an outer band, accounted for at line 21. If that were the only band then the answer is one. However, if we find that an inner bullseye exists, we must account for the bands that are contained within that inner portion. We use recursion at line 23 to ask the inner bullseye how many bands it contains, and then add that to our band count to get the final answer that we return to the original caller. We trace a sample execution of this method in Figure 11.5.

In contrast to the implementation of `getNumBands`, the `getRadius` method is naturally expressed without any use of recursion. To determine the radius of our bullseye, we simply need to know the radius of the outer circle.

```

26 def getRadius(self):
27     return self._outer.getRadius()    # ask the circle
    
```

We note that the call to `getRadius` at line 27 is *not* recursive; rather this invokes the `getRadius` method of the `Circle` instance (not of another `Bullseye` instance). This is an example of polymorphism, as both classes support a `getRadius` method with different underlying implementations.

The mutator `setColors(primary, secondary)` recolors an existing bullseye. Thinking recursively, we simply change the outer circle to be the desired primary color and then tell the inner bullseye (if any) to recolor itself.

```

29 def setColors(self, primary, secondary):
30     self._outer.setFillColor(primary)
31     if self._rest:
32         self._rest.setColors(secondary, primary)           # color inversion

```

The only catch is that we need to instruct that inner bullseye to recolor itself so that it uses our secondary color as its primary, and vice versa. This inversion of the color parameters is quite similar to the technique we used in our original constructor.

Finally, having inherited from `Drawable`, our bullseye automatically supports interesting behaviors such as `move`, `scale`, `rotate`,¹ and `clone`. However, based on the protocol of `cs1graphics`, we are responsible for implementing the `_draw` method to display the bullseye. The core of our routine includes the following commands:

```

36     self._outer._draw()                                     # draw the circle
37     if self._rest:
38         self._rest._draw()                                 # recursively draw the rest

```

Notice that we always draw the outer circle, and in the case where there remains a rest of the bullseye, we draw that (recursively) on top of the outer circle. The order of these commands is significant to achieve the desired effect, as the depth attributes are not relevant when implementing the low-level `_draw`.



A WORD OF WARNING

Every recursion must have a base case. Although recursion is a powerful technique, there must always exist at least one case that can be described without further use of recursion. Just as an improperly formed loop can repeat infinitely, so can an ill-formed recursion. Using our contractor analogy, we cannot have everyone passing off work to another person. For recursion to work, we must reach a scenario in which a person handles a request without assistance.

In our `Bullseye` class, the case of a single-banded bullseye serves as the base case. This is true not just of the initial construction, but of our implementation of the various behaviors. For example, a single-banded bullseye answers `getNumBands` without a recursive call because it recognizes its own lack of an inner bullseye.

1. Admittedly, rotating a bullseye is not the most interesting behavior. But rotations are helpful for other recursive figures.

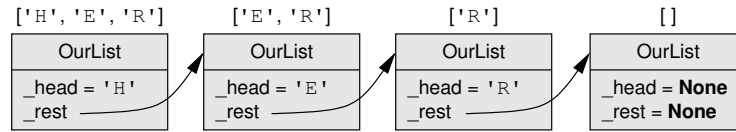


FIGURE 11.6: Underlying recursive structure for our list ['H', 'E', 'R'].

11.2 Our Own List Implementation

As our next example, we develop our own list implementation from scratch. Obviously, Python already includes the built-in `list` class,² so there is no need for us to do this. At the same time, it is empowering to realize that we can develop such an interesting class ourselves, had it not already been provided. The idea is to view a list as a recursive structure. Each list will be represented with two attributes: `_head`, which represents the first element of the list, and `_rest`, which is itself a list of all remaining items. So a list of three elements is represented as a first element and a remaining list of two items. That list of two items is represented as a single element followed by a remaining list of one item. Even the list of length one is viewed as a single element followed by a list of zero remaining elements. We use an empty list as our base case and represent it by setting both `_head` and `_rest` to `None`. An example of our representation is shown in Figure 11.6, for the list ['H', 'E', 'R'].

Preliminaries

To begin our class definition, we provide a constructor that produces an initially empty list.

```
class OurList:
    def __init__(self):
        self._head = None
        self._rest = None
```

Many of our remaining methods use an empty list as the base case for the recursion, so to improve the legibility of these other methods, we introduce a private utility function `_isEmpty` that determines whether a given list matches the empty configuration.

```
def _isEmpty(self):
    return self._rest is None
```

append method

The `append` method is responsible for adding a given element to the end of the list. With our recursive view, this is rather straightforward. If we are adding a value to an empty list, we make the necessary modifications directly. That empty list should be turned into a list of one item, with the new value at its head and a new empty list to represent the rest. Alternatively, if we receive a request to append a value to a nonempty list, we pass the buck.

2. For the record, the actual implementation used for Python’s version of `list` is not recursive; see Chapter 12 for discussion.

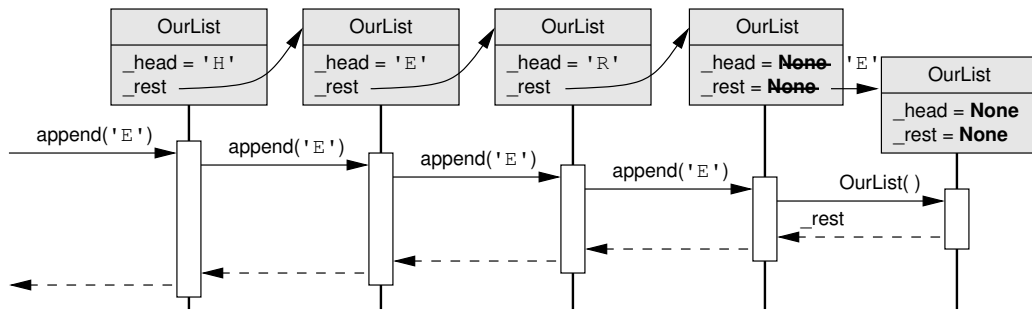


FIGURE 11.7: A call to `append('E')` upon list `['H', 'E', 'R']`.

Appending the new item to the end of the *remaining* sublist serves the purpose of adding it to the end of the complete list. These intuitions lead to the following implementation:

```
def append(self, value):
    if self._isEmpty():
        self._head = value           # we now have one element
        self._rest = OurList()      # followed by new empty list
    else:
        self._rest.append(value)     # pass it on
```

A sample trace of this code is shown in Figure 11.7. As the recursion unfolds we eventually reach our base case, and therefore make the local modification to reflect the change.

count method

We next consider the `count` method. This accessor is responsible for counting the number of occurrences of a given value on the list. We implement it recursively as follows:

```
def count(self, value):
    if self._isEmpty():
        return 0
    else:
        subtotal = self._rest.count(value) # recursion
        if self._head == value:           # additional match
            subtotal += 1
        return subtotal
```

An empty list serves as our base case, as it clearly has a count of zero. Otherwise, we use recursion to ask the rest of our list how many occurrences of the given value it has. Once we know that piece of information, we are almost ready to answer the question that was asked of us. But we must also consider the head of our list. If it matches the target value then the overall number of occurrences on our list is one more than that reported by our sublist.

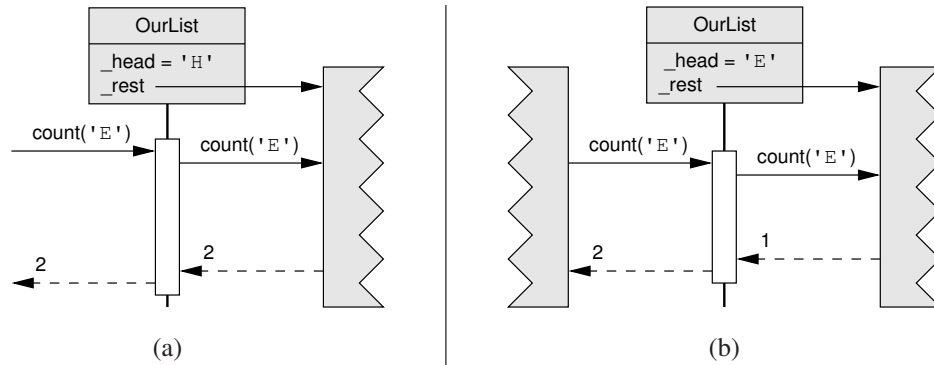


FIGURE 11.8: A local view of calls to `count('E')` upon list `['H', 'E', 'R', 'E']`. (a) Shows a view of the initial call; (b) shows a view of the next call.

