Chapter 7 Introduction to recursion

In computing, a recursive process is one that is 'described in terms of itself'. Recursion is viewed with suspicion beginners - how on earth cab an operation be defined in terms of itself? One of our students recently remarked that writing a recursive program seemed like an act of faith. However, once mastered, recursion is a powerful tool should not be neglected. There are many programming problems where a recursive solution is elegant and easy to write and the non-recursive solution is difficult and tricky. Many human problem-solving activities are recursive in nature. For example, let us consider the problen of planning a route for walking through London from Trafalgar Square to the British Museum. One way of solving this problem might be to pick an intermediate point such as Covent Garden and break our original problem down into the problean of getting from Trafalgar Square to Covent Garden and from Covent Garden to the British Museum. A problem of navigation has been broken down into two easier subproblems, also of navigation.

This is the essence of recursion. The solution to a problem is described in terms of solutions to easier or smaller versions of the same problem. We could (rather fancifully) describe how to find a route between two points as:

```
100  DEF PROCfind_route_between(a, b)
110    IF getting from a to b is 'easy' (one street say)
       THEN  PROCprint_route(a, b) : ENDPROC
120    n = a point midway between a and b
130    PROCfind_route_between(a, n)
140    PROCfind_route_between(n, b)
150  ENDPROC
```

We shall not expand this into a complete BASIC program. In order to do so we would need to store a street map of London, lists of landmarks and their locations, a definition at what we mean by an 'easy' problem: and so on. However, this outline procedure describes a process with which we are all subconsciously familiar. It also exhibits the essential features of a recursive procedure.

When a procedure is called, the particular problem to be specified by means of its parameters:

```
PROCfind_route_between("Trntalgar Square", "British Museum")
```

The first thing the procedure does is to decide whether the problem represented by its parameters can be solved directly without breaking it down into further subproblems. If this can be done, no recursion takes place. This is essential, otherwise the process of breaking the problem down into subproblems would never stop.

Finally if the problem to be solved by the call of the procedure is not an easy one, it breaks it down into easier subproblems and requests the solution to each of these subproblems in turn. The solutions to the subproblems are requested by calling the same procedure, but with different parameters. You might find it easier to think of the subproblems being solved by different copies of the procedure, although it does not happen like this behind the scenes. This is the classic 'divide and conquer' approach to problem solving, so important in areas like Artificial Intelligence.

Learning to use recursion successfully means learning to recognise when a problem can be broken down into easier, or smaller, versions of itself and remembering to start a recursive procedure with a test that recognises when a given problem does not need to be further broken down. It is usually easier to write a recursive procedure without worrying in detail about what the exact sequence of operations will be when the procedure is called (an 'act of faith' if you like). Just remember the two basic ingredients - the stopping condition and the breakdown into easier subproblems.

It is of course interesting to understand what does happen when we call a recursive procedure. In fact, when a program does not work as intended, such an understanding is essential. Later, we shall explain in detail how recursive programs work, but first let us write some simple programs that use recursion.

7.1 Some easy recursive programs

Many of the programs presented in this section could very easily be written without recursion using simple loops. However, such 'inappropriate' use of recursion provides a useful introduction to the subject using problems with which we are familiar.

The first example simply prints the positive integers from 1 to n using a procedure that can be called by:

```
10  INPUT n
20  PROCprintupto(n)
30  END
```

We shall break down the process of printing the numbers up to a into the problem of printing the numbers up to n-1 followed by the use of a PRINT statement to print n. Of course if n = 0, then there are no values to be printed and this is the condition that we shall use to terminate the recursion.

```
100  DEF PROCprintupto(n)
110    IF n =0 THEN ENDPROC
120    PROCprintupto(n-1)
130    PRINT n
140  ENDPROC
```

In the next section we shall discuss in detail what happens when this program is obeyed. For the time being we shall take on trust the fact that a recursive program works!

