Chapter 11

Recursion

Recursion is a powerful concept that is important in computer science for two reasons. First, many seemingly complex problems can be solved simply if we use recursion. Second, certain types of information can be represented easily using recursively defined data structures. Recursion is, essentially, a way of defining something in terms of itself. A recursive solution to a problem is obtained through solving simpler versions of the original problem while a recursive description of a structure is obtained through the description of simpler versions of the original structure. In this chapter, we begin by examining situations from everyday life that illustrate recursion and then apply recursive solutions to several types of problems that can be solved using a computer.

11.1 Everyday Recursion

We begin our study of recursion by examining situations from everyday life that illustrate recursion.

If you wanted to draw a family tree showing an individual and all of that person's direct descendants, you would probably start by writing down the name of the person. Under his or her name, you could then draw lines to the name of each child of the person. If any of these children had children, you could then draw lines to the names of their children. This process could be continued, drawing lines to the names of the children's children, and so on until you reached a point (for each descendant) at which there were no further descendants.

We can describe this process easily and concisely if we use recursion.

Example 1

To draw a family tree showing a person and all of the person's direct descendants, we can use the following algorithm.

```
To DRAW THE FAMILY TREE of a person
  write the name of the person
  if the person had children
    for each child
      draw a line from the person and
      DRAW THE FAMILY TREE of the child
```

The example illustrates two features typical of recursive algorithms.

1. The simplest case of the problem can be solved directly. In the example, a childless person is the simplest case; after writing the name of such a person, that person's family tree is complete.
2. All other cases of the problem are solved by doing a bit of work in the direction of a solution and then solving one or more sub-problems of the original problem. In the example, after writing the name of a person with one or more children, we must draw the family tree of each of these children.

This type of analysis can be applied to many activities. We illustrate a fairly silly one in the next example.

Example 2

Let us consider the structure of an onion without its brown papery covering. Suppose that we want to describe the process of finding the centre of the onion. If we peel away the outermost thick, fleshy layer, we are either at the core or we are left with what looks like a smaller version of the original onion. Thus, finding the centre of an onion could be described by the following recursive procedure.

```
To FIND THE CENTRE OF THE ONION
  if you are not at the core
    remove one layer
    FIND THE CENTRE OF THE ONION
```

The simplest case for finding the centre of the onion is finding that we are at the core. In the more complex case, we consider an onion as being composed of a fleshy layer that surrounds a smaller onion or the core. By removing one layer, we get a little closer to the simple case.

In any recursive algorithm, it is crucial that we have a way of stopping the process. This is often called a *terminating condition* of a recursive algorithm. In Example 1, the terminating condition was the case in which a person had no children. In Example 2, the terminating condition was the arrival at the core of the onion.

Exercises 11.1

1. The following is a recursive definition of the process of climbing a set of stairs.

```
To CLIMB THE STAIRS
  if you are at the top
    you are done
  else
    take one step and
    CLIMB THE STAIRS
```

   (a) Identify the simple case in CLIMB THE STAIRS.

   (b) Identify the complex case in CLIMB THE STAIRS.

2. Why is the terminating condition so important in a recursive algorithm?

3. An old children's rhyme begins: "There once was a man who had a hollow tooth. In the tooth was a treasure chest. In the treasure chest was a piece of paper. On the piece of paper was written 'There once was a man who had a hollow tooth ...' ".

   Is this an example of recursion? Justify your answer.

4. Which of the following can be said to be examples of recursion?

   (a) closing a zipper

   (b) nested Russian wooden dolls

   (c) a reflection of a mirror in a mirror

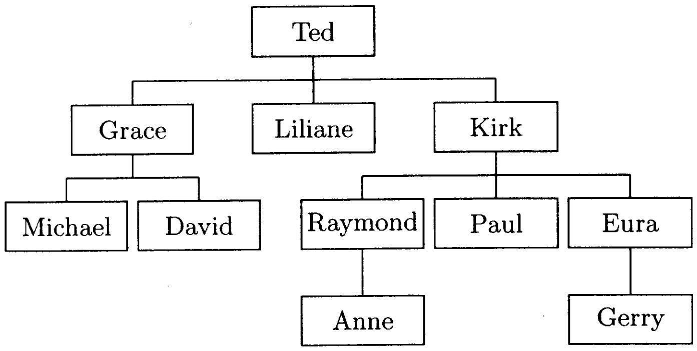
   (d) the song that begins "There's a hole in my bucket ..."

