

Chapter 13

Algorithm Design and Recursion

Objectives

- To understand basic techniques for analyzing the efficiency of algorithms.
- To know what searching is and understand the algorithms for linear and binary search.
- To understand the basic principles of recursive definitions and functions and be able to write simple recursive functions.
- To understand sorting in depth and know the algorithms for selection sort and merge sort.
- To appreciate how the analysis of algorithms can demonstrate that some problems are intractable and others are unsolvable.

If you have worked your way through to this point in the book, you are well on the way to becoming a programmer. Way back in Chapter 1, I discussed the relationship between programming and the study of computer science. Now that you have some programming skills, you are ready to start considering some broader issues in the field. Here we will take up one of the central issues, namely the design and analysis of algorithms. Along the way, you'll get a glimpse of recursion, a particularly powerful way of thinking about algorithms.

13.1 Searching

Let's begin by considering a very common and well-studied programming problem: *searching*. Searching is the process of looking for a particular value in a collection. For example, a program that maintains the membership list for a club might need to look up the information about a particular member. This involves some form of search process.

13.1.1 A Simple Searching Problem

To make the discussion of searching algorithms as simple as possible, let's boil the problem down to its essence. Here is the specification of a simple searching function:

```
def search(x, nums):
    # nums is a list of numbers and x is a number
    # Returns the position in the list where x occurs or -1 if
    # x is not in the list.
```

Here are a couple interactive examples that illustrate its behavior:

```
>>> search(4, [3, 1, 4, 2, 5])
2
>>> search(7, [3, 1, 4, 2, 5])
-1
```

In the first example, the function returns the index where 4 appears in the list. In the second example, the return value -1 indicates that 7 is not in the list.

You may recall from our discussion of list operations that Python actually provides a number of built-in search-related methods. For example, we can test to see if a value appears in a sequence using `in`.

```
if x in nums:
    # do something
```

If we want to know the position of `x` in a list, the `index` method fills the bill nicely.

```
>>> nums = [3,1,4,2,5]
>>> nums.index(4)
2
```

In fact, the only difference between our `search` function and `index` is that the latter raises an exception if the target value does not appear in the list. We could implement `search` using `index` by simply catching the exception and returning `-1` for that case.

```
def search(x, nums):
    try:
        return nums.index(x)
    except:
        return -1
```

This approach avoids the question, however. The real issue is how does Python actually search the list? What is the algorithm?

13.1.2 Strategy 1: Linear Search

Let's try our hand at developing a search algorithm using a simple "be the computer" strategy. Suppose that I gave you a page full of numbers in no particular order and asked whether the number 13 is in the list. How would you solve this problem? If you are like most people, you would simply scan down the list comparing each value to 13. When you see 13 in the list, you quit and tell me that you found it. If you get to the very end of the list without seeing 13, then you tell me it's not there.

This strategy is called a *linear search*. You are searching through the list of items one by one until the target value is found. This algorithm translates directly into simple code.

```
def search(x, nums):
    for i in range(len(nums)):
        if nums[i] == x: # item found, return the index value
            return i
    return -1 # loop finished, item was not in list
```

This algorithm was not hard to develop, and it will work very nicely for modest-sized lists. For an unordered list, this algorithm is as good as any. The Python `in` and `index` operations both implement linear searching algorithms.

If we have a very large collection of data, we might want to organize it in some way so that we don't have to look at every single item to determine where, or if, a particular value appears in the list. Suppose that the list is stored in sorted order (lowest to highest). As soon as we encounter a value that is greater than

the target value, we can quit the linear search without looking at the rest of the list. On average, that saves us about half of the work. But, if the list is sorted, we can do even better than this.

13.1.3 Strategy 2: Binary Search

When a list is ordered, there is a much better searching strategy, one that you probably already know. Have you ever played the number guessing game? I pick a number between 1 and 100, and you try to guess what it is. Each time you guess, I will tell you if your guess is correct, too high, or too low. What is your strategy?

If you play this game with a very young child, they might well adopt a strategy of simply guessing numbers at random. An older child might employ a systematic approach corresponding to linear search, guessing 1, 2, 3, 4, . . . until the mystery value is found.

Of course, virtually any adult will first guess 50. If told that the number is higher, then the range of possible values is 50–100. The next logical guess is 75. Each time we guess the middle of the remaining numbers to try to narrow down the possible range. This strategy is called a *binary search*. Binary means two, and at each step, we are dividing the remaining numbers into two parts.

We can employ a binary search strategy to look through a sorted list. The basic idea is that we use two variables to keep track of the endpoints of the range in the list where the item could be. Initially, the target could be anywhere in the list, so we start with variables `low` and `high` set to the first and last positions of the list, respectively.

The heart of the algorithm is a loop that looks at the item in the middle of the remaining range to compare it to `x`. If `x` is smaller than the middle item, then we move `top`, so that the search is narrowed to the lower half. If `x` is larger, then we move `low`, and the search is narrowed to the upper half. The loop terminates when `x` is found or there are no longer any more places to look (i.e., `low > high`). Here is the code:

```
def search(x, nums):
    low = 0
    high = len(nums) - 1
    while low <= high:           # There is still a range to search
        mid = (low + high) / 2   # position of middle item
        item = nums[mid]
        if x == item :          # Found it! Return the index
```

```
    return mid
elif x < item:          # x is in lower half of range
    high = mid - 1     #   move top marker down
else:                  # x is in upper half
    low = mid + 1     #   move bottom marker up
return -1              # no range left to search,
                      # x is not there
```

This algorithm is quite a bit more sophisticated than the simple linear search. You might want to trace through a couple of example searches to convince yourself that it actually works.

13.1.4 Comparing Algorithms

So far, we have developed two solutions to our simple searching problem. Which one is better? Well, that depends on what exactly we mean by better. The linear search algorithm is much easier to understand and implement. On the other hand, we expect that the binary search is more efficient, because it doesn't have to look at every value in the list. Intuitively, then, we might expect the linear search to be a better choice for small lists and binary search a better choice for larger lists. How could we actually confirm such intuitions?

One approach would be to do an empirical test. We could simply code up both algorithms and try them out on various sized lists to see how long the search takes. These algorithms are both quite short, so it would not be difficult to run a few experiments. When I tested the algorithms on my particular computer (a somewhat dated laptop), linear search was faster for lists of length 10 or less, and there was not much noticeable difference in the range of length 10–1000. After that, binary search was a clear winner. For a list of a million elements, linear search averaged 2.5 seconds to find a random value, whereas binary search averaged only 0.0003 seconds.

The empirical analysis has confirmed our intuition, but these are results from one particular machine under specific circumstances (amount of memory, processor speed, current load, etc.). How can we be sure that the results will always be the same?

Another approach is to analyze our algorithms abstractly to see how efficient they are. Other factors being equal, we expect the algorithm with the fewest number of “steps” to be the more efficient. But how do we count the number of steps? For example, the number of times that either algorithm goes through its

main loop will depend on the particular inputs. We have already guessed that the advantage of binary search increases as the size of the list increases.

Computer scientists attack these problems by analyzing the number of steps that an algorithm will take relative to the size or difficulty of the specific problem instance being solved. For searching, the difficulty is determined by the size of the collection. Obviously, it takes more steps to find a number in a collection of a million than it does in a collection of ten. The pertinent question is *how many steps are needed to find a value in a list of size n* . We are particularly interested in what happens as n gets very large.

Let's consider the linear search first. If we have a list of ten items, the most work our algorithm might have to do is to look at each item in turn. The loop will iterate at most ten times. Suppose the list is twice as big. Then we might have to look at twice as many items. If the list is three times as large, it will take three times as long, etc. In general, the amount of time required is linearly related to the size of the list n . This is what computer scientists call a *linear time* algorithm. Now you really know why it's called a linear search.

What about the binary search? Let's start by considering a concrete example. Suppose the list contains sixteen items. Each time through the loop, the remaining range is cut in half. After one pass, there are eight items left to consider. The next time through there will be four, then two, and finally one. How many times will the loop execute? It depends on how many times we can halve the range before running out of data. This table might help you to sort things out:

List size	Halvings
1	0
2	1
4	2
8	3
16	4

Can you see the pattern here? Each extra iteration of the loop doubles the size of the list. If the binary search loops i times, it can find a single value in a list of size 2^i . Each time through the loop, it looks at one value (the middle) in the list. To see how many items are examined in a list of size n , we need to solve this relationship: $n = 2^i$ for i . In this formula, i is just an exponent with a base of 2. Using the appropriate logarithm gives us this relationship: $i = \log_2 n$. If you are not entirely comfortable with logarithms, just remember that this value is the number of times that a collection of size n can be cut in half.