An interesting variation on this program is to change it so that it prints the integers up to n, but in reverse order. In this case, the breakdown into an easier subproblem gives

```
PRINT n
print numbers up to n-1 in reverse order.
```

The only change that needs to be made to the previous program is to switch lines 120 and 130.

```
100  DEF PROCprintupto(n)
110    IF n=0 THEN ENDPROC
120    PRINT n
130    PROCprintupto(n-1)
140  ENDPROC
```

The above two programs are examples of what is sometimes called 'unary recursion' - a problem is broken down into one easier version of itself together with straightforward operations such as PRINT.

A simple example of 'binary recursion', where a problem is broken down into two simpler versions of itself, is provided by an alternative approach to printing the first a integers. We can define a procedure that prints the integers in a given range. For example,

```
PROCprintbetween(3,7)
```

will print the integers 3, 4, 5, 6, 7.

```
PROCprintbetween(4,4)
```

will print the single integer 4. This procodure could be used to print the positive integers up to n by calling

```
PROCprintbetween(1,n)
```

PROCprintbetween can be defined using binary recursion if we break down the problem of printing a given sequence into:

print the first half of the sequence
print the second half of the sequence

If only one value is to be printed, then this breakdown will not be needed.

```
10   INPUT max
20   PROCprintupto(max)
30   END

100   DEF PROCprintupto(n)
110   PROCprintbetween(1, n)
120   ENDPROC

130   DEFPROCprintbetween(i, j)
140   LOCAL mid
150   IF i=j THEN PRINT i : ENDPROC
160   mid = (i+j) DIV 2
170   PROCprintbetween(i, laid)
180   PROCprintbetween(mid+1, j)
190   ENDPROC
```

Again, we leave a detailed study of what happens when this program is run until the next section. For the time being, note that it is vital when writing recursive program that variables should be declared to be LOCAL wherever appropriate. The reasons for this are discussed in the next section.

The problem of printing the first n integers is, of course, a rather trivial problem. We finish this section with a simple recursive program that could not be so easily written without recursion. The problem we consider is that of printing a given positive integer in binary. For example,

```
PROCbinaryprint(5)
```

should display

```
101
```

and

```
PROCbinaryprint(179)
```

should display

    10110011

The easiest way to convert an integer into binary is keep dividing by 2 and collect all the remainders. The remainders represent the bits required, but they are generated in reverse order.

		remainders
2	0	
2	1	1
2	2	0
2	5	1
2	11	1
2	22	0
2	44	0
2	89	1
	179	1

remainders in reverse order give 10110011

One way of programming this process without recursion would be to store the reanainders in an array and print them out only when the repeated division has terminated with zero. With recursion, the solution is considerably simpler. We break down the problem of printing the number n in binary:

```
print n DIV 2 in binary
PRINT ; n MOD 2;
```

where n MOD 2 is the last bit of the number.

```
 10   INPUT "Integer to te expressed in binary", int
 20   PROCbinaryprint(int)
 30   END

100   DEFPROCbinaryprint(n)
110   IF n<2 THEN PRINT ;n ; : ENDPROC
120   PROCbinaryprint(n DIV 2)
130   PRINT ; n MOD 2;
140   ENDPROC
```

You might like to experiment with the effect of omitting some of the semicolons in the above program. Changing line 110 affects only the first bit of the number printed while changing line 130 affects all the other bits apart from the first one.
Another experiment worth trying is to replace the stopping condition at line 110 with

```
110 IF n=0 THEN ENDPROC
```

The program will then work correctly in all cases except whn the original input value is 0. Taking no action on a zero parameter is correct if the case 'n=0' arises as a 'subproblem'. We do not want to print a leading zero at the start of a non-zero number. However, if the original number is zero, then this number must be printed. We must ensure that our procedure correctly handles the case where the dstopping condition is true on the first call of the procedure as well as the case where it is true for a subproblem.

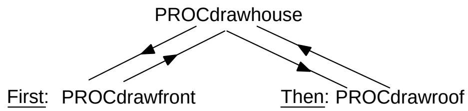
7.2 How it works

We start this section by introducing a model - the 'tree of procedure calls' - that will be valuable in understanding the behaviour of recursive programs. To introduce this model, we first look at a program that involves procedures, but no recursion. This program draws a simple house.

