Chapter 12

Recursion

12.1 Background

Recursion is a powerful programming construct that provides an elegant way to solve large problems by breaking them down into simpler subproblems. In this way it is often possible to solve seemingly complex problems via repeated application of very simple solutions to the subproblems.

In programming languages, when we talk about recursion we talk in terms of recursive functions.[1] A recursive function is a function that calls itself somewhere in its definition, i.e., in some sense, the function is defined in terms of itself. At first this may seem confusing. After all, if we use a word to define itself, for example, if we say, "a tautology is a tautology," then we haven't helped to clarify what a tautology is.[2] So, clearly there must be something more to a useful recursive function than the mere fact that it uses itself within its own definition.

12.2 Flawed Recursion

We will consider the proper implementation of a recursive function in a moment, but let's start by considering some flawed implementations because it is important to recognize how you must *not* implement a recursive function. You could easily write a function that uses itself in its own definition and hence is arguably a recursive function. Consider the function `r1()` defined in Listing 12.1 which is the simplest possible (albeit flawed) implementation of a recursive function. This function has no arguments and the entire body of the function is merely a call to the function itself. Thus, when we call `r1()`, the body of the function says to call the function `r1()`. So, `r1()` will be called again only to find that `r1()` should be called again, and so on. In theory, these successive calls would go on forever, never doing anything useful. However, in practice, these successive calls can't go on forever; something must ultimately break if `r1()` is called (for example, it requires some memory to keep track of function calls so perhaps your computer ultimately runs out of memory).

From the file: `recursion.tex`

[1] Not all computer languages support recursive functions, but nearly all modern languages do.

[2] A tautology is, according to Webster's Collegiate Dictionary, 5th ed., "needless repetition of meaning in other words; also, an instance of this as 'audible to the ear'."

Listing 12.1 Implementation of the simplest (albeit flawed) recursive function. The body of the function is merely a call to the function itself.

```
>>> def r1():
...     r1()
```

Listing 12.2 demonstrates what happens when we call `r1()`. In line 1 `r1()` is called. This produces the `Traceback` message which contains hundreds of repeated lines (most of which have been removed from the listing). This is followed by a statement telling us there was a `RuntimeError` because the "maximum recursion depth [was] exceeded." Essentially Python kept track of how many times `r1()` called `r1()`. Once the number of calls exceeded a certain value, Python stepped in, thinking that something must be wrong, and halted the entire process.

Listing 12.2 Calling the recursive function `r1()` produces an error because the maximum recursion depth is exceeded.

```
>>> r1()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in r1
  File "<stdin>", line 2, in r1
                  .
                  .   <<OUTPUT DELETED>>
                  .
RuntimeError: maximum recursion depth exceeded
```

Let's implement another (flawed) recursive function but add a bit of code to the function's body to help shed light on what is happening. Listing 12.3 starts by defining the function `r2()` that takes a single argument (which is assumed to be an integer). In line 2, the first statement in the body of the function prints the value of the argument together with the word `Start`. Then, in the second statement of the body (line 3), `r2()` is called *but* the argument passed to `r2()` is incremented by one from the value that was originally passed to the function. The last statement in the body of the function (line 4) again prints the argument but now it is paired with the word `End`. In line 6 `r2()` is called with an argument of `0`. The subsequent output is discussed following the listing.

Listing 12.3 Another flawed recursion function. This function accepts an argument and increments the value of the argument with each successive recursive call.

```
>>> def r2(n):
...     print(n, "Start")
...     r2(n + 1)
...     print(n, "End")
...
```

```
>>> r2(0)
0 Start
1 Start
2 Start
 .
 .   <<OUTPUT DELETED>>
 .
994 Start
995 Start
996 Start
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in r2
  File "<stdin>", line 3, in r2
                  .
                  .   <<OUTPUT DELETED>>
                  .
  File "<stdin>", line 2, in r2
RuntimeError: maximum recursion depth exceeded while calling a Python
object
```