5. Rewrite the definition of CLIMB THE STAIRS in Question 1 using a `while` statement.

6. Give a recursive description of the process of reading a book.

7. In the text, we gave recursive algorithms for the tasks of drawing a family tree and finding the centre of an onion. Find another task that can be defined recursively and give the recursive algorithm for it.

8. The diagram shows a family tree. If we were to draw this tree using the recursive algorithm of Example 1, in what order would the members of the family be inserted in the tree? (Assume that the children of any individual are added from left to right. For example, Michael would be added before David.)



11.2 Recursion in Mathematics

Suppose that we are given the following sequence of integers:

$$3, 6, 12, 24, \ldots$$

in which any term is twice as large as the preceding one. If we call the terms $t_1, t_2, t_3, \ldots$ then we can express the relationship concisely as follows:

$$\begin{aligned} t_1 &= 3 \\ t_n &= 2 \times t_{n-1}, \text{ if } n > 1 \end{aligned}$$

This definition of the terms of the sequence satisfies the conditions that we set out for any recursive definition:

1. An object is defined in terms of another object of the same type. Here a term is defined in terms of the preceding term.

2. There is some way of stopping the recursion. This is done here by explicitly defining the first term, t_1, as having the value 3.

Example 1

Given the sequence defined recursively by the equations

$$\begin{aligned} t_1 &= 2 \\ t_n &= 3 \times t_{n-1} + 1, \text{ if } n > 1 \end{aligned}$$

find the value of the fifth term, t_5.
From the definition, we know that

$$\begin{aligned} & t_5 &= 3 \times t_4 + 1 \\ \text{and} \quad & t_4 &= 3 \times t_3 + 1 \\ \text{and} \quad & t_3 &= 3 \times t_2 + 1 \\ \text{and} \quad & t_2 &= 3 \times t_1 + 1 \\ \text{and} \quad & t_1 &= 2 \end{aligned}$$

Having found the value of t_1, we can now substitute back into the expressions for the other terms.

$$\begin{aligned} & t_2 = 3 \times 2 + 1 &= 7 \\ \text{and} \quad & t_3 = 3 \times 7 + 1 &= 22 \\ \text{and} \quad & t_4 = 3 \times 22 + 1 &= 67 \\ \text{and} \quad & t_5 = 3 \times 67 + 1 &= 202 \end{aligned}$$

The value of the fifth term is 202.

One of the most famous algorithms known was stated over two thousand years ago by the Greek mathematician Euclid. *Euclid's algorithm*, as it is called, provides a method of finding the greatest common divisor (gcd) of a pair of natural numbers. The algorithm is based on the following properties of a gcd:

Rule 1. The gcd of two equal values, m and n, is simply m (or n).

Rule 2. If two numbers are not equal, then the gcd of the two values will also divide their difference and will, in fact, be the gcd of this difference. Thus, if $m > n$, then the gcd of m and n will be equal to the gcd of n and $m - n$.

Rule 3. If the numbers are not equal and the first number is smaller than the second, then we can simply switch the numbers and apply Rule 2.