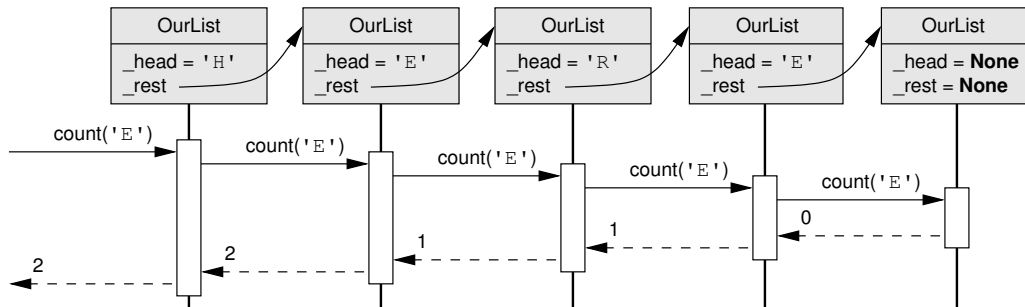


FIGURE 11.9: The complete trace of `count('E')` upon list `['H', 'E', 'R', 'E']`.

The code for `count` is written from the perspective of a single level of the recursion. We can visualize the perspective of a single object as shown in Figure 11.8. Part (a) of that figure shows the top-level call. From the perspective of that instance, it learns (through recursion) that the remainder of the list has two occurrences of E. Given that its head is an H, it concludes that its entire list has only those two occurrences. Part (b) shows the perspective of the secondary recursion. That object is asked how many E’s are on its list, and determines that there are two, one on its sublist and one at the head. If we were to stitch together all such local views, we get the fully unfolded recursion as shown in Figure 11.9.

The `__contains__` method

The `__contains__` method is used to provide support for a shorthand syntax of the form `val in data`. This method is responsible for returning `True` when the value is found within the list and `False` otherwise. What makes the style of this recursion different from `count` is the form of the base case. For `count`, the base case is an empty list. In any other scenario, there is no way to accurately report the count without examining the sublist. Our implementation of `__contains__` relies upon two possible base cases, as shown in the following code.

```

def __contains__(self, value):
    if self._isEmpty():
        return False
    elif self._head == value:
        return True
    else:
        return value in self._rest           # recurse

```

If a list is empty, then clearly the element is not found. Alternatively, if the head of our list matches the target, then we may confidently return **True** without bothering to look at the remaining sublist. It is only in the third case that we apply recursion. If the value occurs on that sublist, then it occurs in the full list; if not on the sublist, it does not occur anywhere. We express this logic simply by returning the result of the expression `value in self._rest`, relying upon the implicit recursion that is used to evaluate the `in` operator.

The `__getitem__` method

A different recursive pattern occurs in our implementation of the `__getitem__` method, which is used to support a syntax such as `data[i]` for retrieving the element at a specified index of a list. There are several interesting aspects of the following implementation:

```

def __getitem__(self, i):
    if self._isEmpty():
        raise IndexError('list index out of range')
    elif i == 0:
        return self._head
    else:
        return self._rest[i-1]             # recurse

```

To begin, if someone attempts to access an element of an empty list, we immediately raise an `IndexError`, in accordance with the behavior of Python’s built-in `list` class in this situation (see Section 5.5 for a discussion of raising exceptions). This serves as an important base case, guarding against use of an illegal index (see Exercise 11.14). Assuming our list is nonempty, the next task is to see whether the given index is 0. If so, the user is interested in the head of our list and we answer the question without examining the rest of the list. In the remaining case, we rely on the use of recursion, although with a changing parameterization. Retrieving the *i*-th element of a given list is equivalent to retrieving the (*i*-1)th element of the rest of the list. A sample trace of this behavior is shown in Figure 11.10.

Completing our class

With practice, we can go on to develop recursive implementations for almost all of the behaviors supported by Python’s lists. A reasonably complete implementation of the class is given in Figure 11.11. Most of those methods are accomplished using similar techniques to the methods we have discussed. For example, `count` is quite similar to `__len__`, reaching an empty list as a base case. The three cases of `__contains__` are very similar to those of `index`, and `__getitem__` closely parallels `__setitem__`. We also provide an

372 Chapter 11 Recursion

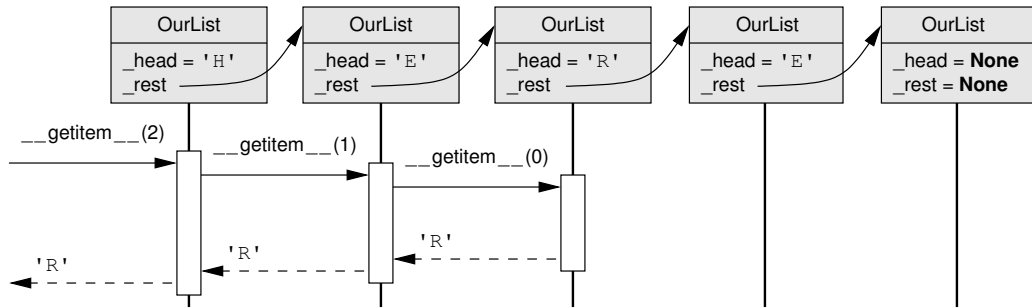


FIGURE 11.10: The underlying evaluation for the syntax `data[2]`, using data `['H', 'E', 'R', 'E']`.

implementation of `__repr__` designed to mimic Python’s list representation. The mutators `insert` and `remove` are similar to `append` in spirit, but their details are considerably more subtle. Figure 11.11 includes a correct implementation of those routines, although we suggest tracing through several sample executions to better understand the process (see Exercise 11.12 and Exercise 11.13).

```

1 class OurList:
2     def __init__(self):
3         self._head = None
4         self._rest = None
5
6     def isEmpty(self):
7         return self._rest is None
8
9     def len(self):
10        if self.isEmpty():
11            return 0
12        else:
13            return 1 + len(self._rest)           # recurse
14
15    def count(self, value):
16        if self.isEmpty():
17            return 0
18        else:
19            subtotal = self._rest.count(value)   # recursion
20            if self._head == value:             # additional match
21                subtotal += 1
22            return subtotal

```

FIGURE 11.11: The implementation of `OurList` (continued on next page).

```
23 def __contains__(self, value):
24     if self._isEmpty():
25         return False
26     elif self._head == value:
27         return True
28     else:
29         return value in self._rest          # recurse
30
31 def index(self, value):
32     if self._isEmpty():
33         raise ValueError('OurList.index(x): x not in list')
34     elif self._head == value:
35         return 0
36     else:
37         return 1 + self._rest.index(value)  # look in remainder of the list
38
39 def __getitem__(self, i):
40     if self._isEmpty():
41         raise IndexError('list index out of range')
42     elif i == 0:
43         return self._head
44     else:
45         return self._rest[i-1]            # recurse
46
47 def __setitem__(self, i, value):
48     if self._isEmpty():
49         raise IndexError('list assignment index out of range')
50     elif i == 0:
51         self._head = value
52     else:
53         self._rest[i-1] = value          # recurse
54
55 def __repr__(self):
56     if self._isEmpty():
57         return ' [] '
58     elif self._rest._isEmpty():
59         return ' [' + repr(self._head) + ' ] '
60     else:
61         return ' [' + repr(self._head) + ', ' + repr(self._rest)[1:] # remove extra [
```

FIGURE 11.11 (continuation): The implementation of OurList (continued on next page).

```

62  def append(self, value):
63      if self._isEmpty():
64          self._head = value           # we now have one element
65          self._rest = OurList()      # followed by new empty list
66      else:
67          self._rest.append(value)    # pass it on
68
69  def insert(self, index, value):
70      if self._isEmpty():             # inserting at end; similar to append
71          self._head = value
72          self._rest = OurList()
73      elif index == 0:                # new element goes here!
74          shift = OurList()
75          shift._head = self._head
76          shift._rest = self._rest
77          self._head = value
78          self._rest = shift
79      else:                           # insert recursively
80          self._rest.insert(index-1, value)
81
82  def remove(self, value):
83      if self._isEmpty():
84          raise ValueError('OurList.remove(x): x not in list')
85      elif self._head == value:
86          self._head = self._rest._head
87          self._rest = self._rest._rest
88      else:
89          self._rest.remove(value)

```

FIGURE 11.11 (continuation): The implementation of OurList.

11.3 Functional Recursion

Our earlier examples of recursion are structural, as we have objects whose states include references to similar objects. Yet those examples also demonstrate the principle of functional recursion, as most of the behaviors depended upon recursive calls to the same algorithm. In the remainder of the chapter, we wish to demonstrate several classic examples of functional recursion (in the absence of any structural recursion).