OK, so what does this bit of math tell us? Binary search is an example of a *log time* algorithm. The amount of time it takes to solve a given problem grows as the log of the problem size. In the case of binary search, each additional iteration doubles the size of the problem that we can solve.

You might not appreciate just how efficient binary search really is. Let me try to put it in perspective. Suppose you have a New York City phone book with, say, twelve million names listed in alphabetical order. You walk up to a typical New Yorker on the street and make the following proposition (assuming their number is listed): “I’m going to try guessing your name. Each time I guess a name, you tell me if your name comes alphabetically before or after the name I guess.” How many guesses will you need?

Our analysis above shows the answer to this question is $\log_2 12,000,000$. If you don’t have a calculator handy, here is a quick way to estimate the result. $2^{10} = 1024$ or roughly 1000, and $1000 \times 1000 = 1,000,000$. That means that $2^{10} \times 2^{10} = 2^{20} \approx 1,000,000$. That is, 2^{20} is approximately one million. So, searching a million items requires only 20 guesses. Continuing on, we need 21 guesses for two million, 22 for four million, 23 for eight million, and 24 guesses to search among sixteen million names. We can figure out the name of a total stranger in New York City using only 24 guesses! By comparison, a linear search would require (on average) 6 million guesses. Binary search is a phenomenally good algorithm!

I said earlier that Python uses a linear search algorithm to implement its built-in searching methods. If a binary search is so much better, why doesn’t Python use it? The reason is that the binary search is less general; in order to work, the list must be in order. If you want to use binary search on an unordered list, the first thing you have to do is put it in order or *sort* it. This is another well-studied problem in computer science, and one that we should look at. Before we turn to sorting, however, we need to generalize the algorithm design technique that we used to develop the binary search.

13.2 Recursive Problem-Solving

Remember, the basic idea behind the binary search algorithm was to successfully divide the problem in half. This is sometimes referred to as a “divide and conquer” approach to algorithm design, and it often leads to very efficient algorithms.

One interesting aspect of divide and conquer algorithms is that the original problem divides into subproblems that are just smaller versions of the original.

To see what I mean, think about the binary search again. Initially, the range to search is the entire list. Our first step is to look at the middle item in the list. Should the middle item turn out to be the target, then we are finished. If it is not the target, we continue *by performing binary search on either the top-half or the bottom half of the list.*

Using this insight, we might express the binary search algorithm in another way.

```
Algorithm: binarySearch -- search for x in nums[low]...nums[high]

mid = (low + high) / 2
if low > high
    x is not in nums
elif x < nums[mid]
    perform binary search for x in nums[low]...nums[mid-1]
else
    perform binary search for x in nums[mid+1]...nums[high]
```

Rather than using a loop, this definition of the binary search seems to refer to itself. What is going on here? Can we actually make sense of such a thing?

13.2.1 Recursive Definitions

A description of something that refers to itself is called a *recursive* definition. In our last formulation, the binary search algorithm makes use of its own description. A “call” to binary search “recurs” inside of the definition—hence, the label “recursive definition.”

At first glance, you might think recursive definitions are just nonsense. Surely you have had a teacher who insisted that you can’t use a word inside its own definition? That’s called a circular definition and is usually not worth much credit on an exam.

In mathematics, however, certain recursive definitions are used all the time. As long as we exercise some care in the formulation and use of recursive definitions, they can be quite handy and surprisingly powerful. The classic recursive example in mathematics is the definition of factorial.

Back in Chapter 3, we defined the factorial of a value like this:

$$n! = n(n - 1)(n - 2) \dots (1)$$

For example, we can compute

$$5! = 5(4)(3)(2)(1)$$

Recall that we implemented a program to compute factorials using a simple loop that accumulates the factorial product.

Looking at the calculation of $5!$, you will notice something interesting. If we remove the 5 from the front, what remains is a calculation of $4!$. In general, $n! = n(n - 1)!$. In fact, this relation gives us another way of expressing what is meant by factorial in general. Here is a recursive definition:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n - 1)! & \text{otherwise} \end{cases}$$

This definition says that the factorial of 0 is, by definition, 1, while the factorial of any other number is defined to be that number times the factorial of one less than that number.

Even though this definition is recursive, it is not circular. In fact, it provides a very simple method of calculating a factorial. Consider the value of $4!$. By definition we have

$$4! = 4(4 - 1)! = 4(3!)$$

But what is $3!$? To find out, we apply the definition again.

$$4! = 4(3!) = 4[(3)(3 - 1)!] = 4(3)(2!)$$

Now, of course, we have to expand $2!$, which requires $1!$, which requires $0!$. Since $0!$ is simply 1, that's the end of it.

$$4! = 4(3!) = 4(3)(2!) = 4(3)(2)(1!) = 4(3)(2)(1)(0!) = 4(3)(2)(1)(1) = 24$$

You can see that the recursive definition is not circular because each application causes us to request the factorial of a smaller number. Eventually we get down to 0, which doesn't require another application of the definition. This is called a *base case* for the recursion. When the recursion bottoms out, we get a closed expression that can be directly computed. All good recursive definitions have these key characteristics:

1. There are one or more base cases for which no recursion is required.
2. All chains of recursion eventually end up at one of the base cases.

The simplest way to guarantee that these two conditions are met is to make sure that each recursion always occurs on a *smaller* version of the original problem. A very small version of the problem that can be solved without recursion then becomes the base case. This is exactly how the factorial definition works.

13.2.2 Recursive Functions

You already know that the factorial can be computed using a loop with an accumulator. That implementation has a natural correspondence to the original definition of factorial. Can we also implement a version of factorial that follows the recursive definition?

If we write factorial as a separate function, the recursive definition translates directly into code.

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

Do you see how the definition that refers to itself turns into a function that calls itself? This is called a *recursive function*. The function first checks to see if we are at the base case `n == 0` and, if so, returns 1. If we are not yet at the base case, the function returns the result of multiplying `n` by the factorial of `n-1`. The latter is calculated by a recursive call to `fact(n-1)`.

I think you will agree that this is a reasonable translation of the recursive definition. The really cool part is that it actually works! We can use this recursive function to compute factorial values.

```
>>> from recfact import fact
>>> fact(4)
24
>>> fact(10)
3628800
```

Some beginning programmers are surprised by this result, but it follows naturally from the semantics for functions that we discussed way back in Chapter 6. Remember that each call to a function starts that function anew. That means it has its own copy of any local values, including the values of the parameters. Figure 13.1 shows the sequence of recursive calls that computes $5!$. Note especially how each return value is multiplied by a value of `n` appropriate for each function invocation. The values of `n` are stored on the way down the chain and then used on the way back up as the function calls return.

There are many problems for which recursion can yield an elegant and efficient solution. The next few sections present examples of recursive problem solving.

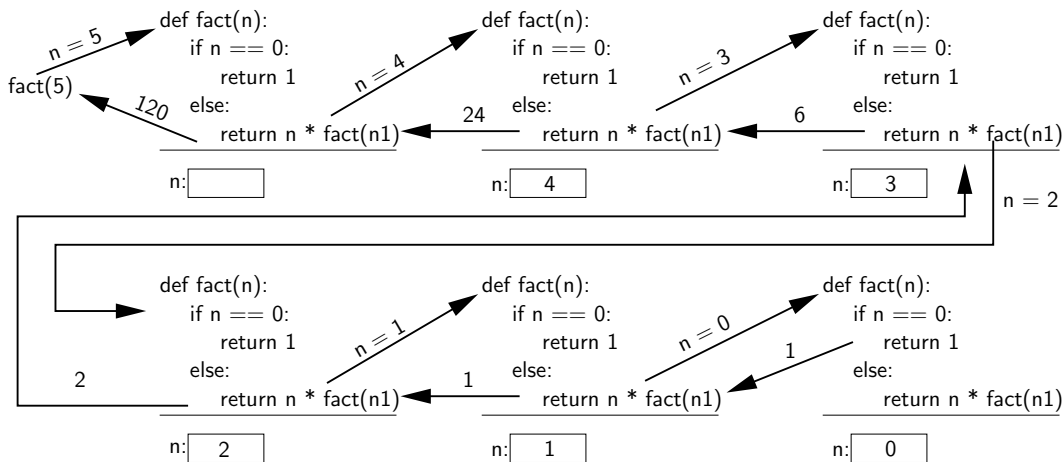


Figure 13.1: Recursive computation of 5!