In line 7 of Listing 12.3, we see that when `r2()` is called with an argument of `0`, the first output that is produced is `0 Start`, i.e., the first statement in the body of the function (line 2) merely prints whatever argument was passed to the function. Then, in line 3, `r2()` is called again (i.e., `r2()` is called from within `r2()`, before the flow of execution returned from the first call to `r2()`). But, the argument is incremented by one. Thus, for this second call to `r2()`, the argument is `1` which results in the output shown on line 8 (`1 Start`. After printing this argument, `r2()` is called again, but the argument is again incremented by one. This leads to the output of `2 Start` shown on line 9. This cycle repeats until, ultimately, Python generates the `RuntimeError` shown in lines 16 to 22 (portions of the output have been deleted).[3] Importantly, notice that we never got to the second `print()` statement in the body of the function, i.e., the statement in line 4 that prints the argument followed by the word `End`. That is because we never *returned* from any of the calls to `r2()` before Python halted the process.

12.3 Proper Recursion

Now we can ask: What causes the errors shown in Listings 12.1 and 12.3? Thinking about how `r1()` and `r2()` are defined, we see that they "never" stop calling themselves, i.e., they keep calling themselves until the recursion limit is reach. When this limit is reached one obtains the `RuntimeError` error messages shown in the listings. To prevent this error and to make a useful recursive function, *a recursive function must possess two vital features*:

[3] On the system on which this particular example was run, the maximum recursion depth was set to 1000. This is true despite the fact that there are 997 integer values displayed in the output. You can obtain the recursion limit with the following two statements: `import sys; print(sys.getrecursionlimit())`. The function `getrecursionlimit()` in the `sys` module is used to set the recursion limit to a different value.

1. There must be a reachable *base case* where the function stops calling itself.

2. The argument of the function must be modified with each call.

Note that `r1()` in Listing 12.1 possessed neither of these features. On the other hand, `r2()` in Listing 12.3 possessed the second feature (the argument was modified with each call), but it did not possess a base case.

The base case specifies when to stop making recursive calls. In order to do this, we need to use what we've learned about conditionals to write the base case and the general case. As will become more clear in a little while, you can think of a base case as the simplest subproblem you have, for which the answer is easily determined while the general case is where we modify the argument to ensure that the base case is reachable. But, before doing anything particularly useful in terms of solving a problem, let's create a modified version of `r2()` so that it has both features necessary to realize a useful recursive function.

Listing 12.4 defines the function `r3()` which is similar to `r2()` except that, as shown in lines 3 and 4, it will only make a recursive call to `r3()` if `n` is less than 4. For this function the "base case" is when the function is passed an argument of 4 (or greater). When this occurs, instead of calling `r3()` again, the function merely prints the start and end messages and then returns. The "general case" results in the printing of the start message, the recursive call to `r3()`, and then the printing of the end message. The discussion continues following the listing.

Listing 12.4 Recursive function that has a (reachable) base case and modifies its argument with each successive call.

```
1  >>> def r3(n):
2  ...     if n < 4:     # General case.
3  ...         print(n, "Start")
4  ...         r3(n + 1)
5  ...         print(n, "End")
6  ...     else:         # Base case.
7  ...         print(n, "Start")
8  ...         print(n, "End")
9  ...
10 >>> r3(0)
11 0 Start
12 1 Start
13 2 Start
14 3 Start
15 4 Start
16 4 End
17 3 End
18 2 End
19 1 End
20 0 End
```

In line 10 `r3()` is called with an argument of 0. Because this argument is less than 4, this leads to the "general-case behavior" of the function. Thus, in accordance with the statement in line

3, the start message is printed (as shown in line 11). Then, in line 4, a recursive call is made to `r3()` but the argument is incremented so that it is now 1. Because an argument of 1 is less than 4, this call to `r3()` again results in the general-case behavior. This ultimately produces all the start messages shown in lines 11 through 14.