We begin by discussing a simple mathematical example. The factorial of a number, commonly written as $n!$, is an important combinatorial concept. It represents the number of ways that n items can be ordered (so-called *permutations*). It is defined as

$$n! = n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdots 3 \cdot 2 \cdot 1$$

since there are n possible choices for the first item, after which $n-1$ possible choices for the second item, $n-2$ choices for the third, and so on. Given this formula, it would

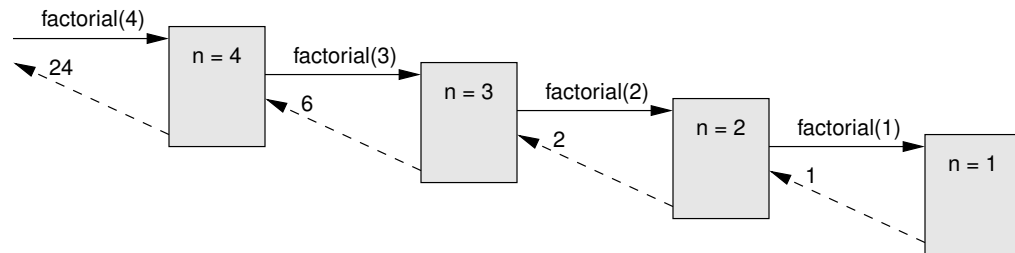


FIGURE 11.12: The trace of a call to factorial(4).

be quite easy to compute the factorial of a number using a loop. However, we wish to demonstrate a simple recursive approach. The key observation is the recognition that the formula corresponding to $(n-1)!$ factorial appears as a part of the $n!$ formula.

$$n! = n \cdot \underbrace{(n-1) \cdot (n-2) \cdot (n-3) \cdots 3 \cdot 2 \cdot 1}_{(n-1)!}$$

We see that $n! = n \cdot (n-1)!$ in general; as a base case we know that $1! = 1$. Going back to our definition of a factorial as the number of ways of arranging n items, we can view the recursion intuitively. We can select any of the n items to be first, and for each such choice we have $(n-1)!$ ways to subsequently order the remaining $n-1$ items. Turning this idea into code, we define the following recursive function:

```

def factorial(n):
    """Compute the factorial of n.

    n is presumed to be a positive integer
    """
    if n <= 1:
        return 1
    else:
        return n * factorial(n-1)
  
```

In our earlier examples of structural recursion, we had a clear notion of distinct objects with their own states. We demonstrated the interactions between these distinct objects using sequence diagrams such as Figure 11.5. When dealing with a pure functional recursion, there are no such objects in play. Yet behind the scene, there is an underlying execution sequence and even a concept of state. This goes back to a lesson from *For the Guru* on page 171 of Section 5.2. Each time a function is called, the system creates an activation record to track the state of that particular invocation. In the context of recursion, each individual call to the function executes the same body of code yet with a separate activation record. We visualize this unfolding recursion as shown in Figure 11.12.

11.4 Binary Search

Assume that we have a list of values and we want to know whether a specific value occurs in that list. This is precisely the functionality provided by the `__contains__` method of Python’s `list` class. In fact for the remainder of the chapter, we presume that we are back to using the built-in `list` class supported by Python (as opposed to our own list implementation from earlier in this chapter). Behind the scene, the algorithm used by Python’s `list` class is known as a *sequential search*. It starts scanning from the beginning of the list until it either finds what it is looking for, or reaches the end of the list and thus reports a failure. In fact, we even considered a sequential search as an example of a `while` loop on page 162.

In this section, we focus our attention on a similar problem, but this time assuming that the list of values is known to be *sorted*. As a concrete example, we revisit the concept of a *lexicon* (a list of strings). We originally considered the task of reading a lexicon from a file as an example on page 280 of Chapter 8. Although we did not presume at the time that the strings were alphabetized in the file, we do so now (or we could explicitly call `lexicon.sort()` after reading the file). In real life, people commonly maintain lexicons in sorted order (e.g., phone books, encyclopedias, guest lists). The reason is that searching within a sorted list is *much* easier. In this section, we leverage our intuition to develop an efficient implementation for searching a sorted list.

11.4.1 Algorithmic Design

Imagine that you are handling security at a party and you have been provided a list of invited guests. As each person arrives you are responsible for checking whether that individual was invited. This is a very large party so there are many invitees and also a rather long line of people waiting to get into the party. It is important that you are as quick as possible in determining whether each name is on the list. If the names on the guest list were arbitrarily ordered, you would have no choice but to rely upon a sequential search. This can be very time consuming; each time someone arrives at the door, you may have to scan the entire list. It is much easier to look for an individual if the guest list is alphabetized. Why is this the case? How might you solve the problem?

The key intuition is the following. Rather than scanning the list from beginning to end, you can jump to a name near the middle of the list and compare that name to the one for which you are looking (let us call this the *target* name). If you are extremely lucky, that middle name may be the target. But the efficiency of our technique is not contingent on us being that lucky. The real key is that even if the middle name is not the same as our target, we get important information by considering whether the target should appear before or after that point in the alphabetized list. If our target should appear before the considered name, then we only have to continue our search on the first half of the original list; similarly, if the target should appear after the middle name, then we only have to search the second half of the original list. With this technique, we are guaranteed to make significant progress in paring down the effective size of the list. More importantly, after narrowing our search to half the list we do not revert to performing a sequential search; we apply recursion! That is, we again consider a name near the middle of the relevant portion of the list and we either find the target or further pare down the relevant portion to half its current size. Continuing in this fashion, we eventually find the target or reach a point when we have eliminated the entire list (and thus conclude that the person was uninvited).

This technique of searching a sorted list is commonly termed *binary search*, because each step narrows the list into one of two halves. What is amazing is how much faster binary search can be versus a sequential search. Imagine that we start with a list of 1000 names. After one step of our process, we either find the target or reduce the range to 500 names. Even if we are not lucky, after a second step, we are down to at most 250 names, after three steps down to at most 125 names, after four steps, at most 62 names and so on. In fact after at most 10 such steps we will have either found the target or eliminated all possibilities. This is quite an improvement, comparing our target to a selection of only 10 names rather than sequentially comparing to all 1000 original names.

Of course, computers are very fast machines. Even a sequential search of 1000 names is evaluated quickly. But computers must often process significantly longer lists. A sequential search of millions or billions of items can become time consuming, even for fast machines. To analyze the cost of a binary search, we note that the overall number of steps is related to the number of times that we can repeatedly divide a list in half before running out of items. Mathematically, this is equivalent to the concept of a *logarithm*, specifically a logarithm with base two. So if we start with n original entries, there will be at most $\log_2 n$ steps of a binary search. Putting this into perspective, we already noted that binary search over a list of 1000 entries requires at most 10 steps. For a list of one million items, it requires at most 20 steps; for a list of one billion items, at most 30 steps. Here we see the real victory. Rather than sifting through each of a billion entries, we can determine whether a value is on the list by examining only 30 well-chosen entries.

11.4.2 A Poor Implementation

Our goal is to convert this high-level idea into an actual implementation. Presumably, a reasonable interface for a user would be to provide a general function `search(lexicon, target)` that returns **True** if the target is found on a given list of strings and **False** otherwise.

In this section, we intentionally provide an awful implementation of the binary search algorithm. This implementation will seem natural and look innocent enough. In fact, it correctly determines the answer. The problem is that it is horribly inefficient — even worse than sequential search. Yet we choose to show this approach first because it is important to understand why it is so flawed. The code for this poor version is shown in Figure 11.13. The conditional at lines 7 and 8 provides an important base case.

```

7   if len(lexicon) == 0:                               # base case
8       return False

```

If we are asked about an empty list, clearly the target is not contained. In fact this is the only case where we definitively conclude that the target is absent.

In any other case, we consider comparing the target to the element at the middle of the list. The concept of the middle is well defined when the list has odd length. If the list has even length, we consider the element slightly right of center. Formally, our definition of the middle index is `len(lexicon) // 2`, as computed at line 10. As an example, if a list has length 7, the middle index is set to 3 (it truly is the middle of indices $\{0, 1, 2, \mathbf{3}, 4, 5, 6\}$). If a list has length 6, the division results in a choice of 3 for the “middle” (although there is slight asymmetry in $\{0, 1, 2, \mathbf{3}, 4, 5\}$).

```

1 def search(lexicon, target):
2     """Search for the target within lexicon.
3
4     lexicon    a list of words (presumed to be alphabetized)
5     target     the desired word
6     """
7     if len(lexicon) == 0:                                # base case
8         return False
9     else:
10        midIndex = len(lexicon) // 2
11        if target == lexicon[midIndex]:                  # found it
12            return True
13        elif target < lexicon[midIndex]:                # check left side
14            return search(lexicon[:midIndex], target)
15        else:                                           # check right side
16            return search(lexicon[midIndex+1:], target)

```

FIGURE 11.13: A bad implementation of the binary search algorithm.

Continuing our examination of the code, lines 11–16 contain the heart of the binary search algorithm. We compare the target to the middle element of the list. One of three things will happen. If we find an exact match, then we immediately report success. Otherwise, if the target is less than the middle item (in alphabetical order), we need only check the left side of the list. We accomplish this at line 14 by recursing upon the slice `lexicon[:midIndex]`. Recalling the definition of a slice, this is a list that contains the elements from the beginning of the original list, up to but not including the item at `midIndex`. We intentionally exclude that middle item because we have already determined that it was not a match. Lines 15 and 16 handle the symmetric case, where we wish to search the portion of the list strictly beyond the middle index.