13.2.3 Example: String Reversal

Python lists have a built-in method that can be used to reverse the list. Suppose that you want to compute the reverse of a string. One way to handle this problem effectively would be to convert the string into a list of characters, reverse the list, and turn the list back into a string. Using recursion, however, we can easily write a function that computes the reverse directly, without having to detour through a list representation.

The basic idea is to think of a string as a recursive object; a large string is composed out of smaller objects, which are also strings. In fact, one very handy way to divide up virtually any sequence is to think of it as a single first item that just happens to be followed by another sequence. In the case of a string, we can divide it up into its first character and “all the rest.” If we reverse the rest of the string and then put the first character on the end of that, we’ll have the reverse of the whole string.

Let’s code up that algorithm and see what happens.

```
def reverse(s):
    return reverse(s[1:]) + s[0]
```

Notice how this function works. The slice `s[1:]` gives all but the first character of the string. We reverse the slice (recursively) and then concatenate the first

character (`s[0]`) onto the end of the result. It might be helpful to think in terms of a specific example. If `s` is the string "abc", then `s[1:]` is the string "bc". Reversing this yields "cb" and tacking on `s[0]` yields "cba". That's just what we want.

Unfortunately, this function doesn't quite work. Here's what happens when I try it out:

```
>>> reverse("Hello")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in reverse
  File "<stdin>", line 2, in reverse
  ...
  File "<stdin>", line 2, in reverse
RuntimeError: maximum recursion depth exceeded
```

I've only shown a portion of the output, it actually consisted of 1000 lines! What's happened here?

Remember, to build a correct recursive function we need a base case for which no recursion is required, otherwise the recursion is circular. In our haste to code up the function, we forgot to include a base case. What we have written is actually an *infinite recursion*. Every call to `reverse` contains another call to `reverse`, so none of them ever return. Of course, each time a function is called it takes up some memory (to store the parameters and local variables), so this process can't go on forever. Python puts a stop to it after 1000 calls, the default "maximum recursion depth."

Let's go back and put in a suitable base case. When performing recursion on sequences, the base case is often an empty sequence or a sequence containing just one item. For our reversing problem we can use an empty string as the base case, since an empty string is its own reverse. The recursive calls to `reverse` are always on a string that is one character shorter than the original, so we'll eventually end up at an empty string. Here's a correct version of `reverse`:

```
def reverse(s):
    if s == "":
        return s
    else:
        return reverse(s[1:]) + s[0]
```

This version works as advertised.

```
>>> reverse("Hello")
'olleH'
```

13.2.4 Example: Anagrams

An anagram is formed by rearranging the letters of a word. Anagrams are often used in word games, and forming anagrams is a special case of generating the possible permutations (rearrangements) of a sequence, a problem that pops up frequently in many areas of computing and mathematics.

Let's try our hand at writing a function that generates a list of all the possible anagrams of a string. We'll apply the same approach that we used in the previous example by slicing the first character off of the string. Suppose the original string is "abc", then the tail of the string is "bc". Generating the list of all the anagrams of the tail gives us ["bc", "cb"], as there are only two possible arrangements of two characters. To add back the first letter, we need to place it in all possible positions in each of these two smaller anagrams: ["abc", "bac", "bca", "acb", "cab", "cba"]. The first three anagrams come from placing "a" in every possible place in "bc", and the second three come from inserting "a" into "cb".

Just as in our previous example, we can use an empty string as the base case for the recursion. The only possible arrangement of characters in an empty string is the empty string itself. Here is the completed recursive function:

```
def anagrams(s):
    if s == "":
        return [s]
    else:
        ans = []
        for w in anagrams(s[1:]):
            for pos in range(len(w)+1):
                ans.append(w[:pos]+s[0]+w[pos:])
        return ans
```

Notice in the else I have used a list to accumulate the final results. In the nested for loops, the outer loop iterates through each anagram of the tail of `s`, and the inner loop goes through each position in the anagram and creates a new string with the original first character inserted into that position. The expression `w[:pos]+s[0]+w[pos:]` looks a bit tricky, but it's not too hard to decipher. Taking `w[:pos]` gives the portion of `w` up to (but not including) `pos`, and `w[pos:]`

yields everything from `pos` through the end. Sticking `s[0]` between these two effectively inserts it into `w` at `pos`. The inner loop goes up to `len(w)+1` so that the new character can be added to the very end of the anagram.

Here is our function in action:

```
>>> anagrams("abc")
['abc', 'bac', 'bca', 'acb', 'cab', 'cba']
```

I didn't use "Hello" for this example because that generates more anagrams than I wanted to print. The number of anagrams of a word is the factorial of the length of the word.

13.2.5 Example: Fast Exponentiation

Another good example of recursion is a clever algorithm for raising values to an integer power. The naive way to compute a^n for an integer n is simply to multiply a by itself n times $a^n = a * a * a * \dots * a$. We can easily implement this using a simple accumulator loop.

```
def loopPower(a, n):
    ans = 1
    for i in range(n):
        ans = ans * a
    return ans
```

Divide and conquer suggests another way to perform this calculation. Suppose we want to calculate 2^8 . By the laws of exponents, we know that $2^8 = 2^4(2^4)$. So if we first calculate 2^4 , we can just do one more multiply to get 2^8 . To compute 2^4 , we can use the fact that $2^4 = 2^2(2^2)$. And, of course, $2^2 = 2(2)$. Putting the calculation together we start with $2(2) = 4$ and $4(4) = 16$ and $16(16) = 256$. We have calculated the value of 2^8 using just three multiplications. The basic insight is to use the relationship $a^n = a^{n/2}(a^{n/2})$.

In the example I gave, the exponents were all even. In order to turn this idea into a general algorithm, we also have to handle odd values of n . This can be done with one more multiplication. For example, $2^9 = 2^4(2^4)(2)$. Here is the general relationship:

$$a^n = \begin{cases} a^{n/2}(a^{n/2}) & \text{if } n \text{ is even} \\ a^{n/2}(a^{n/2})(a) & \text{if } n \text{ is odd} \end{cases}$$

In this formula I am exploiting integer division; if n is 9 then $n/2$ is 4.


```
        return -1
    mid = (low + high) / 2
    item = nums[mid]
    if item == x:                # Found it! Return the index
        return mid
    elif x < item:              # Look in lower half
        return recBinSearch(x, nums, low, mid-1)
    else:                       # Look in upper half
        return recBinSearch(x, nums, mid+1, high)
```

We can then implement our original search function using a suitable call to the recursive binary search, telling it to start the search between 0 and `len(nums)-1`.

```
def search(x, nums):
    return recBinSearch(x, nums, 0, len(nums)-1)
```

Of course, our original looping version is probably a bit faster than this recursive version because calling functions is generally slower than iterating a loop. The recursive version, however, makes the divide-and-conquer structure of binary search much more obvious. Below we will see examples where recursive divide-and-conquer approaches provide a natural solution to some problems where loops are awkward.

13.2.7 Recursion vs. Iteration

I'm sure by now you've noticed that there are some similarities between iteration (looping) and recursion. In fact, recursive functions are a generalization of loops. Anything that can be done with a loop can also be done by a simple kind of recursive function. In fact, there are programming languages that use recursion exclusively. On the other hand, some things that can be done very simply using recursion are quite difficult to do with loops.

For a number of the problems we've looked at so far, we have had both iterative and recursive solutions. In the case of factorial and binary search, the loop version and the recursive version do basically the same calculations, and they will have roughly the same efficiency. The looping versions are probably a bit faster because calling functions is generally slower than iterating a loop, but in a modern language the recursive algorithms are probably fast enough.

In the case of the exponentiation algorithm, the recursive version and the looping version actually implement very different algorithms. If you think about it a bit, you will see that the looping version is linear and the recursive version

executes in log time. The difference between these two is similar to the difference between linear search and binary search, so the recursive algorithm is clearly superior. In the next section, you'll be introduced to a recursive sorting algorithm that is also very efficient.

As you have seen, recursion can be a very useful problem-solving technique that can lead to efficient and effective algorithms. But you have to be careful. It's also possible to write some very inefficient recursive algorithms. One classic example is calculating the *n*th Fibonacci number.

The Fibonacci sequence is the sequence of numbers 1, 1, 2, 3, 5, 8, . . . It starts with two 1s and successive numbers are the sum of the previous two. One way to compute the *n*th Fibonacci value is to use a loop that produces successive terms of the sequence.