When `r3()` is called with an argument of 4, we have reached the base case (which corresponding to the statements in lines 7 and 8). The start message is again printed, as shown in line 15, i.e., `4 Start`. But then, rather than calling `r3()` again, the function prints the end message (shown in line 16) and then returns. The flow of execution returns to the point where this function was called. This happens to be from `r3()` but where the arguments was 3. So, now that function prints the end message (i.e., `3 End`) and returns. Execution now returns to prior invocation of `r3()` where the argument was 2, and so on, until we ultimately get back to where `r3()` was called with an argument of 0 (and thus back to the interactive prompt).

Again, `r3()` is a proper recursive function in that it has a reachable base case (where the function stops making calls to itself) and the argument is modified with each call. You should spend a bit of time looking at the output in Listing 12.4. Note that both the *first* line of output and the *last* line of output are associated with the call to `r3()` with an argument of zero. This call to `r3()` executes a `print()` statement, then `r3()` is called so that nothing else is executed from this original call to `r3()` until all the other calls to `r3()` are complete. Once they are complete and control is returned to the first call to `r3()`, the final `print()` statement is executed.

The implementation of the `r3()` function in Listing 12.4 separates the general case and the base case into two distinct blocks of code. However, because there is an overlap in what the general case and the base case actually do, there is an needless duplication of code. A cleaner implementation is shown in Listing 12.5.

Listing 12.5 A alternate implementation of the `r3()` function of 12.4 that removes the unnecessary duplication of code.

```
1 def r3(n):
2     print(n, "Start")
3     if n < 4:
4         r3(n + 1)
5     print(n, "End")
```

In mathematics you will often see the Fibonacci sequence defined recursively, so let's consider an implementation of that. Assume $F(n)$ yields the nth Fibonacci number. $F(n)$ can be defined as follows[4][5]

$$\begin{aligned} F(1) &= 1 \\ F(2) &= 1 \\ F(n) &= F(n-1) + F(n-2) \end{aligned}$$

[4]The implementation of the Fibonacci sequence given in Sec. 6.9.3 is actually much more efficient than the recursive implementation we will present here. We are defining it recursively here for purposes of illustration.

[5]Sometimes the sequence is defined with the first two numbers being 0 and 1 instead of 1 and 1.

The Fibonacci function $F(n)$ is defined in terms of itself. The third line above is the general case and says that any number in the sequence is the sum of the previous two numbers in the sequence. There are actually two possible base cases. Both $F(1)$ and $F(2)$ simply return the number 1. Using this definition let's write, as shown in Listing 12.6, a function that uses recursion to generate numbers in the Fibonacci sequence.

Listing 12.6 Recursive implementation of a function to generate numbers in the Fibonacci sequence.

```
1 >>> def fib(n):
2 ...     if n == 1 or n == 2:            # Base case.
3 ...         return 1
4 ...     return fib(n - 1) + fib(n - 2) # General case.
```

Using the mathematical definition above, we can easily determine the base case: if n is 1 or 2, then the function merely returns 1. For the general case we want to return the sum of the previous two numbers in the sequence, i.e., the sum of the $(n-1)$th number and the $(n-2)$th number. Since `fib(n)` generates the nth number in the sequence, we can call `fib()` with arguments of `n - 1` and `n - 2` and return the sum of the results.

These separate calls to `fib()` will in turn make more calls to `fib()` until the base case is reached. Once this happens, the results will build from the base case to find the $(n-1)$th and $(n-2)$th Fibonacci number, and then, from these, the nth number. Note that the body of the function definition in Listing 12.6 looks almost identical to the mathematical definition! Note that our implementation does meet the requirements for a useful recursive function: there is a reachable base case and the argument is modified for each recursive call.[6] Listing 12.7 demonstrates the behavior of this recursive implementation of a Fibonacci number generator.