This implementation is technically correct. As an illustrative example, Figure 11.14 gives a trace of the call `search(['B', 'E', 'G', 'I', 'N', 'S'], 'F')` upon this alphabetized list. For the initial call, the length of the lexicon is 6 and so the value of `midIndex` is set to `6 // 2` which is 3. Then the target 'F' is compared to the middle entry 'I'. This is not a match and since `'F' < 'I'`, line 14 of the code is executed. Therefore we recurse on the list `lexicon[:3]`, which in this case is `['B', 'E', 'G']`. The activation of that recursion is on a list with length 3, and so `midIndex` is set to `3 // 2` which is 1. Therefore the target is compared to 'E' and as `'F' > 'E'`, line 16 is executed with the slice `lexicon[2:]`, which is simply `['G']`. This list has length 1, and so we set `midIndex` to `1 // 2` which is 0 and so our target is compared to 'G'. We find that `'F' < 'G'` and again line 14 is executed with `lexicon[:0]`. This slice is technically an empty list, as there are no elements at index strictly before 0. So we reach a base case as `search([], 'F')` is called. Having realized that 'F' is not contained in the empty list, we see that 'F' is not on the list `['G']`, nor on the list `['B', 'E', 'G']` nor on the original list `['B', 'E', 'G', 'I', 'N', 'S']`.

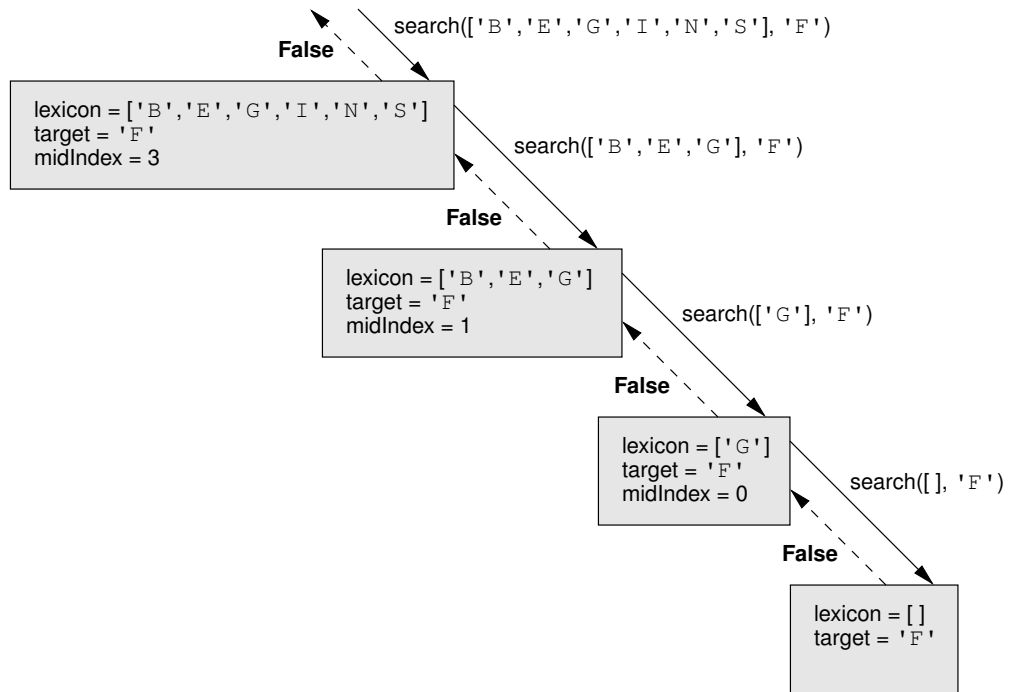


FIGURE 11.14: The trace of the call `search(['B', 'E', 'G', 'I', 'N', 'S'], 'F')` for the inferior implementation of binary search from Figure 11.13.

So what is wrong?

This version of the code correctly answers the question. However, it is very inefficient. Our implementation relies upon the use of Python’s **list** class and in particular the following three operations upon those lists: determining the length of a list (lines 7 and 10), retrieving the middle element of a list (lines 11 and 13), and creating a slice of the list (lines 14 and 16). Python’s lists are very efficient in reporting the length and in retrieving one particular element at a specified index (see Section 13.2 for more discussion). The culprit involves our use of slices when representing a sublist, as with the syntax `lexicon[:midIndex]` to designate the left half of the original list. Although a convenient Python syntax, slicing causes a new list to be created that is essentially a copy of the relevant portion of the original list. Creating this slice takes time proportional to the length of the slice, ruining the potential benefits of the binary search algorithm.

For intuition, consider again the analogy of checking a guest list at the door of a party. When processing a guest list with one thousand entries, we quickly compare the target to the middle entry. So far, so good. But if we determine that the first half of the list must be recursively searched, we do not want to be in a position of having to take time to copy those 500 names onto a new sheet of paper so that the sublist can be passed as a parameter to an assistant. We will have spent so much time creating the copy that we may as well have just looked for the target ourselves.

```

1 def search(lexicon, target, start=0, stop=None):
2     """Search for the target within lexicon[start:stop].
3
4     lexicon    a list of words (presumed to be alphabetized)
5     target     the desired word
6     start      the smallest index at which to look (default 0)
7     stop       the index before which to stop (default len(lexicon) )
8     """
9     if stop is None:
10        stop = len(lexicon)
11    if start >= stop:                                # nothing left
12        return False
13    else:
14        midIndex = (start + stop) // 2
15        if target == lexicon[midIndex]:              # found it
16            return True
17        elif target < lexicon[midIndex]:            # check left side
18            return search(lexicon, target, start, midIndex)
19        else:                                       # check right side
20            return search(lexicon, target, midIndex+1, stop)

```

FIGURE 11.15: Preferred implementation of the binary search algorithm.

11.4.3 A Good Implementation

Having recognized the problem with the preceding implementation, our goal in this section is to provide a truly efficient implementation of binary search. That approach is given in Figure 11.15. The key to our new approach is avoiding the use of explicit slicing when designating a sublist. Instead, we describe a sublist implicitly, passing a reference to the original list as well as the appropriate start and stop indices.

Passing a reference to the list is quite efficient, as it does not involve copying the object (see Section 10.3.1 for further discussion). We mimic slicing notation by using indices `start` and `stop`, with the convention that the search proceeds beginning at index `start` of the lexicon, going up to but not including index `stop`. However, we want to support the original calling syntax, as in `search(['B', 'E', 'G', 'I', 'N', 'S'], 'F')`, so that our user need not be concerned with the extra parameters. We do this by making careful use of default parameter values. We set the value of `start` to zero by default, as this is the leftmost index of a full list. Setting a default for `stop` is more subtle. The problem is that we want to set it to `len(lexicon)`, yet the list `lexicon` does not actually exist at the time we are declaring the function. Trying to use this expression in the signature would cause a syntax error if the default value were evaluated (see Section 10.3.2 for more on default parameter values). Instead, we initially set `stop` to the special value **None** in our signature. The desired value is computed, if necessary, at line 10 within the function body.

Other than the change to an implicit representation of the slice, our new version of the code is modeled upon the preceding bad version. The condition at line 11 is our way to

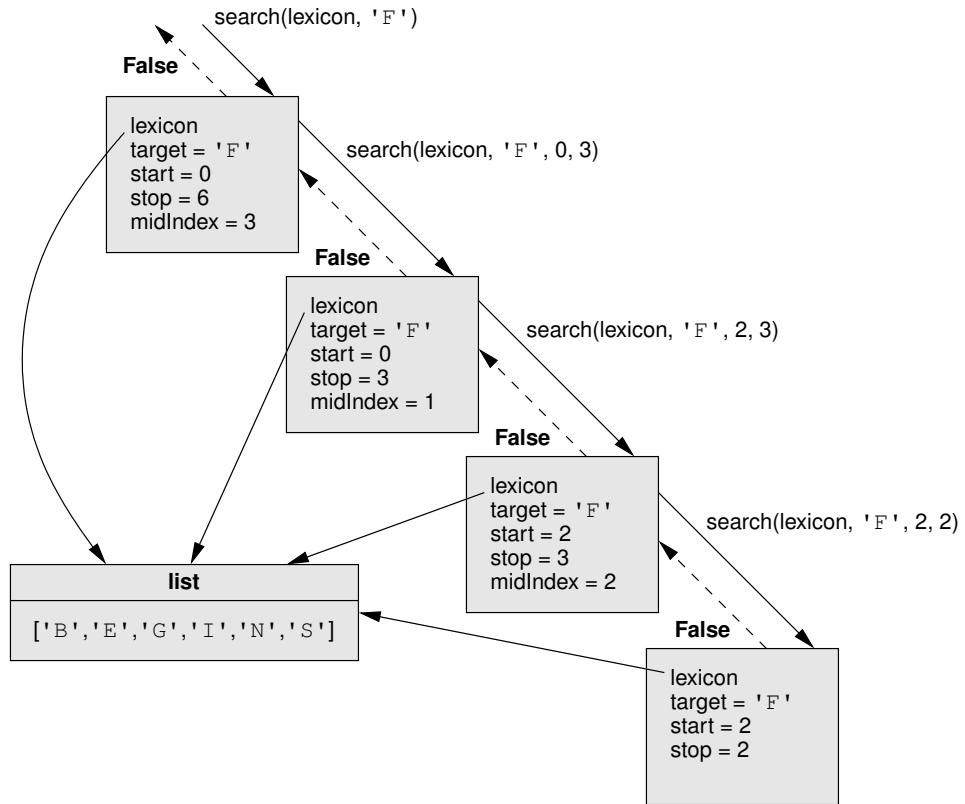


FIGURE 11.16: The trace of the call, `search(['B', 'E', 'G', 'I', 'N', 'S'], 'F')` for the improved implementation of binary search from Figure 11.15.

recognize a request to search an “empty” portion of the list and lines 14 computes the middle index of the current portion of the list and lines 18 and 20 recurse on the appropriate sublist. In Figure 11.16, we retrace `search(['B', 'E', 'G', 'I', 'N', 'S'], 'F')` using the revised implementation. Although very similar in style to the earlier version in Figure 11.14, the new style of parameter passing is more efficient.