In order to compute the next Fibonacci number, we always need to keep track of the previous two. We can use two variables, *curr* and *prev*, to keep track these values. Then we just need a loop that adds these together to get the next value. At that point, the old value of *curr* becomes the new value of *prev*. Here is one way to do it in Python:

```
def loopfib(n):
    # returns the nth Fibonacci number

    curr = 1
    prev = 1
    for i in range(n-2):
        curr, prev = curr+prev, curr
    return curr
```

I used simultaneous assignment to compute the next values of *curr* and *prev* in a single step. Notice that the loop only goes around $n - 2$ times, because the first two values have already been assigned and do not require an addition.

The Fibonacci sequence also has an elegant recursive definition.

$$fib(n) = \begin{cases} 1 & \text{if } n < 3 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

We can turn this recursive definition directly into a recursive function.

```
def fib(n):
    if n < 3:
        return 1
```


13.3 Sorting Algorithms

The sorting problem provides a nice test bed for the algorithm design techniques we have been discussing. Remember, the basic sorting problem is to take a list and rearrange it so that the values are in increasing (actually, nondecreasing) order.

13.3.1 Naive Sorting: Selection Sort

Let's start with a simple "be the computer" approach to sorting. Suppose you have a stack of index cards, each with a number on it. The stack has been shuffled, and you need to put the cards back in order. How would you accomplish this task?

There are any number of good systematic approaches. One simple method is to look through the deck to find the smallest value and then place that value at the front of the stack (or perhaps in a separate stack). Then you could go through and find the smallest of the remaining cards and put it next in line, etc. Of course, this means that you'll also need an algorithm for finding the smallest remaining value. You can use the same approach we used for finding the max of a list (see Chapter 7). As you go through, you keep track of the smallest value seen so far, updating that value whenever you find a smaller one.

The algorithm I just described is called *selection sort*. Basically, the algorithm consists of a loop and each time through the loop, we select the smallest of the remaining elements and move it into its proper position. Applying this idea to a list of n elements, we proceed by finding the smallest value in the list and putting it into the 0^{th} position. Then we find the smallest remaining value (from positions $1-(n-1)$) and put it in position 1. Next, the smallest value from positions $2-(n-1)$ goes into position 2, etc. When we get to the end of the list, everything will be in its proper place.

There is one subtlety in implementing this algorithm. When we place a value into its proper position, we need to make sure that we do not accidentally lose the value that was originally stored in that position. For example, if the smallest item is in position 10, moving it into position 0 involves an assignment.

```
nums[0] = nums[10]
```

But this wipes out the value currently in `nums[0]`; it really needs to be moved to another location in the list. A simple way to save the value is to swap it with the one that we are moving. Using simultaneous assignment, the statement

```
nums[0], nums[10] = nums[10], nums[0]
```

places the value from position 10 at the front of the list, but preserves the original first value by stashing it into location 10.

Using this idea, it is a simple matter to write a selection sort in Python. I will use a variable called `bottom` to keep track of which position in the list we are currently filling, and the variable `mp` will be used to track the location of the smallest remaining value. The comments in this code explain this implementation of selection sort:

```
def selSort(nums):
    # sort nums into ascending order

    n = len(nums)

    # For each position in the list (except the very last)
    for bottom in range(n-1):
        # find the smallest item in nums[bottom]..nums[n-1]

        mp = bottom                # bottom is smallest initially
        for i in range(bottom+1,n): # look at each position
            if nums[i] < nums[mp]: # this one is smaller
                mp = i            # remember its index

        # swap smallest item to the bottom
        nums[bottom], nums[mp] = nums[mp], nums[bottom]
```

One thing to notice about this algorithm is the accumulator for finding the minimum value. Rather than actually storing the minimum seen so far, `mp` just remembers the position of the minimum. A new value is tested by comparing the item in position `i` to the item in position `mp`. You should also notice that `bottom` stops at the second to last item in the list. Once all of the items up to the last have been put in the proper place, the last item has to be the largest, so there is no need to bother looking at it.

The selection sort algorithm is easy to write and works well for moderate-sized lists, but it is not a very efficient sorting algorithm. We'll come back and analyze it after we've developed another algorithm.

13.3.2 Divide and Conquer: Merge Sort

As discussed above, one technique that often works for developing efficient algorithms is the divide-and-conquer approach. Suppose a friend and I were working together trying to put our deck of cards in order. We could divide the problem up by splitting the deck of cards in half with one of us sorting each of the halves. Then we just need to figure out a way of combining the two sorted stacks.

The process of combining two sorted lists into a single sorted result is called *merging*. The basic outline of our divide and conquer algorithm, called *mergeSort* looks like this:

```
Algorithm: mergeSort nums
```

```
split nums into two halves
sort the first half
sort the second half
merge the two sorted halves back into nums
```

The first step in the algorithm is simple, we can just use list slicing to handle that. The last step is to merge the lists together. If you think about it, merging is pretty simple. Let's go back to our card stack example to flesh out the details. Since our two stacks are sorted, each has its smallest value on top. Whichever of the top values is the smallest will be the first item in the merged list. Once the smaller value is removed, we can look at the tops of the stacks again, and whichever top card is smaller will be the next item in the list. We just continue this process of placing the smaller of the two top values into the big list until one of the stacks runs out. At that point, we finish out the list with the cards from the remaining stack.

Here is a Python implementation of the merge process. In this code, `lst1` and `lst2` are the smaller lists and `lst3` is the larger list where the results are placed. In order for the merging process to work, the length of `lst3` must be equal to the sum of the lengths of `lst1` and `lst2`. You should be able to follow this code by studying the accompanying comments:

```
def merge(lst1, lst2, lst3):
    # merge sorted lists lst1 and lst2 into lst3

    # these indexes keep track of current position in each list
    i1, i2, i3 = 0, 0, 0 # all start at the front
```

```
n1, n2 = len(lst1), len(lst2)

# Loop while both lst1 and lst2 have more items
while i1 < n1 and i2 < n2:
    if lst1[i1] < lst2[i2]: # top of lst1 is smaller
        lst3[i3] = lst1[i1] # copy it into current spot in lst3
        i1 = i1 + 1
    else: # top of lst2 is smaller
        lst3[i3] = lst2[i2] # copy it into current spot in lst3
        i2 = i2 + 1
    i3 = i3 + 1 # item added to lst3, update position

# Here either lst1 or lst2 is done. One of the following loops will
# execute to finish up the merge.

# Copy remaining items (if any) from lst1
while i1 < n1:
    lst3[i3] = lst1[i1]
    i1 = i1 + 1
    i3 = i3 + 1
# Copy remaining items (if any) from lst2
while i2 < n2:
    lst3[i3] = lst2[i2]
    i2 = i2 + 1
    i3 = i3 + 1
```

OK, now we can slice a list into two, and if those lists are sorted, we know how to merge them back into a single list. But how are we going to sort the smaller lists? Well, let's think about it. We are trying to sort a list, and our algorithm requires us to sort two smaller lists. This sounds like a perfect place to use recursion. Maybe we can use `mergeSort` itself to sort the two lists. Let's go back to our recursion guidelines to develop a proper recursive algorithm.

In order for recursion to work, we need to find at least one base case that does not require a recursive call, and we also have to make sure that recursive calls are always made on smaller versions of the original problem. The recursion in our `mergeSort` will always occur on a list that is half as large as the original, so the latter property is automatically met. Eventually, our lists will be very small, containing only a single item. Fortunately, a list with just one item is already sorted! Voilà, we have a base case. When the length of the list is less

than 2, we do nothing, leaving the list unchanged.

Given our analysis, we can update the mergeSort algorithm to make it properly recursive.

```
if len(nums) > 1:
    split nums into two halves
    mergeSort the first half
    mergeSort the second half
    merge the two sorted halves back into nums
```

We can translate this algorithm directly into Python code.

```
def mergeSort(nums):
    # Put items of nums in ascending order
    n = len(nums)
    # Do nothing if nums contains 0 or 1 items
    if n > 1:
        # split into two sublists
        m = n / 2
        nums1, nums2 = nums[:m], nums[m:]
        # recursively sort each piece
        mergeSort(nums1)
        mergeSort(nums2)
        # merge the sorted pieces back into original list
        merge(nums1, nums2, nums)
```

You might try tracing this algorithm with a small list (say eight elements), just to convince yourself that it really works. In general, though, tracing through recursive algorithms can be tedious and often not very enlightening.

Recursion is closely related to mathematical induction, and it requires practice before it becomes comfortable. As long as you follow the rules and make sure that every recursive chain of calls eventually reaches a base case, your algorithms *will* work. You just have to trust and let go of the grungy details. Let Python worry about that for you!