Listing 12.7 Demonstration of the behavior of the `fib()` function as implemented in Listing 12.6.

```
1 >>> fib(1)
2 1
3 >>> fib(2)
4 1
5 >>> fib(3)
6 2
7 >>> fib(10)
8 55
```

In lines 1 and 3 of Listing 12.7, `fib()` is called with arguments that result in base-case behavior where the function immediately returns 1. When `fib()` is called with an argument of 3,

[6] However, it may also be worth pointing out that the base case is only reachable if the function is called with a positive argument. You may wish to confirm for yourself that if the function is called with a non-positive argument, the subsequent recursive calls have arguments that progressively become more negative until eventually the recursion limit is reached.

as is done in line 5, the return value is the sum of `fib(1)` and `fib(2)`, i.e., the two base-case values, which is simply 2. In line 7 `fib()` is called with an argument of 10. Behind the scenes this actually triggers a rather large cascade of recursive calls. In fact, the argument does not have to be large before the computer takes a noticeably long time to return a value. Thus, as mentioned in Footnote 4 on page 301, a purely recursive implementation of a Fibonacci number generator is not efficient. It is much more efficient to start from the base values and then, in an iterative loop (such as a `for`-loop), directly build up to the number n rather than starting from n and recursively working down to the base values.

Now let's consider a recursive implementation of the factorial function (in this case the recursive implementation is nearly as efficient as an iterative [non-recursive] implementation). Recall that $n!$, read as n factorial, is defined as $n \times (n-1) \times (n-2) \cdots \times 1$. Listing 12.8 shows both a recursive and a non-recursive implementation of the factorial function.

Listing 12.8 Recursive and non-recursive implementation of the factorial function.

```
1  >>> def fact_r(n):  # Recursive implementation.
2  ...     if n == 1:                # Base case.
3  ...         return 1
4  ...     return n * fact_r(n - 1) # General case.
5  ...
6  >>> fact_r(1)
7  1
8  >>> fact_r(5)
9  120
10 >>> fact_r(40)
11 815915283247897734345611269596115894272000000000
12 >>> def fact_nr(n): # Non-recursive implementation.
13 ...     fact = 1
14 ...     for i in range(1, n + 1):
15 ...         fact = fact * i
16 ...     return fact
17 ...
18 >>> fact_nr(40)
19 815915283247897734345611269596115894272000000000
```

The recursive function is defined in lines 1 through 4. The base case for this function is when the argument is 1, in which case the function returns 1. This is handled by the first two lines in the body of the function, i.e., lines 2 and 3. Thus, if 1 is supplied as an argument then we can simply return 1 as the answer. The fact that the function does this is demonstrated in lines 6 and 7. Recalling the definition of $n!$ we recognize that $n!$ is equal to $n \times (n-1)!$. We can use this fact to define our general case by multiplying the argument n by a recursive call to `fact_r` with an argument of $(n-1)$ as is done in the last statement in the body of the function. Lines 8 through 11 demonstrate that the function works properly for arguments that do not immediately trigger the base case.

A non-recursive implementation of the factorial function is given in lines 12 through 16 of Listing 12.8. In the body of function the identifier `fact` is assigned the initial value of 1. Then, in

lines 14 and 15, `fact` is used as an accumulator in a `for`-loop to accumulate all the multiplication needed to build up from 1 to n. The last two lines of the listing demonstrate the function works properly. The non-recursive implementation is actually slightly more efficient than the recursive implementation because calls to a function are typically more computationally expensive than iterating a `for`-loop.

Recursion is applicable to more than just mathematical problems. For example, consider the function in Listing 12.9 that returns the reverse of the string it was passed.

Listing 12.9 Recursive function to reverse a string.

```
>>> def reverse(s):
...     if s == "":
...         return ""
...     return s[-1] + reverse(s[ : -1])
```

Inspect this code and then think about how you might describe, in words, the solution. How can we use recursion to reverse the characters of a string and what is the base case? We know that in order to reverse a string recursively we're going to have to break it down into subproblems of the same form, but when should we stop trying to reverse the string? The answer is: when there isn't anything left to reverse, i.e., when the string we want to reverse is empty. If the string is empty then we can simply return an empty string because there is nothing to reverse.[7]