A WORD OF WARNING

If you take a slice of a list, this creates a new list that is a copy of the relevant portion of the original (see Section 10.2.2). This may be an unwanted expense in a recursive setting.

11.4.4 Searching for a Partial Match

We originally described binary search as an algorithm for finding an exact match of a value within a sorted list (for example, searching for 1492 within a sorted list of numbers). Yet the approach can be easily adapted for more general types of searches known as *range searches*. For example, we can efficiently locate all numbers from a sorted list that are at least 1450 yet strictly less than 1500.

In the context of a lexicon of words, a common form of a range search is to check if any entry of the lexicon starts with a given *prefix*. For example, we might want to know if any word starts with the pattern `pyt`. Such a prefix-based search is quite easily accomplished, as shown in Figure 11.17. This implementation is almost identical to that given in Figure 11.15. Only one line of code has been changed. Originally, we looked for an exact match using the syntax

```
15 if target == lexicon[midIndex]: # found it
```

This time, we consider the search successful if the middle entry starts with the target string.

```
15 if lexicon[midIndex].startswith(target): # found prefix
```

```
1 def prefixSearch(lexicon, target, start=0, stop=None):
2     """Search to see if target occurs as a prefix of a word in lexicon[start:stop].
3
4     lexicon    a list of words (presumed to be alphabetized)
5     target     the desired word
6     start     the smallest index at which to look (default 0)
7     stop      the index before which to stop (default len(lexicon) )
8     """
9     if stop is None:
10        stop = len(lexicon)
11    if start >= stop:
12        return False
13    else:
14        midIndex = (start + stop)//2
15        if lexicon[midIndex].startswith(target): # found prefix
16            return True
17        elif target < lexicon[midIndex]: # check left side
18            return prefixSearch(lexicon, target, start, midIndex)
19        else: # check right side
20            return prefixSearch(lexicon, target, midIndex+1, stop)
```

FIGURE 11.17: Using binary search to check whether a prefix occurs within a list of words.

11.5 Case Study: Solving a Puzzle

Our next goal is to design a program that computes *anagrams* of a given word. An anagram is defined as a word that can be formed by rearranging the letters of another word. For example the word `trace` can be rearranged to form anagrams including `caret`, `cater`, `crate`, and `react`. We will develop a function that computes anagrams using a syntax such as `anagrams(lexicon, 'trace')`, where `lexicon` is again presumed to be an alphabetized list of words in a language. The return value of this call will be a list of `lexicon` entries that can be formed by rearranging the given string of characters.

We begin by noting a connection between this problem and the discussion of factorials from Section 11.3. Remember that a factorial, written as $n!$, is the number of ways there are to order n items. So if we start with an n -letter word, there are $n!$ possible ways to arrange the characters of that word when looking for anagrams (including the original ordering). Of course our program for computing factorials was only designed to calculate the *number* of orderings. For this case study, we must construct the various rearrangements.

The intuition introduced in the context of factorials can be used to develop a new recursion for computing anagrams. Rather than tackling the entire rearrangement problem at once, we begin by picking the first character. Of course we do not know which character to use as the first; instead we explore each possible choice, one at a time. For one such choice, we subsequently try all possible arrangements of the remaining characters. For example, if using the `c` from `trace` as the first character, we must subsequently decide how to arrange the remaining letters `trae`. We use recursion to handle this subtask. There is one catch in implementing this. The original goal of our function is to find solutions that are words in the `lexicon`. Yet when rearranging the rest of the letters, we need to recognize that `aret`, `ater`, and `rate` are meaningful, because when following the initial `c`, these form the words `caret`, `cater`, and `crate`. We therefore design a recursion that accepts a third parameter to designate an existing prefix. Our precise signature appears as

```
def anagrams(lexicon, charsToUse, prefix=' '):
```

The goal of the function is to return a list of words from the `lexicon` that start with the given prefix and are followed by an arrangement of `charsToUse`. We use a default parameter of an empty string for the prefix so that users may rely upon the two-parameter syntax, such as `anagrams(lexicon, 'trace')`.

The heart of our implementation is based on trying each possible character as the next subsequent character. When facing an initial call of `anagrams(lexicon, 'trace')`, there are five choices for the first character, resulting in subsequent calls of the form:

```
anagrams(lexicon, 'race', 't')
anagrams(lexicon, 'tace', 'r')
anagrams(lexicon, 'trce', 'a')
anagrams(lexicon, 'trae', 'c')
anagrams(lexicon, 'trac', 'e')
```

Each one of these recursive calls will report a list of solutions (possibly empty). The overall list of solutions should be the union of these individual lists. We accomplish this with a simple loop shown at lines 12–16 of Figure 11.18. For each character of `charsToUse` we

```

1 def anagrams(lexicon, charsToUse, prefix=' '):
2     """
3     Return a list of anagrams, formed with prefix followed by charsToUse.
4
5     lexicon        a list of words (presumed to be alphabetized)
6     charsToUse     a string which represents the characters to be arranged
7     prefix         a prefix which is presumed to come before the arrangement
8                   of the charsToUse (default empty string)
9     """
10    solutions = []
11    if len(charsToUse) > 1:
12        for i in range(len(charsToUse)):          # pick charsToUse[i] next
13            newPrefix = prefix + charsToUse[i]
14            newCharsToUse = charsToUse[: i] + charsToUse[i+1 : ]
15            solutions.extend(anagrams(lexicon, newCharsToUse, newPrefix))
16    else:      # check to see if we have a good solution
17        candidate = prefix + charsToUse
18        if search(lexicon, candidate):          # use binary search
19            solutions.append(candidate)
20    return solutions

```

FIGURE 11.18: Preliminary implementation of an anagram solver.

compute a new prefix which includes that character (line 13), we then compute a new string of remaining characters to use that does not include the selected one (line 14), and then we extend our list of solutions by the result of a recursion (line 15).

We must also define a base case for our recursion, namely a situation in which we can determine the answer without need for further recursive applications. Recall that our formal goal is to return a list of words that can be formed by taking the prefix followed by an arrangement of the remaining characters. If there is only one character remaining, we simply have to check whether the complete candidate word is in the lexicon. We rely upon the binary search function we implemented in Section 11.4.3 to perform this check. This base case is handled by lines 17–19 of Figure 11.18.

To fully understand how our anagram solver works, it may help to see a detailed trace as the recursion unfolds. For the sake of illustration, Figure 11.19 considers a very small example, computing anagrams of the word 'are'. The original activation relies upon three separate recursive calls, exploring the use of 'a', 'r', and then 'e' as the first character in a presumed anagram. Of course, the second call `anagrams(lexicon, 'ae', 'r')` does not actually begin until the first call `anagrams(lexicon, 're', 'a')` and all of its ancillary calls completes, returning control back to the original level.

The overall return value is the concatenation of the three lists returned by the secondary calls. We should note that the current version of our recursion does not make any explicit attempt to alphabetize the resulting list or to avoid duplicates. We explore these issues in Exercise 11.34 and Exercise 11.35 respectively.