These three rules can be stated more concisely (and possibly more clearly) in symbolic form:

Rule 1. If $m = n$, then $\gcd(m, n) = m$

Rule 2. If $m > n$, then $\gcd(m, n) = \gcd(n, m - n)$

Rule 3. If $m < n$, then $\gcd(m, n) = \gcd(n, m)$

Example 2

Use Euclid's algorithm to find the greatest common divisor of 54 and 90.

$$\begin{aligned}
\gcd(54, 90) &= \gcd(90, 54) && \text{by Rule 3} \\
&= \gcd(54, 36) && \text{by Rule 2} \\
&= \gcd(36, 18) && \text{by Rule 2} \\
&= \gcd(18, 18) && \text{by Rule 2} \\
&= 18 && \text{by Rule 1}
\end{aligned}$$

Exercises 11.2

1. Write the first five terms of each sequence (correct to two decimal places).

   (a) $t_1 = 3$
   $t_n = t_{n-1} + 2,\ n > 1$

   (b) $t_1 = 2$
   $t_{n+1} = 1 - \frac{1}{t_n},\ n > 0$

   (c) $t_1 = 2$
   $t_{n+1} = n + t_n,\ n > 0$

   (d) $t_1 = 1$
   $t_2 = 3$
   $t_n = t_{n-1} + t_{n-2},\ n > 2$

   (e) $t_n = \begin{cases} 1 & \text{if } n = 1 \\ \sqrt{1 + t_{n-1}} & \text{if } n > 1 \end{cases}$

   (f) $t_n = \begin{cases} 2 & \text{if } n = 1 \\ \sqrt{3t_{n-1} + 4} & \text{if } n > 1 \end{cases}$

2. Write a possible recursive definition of each sequence.

(a) $10, 13, 16, 19, \ldots$ (b) $\frac{1}{2}, \frac{3}{2}, \frac{5}{2}, \ldots$

(c) $5, 15, 45, 135, \ldots$ (d) $-1, -3, -5, \ldots$

(e) $256, 64, 16, 4, \ldots$ (f) $1.1, 1.2, 1.3, 1.4, \ldots$

3. Use Euclid's algorithm to find the gcd of each pair of numbers.

   (a) 18 and 24 (b) 84 and 144

   (c) 35 and 16 (d) 515 and 206

4. A frog, sitting at one end of a two-metre log, jumps toward the other end but can only make it half way. The frog continues jumping toward the end but each time becomes more exhausted and only jumps half the length of the preceding jump.

   (a) State the lengths of the first four jumps.

   (b) State an expression for the length in metres of the n^{th} jump, j_n, in terms of the $(n-1)^{\text{th}}$ jump, j_{n-1}.

   (c) Use the recursive relationship established in part b) to find the distance jumped in the sixth jump.

   (d) Give a non-recursive expression for j_n, the length in metres of the n^{th} jump.

5. (a) If we were to use Euclid's algorithm, as given in the text, to find the gcd of 44 and 6, how many subtractions would be required to reduce the problem to that of finding the gcd of 6 and 2?

   (b) What operation would achieve the same effect as this sequence of subtractions?

   (c) Suggest a modification of the algorithm that avoids this sequence of subtractions.

   (d) Use the modified algorithm to find the gcd of each pair of numbers in Question 3.

6. Prove that Euclid's algorithm, as stated in the text, correctly finds the gcd of two positive integers, m and n.

11.3 Recursive Queries

Suppose that we are given the following recursive definition of a sequence

$$\begin{aligned} t_1 &= 2 \\ t_n &= 3t_{n-1} - 1, \; n > 1 \end{aligned}$$

We can express such a sequence using function notation if we write

$$\begin{aligned} t(1) &= 2 \\ t(n) &= 3t(n-1) - 1, \; n > 1 \end{aligned}$$

To implement such a function with a Java method is simple. (The tricky part is understanding how the method operates.)

Example 1

The following method returns the value of the function defined by

$$\begin{aligned} t(1) &= 2 \\ t(n) &= 3t(n-1) - 1, \; n > 1 \end{aligned}$$

If the method is given an invalid value of the parameter, n, it throws an exception.

```
public static int term (int n)
{
  if (n < 1)
    throw new RuntimeException("Invalid parameter");
  else if (n == 1)
    return 2;
  else
    return 3 * term(n-1) - 1;
}
```

At first glance, the method may not appear to do anything; it seems to be nothing more than a definition of the sequence. To see how it works, suppose that **term** is called with **n** = 4. Since **n** > 1, the method executes the statement

```
return 3 * term(3) - 1;
```

The right side of this assignment statement involves a call to the method **term**. We have seen methods call other methods before. In such a case, execution of the first method is suspended until the called method returns the value it has been asked to compute. The same thing happens here; the only difference is that here both the calling method and the called method are the same. To visualize the process, we can think of a new copy of **term** being created by the call. This copy is identical to the original but it has its own copy of the parameter **n**. This version of the parameter **n** has the value 3. If the method had any local variables, there would also be separate copies of them in both the original and the called versions of the method.