The more challenging part of the string-reversal problem is coming up with the general case. Think of this in terms of adding something to the answer of a smaller subproblem that has already been solved. We know that the first character in a reversed string is the last character of the given string (e.g., the first character in the reverse of `cat` is `t` because `t` is the last character of the given string [and will ultimately be the first character of `tac`]). The first character of the reversed string can be easily obtained using negative indexing on the given string, i.e., using `s[-1]`. In order to get a complete reversed string, we concatenate the last character of the string with the reverse of the rest of the string, i.e., the reverse of the string after removing the last character. This suggests a recursive solution. We are defining a function that gives us the reverse of the string so we can call this function to find the reverse of the rest of the string! We just have to ensure that the argument we pass to the function is the string with the final character removed, i.e., `s[ : -1]`. So, the recursive implementation could be described as, "Write the last character and then write the remaining characters in reverse order." Listing 12.10 demonstrates that the function works properly.

Listing 12.10 Demonstration that the recursive function `reverse()` defined in Listing 12.9 works properly.

```
>>> reverse("recursion")
'noisrucer'
>>> reverse("Hello there.")
.ereht olleH
```

[7] An alternate base case would be to check for a string of length 1. In that case the function should return that single-character string since one cannot reverse a single character.

This same recursive approach can be used to reverse a `list`, except for a list the base case would be when the `list` had a length of zero and the return value would be an empty `list`.

There are, in fact, many ways to reverse a string (or a `list`). The easiest and clearest implementation is simply to use a slice with a increment of `-1`, i.e., the reverse of the string `s` is the string `s[ : : -1]`. A solution that requires slightly more code, but is still simpler than the recursive approach, is to use a `for`-loop where the index would pick out the characters of the string in reverse order. However, for the sake of illustration, we will consider one more recursive function that reverses the characters of a string.

Listing 12.11 defines two functions. The first, `reverse_main()`, in lines 1 and 2, takes a single string argument and is not a recursive function. It merely returns whatever the function `reverse_help()` returns when that function is called with two arguments: the same string as was passed to `reverse_main()` and the integer 0. The discussion continues following the listing.

Listing 12.11 Another recursive solution to the string reversal problem. In this case the solution is broken into two functions: a "main" function that takes a single string argument and a "help" function that takes two arguments corresponding to the string and what is effectively an integer index.

```
1  >>> def reverse_main(s):
2  ...     return reverse_help(s, 0)
3  ...
4  >>> def reverse_help(s, n):
5  ...     if n == len(s):
6  ...         return ""
7  ...     return reverse_help(s, n + 1) + s[n]
8  ...
9  >>> reverse_main("pleh")
10 'help'
11 >>> reverse_main("Hello there.")
12 .ereht olleH
```

The function `reverse_help()`, given in lines 3 through 6, is a recursive function. It returns the reserve of its first (string) argument starting from the (integer) index specified by the second argument. Thus, when `reverse_help()`'s second argument is 0, the entire string is reversed. If, as in the code, the second argument is called `n`, the general case is to return the reverse of the string starting from the `(n + 1)`th character concatenated with the `n`th character. The base case is, as shown in lines 5 and 6, to return the empty string if the index is equal to the length of the string (i.e., return the empty string if there are no more characters that must be rearranged). In some sense this code is similar to the end message in `r3()` in Listing 12.4 where the printed arguments counted down from the maximum value to zero. Lines 9 through 12 demonstrate the function works properly.

12.4 Merge Sort

All the previous examples were intended to illustrate the implementation and behavior of recursive functions. They were not, however, very practical and there are much simpler ways to obtain identical results without using recursion. Now we want to consider a much more practical, "real world" application of recursion. We want to demonstrate how recursion can be used to efficiently sort the elements of a `list`. Before digging into the details, there are two important things to note. First, the subject of sorting is actually quite a complicated one! Though quite interesting at times, we will not delve into any of the complexities of the subject. Suffice it to say that, although the algorithm we will consider here is quite good, it is not optimum. Second, don't forget that `lists` have a `sort()` method and, in fact, there is a built-in function called `sorted()` that can be used to sort iterables. These can be used to sort iterables more efficiently than the approach described here.