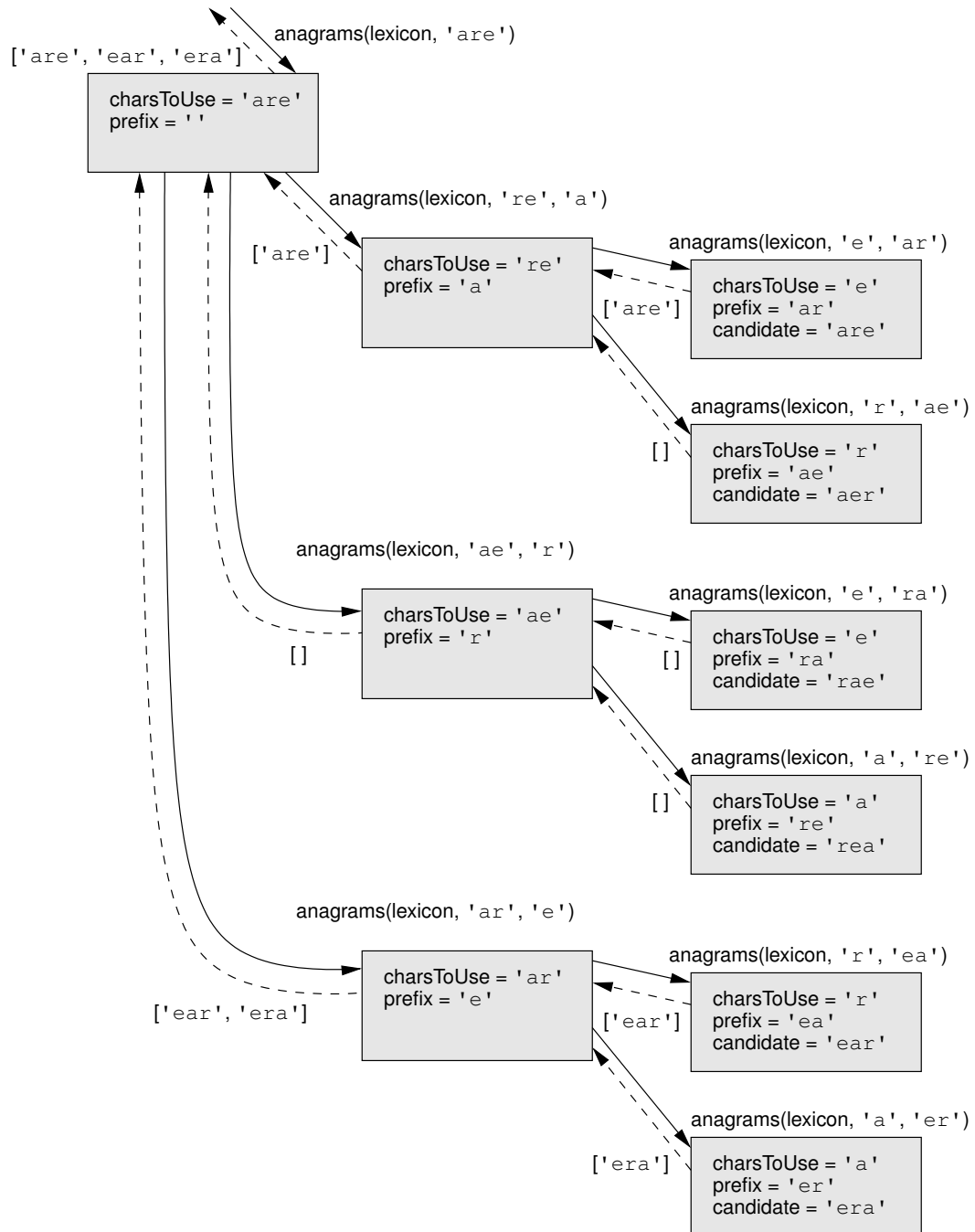


FIGURE 11.19: The trace of a call to `anagrams(lexicon, 'are')`.

11.5.1 Improving Efficiency: Pruning a Recursion

As our final lesson of the chapter, we examine the efficiency of our preliminary anagram solver. The precise computation time depends upon the speed of the computer as well as the size of the lexicon being used. Of course, it also depends greatly on the number of characters in the anagram. For the sake of argument, we report running times observed on our own computer using a lexicon of over 100,000 words. We solve 7-letter anagrams in a fraction of a second, 8-letter anagrams in less than two seconds, even 9-letter anagrams in 15 seconds or so. But things soon get much worse. Finding anagrams of a 10-letter word (such as `coordinate` → `decoration`) takes 2.5 minutes. Analyzing an 11-letter word (e.g., `description` → `predictions`) takes about 28 minutes; we can solve a 12-letter puzzle (e.g., `impersonated` → `predominates`), yet only after 5.6 hours. Projecting this, we would have to wait over 3 days to find that `relationships` and `rhinoplasties` are anagrams and over 42 days to find that `disintegrations` and `disorientating` are anagrams.

The biggest impediment to efficiency is the growing number of ways to rearrange a set of characters, precisely what factorials describe. For example, there are $3! = 3 \cdot 2 \cdot 1 = 6$ ways to rearrange three letters. In fact, we saw those six candidate orderings as base cases in the example of Figure 11.19. There are similarly 24 orderings for a 4-letter anagram and 120 orderings for a 5-letter anagram. The number of arrangements soon grows out of control. There are over 3.6 million ways to arrange 10 letters, 40 million ways for 11 letters, 479 million for 12 letters, 6.2 billion for 13 letters, 87 billion for 14 letters.

Even the fastest computers will require significant time to evaluate this many possibilities, especially given the many intermediate recursions used to generate those candidates and the subsequent search for each candidate within a large lexicon. Figure 11.20 summarizes the results of our own experiments. We measure the performance in two ways. We report the actual time it took to complete (using our computer). Although times may vary on your own computer, the comparison of the running times is telling. The other metric we display is an overall count of the number of intermediate recursive calls that are made along the way for each trial. The performance of the original version of our anagram solver is given in the middle of that figure.

Amazingly, with one simple change to our code, we can process all of these cases efficiently. For example, we can evaluate a 14-letter word in just under 16 seconds (rather than the projected 42 days that our original code would require). Although we might try to speed up the basic computations, the real improvement comes from simply avoiding as much of the work as possible. We use some intuition that would be very obvious to a person trying to solve anagrams by hand. Let's start by unfolding the recursion on the example `anagrams(lexicon, 'trace')`. We begin with a loop that explores each possible first letter. The first such recursive call is to `anagrams(lexicon, 'race', 't')`. Each of the four remaining letters is next considered. Exploring some combinations, such as `'tr'`, lead to a real word (e.g., `trace`). Other prefixes, like `'ta'`, might seem reasonable even though they turn out not to uncover any solutions. Still other prefixes, such as `'tc'`, seem impossible to a person. If there are not any legitimate words in English starting with those letters, then it is a waste of time to consider all six arrangements of the final three letters (i.e., `tcrae`, `tcrea`, `tcare`, `tcaer`, `tcerar`, `tcear`). Yet to avoid such waste, we have to know whether a prefix is impossible. Humans rely on intuition for this, but

| word | len | Original | | Improved | |
|----------------|-----|------------|-----------------|----------|-------------|
| | | time (sec) | #recursions | time | #recursions |
| trace | 5 | 0.006 | 206 | 0.010 | 108 |
| parsed | 6 | 0.031 | 1,237 | 0.040 | 256 |
| editing | 7 | 0.214 | 8,660 | 0.082 | 378 |
| altering | 8 | 1.72 | 69,281 | 0.193 | 1,097 |
| diameters | 9 | 15.3 | 623,530 | 0.521 | 2,625 |
| coordinate | 10 | 153 | 6,235,301 | 0.836 | 3,359 |
| description | 11 | 1668 | 68,588,312 | 1.60 | 5,842 |
| impersonated | 12 | ~5.6 hrs | 823,059,745 | 3.90 | 12,371 |
| relationships | 13 | ~3 days | 10,699,776,686 | 7.11 | 20,228 |
| disintegration | 14 | ~42 days | 149,796,873,605 | 15.8 | 42,714 |

FIGURE 11.20: Efficiency of original and improved anagram implementations.

intuition might be flawed.³ The computer has a way to accurately check whether a prefix occurs within a large lexicon; this is precisely what we accomplished in Section 11.4.4. Therefore, we rewrite the body of the for loop, changing it from the original version

```

13     newPrefix = prefix + charsToUse[i]
14     newCharsToUse = charsToUse[:i] + charsToUse[i+1:]
15     solutions.extend(anagrams(lexicon, newCharsToUse, newPrefix))

```

to the following improved version:

```

13     newPrefix = prefix + charsToUse[i]
14     if prefixSearch(lexicon, newPrefix): # worth exploring
15         newCharsToUse = charsToUse[:i] + charsToUse[i+1:]
16         solutions.extend(anagrams(lexicon, newCharsToUse, newPrefix))

```

We only rearrange the remaining characters when we find that the new prefix exists in the lexicon. This general technique is called *pruning a recursion*.

The code reflecting this change is given in Figure 11.21. Although this may appear to be a minor cosmetic change, its impact is enormous. On small examples, our “improvement” does not actually help; in fact for 5- and 6-letter words, our running times increase slightly. As shown in Figure 11.20, the overall number of recursions has been reduced somewhat in the new version, going from 206 down to 108 in the 5-letter example. However, we spend a bit more time at each step due to the additional prefix search.

On bigger examples, the extra time checking prefixes is well spent. The overall number of recursions still grows with the size of the word, but not nearly as fast. The savings occurs because as we process longer words, the majority of prefixes are not viable. At the extreme, we find that we can handle 14-letter words in a fraction of a minute because we use fewer than 43000 recursions rather than the 149 billion required by the naive approach.