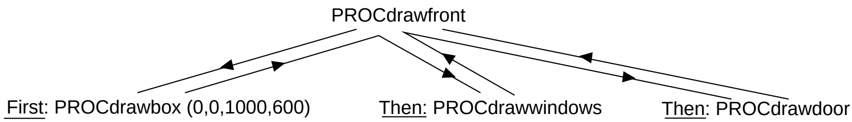
```
 10   height=600:width=1000
 20   MODE 4
 30   PROCdrawhouse
 40   k=GET : MODE 7
 50   END

 60   DEF PROCdrawhouse
 70     PROCdrawfront
 80     PROCdrawroof
 90   ENDPROC

100   DEF PROCdrawfront
110     PROCdrawbox(0,0,width,height)
120     PROCdrawwindows
130     PROCdrawdoor
140   ENDPROC

150   DEF PROCdrawwindows
160     LOCAL ww ,wh
170     ww=2*width/10 : wh=height/3
180     PROCdrawbox(ww/2,wh,ww,wh)
190     PROCdrawbox(7*ww/2,wb,ww,wh)
200   ENDPROC

210   DEF PROCdrawdoor
220     PROCdrawbox(4*width/10,0,width/5,height*2/3)
230   ENDPROC
```

```
240  DEF PROCdrawbox(x,y,w,h)
250    MOVE x,y
260    PLOT 1,0,h:PLOT 1,w,0
270    PLOT 1,0,-h:PLOT 1,-w,0
280  ENDPROC

290  DEFPROCdrawroof
300    MOVE 0,height
310    PLOT 1,width/2,height/3
320    PLOT 1,width/2,-height/3
330  ENDPROC
```

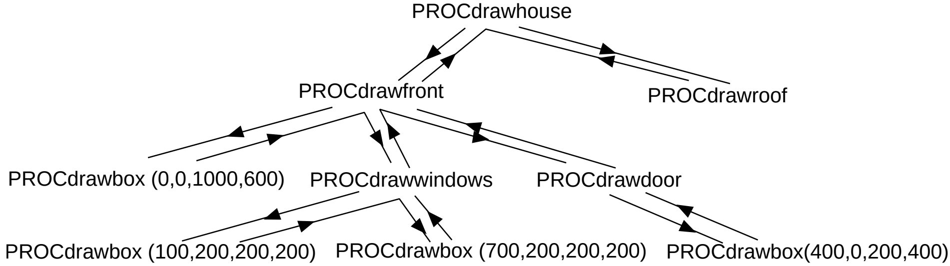
The process of drawing a house is broken down into the process of drawing a 'front' and then drawing a roof. We can illustrate this by



PROCdrawroof is defined in terms of primitive operations, MOVE and DRAW, but PROCdrawfront is broken down into further 'subproblems'.



PROCdrawbox is primitive, but PROCdrawwindows and PROCdrawdoor are themselves defined in terms of other procedure calls. We can represent all this information as a complete 'tree of procedure calls' for the program, together with arrows representing the 'flow of control' through the program.



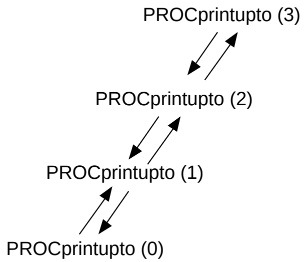
We shall often abbreviate such a diagram by using single lines in place of the double arrows and by omitting the word PROC.

Notice in this example that PROCdrawbox is obeyed on several different occasions with different sets of parameters. Each time it is used, this procedure behaves differently. However one call of PROCdrawbox is terminated before another is activated.