The algorithm we will implement is called merge-sort. A key component of the algorithm is the ability to merge two `lists` that are assumed to be in sorted order. The resulting `list` is also in sorted order. Given the ability to merge `lists` in this way, we can start with an unsorted `list` and (recursively) break it down into a collection of single-element `lists`. A single-element `list` is inherently sorted. We can merge the single-element `lists` into sorted two-element `lists`. We can then merge the resulting two-element `lists` into four-element `lists`, and so on, until we have obtained all the elements in sorted order. (Despite the description here, the number of elements does not have to be a power of two as will be made more clear below.)

Let's start by considering the `merge()` function shown in Listing 12.12. This is a non-recursive function that returns the a single `list` that contains all the elements of the two `lists` it is given as arguments. It is assumed the two `lists` this function is passed are already sorted. The `list` the function returns is also sorted. So, for example, if the function is passed the `lists [1, 4]` and `[2, 3]`, the resulting `list` is `[1, 2, 3, 4]`. The discussion continues following the listing.

Listing 12.12 The `merge()` function that merges two sorted `lists` and returns a single sorted `list`.

```
>>> def merge(left, right):
...     result = []      # Accumulator for merged list.
...     while len(left) > 0 or len(right) > 0:
...         if len(left) > 0 and len(right) > 0:
...             if left[0] <= right[0]:     # left smaller than right.
...                 result.append(left[0])  # Append element from left.
...                 left = left[1 : ]       # Remove first left element
...             else:                       # right smaller than left.
...                 result.append(right[0]) # Append element from right.
...                 right = right[1 : ]     # Remove first right element.
...         elif len(left) == 0:      # No elements in left.
...             result.extend(right)  # Extend by remaining right elements.
...             break                 # Terminate loop.
...         else:                     # No elements in right.
...             result.extend(left)   # Extend by remaining left elements.
```

```
16 ...             break                    # Terminate loop.
17 ...         print("result: ", result) # For sake of illustration.
18 ...     return result
```

The two `lists` passed to the function are called `left` and `right`. Values are taken from these `lists` and appended to the `list` `result` which is initialized to the empty `list` in line 2. As an element is taken from `left` or `right`, the `list` from which the element was taken is reduced in size to now exclude this element. This continues until all the elements have been move from `left` and `right` to `result`. To accomplish this, a `while` loop, whose header is in line 3, continues to iterate while there is at least one element in either `left` or `right`. The first statement in the body of the loop, line 4, checks if both `left` *and* `right` have at least one element. If this is true, then the first elements of the two `lists` are compared (the first elements are the smallest of both `lists`). The smaller of the two elements is appended to `result` and the `list` from which the element came is reset to exclude the element.

If, however, `left` and `right` don't both have at least one element, then, in line 11, we check to see if length of `left` is zero. If that's the case, then we *extend* `result` by whatever elements remain in `right`. Finally, the `else` clause in line 14 ensures that if there are no more elements in `right`, then `result` will be extended by any elements that remain in `left`. The `print()` statement in line 17 is merely for the sake of illustration: it displays the contents of the `result` list for each iteration of the loop.

The `merge()` function is demonstrated in Listing 12.13. In line 1 the function is called with two `lists`, each of two elements. The resulting `list` shown in line 5 is the correctly merged sorted `list`. In line 6 `merge()` is called with a `list` of three elements and another of five elements. The result show in line 14 is again the correctly sorted `list`.

Listing 12.13 Demonstration of the `merge()` function.

```
1  >>> merge([1, 4], [2, 3])
2  result:  [1]
3  result:  [1, 2]
4  result:  [1, 2, 3]
5  [1, 2, 3, 4]
6  >>> merge([1, 5, 10], [0, 4, 6, 7, 8])
7  result:  [0]
8  result:  [0, 1]
9  result:  [0, 1, 4]
10 result:  [0, 1, 4, 5]
11 result:  [0, 1, 4, 5, 6]
12 result:  [0, 1, 4, 5, 6, 7]
13 result:  [0, 1, 4, 5, 6, 7, 8]
14 [0, 1, 4, 5, 6, 7, 8, 10]
```