We can illustrate this process using a diagram.

```
original call
  term (n = 4)
    executes
      return 3 * (term(3)) - 1;
          calls       |
                      v
            term (n = 3)
```

This copy of **term** (with **n** = 3) now executes the statement

```
term = 3 * term(2) - 1;
```

Before it can complete execution of this statement, it must call another copy of **term**, this time with **n** = 2. The process continues until a copy is called with **n** = 1, as shown in the next diagram.

```
original call
  term (n = 4)
    executes
      return 3 * term(3) - 1;
          calls
            term (n = 3)
              executes
                return 3 * term(2) - 1;
                    calls
                      term (n = 2)
                        executes
                          return 3 * term(1) - 1;
                              calls
                                term (n = 1)
                                  executes
                                    return 2;
```

The copy of `term` with `n = 1` now returns the value 2 to the point from which it was called and then ceases to exist. Knowing the value of `term(1)`, the value of the expression `3 * term(1) - 1` can now be completed; it gives $3 \times 2 - 1 = 5$. This value can then be returned to the point at which this version of `term` was called. This process continues until the original call computes and returns the desired value. Once again, we can illustrate the process with a diagram.

```
term (n = 1) returns 2

to term (n = 2)
  which returns 3 × 2 − 1 = 5

to term (n = 3)
  which returns 3 × 5 − 1 = 14

to term (n = 4)
  which returns 3 × 14 − 1 = 41

to the point from which it was called
```

A silly but possibly useful way of looking at this process is to imagine yourself trying to find the value of $t(4)$. Rather than doing all the calculations yourself, you phone your friend Alex. Alex knows the formula for $t(n)$, so he can see that

$t(4) = 3 \times t(3) - 1.$

Unfortunately, he doesn't know $t(3)$ so he puts you on hold while he calls his friend Kim. Kim also knows the formula so she can quickly determine that

$t(3) = 3 \times t(2) - 1.$

To help her complete her calculations, Kim must have the value of $t(2)$. To get this, she puts Alex on hold while she phones her friend Chris. Chris also knows the formula so he can see that the answer to Kim's problem is

$t(2) = 3 \times t(1) - 1$

Like the other callers before him, Chris doesn't want to do all the work so he puts Kim on hold while he phones his friend Wei Ting to find $t(1)$. Wei Ting can see from the formula that

$t(1) = 2$

Wei Ting tells Chris that the answer is 2 and then hangs up (wondering why Chris would phone to ask her such a question).
Chris can now complete his calculations:

$t(2) = 3 \times 2 - 1 = 5$

Chris passes this information back to Kim and then he too hangs up.
Kim can now determine that:

$t(3) = 3 \times 5 - 1 = 14$

Kim tells Alex her result and then she also hangs up.
Alex can now complete his calculation:

$t(4) = 3 \times 14 - 1 = 41$

He passes this back to you and hangs up, leaving you to do what you want with the knowledge that

$t(4) = 41$

If all of this calling and returning seems to you to be a waste of time, you are right! Recursion is a very useful technique but it does *not* provide the most efficient way of evaluating a function like the one that we have here. We have shown this example only because it illustrates some aspects of recursion — not because we recommend that you use it in such circumstances.