13.3.3 Comparing Sorts

Now that we have developed two sorting algorithms, which one should we use? Before we actually try them out, let's do some analysis. As in the searching problem, the difficulty of sorting a list depends on the size of the list. We need to

figure out how many steps each of our sorting algorithms requires as a function of the size of the list to be sorted.

Take a look back at the algorithm for selection sort. Remember, this algorithm works by first finding the smallest item, then finding the smallest of the remaining items, and so on. Suppose we start with a list of size n . In order to find the smallest value, the algorithm has to inspect each of the n items. The next time around the outer loop, it has to find the smallest of the remaining $n - 1$ items. The third time around, there are $n - 2$ items of interest. This process continues until there is only one item left to place. Thus, the total number of iterations of the inner loop for the selection sort can be computed as the sum of a decreasing sequence.

$$n + (n - 1) + (n - 2) + (n - 3) + \cdots + 1$$

In other words, the time required by selection sort to sort a list of n items is proportional to the sum of the first n whole numbers. There is a well-known formula for this sum, but even if you do not know the formula, it is easy to derive. If you add the first and last numbers in the series you get $n + 1$. Adding the second and second to last values gives $(n - 1) + 2 = n + 1$. If you keep pairing up the values working from the outside in, all of the pairs add to $n + 1$. Since there are n numbers, there must be $\frac{n}{2}$ pairs. That means the sum of all the pairs is $\frac{n(n+1)}{2}$.

You can see that the final formula contains an n^2 term. That means that the number of steps in the algorithm is proportional to the square of the size of the list. If the size of the list doubles, the number of steps quadruples. If the size triples, it will take nine times as long to finish. Computer scientists call this a *quadratic* or n^2 algorithm.

Let's see how that compares to the merge sort algorithm. In the case of merge sort, we divided a list into two pieces and sorted the individual pieces before merging them together. The real work is done during the merge process when the values in the sublists are copied back into the original list.

Figure 13.3 depicts the merging process to sort the list [3, 1, 4, 1, 5, 9, 2, 6]. The dashed lines show how the original list is continually halved until each item is its own list with the values shown at the bottom. The single-item lists are then merged back up into the two item lists to produce the values shown in the second level. The merging process continues up the diagram to produce the final sorted version of the list shown at the top.

The diagram makes analysis of the merge sort easy. Starting at the bottom level, we have to copy the n values into the second level. From the second to

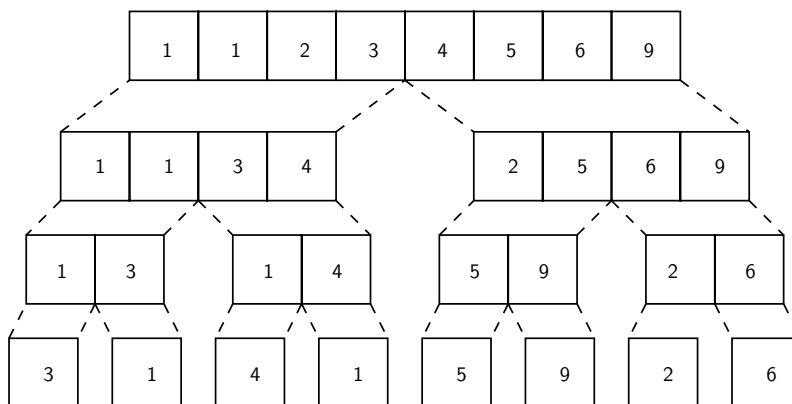


Figure 13.3: Merges required to sort [3, 1, 4, 1, 5, 9, 2, 6].

third level, the n values need to be copied again. Each level of merging involves copying n values. The only question left to answer is how many levels are there? This boils down to how many times a list of size n can be split in half. You already know from the analysis of binary search that this is just $\log_2 n$. Therefore, the total work required to sort n items is $n \log_2 n$. Computer scientists call this an $n \log n$ algorithm.

So which is going to be better, the n^2 selection sort or the $n \log n$ merge sort? If the input size is small, the selection sort might be a little faster because the code is simpler and there is less overhead. What happens, though as n gets larger? We saw in the analysis of binary search that the log function grows *very* slowly ($\log_2 16,000,000 \approx 24$) so $n(\log_2 n)$ will grow much slower than $n(n)$.

Empirical testing of these two algorithms confirms this analysis. On my computer, selection sort beats merge sort on lists up to size about 50, which takes around 0.008 seconds. On larger lists, the merge sort dominates. Figure 13.4 shows a comparison of the time required to sort lists up to size 3000. You can see that the curve for selection sort veers rapidly upward (forming half of a parabola), while the merge sort curve looks almost straight (look at the bottom). For 3000 items, selection sort requires over 30 seconds while merge sort completes the task in about $\frac{3}{4}$ of a second. Merge sort can sort a list of 20,000 items in less than six seconds; selection sort takes around 20 minutes. That's quite a difference!

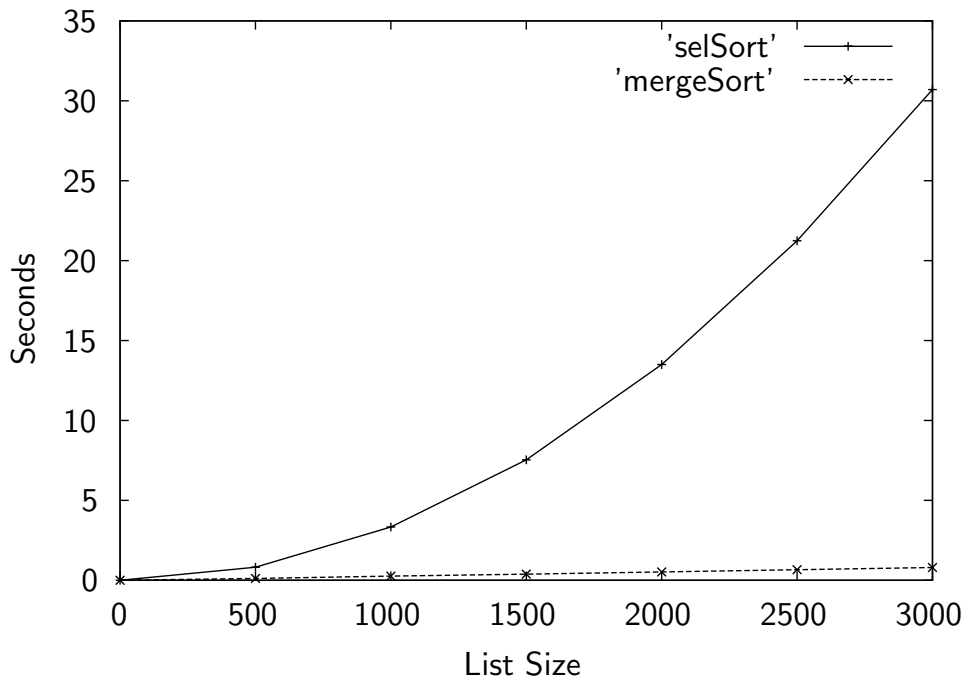


Figure 13.4: Experimental comparison of selection sort and merge sort.

13.4 Hard Problems

Using our divide-and-conquer approach we were able to design good algorithms for the searching and sorting problems. Divide and conquer and recursion are very powerful techniques for algorithm design. However, not all problems have efficient solutions.

13.4.1 Towers of Hanoi

One very elegant application of recursive problem solving is the solution to a mathematical puzzle usually called the Tower of Hanoi or Tower of Brahma. This puzzle is generally attributed to the French mathematician Édouard Lucas, who published an article about it in 1883. The legend surrounding the puzzle goes something like this:

Somewhere in a remote region of the world is a monastery of a very devout religious order. The monks have been charged with a sacred task that keeps time for the universe. At the beginning of all things, the monks were given a table that supports three vertical posts. On one of the posts was a stack of 64 concentric golden disks. The disks are of varying radii and stacked in the shape of a beautiful pyramid. The monks were charged with the task of moving the disks from the first post to the third post. When the monks have completed their task, all things will crumble to dust and the universe will end.

Of course, if that's all there were to the problem, the universe would have ended long ago. To maintain divine order, the monks must abide by certain rules.

1. Only one disk may be moved at a time.
2. A disk may not be “set aside.” It may only be stacked on one of the three posts.
3. A larger disk may never be placed on top of a smaller one.

Versions of this puzzle were quite popular at one time, and you can still find variations on this theme in toy and puzzle stores. Figure 13.5 depicts a small version containing only eight disks. The task is to move the tower from the first post to the third post using the center post as sort of a temporary resting place during the process. Of course, you have to follow the three sacred rules given above.