3. Did you remember the word `tchaviche`, which is on the Consolidated Word List for the Scripps National Spelling Bee? It is apparently a breed of salmon.

```

1 def anagrams(lexicon, charsToUse, prefix=' '):
2     """
3     Return a list of anagrams, formed with prefix followed by charsToUse.
4
5     lexicon          a list of words (presumed to be alphabetized)
6     charsToUse      a string which represents the characters to be arranged
7     prefix           a prefix which is presumed to come before the arrangement
8                     of the charsToUse (default empty string)
9     """
10    solutions = []
11    if len(charsToUse) > 1:
12        for i in range(len(charsToUse)):          # pick charsToUse[i] next
13            newPrefix = prefix + charsToUse[i]
14            if prefixSearch(lexicon, newPrefix):  # worth exploring
15                newCharsToUse = charsToUse[ : i] + charsToUse[i+1 : ]
16                solutions.extend(anagrams(lexicon, newCharsToUse, newPrefix))
17    else:    # check to see if we have a good solution
18        candidate = prefix + charsToUse
19        if search(lexicon, candidate):          # use binary search
20            solutions.append(candidate)
21    return solutions

```

FIGURE 11.21: Improved implementation of our anagram solver. This prunes the recursion whenever an impossible prefix occurs.

11.6 Chapter Review

11.6.1 Key Points

General Issues

- Recursion provides another technique for expressing repetition.
- Structural recursion occurs when an instance of a class has an attribute that is itself another instance of the same class.
- Functional recursion occurs when the body of a function includes a command that calls the same function.
- A base case of a recursion is a scenario resolved without further use of recursion.
- One way to envision the recursive process is to maintain the local perspective for one particular recursive call. If the execution of that call causes another call to be made, we simply trust that the other call works and consider its impact on the original task.
- Another way to envision the recursive process is to *unfold* the entire recursion. Starting with the initial call, we can trace the precise execution of all subsequent calls.
- Each activation of a recursive function is managed separately, with the parameters and local variables for that call stored in a dedicated activation record.

OurList Implementation

- A list can be represented recursively by considering the element at the head of the list, and a subsequent list of all remaining items.
- We want to be able to represent an empty list, and so we use that as the base case for our recursion. A list of length one is represented by an element followed by a subsequent empty list.
- The empty list serves as a base case for all of our functions because when a command is issued to the empty list, there is no subsequent list for recursion.
- Several of the methods rely upon a second form of a base case, in scenarios when access to the head of the list suffices.

Binary Search

- A standard technique for finding a value in a list is known as *sequential search*. A loop is used to iterate through the elements one at a time, until either the desired value is found or the end of the list is reached (in which case the search was unsuccessful).
- When a list of values is known to be sorted, there is a more efficient approach known as *binary search*.
- The general idea of binary search is to compare a target value to an item near the middle of the list. If that is not the target value, we can rule out half of the original list based upon whether the target was less than or greater than the middle entry. The remaining sublist can be searched recursively.
- When using Python’s slicing syntax, the slice is a newly constructed list and creating it takes time proportional to the length of that slice. For binary search, all of the potential gains in efficiency are lost if slices are created for recursion.
- Instead of creating slices, an efficient recursion can be designed by sending a reference to the original list, together with indices that delimit the “slice” of interest.
- Binary search can easily be adapted not just for finding an exact match, but also for finding matches that lie within a given range of values.

Solving a Puzzle

- Discrete puzzles can often be solved by recursively constructing and exploring partial solutions until finding a complete solution that suffices.
- The time for solving a puzzle depends greatly on the number of possible solutions that must be explored.
- Such a recursion can be made significantly faster by *pruning* the recursion, that is, by recursing only on those partial solutions that could lead to success.

11.6.2 Glossary

anagram A word that can be formed by rearranging the letters of another word.

base case A case in a recursive process that can be resolved without further recursion.

binary search A technique for searching a sorted list by comparing a target value to the value at the middle of the list and, if not the target, searching recursively on the appropriate half of the list.

functional recursion A technique in which a function calls itself.

lexicon A collection of strings, for example all words or phrases in the English language.

pruning a recursion A technique for speeding recursive searches by avoiding examination of branches that are clearly not useful.

range search A search for elements of a collection that have value in a given range (e.g., from 80 to 90), as opposed to a search for a single value.

recursion A technique in which a structure or behavior is defined based upon a “smaller” version of that same structure or behavior.

sequential search An algorithm used to find a value in a sequence by scanning from beginning to end, until either finding the value or exhausting the entire sequence.

structural recursion A technique in which a structure is defined using a “smaller” version of the same structure.

unfolding a recursion Tracing execution through all levels of a recursion.

11.6.3 Exercises

Bullseye

Practice 11.1: Using the style of Figure 11.4, give a sequence diagram showing a complete trace for the subsequent call `sample.setColors('blue', 'white')` upon that bullseye.

Exercise 11.2: Our original Bullseye class alternated between two colors. Write a new version supporting the constructor signature `Bullseye(numBands, radius, colorlist)`, such that `colorlist` is a list of one or more colors. The bullseye should be drawn so that `colorlist[0]` is the primary color, `colorlist[1]` the secondary color, and so on, cycling when reaching the end of the list. Revise the `setColors` method accordingly.

Exercise 11.3: As a `Drawable` object, our bullseyes can be rotated. Of course, since the reference point is implicitly the center of many concentric circles, rotation is not very interesting. However, it can be used to produce an interesting optical illusion if we slightly alter our bullseye definition so that the circles are not perfectly concentric. We can bring the two-dimensional image to life, making it appear as a three-dimensional cone when rotated.

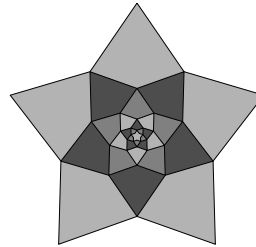
Modify the Bullseye class so that the rest of a bullseye is moved slightly to the right of center. Specifically, the default width of each band is `radius/numBands`; move the inner bullseye to the right by an amount equal to the band width times a coefficient. With a small coefficient, say 0.1, the perspective will appear to be above the tip of the cone; with larger coefficients, the view is from an angle (until eventually the illusion breaks down).

Exercise 11.4: At the beginning of the chapter, we discussed a pyramid as another natural example of a recursive structure. Implement a `Pyramid` class recursively, using an approach similar to that used for the Bullseye class. Allow the user to specify the number of levels and the overall width (which can be the same as the height).

The main difference in techniques involves the positioning of the components. For the bullseye, the outer circle and inner bullseye were concentric. For the pyra-

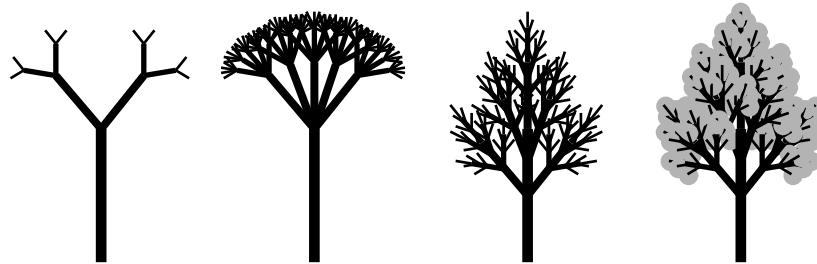
mid, you will need to relocate the bottom rectangle and the upper pyramid to achieve the desired effect. Move the components so that the completed pyramid sits with its bottom edge centered on the origin (the default reference point).

Exercise 11.5: The following flower is based upon a recursive nesting of stars:



It is composed of an outer Star, as defined in Section 9.4.1, together with a inner flower. The outer radius of the inner flower is equal to the inner radius of the outer star and the inner flower is rotated so that its outer points coincide with the inner points of the star (try saying that ten times fast).

Exercise 11.6: Implement a Tree class for creating somewhat realistic looking trees with the use of recursion. Consider the following examples:



The general approach is to define a tree of order k that has a trunk and then two or more branches, which are actually trees of order $k - 1$. By defining a tree with the reference point at the bottom of the trunk, it is easy to rotate a subtree and attach it where desired. Variations can be achieved (as shown above), by altering the factor at which the trunk height and width decrease at each level, by altering the number of branches attached to the trunk or the angle between those branches. In our left two figures, all branches are attached to the top of the trunk. In the right two, the branch points are distributed vertically. The third and fourth figures are identical, except that leaves have been added as a base case, using a single small green circle at the end of each branch.

OurList

Practice 11.7: Several different recursive patterns are used by the methods of the OurList class. When one call results in a subsequent recursive call, we can examine whether the base case involves an empty list, the head of the list, or an index of zero. The parameterization may be the same at each level or vary between levels (or there may be no parameters whatsoever). Similarly, the return value might always be the

same or might vary from level to level. For each method of Figure 11.11, fill in the following chart (we’ve gotten you started by doing `__len__`).

| method | base case | | | parameters | | | return value | | |
|---------------------------|-----------|------|-------|------------|------|------|--------------|------|------|
| | empty | head | index | same | vary | none | same | vary | none |
| <code>__len__</code> | ✓ | | | | | ✓ | | ✓ | |
| <code>__contains__</code> | | | | | | | | | |
| <code>__getitem__</code> | | | | | | | | | |
| <code>__setitem__</code> | | | | | | | | | |
| <code>__repr__</code> | | | | | | | | | |
| <code>count</code> | | | | | | | | | |
| <code>index</code> | | | | | | | | | |
| <code>append</code> | | | | | | | | | |
| <code>insert</code> | | | | | | | | | |
| <code>remove</code> | | | | | | | | | |

Practice 11.8: Give a recursive implementation of a method `OurList.min` that returns the smallest value on the list.