Now that we know how to merge two sorted `lists`, let's write a function called `merge_sort()` that takes a single unsorted `list` as its argument. This function uses recursion as well as the `merge()` function to return a sorted version of the `list` it was passed. Before considering the

code to implement this, let's consider how it works. Any `list` that has more than one element can be divided into two "sublists" of (nearly) equal length. If the original `list` has an even number of elements, then the two sublists will have an equal number of elements. If the original `list` had an odd number of elements, then one of the sublists will have one element more than the other sublist. Continuing this process, we can divide the original `list` into a collection of single-element `lists`. Note that even if the original `list` is relatively large, this can be done very quickly! If there are N elements in the original `list`, then it takes at most $\log_2(N)$ steps to divide this into a collection of N single-element `lists`. So, for example, if N is 1,000,000,000, then it takes only about 30 sets of subdivisions to divide this into individual elements!

As mentioned above, a single element `list` is inherently sorted. We can merge the single-element `lists`, using the `merge()` function of Listing 12.12 to form two-element `lists`. From these we can form four-element `lists`, and so on. (However, when reassembling the sublists, we may obtain `lists` with an odd number of elements if, in the process of creating the sublists, we encountered an odd number of elements.) Eventually we get to the point where all the sublists have been merged to obtain a new `list` that contains all of the elements of the original `list`, but now in sorted order. This process is illustrated in Fig. 12.1 where the merge-sort algorithm is used to sort a `list` of ten integers. The original order of these integers is shown in "Step 1."

In Fig. 12.1, Steps 2 through 5 involve dividing the original `list` into progressively smaller sublists. Note that Step 3 yields `lists` of unequal length owing to the fact that `lists` of five elements had to be divided in two. At Step 5 all that remains are single-element `lists`.

At Step 6 (i.e., the step following the double lines), we begin to merge the single-element `lists` into two-element `lists`. The last elements to be subdivided are the first to be merged (these are the pairs of numbers (5, 6) and (0, 2)). The merged `lists` are always in sorted order. In Step 9 we obtain the complete `list` of sorted values.

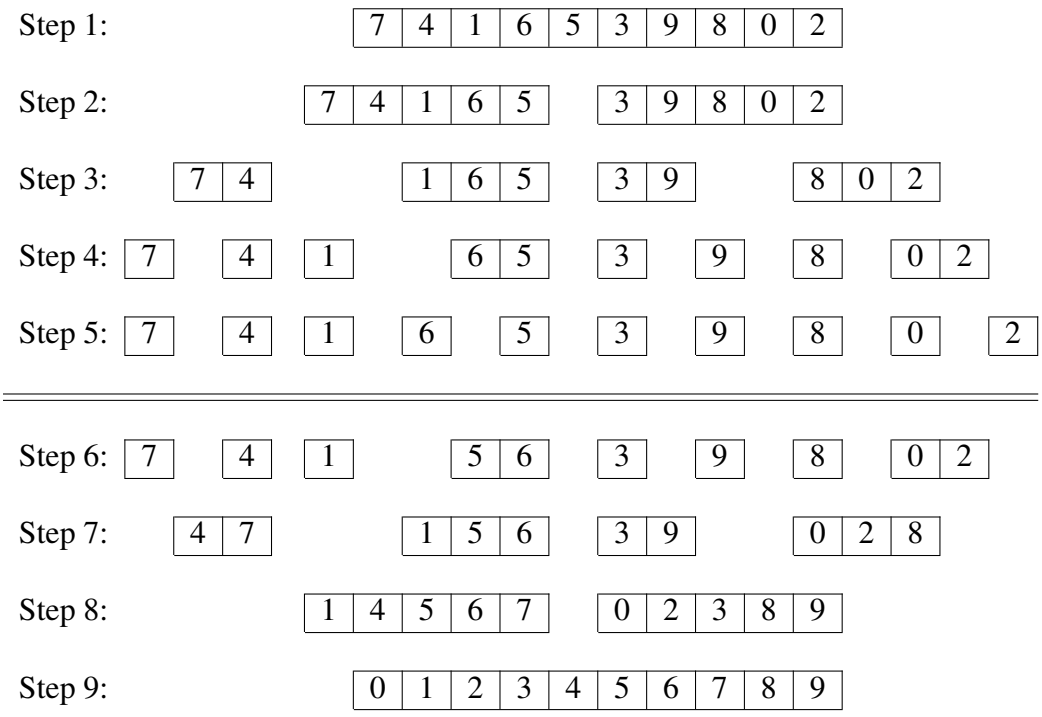
With that understanding, a suitable function to implement the merge-sort alogorithm is shown in Listing 12.14. Th `merge_sort()` functions starts, in lines 2 and 3, by checking if the `list` the function was passed has a single element (or no elements). If so, there is nothing to sort and the function merely returns this `list`. This is the base case. The discussion continues following the listing.