We want to develop an algorithm for this puzzle. You can think of our algorithm either as a set of steps that the monks need to carry out, or as a program that generates a set of instructions. For example, if we label the three posts A, B, and C. The instructions might start out like this:

```
Move disk from A to C.  
Move disk from A to B.  
Move disk from C to B.  
...
```

This is a difficult puzzle for most people to solve. Of course, that is not surprising, since most people are not trained in algorithm design. The solution process is actually quite simple—if you know about recursion.

Let's start by considering some really easy cases. Suppose we have a version of the puzzle with only one disk. Moving a tower consisting of a single disk is simple enough; we just remove it from A and put it on C. Problem solved. OK,

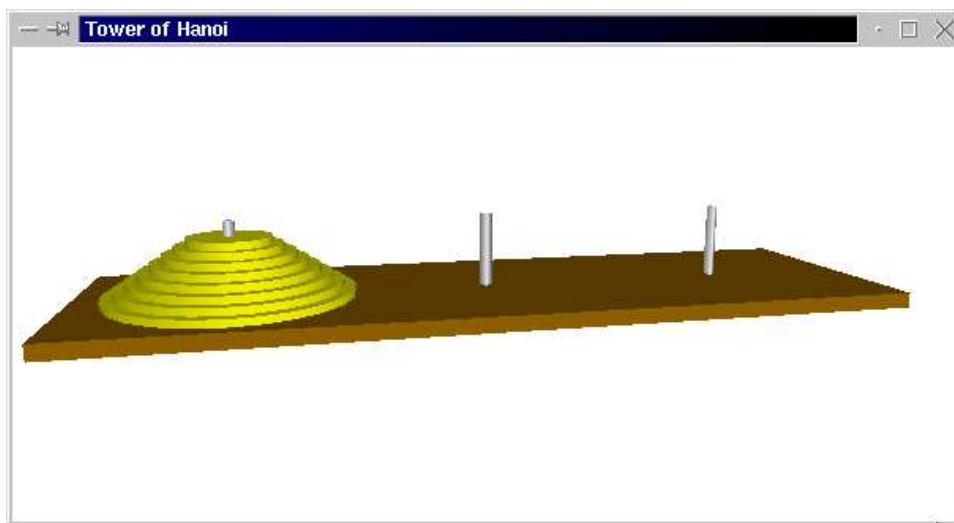


Figure 13.5: Tower of Hanoi puzzle with eight disks.

what if there are two disks? I need to get the larger of the two disks over to post C, but the smaller one is sitting on top of it. I need to move the smaller disk out of the way, and I can do this by moving it to post B. Now the large disk on A is clear; I can move it to C and then move the smaller disk from post B onto post C.

Now let's think about a tower of size three. In order to move the largest disk to post C, I first have to move the two smaller disks out of the way. The two smaller disks form a tower of size two. Using the process I outlined above, I could move this tower of two onto post B, and that would free up the largest disk so that I can move it to post C. Then I just have to move the tower of two disks from post B onto post C. Solving the three disk case boils down to three steps:

1. Move a tower of two from A to B.
2. Move one disk from A to C.
3. Move a tower of two from B to C.

The first and third steps involve moving a tower of size two. Fortunately, we have already figured out how to do this. It's just like solving the puzzle with

two disks, except that we move the tower from A to B using C as the temporary resting place, and then from B to C using A as the temporary.

We have just developed the outline of a simple recursive algorithm for the general process of moving a tower of any size from one post to another.

Algorithm: move n -disk tower from source to destination via resting place

```
move n-1 disk tower from source to resting place
move 1 disk tower from source to destination
move n-1 disk tower from resting place to destination
```

What is the base case for this recursive process? Notice how a move of n disks results in two recursive moves of $n - 1$ disks. Since we are reducing n by one each time, the size of the tower will eventually be 1. A tower of size 1 can be moved directly by just moving a single disk; we don't need any recursive calls to remove disks above it.

Fixing up our general algorithm to include the base case gives us a working `moveTower` algorithm. Let's code it up in Python. Our `moveTower` function will need parameters to represent the size of the tower, n ; the source post, `source`; the destination post, `dest`; and the temporary resting post, `temp`. We can use an int for n and the strings "A," "B," and "C" to represent the posts. Here is the code for `moveTower`:

```
def moveTower(n, source, dest, temp):
    if n == 1:
        print "Move disk from", source, "to", dest+ "."
    else:
        moveTower(n-1, source, temp, dest)
        moveTower(1, source, dest, temp)
        moveTower(n-1, temp, dest, source)
```

See how easy that was? Sometimes using recursion can make otherwise difficult problems almost trivial.

To get things started, we just need to supply values for our four parameters. Let's write a little function that prints out instructions for moving a tower of size n from post A to post C.

```
def hanoi(n):
    moveTower(n, "A", "C", "B")
```

Now we're ready to try it out. Here are solutions to the three- and four-disk puzzles. You might want to trace through these solutions to convince yourself that they work.

```
>>> hanoi(3)
Move disk from A to C.
Move disk from A to B.
Move disk from C to B.
Move disk from A to C.
Move disk from B to A.
Move disk from B to C.
Move disk from A to C.
```

```
>>> hanoi(4)
Move disk from A to B.
Move disk from A to C.
Move disk from B to C.
Move disk from A to B.
Move disk from C to A.
Move disk from C to B.
Move disk from A to B.
Move disk from A to C.
Move disk from B to C.
Move disk from B to A.
Move disk from C to A.
Move disk from B to C.
Move disk from A to B.
Move disk from A to C.
Move disk from B to C.
```

So, our solution to the Tower of Hanoi is a “trivial” algorithm requiring only nine lines of code. What is this problem doing in a section labeled *hard problems*? To answer that question, we have to look at the efficiency of our solution. Remember, when I talk about the efficiency of an algorithm I mean how many steps it requires to solve a given size problem. In this case, the difficulty is determined by the number of disks in the tower. The question we want to answer is *how many steps does it take to move a tower of size n ?*

Just looking at the structure of our algorithm, you can see that moving a tower of size n requires us to move a tower of size $n - 1$ twice, once to move

it off the largest disk, and again to put it back on top. If we add another disk to the tower, we essentially double the number of steps required to solve it. The relationship becomes clear if you simply try out the program on increasing puzzle sizes.

Number of Disks	Steps in Solution
1	1
2	3
3	7
4	15
5	31

In general, solving a puzzle of size n will require $2^n - 1$ steps.

Computer scientists call this an *exponential time* algorithm, since the measure of the size of the problem, n , appears in the exponent of this formula. Exponential algorithms blow up very quickly and can only be practically solved for relatively small sizes, even on the fastest computers. Just to illustrate the point, if our monks really started with a tower of just 64 disks and moved one disk every second, 24 hours a day, every day, without making a mistake, it would still take them over 580 *billion* years to complete their task. Considering that the universe is roughly 15 billion years old now, I'm not too worried about turning to dust just yet.

Even though the algorithm for Towers of Hanoi is easy to express, it belongs to a class known as *intractable* problems. These are problems that require too much computing power (either time or memory) to be solved in practice, except for the simplest cases. And in this sense, our toy-store puzzle does indeed represent a hard problem. But some problems are even harder than intractable, and we'll meet one of those in the next section.

13.4.2 The Halting Problem

Let's just imagine for a moment that this book has inspired you to pursue a career as a computer professional. It's now six years later, and you are a well-established software developer. One day, your boss comes to you with an important new project, and you are supposed to drop everything and get right on it.

It seems that your boss has had a sudden inspiration on how your company can double its productivity. You've recently hired a number of rather inexperienced programmers, and debugging their code is taking an inordinate amount of

time. Apparently, these wet-behind-the-ears newbies tend to accidentally write a lot of programs with infinite loops (you've been there, right?). They spend half the day waiting for their computers to reboot so they can track down the bugs. Your boss wants you to design a program that can analyze source code and detect whether it contains an infinite loop before actually running it on test data. This sounds like an interesting problem, so you decide to give it a try.

As usual, you start by carefully considering the specifications. Basically, you want a program that can read other programs and determine whether they contain an infinite loop. Of course, the behavior of a program is determined not just by its code, but also by the input it is given when it runs. In order to determine if there is an infinite loop, you will have to know what the input will be. You decide on the following specification:

Program: Halting Analyzer

Inputs: A Python program file.

The input for the program.

Outputs: "OK" if the program will eventually stop.

"FAULTY" if the program has an infinite loop.

Right away you notice something interesting about this program. This is a program that examines other programs. You may not have written many of these before, but you know that it's not a problem in principle. After all, compilers and interpreters are common examples of programs that analyze other programs. You can represent both the program that is being analyzed and the proposed input to the program as Python strings.