Example 2

An efficient method for calculating the value of the function of the previous example would use a loop. Here is an efficient, non-recursive method that does the job.

```
public static int term (int n)
{
  if (n < 1)
    throw new RuntimeException("Invalid parameter");
  else
  {
    int value = 2;
    for (int i = 2; i <= n; i++)
      value = 3 * value - 1;
    return value;
  }
}
```

Recursive methods in Java are not limited to evaluating terms of sequences. They can be used to evaluate any function for which we have a recursive definition.

Example 3

A more efficient form of Euclid's algorithm than the one given in the previous section replaces the successive subtractions by one mod operation (implemented by the % operator in Java). The revised algorithm for determining the gcd of two non-negative integers is

$$\gcd(m,n) = \begin{cases} m & \text{if } n = 0 \\ \gcd(n, m \bmod n) & \text{if } n > 0 \end{cases}$$

The algorithm can be implemented easily in Java, as shown in the following method. (The method assumes that the values of both parameters are non-negative integers.)

```
public static int gcd (int m, int n)
{
  if (n == 0)
    return m;
  else
    return gcd(n, m % n);
}
```

If we make a call to gcd(24,18), we produce the following sequence of actions:

```
original call
  gcd (m = 24, n = 18)
    executes
      return [gcd(18,24 % 18)];
                    |
          calls     ↓
            gcd (m = 18, n = 6)
              executes
                return [gcd(6,18 % 6)];
                             |
                    calls    ↓
                      gcd (m = 6, n = 0)
                        executes
                          return 6;
```

Now, each copy of gcd returns, as shown in the next diagram.

```
gcd (m = 6, n = 0)
  returns 6
    to gcd (m = 18, n = 6)
      which returns 6
        to gcd (m = 24, n = 18)
          which returns 6
            to the point from which it was called
```

Exercises 11.3

1. Trace the execution of the method gcd in Example 2 to find the greatest common divisor of each pair of values.

   (a) m = 20   n = 28   (b) m = 991   n = 129

2. Trace the execution of the method `term` shown below, given an initial parameter value of 3.

```
public static double term (int n)
{
  if (n < 0)
  {
    System.out.println("Invalid argument to method"
                       + " term\nzero returned");
    return 0;
  }
  else if (n == 0)
    return 1;
  else
    return 2 + 0.5 * term(n-1);
}
```

3. What happens if the function `term` in Example 1 is called with a parameter whose value is zero?

4. What is wrong with these methods?

   (a)
```
public static double bad (double a, double b)
{
  a = a/2;
  b = b*2;
  return bad(a,b);
}
```

   (b)
```
public static int badToo (int n)
{
  if (n < 1)
    return 0;
  else if (n == 1)
    return 5;
  else return 2*badToo(n+1) + 3;
}
```

5. The *factorial* of a non-negative integer n (written as $n!$), is a function that is useful in many branches of mathematics. It can be defined recursively as follows:

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

   (a) Write a recursive method `factorial` that returns the value of $n!$ as a `long` value. (A `long` can represent $n!$ for values of n up to 20.) If an invalid parameter value is given to the method, it should print an error message and return zero.

   (b) Write a non-recursive version of `factorial`.

   (c) Which version do you think would be more efficient? Justify your answer.

6. A function f is defined for integers x and y as follows:

$$f(x,y) = \begin{cases} -f(y,x) & \text{if } x < y \\ 0 & \text{if } x = y \\ 1 + f(x-1,y) & \text{if } x > y \end{cases}$$

   (a) Use this definition to evaluate
   i) $f(5,3)$ ii) $f(2,2)$ iii) $f(1,4)$ iv) $f(-5,-2)$

   (b) State, in a few words, the effect of the function f if it has integer arguments x and y.

7. A function that is important in the study of the theory of computation (but quite useless otherwise) is known as *Ackermann's function*. It can be defined for non-negative integers m and n as follows:

$$a(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ a(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ a(m-1, a(m, n-1)) & \text{otherwise} \end{cases}$$

   (a) Evaluate $a(1,1)$ and $a(2,1)$ by hand.

   (b) Write a Java implementation of Ackermann's function.

   (c) Try to use your implementation to evaluate $a(5,5)$. Account for the result.