Exercise 11.9: The standard notion for comparing two Python lists is based on what is termed *lexicographical order*. The list with the smaller first element is considered the “smaller” list. However, in case the first elements are equivalent, the second elements are used as a tie-breaker, and so on. If all elements are pairwise equivalent yet one list has additional elements at the end, that list is considered “larger.” If all elements are pairwise equivalent and they have the same length, the two lists are considered equivalent. Give an implementation of the method `__le__(self, other)` that returns **True** precisely when the original list is less than or equal to the other list, by lexicographical convention.

Exercise 11.10: In the chapter, we gave an implementation of the `OurList.index` method that took a single parameter `value`. That method returned the leftmost index at which the value occurs in the list. Python’s lists support a more general signature, of the form

```
def index(self, value, start=0):
```

that returns the smallest index, at least as great as `start`, at which the value occurs. Use recursion to implement the more general form of `OurList.index`.

Exercise 11.11: Use recursion to implement the method `OurList.pop(index)`. For simplicity, you do not need to support negative indices nor provide a default parameter.

Exercise 11.12: Assume that `sample` is an instance of `OurList` representing the corresponding list `['H', 'E', 'R', 'E']`. Give an explicit trace showing the execution of `sample.insert(2, 'N')`.

Exercise 11.13: Assume that `sample` is an instance of `OurList` representing the corresponding list `['H', 'E', 'R', 'E']`. Give an explicit trace showing the execution of `sample.remove('E')`.

Exercise 11.14: In the style of Figure 11.10, give a trace of the execution of `data[6]` on the list `['H', 'E', 'R', 'E']`.

Exercise 11.15: Our implementation of `__getitem__` does not support Python’s standard notion of negative indices. Rewrite it so that it does. Hint: would it help if you knew the overall length?

Exercise 11.16: In our implementation, a call to `demo.index('T')` on a list with contents `['H', 'E', 'R', 'E']` raises a `ValueError`. This is a reasonable response since the value was not found, yet our implementation exposes the underlying recursion to the original caller, with the error appearing as

```
>>> demo.index('T')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "OurList.py", line 61, in index
    return 1 + self._rest.index(value)
  File "OurList.py", line 61, in index
    return 1 + self._rest.index(value)
  File "OurList.py", line 61, in index
    return 1 + self._rest.index(value)
  File "OurList.py", line 61, in index
    return 1 + self._rest.index(value)
  File "OurList.py", line 61, in index
    return 1 + self._rest.index(value)
  File "OurList.py", line 57, in index
    raise ValueError('OurList.index(x): x not in list')
ValueError: OurList.index(x): x not in list
```

The problem is that the actual error is raised from deep within the recursion. From the original caller’s perspective, it would be better to see that error raised from the context of the top-level call. Use a **try-except** clause to accomplish this.

Exercise 11.17: Provide an implementation of `OurList.reverse`. Think carefully about your design. Hint: it’s okay to rely upon calls to additional behaviors.

Exercise 11.18: Develop a strategy for implementing `OurList.sort`. One approach, known as selection sort, involves finding the overall minimum item, rearranging so that it is moved to the beginning of the list, and then sorting the rest accordingly.

Functional Recursion

Practice 11.19: Predict the output that results when `TestA(1)` is invoked on the following:

```
def TestA(count):
    if count != 4:
        print count
        TestA(count+1)
```

Practice 11.20: Predict the output that results when `TestB(1)` is invoked on the following:

```
def TestB(count):
    if count != 4:
        TestB(count+1)
    print count
```

Practice 11.21: Consider the following small program:

```
def fib(n):
    print 'n = ', n
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)

print fib(4)
```

Predict the precise output of the program (not just the return value, all output).

Exercise 11.22: Consider the following program, where `data` is a Python **list**:

```
def getmax(data, start, stop):
    print 'What is max(data[%d:%d])?' % (start, stop)
    if stop == start + 1:
        answer = data[start]
    else:
        mid = (start + stop - 1) // 2
        sub1 = getmax(data, start, mid+1)
        sub2 = getmax(data, mid+1, stop)
        if sub1 > sub2:
            answer = sub1
        else:
            answer = sub2
    print answer, ' is max(data[%d:%d]).' % (start, stop)
    return answer
```

Predict the complete output that is printed when `getmax([12, 35, 48, 19], 0, 4)` is invoked.

Exercise 11.23: Give a natural recursive implementation of a `gcd(u,v)` function, based upon the intuition from *For the Guru* on page 10. Be careful to consider the base case.

Exercise 11.24: Write a function `binary(n)` that takes a nonnegative integer `n`, and returns a string of '0' and '1' characters that is the binary representation of the integer. Notice that the *rightmost* bit of the result is equal to `str(n % 2)`, as even numbers end with 0 and odd numbers end with 1. The remaining prefix is exactly the binary representation of `n // 2` (assuming that number is nonzero).

Exercise 11.25: A problem that is easily solved by recursion, but very difficult to solve by other means, is the classic Towers of Hanoi. The puzzle begins with three pegs and a tower of n disks on the first peg, stacked from largest at bottom to smallest at top. The goal is to move them all to the third peg, moving only one disk at a time. Moreover, you are not allowed to place a disk on top of a smaller disk.

Instructions for a correct solution can easily be generated by recognizing the following pattern. First, the top $n - 1$ disks can be moved from the original peg to the intermediate one (not as one step, but recursively). Then the bottom disk can be moved from the original to the end. Finally, the other $n - 1$ disks can be moved from the intermediate peg to the end. Write a program that generates a solution to the problem.

Exercise 11.26: Animate Exercise 11.25 with `cs1graphics`.

Binary Search

Practice 11.27: Using the style of Figure 11.14, diagram a full trace of the execution of call `search(['A', 'L', 'M', 'O', 'S', 'T'], 'S')` using the poor search implementation of Section 11.4.2.

Practice 11.28: Using the style of Figure 11.16, diagram a full trace of the execution of call `search(['A', 'L', 'M', 'O', 'S', 'T'], 'S')` using the good search implementation of Section 11.4.3.

Exercise 11.29: Had line 20 of Figure 11.15 read

```
return search(lexicon, target, midIndex, stop)
```

the code has the potential of entering an infinite recursion. Give a trace of a small example that demonstrates the flaw.

Exercise 11.30: The good implementation of binary search from Figure 11.15 produces the correct answer, even when executed on an instance of the `OurList` class (rather than the built-in `list` class). Unfortunately, the process is horribly inefficient. Explain why, noting which lines of the binary search code are bottlenecks.

Exercise 11.31: Use binary search to implement a more efficient version of the function `SortedSet.insertAfter`, as originally described in Section 9.2.

Exercise 11.32: Our version of `prefixSearch` simply returns **True** or **False**. Provide a new implementation that returns a list of all words from the lexicon with a prefix that matches the given target.

Anagrams

Practice 11.33: Using the style of Figure 11.19, trace the call `anagrams(lexicon, 'tea')`.

Exercise 11.34: The list of solutions reported by our anagram solver are not necessarily alphabetized. Prove that if `charsToUse` is provided in alphabetical order, then the list of solutions will automatically be alphabetized.

Exercise 11.35: If the original word contains multiple occurrences of the same character, our anagram solver may report a solution multiple times in the results. Modify the code to avoid such duplicates. Hint: when looping over `charsToUse`, only start the recursion once for each distinct character.

Exercise 11.36: Since any given execution of our anagram program only depends upon words with the same length as the original, it would be more efficient to keep a separate lexicon for each word length. Modify the program to read the lexicon from a file, creating a list of sorted lists (one list for each length). Implement the `anagrams` function to take advantage of this new structure and see if you can measure any noticeable improvement in efficiency.

Exercise 11.37: A more interesting version of anagrams involve phrases with multiple words (e.g., `'use python'` is an anagram for `'pushy note'`). Write a function that locates such multiword anagrams. Hint: we suggest the parameterization `anagrams(lexicon, charsToUse, precedingWords=' ', partialWord='')`.

Projects

Exercise 11.38: When computing anagrams, all of the original letters must be used as part of the solution. In the game of Scrabble™, a player has a collection of letters and must form a word by rearranging some (but not necessarily) all of those letters. Modify the anagram program to compute all possible words that can be formed from a given collection of letters.

Exercise 11.39: In the game of Boggle™, there is a four-by-four grid of letters and a player forms a word by spelling the word while tracing a path on that grid. The path can move horizontally, vertically, or diagonally, but the same spot cannot be used more than once for a given word. Write a program that computes all words from a lexicon that can be obtained from a given board configuration. One approach is to recursively trace paths through the board, pruning the search whenever a prefix is reached that does not occur in the lexicon.