There is something else very interesting about this assignment. You are being asked to solve a very famous puzzle known as the *Halting Problem*, and it's unsolvable. There is no possible algorithm that can meet this specification! Notice, I'm not just saying that no one has been able to do this before; I'm saying that this problem can never be solved, in principle.

How do I know that there is no solution to this problem? This is a question that all the design skills in the world will not answer. Design can show that problems are solvable, but it can never prove that a problem is not solvable. To do that, we need to use our analytical skills.

One way to prove that something is impossible is to first assume that it is possible and show that this leads to a contradiction. Mathematicians call this proof by contradiction. We'll use this technique to show that the halting problem cannot be solved.

We begin by assuming that there is some algorithm that can determine if any program terminates when executed on a particular input. If such an algorithm could be written, we could package it up in a function.

```
def terminates(program, inputData):
    # program and inputData are both strings
    # Returns true if program would halt when run with inputData
    #   as its input.
```

Of course, I can't actually write the function, but let's just assume that this function exists.

Using the `terminates` function, we can write an interesting program.

```
# turing.py
import string

def terminates(program, inputData):
    # program and inputData are both strings
    # Returns true if program would halt when run with inputData
    #   as its input.

def main():
    # Read a program from standard input
    lines = []
    print "Type in a program (type 'done' to quit)."
    line = raw_input("")
    while line != "done":
        lines.append(line)
        line = raw_input("")
    testProg = string.join(lines, "\n")

    # If program halts on itself as input, go into an infinite loop
    if terminates(testProg, testProg):
        while True:
            pass    # a pass statement does nothing

main()
```

I have called this program `turing` in honor of Alan Turing, the British mathematician considered by many to be the “Father of Computer Science.” He was the one who first proved that the halting problem could not be solved.

The first thing `turing.py` does is read in a program typed by the user. This is accomplished with a sentinel loop that accumulates lines in a list one at a time. The `string.join` function then concatenates the lines together using a newline character ("`\n`") between them. This effectively creates a multi-line string representing the program that was typed.

`Turing.py` then calls the `terminates` function and sends the input program as both the program to test and the input data for the program. Essentially, this is a test to see if the program read from the input terminates when given itself as input. The `pass` statement actually does nothing; if the `terminates` function returns `true`, `turing.py` will go into an infinite loop.

OK, this seems like a silly program, but there is nothing in principle that keeps us from writing it, provided that the `terminates` function exists. `Turing.py` is constructed in this peculiar way simply to illustrate a point. Here's the million dollar question: What happens if we run `turing.py` and, when prompted to type in a program, type in the contents of `turing.py` itself? Put more specifically, does `turing.py` halt when given itself as its input?

Let's think it through. We are running `turing.py` and providing `turing.py` as its input. In the call to `terminates`, both the program and the data will be a copy of `turing.py`, so if `turing.py` halts when given itself as input, `terminates` will return `true`. But if `terminates` returns `true`, `turing.py` then goes into an infinite loop, so it *doesn't* halt! That's a contradiction; `turing.py` can't both halt and not halt. It's got to be one or the other.

Let's try it the other way around. Suppose that `terminates` returns a false value. That means that `turing.py`, when given itself as input goes into an infinite loop. But as soon as `terminates` returns `false`, `turing.py` quits, so it does halt! It's still a contradiction.

If you've gotten your head around the previous two paragraphs, you should be convinced that `turing.py` represents an impossible program. The existence of a function meeting the specification for `terminates` leads to a logical impossibility. Therefore, we can safely conclude that no such function exists. That means that there cannot be an algorithm for solving the halting problem.

There you have it. Your boss has assigned you an impossible task. Fortunately, your knowledge of computer science is sufficient to recognize this. You can explain to your boss why the problem can't be solved and then move on to more productive pursuits.

13.4.3 Conclusion

I hope this chapter has given you a taste of what computer science is all about. As the examples in this chapter have shown, computer science is much more than “just” programming. The most important computer for any computing professional is still the one between the ears.

Hopefully this book has helped you along the road to becoming a computer programmer. Along the way, I have tried to pique your curiosity about the science of computing. If you have mastered the concepts in this text, you can already write interesting and useful programs. You should also have a firm foundation of the fundamental ideas of computer science and software engineering. Should you be interested in studying these fields in more depth, I can only say “go for it.” Perhaps one day you will also consider yourself a computer scientist; I would be delighted if my book played even a very small part in that process.

13.5 Chapter Summary

This chapter has introduced you to a number of important concepts in computer science that go beyond just programming. Here are the key ideas:

- One core subfield of computer science is analysis of algorithms. Computer scientists analyze the time efficiency of an algorithm by considering how many steps the algorithm requires as a function of the input size.
- Searching is the process of finding a particular item among a collection. Linear search scans the collection from start to end and requires time linearly proportional to the size of the collection. If the collection is sorted, it can be searched using the binary search algorithm. Binary search only requires time proportional to the log of the collection size.
- Binary search is an example of a divide and conquer approach to algorithm development. Divide and conquer often yields efficient solutions.
- A definition or function is recursive if it refers to itself. To be well-founded, a recursive definition must meet two properties:
 1. There must be one or more base cases that require no recursion.
 2. All chains of recursion must eventually reach a base case.

A simple way to guarantee these conditions is for recursive calls to always be made on smaller versions of the problem. The base cases are then simple versions that can be solved directly.

- Sequences can be considered recursive structures containing a first item followed by a sequence. Recursive functions can be written following this approach.
- Recursion is more general than iteration. Choosing between recursion and looping involves the considerations of efficiency and elegance.
- Sorting is the process of placing a collection in order. A selection sort requires time proportional to the square of the size of the collection. Merge sort is a divide and conquer algorithm that can sort a collection in $n \log n$ time.
- Problems that are solvable in theory but not in practice are called intractable. The solution to the famous Towers of Hanoi can be expressed as a simple recursive algorithm, but the algorithm is intractable.
- Some problems are in principle unsolvable. The Halting problem is one example of an unsolvable problem.
- You should consider becoming a computer scientist.

13.6 Exercises

Review Questions

True/False

1. Linear search requires a number of steps proportional to the size of the list being searched.
2. The Python operator `in` performs a binary search.
3. Binary search is an $n \log n$ algorithm.
4. The number of times n can be divided by 2 is $exp(n)$.
5. All proper recursive definitions must have exactly one non-recursive base case.

6. A sequence can be viewed as a recursive data collection.
7. A word of length n has $n!$ anagrams.
8. Loops are more general than recursion.
9. Merge sort is an example of an $n \log n$ algorithm.
10. Exponential algorithms are generally considered intractable.

Multiple Choice

1. Which algorithm requires time directly proportional to the size of the input?
a) linear search b) binary search
c) merge sort d) selection sort
2. Approximately how many iterations will binary search need to find a value in a list of 512 items?
a) 512 b) 256 c) 9 d) 3
3. Recursions on sequences often use this as a base case:
a) 0 b) 1 c) an empty sequence d) None
4. An infinite recursion will result in
a) a program that “hangs”
b) a broken computer
c) a reboot
d) a run-time exception
5. The recursive Fibonacci function is inefficient because
a) it does many repeated computations
b) recursion is inherently inefficient compared to iteration
c) calculating Fibonacci numbers is intractable
d) fibbing is morally wrong
6. Which is a quadratic time algorithm?
a) linear search b) binary search
c) tower of Hanoi d) selection sort
7. The process of combining two sorted sequences is called
a) sorting b) shuffling c) dovetailing d) merging

8. Recursion is related to the mathematical technique called
 - a) looping
 - b) sequencing
 - c) induction
 - d) contradiction
9. How many steps would be needed to solve the Towers of Hanoi for a tower of size 5?
 - a) 5
 - b) 10
 - c) 25
 - d) 31
10. Which of the following is *not* true of the Halting Problem?
 - a) It was studied by Alan Turing
 - b) It is harder than intractable
 - c) Someday a clever algorithm may be found to solve it
 - d) It involves a program that analyzes other programs

Discussion

1. Place these algorithm classes in order from fastest to slowest: $n \log n$, n , n^2 , $\log n$, 2^n
2. In your own words, explain the two rules that a proper recursive definition or function must follow.
3. What is the exact result of `anagram("foo")`?
4. Trace `recPower(3,6)` and figure out exactly how many multiplications it performs.
5. Why are divide-and-conquer algorithms often very efficient?