Listing 12.14 The `merge_sort()` function takes an unsorted `list` as its argument and returns a `list` with the elements sorted.

```
1 >>> def merge_sort(xlist):
2 ...     if len(xlist) <= 1:
3 ...         return xlist
4 ...     left = xlist[ : len(xlist) // 2]  # First "half" of list.
5 ...     right = xlist[len(xlist) // 2 : ] # Second "half" of list.
6 ...     sorted_left = merge_sort(left)
7 ...     sorted_right = merge_sort(right)
8 ...     return merge(sorted_left, sorted_right)
9 ...
```

If the argument `xlist` has more than one element, then statements in lines 4 and 5 break this `list` into two `lists` called `left` and `right`. (These `lists` will have the same number of

Figure 12.1: Steps used in the merge-sort algorithm to sort a `list` of ten integers. The original `list` is shown in Step 1. The `list` is divided into smaller `lists` until, in Step 5, the `lists` contain a single element. In Steps 5 through 9 these `lists` are merged so that eventually the final `list` correspond to the original `list`, but in sorted order.



elements if `xlist` has an even number of elements. If `xlist` has an odd number of elements `right` will have one more element than `left`.) Note that `left` and `right` are, in general, unsorted `lists`. Next, in line 6 and 7, `merge_sort()` is called to sort `left` and `right`. Then, finally, in line 8, these sorted `lists` are passed to `merge()` and the (sorted) `list` it produces is returned.

Though it only requires a few lines of code, `merge_sort()` is *not* a simple function. However, even without thinking about the function very much, you almost certainly have no problem understanding how it behaves for the base case when it is passed a single-element `list`. From there, it is not hard to figure out how `merge_sort()` behaves when it is passed a two-element `list`. In this case, with the initial call, the base case is not triggered. Instead, the two-element `list` is split into two single-element `lists`. When `merge_sort()` is called with these single-element `lists`, we simply get back these `lists` and then they are merged with `merge()` and the resulting two-element `list` is returned. `merge()` ensures the elements of the two-element `list` are in order. You can extend this line of logic to `lists` of any number of elements. Essentially what happens is the `list` is broken into a collection of single-element `lists` which are, in turn, merged into two-element `lists`. These are then merged, and so on.

Listing 12.15 demonstrates that `merge_sort()` behaves as advertised. In line 1, it is called with a `list` of integers. The resulting list, in line 2, is in sorted order. In line 3, `merge_sort()` is called with a `list` of strings. The resulting `list` in line 4 is again in sorted order. Provided all the element of the `list` can be compared using the relation operator <= (less than or equal to), then the `list` can be sorted using this implementation of `merge_sort()`. (The `print()` statement was removed from `merge()` for the sake of this demonstration.)

Listing 12.15 Demonstration of the `merge_sort()` function.

```
1 >>> merge_sort([7, 4, 1, 6, 5, 3, 9, 8 0, 2])
2 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
3 >>> merge_sort(['zebra', 'bat', 'dog', 'cat', 'ape', 'eel'])
4 ['ape', 'bat', 'cat', 'dog', 'eel', 'zebra']
```