Programming Exercises

1. Modify the recursive Fibonacci program given in the chapter so that it prints tracing information. Specifically, have the function print a message when it is called and when it returns. For example, the output should contain lines like these:

```
Computing fib(4)
...
Leaving fib(4) returning 3
```

Use your modified version of `fib` to compute `fib(10)` and count how many times `fib(3)` is computed in the process.

2. This exercise is another variation on “instrumenting” the recursive Fibonacci program to better understand its behavior. Write a program that counts how many times the `fib` function is called to compute `fib(n)` where `n` is a user input.

Hint: To solve this problem, you need an accumulator variable whose value “persists” between calls to `fib`. You can do this by making the count an instance variable of an object. Create a `FibCounter` class with the following methods:

`__init__(self)` Creates a new `FibCounter` setting its count instance variable to 0.

`getCount(self)` Returns the value of count.

`fib(self, n)` Recursive function to compute the `n`th Fibonacci number. It increments the count each time it is called.

`resetCount(self)` Set the count back to 0

3. A palindrome is a sentence that contains the same sequence of letters reading it either forwards or backwards. A classic example is: “Able was I, ere I saw Elba.” Write a recursive function that detects whether a string is a palindrome. The basic idea is to check that the first and last letters of the string are the same letter; if they are, then the entire string is a palindrome if everything between those letters is a palindrome. There are a couple of special cases to check for. If either the first or last character of the string is not a letter, you can check to see if the rest of the string is a palindrome with that character removed. Also, when you compare letters, make sure that you do it in a case-insensitive way.

Use your function in a program that prompts a user for a phrase and then tells whether or not it is a palindrome. Here’s another classic for testing: “A man, a plan, a canal, Panama!”

4. Write and test a recursive function `max` to find the largest number in a list. The `max` is the larger of the first item and the `max` of all the other items.
5. Computer scientists and mathematicians often use numbering systems other than base 10. Write a program that allows a user to enter a number and a base and then prints out the digits of the number in the new base. Use a recursive function `baseConversion(num, base)` to print the digits.

Hint: Consider base 10. To get the rightmost digit of a base 10 number, simply look at the remainder after dividing by 10. For example, `153%10` is

3. To get the remaining digits, you repeat the process on 15, which is just $153/10$. This same process works for any base. The only problem is that we get the digits in reverse order (right to left).

Write a recursive function that first prints the digits of $num/base$ and then prints the last digit, namely $num \% base$. You should put a space between successive digits, since bases greater than 10 will print out with multi-character digits. For example, `baseConversion(245, 16)` should print 15 5.

6. Write a recursive function to print out the digits of a number in English. For example, if the number is 153, the output should be “One Five Three.” See the hint from the previous problem for help on how this might be done.
7. In mathematics, C_k^n denotes the number of different ways that k things can be selected from among n different choices. For example, if you are choosing among six desserts and are allowed to take two, the number of different combinations you could choose is C_2^6 . Here’s one formula to compute this value:

$$C_k^n = \frac{n!}{k!(n-k)!}$$

This value also gives rise to an interesting recursion:

$$C_k^n = C_{k-1}^{n-1} + C_k^{n-1}$$

Write both an iterative and a recursive function to compute combinations and compare the efficiency of your two solutions. Hints: when $k = 1$, $C_k^n = n$ and when $n < k$, $C_k^n = 0$.

8. Some interesting geometric curves can be described recursively. One famous example is the Koch curve. It is a curve that can be infinitely long in a finite amount of space. It can also be used to generate pretty pictures.

The Koch curve is described in terms of “levels” or “degrees.” The Koch curve of degree 0 is just a straight line segment. A first degree curve is formed by placing a “bump” in the middle of the line segment (see Figure 13.6). The original segment has been divided into four, each of which is $1/3$ the length of the original. The bump rises at 60 degrees, so it forms two sides of an equilateral triangle. To get a second degree curve, you put a bump in each of the line segments of the first degree curve.

Successive curves are constructed by placing bumps on each segment of the previous curve.

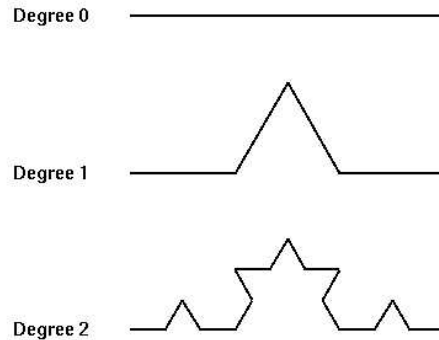


Figure 13.6: Koch curves of degree 0 to 2

You can draw interesting pictures by “Kochizing” the sides of a polygon. Figure 13.7 shows the result of applying a fourth degree curve to the sides of an equilateral triangle. This is often called a “Koch snowflake.” You are to write a program to draw a snowflake.

Hints: Think of drawing a Koch curve as if you were giving instructions to a turtle. The turtle always knows where it currently sits and what direction it is facing. To draw a Koch curve of a given length and degree, you might use an algorithm like this:

```
Algorithm Koch(Turtle, length, degree):
  if degree == 0:
    Tell the turtle to draw for length steps
  else:
    length1 = length/3
    degree1 = degree-1
    Koch(Turtle, length1, degree1)
    Tell the turtle to turn left 60 degrees
    Koch(Turtle, length1, degree1)
    Tell the turtle to turn right 120 degrees
    Koch(Turtle, length1, degree1)
```

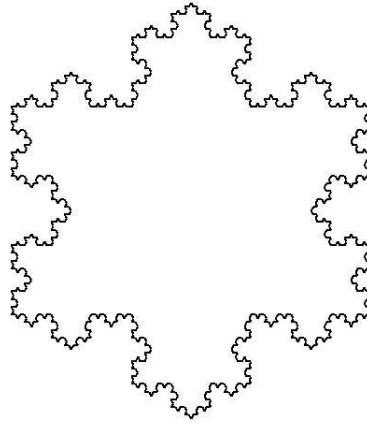


Figure 13.7: Koch snowflake

```
Tell the turtle to turn left 60 degrees
Koch(Turtle, length1, degree1)
```

Implement this algorithm with a `Turtle` class that contains instance variables `location` (a `Point`) and `Direction` (a float) and methods such as `moveTo(somePoint)`, `draw(length)`, and `turn(degrees)`. If you maintain `direction` as an angle in radians, the point you are going to can easily be computed from your current location. Just use $dx = length * \cos(direction)$ and $dy = length * \sin(direction)$.

9. Another interesting recursive curve (see previous problem) is the C-curve. It is formed similarly to the Koch curve except whereas the Koch curve breaks a segment into four pieces of $length/3$, the C-curve replaces each segment with just two segments of $length/\sqrt{2}$ that form a 90 degree elbow. Figure 13.8 shows a degree 12 C-curve.

Using an approach similar to the previous exercise, write a program that draws a C-curve. Hint: your turtle will do the following:

```
turn left 45 degrees
draw a c-curve of size length/sqrt(2)
```

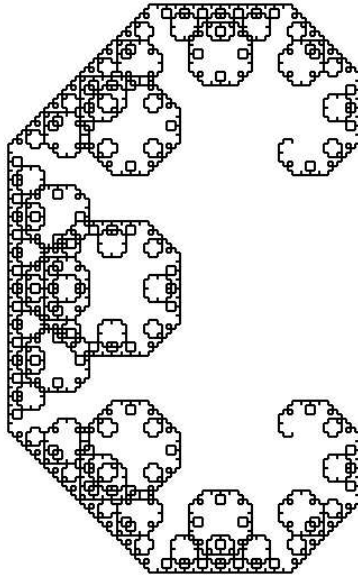


Figure 13.8: C-curve of degree 12

```
turn right 90 degrees
draw a c-curve of size length/sqrt(2)
turn left 45 degrees
```

10. Automated spell checkers are used to analyze documents and locate words that might be misspelled. These programs work by comparing each word in the document to a large dictionary (in the non-Python sense) of words. If the word is not found in the dictionary, it is flagged as potentially incorrect.

Write a program to perform spell-checking on a text file. To do this, you will need to get a large file of English words in alphabetical order. If you have a Unix or Linux system available, you might poke around for a file called `words`, usually located in `/usr/dict` or `/usr/share/dict`. Otherwise, a quick search on the Internet should turn up something usable.

Your program should prompt for a file to analyze and then try to look up every word in the file using binary search. If a word is not found in the dictionary, print it on the screen as potentially incorrect.

11. Write a program that solves word jumble problems. You will need a large dictionary of English words (see previous problem). The user types in a scrambled word, and your program generates all anagrams of the word and then checks which (if any) are in the dictionary. The anagrams appearing in the dictionary are printed as solutions to the puzzle.