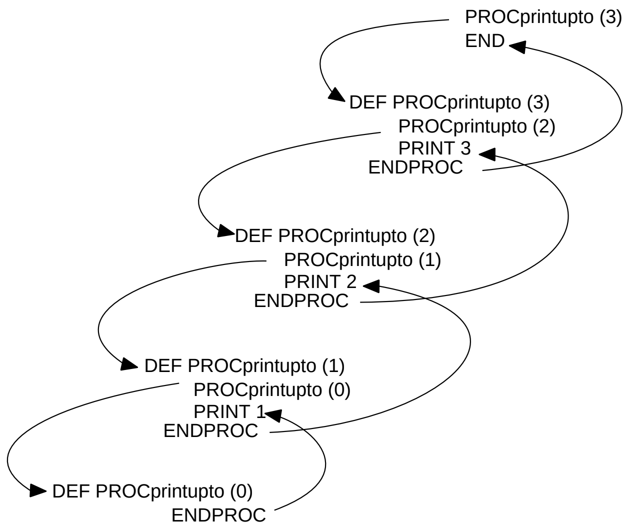
Now let us consider the behaviour of the first recursive program of the last zzection. We can illustrate the behaviour of this program for a call of

```
PROCprintupto(3)
```

by the following 'tree' of procedure calls.



(The tree has only one branch at each level because we are using unary recursion.) Like PROCdrawbox, PROCprintupto is called at several points with a different parameter each time. The only difference is that successive calls of PROCprintupto take place before the previous call has finished. The easiest way to understand what is happening is to imagine a separate copy of the procedure being created each time it is called. Of course, such copying would be extremely wasteful of computer store (and time) and recursion is organised much more efficiently behind the scenes. Only the storage space for parameters and local variables need be copied when a procedure is called. However, in appreciating how a recursive procedure works, it is convenient to imagine the whole procedure being copied. We shall refer to these copies of a procedure as 'activations' of the procedure. We can expand the above tree of procedure calls in more detail:



Now let us consider the behaviour of PROCprintbetween the procedure that used binary recursion. In this program, a call of

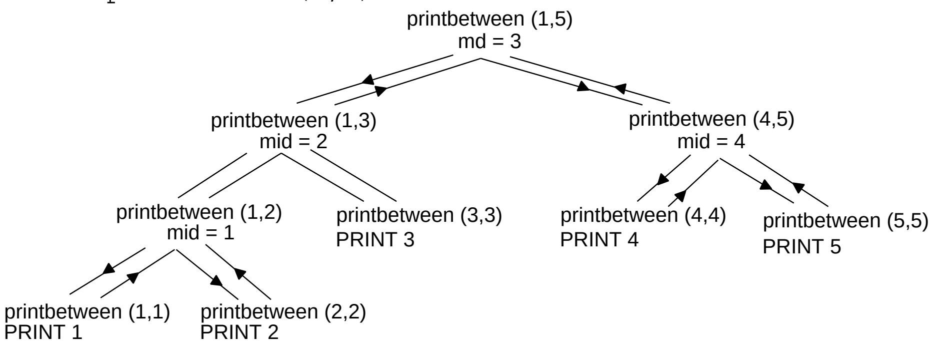
```
PROCprintupto(5)
```

results in a call of

```
PROCprintbetween(1,5)
```

This executes the following:

```
mid = (1+5) DIV 2 i.e. mid = 3
PROCprintbetween(1,3)
PROCprintbetween(4,5)
```

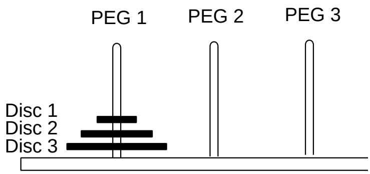


Each of the two recursive calls of PROCprintbetween behave in a similar way. You should be able to follow the arrows through the tree and see exactly how the sequence of procedure calls results in the numbering printed in the required order.

Note the importance of declaring 'mid' to be LOCAL to PROCprintbetween. This results in each recursive call of the procedure having its own private variable called 'mid'. Changing the value of this variable does not affect the current value of 'mid' in other activations or copies of the procedure. Thus, for example, when the activation PROCprintbetween(1,3) is terminated, control returns to PROCprintbetween(1,5) and the value of 'mid' in that procedure activation is still set to 3. The other procedure activations that have been obeyed since setting that value each used different storage locations for holding their LOCAL value for 'mid'. The value mid = 3' is needed in PROCprintbetween(1,5) for calculating the first parameter of the next recursive call (at line 180 of the program).

7.3 Towers of Hanoi

In this section we shall discuss the classic 'Towers of Hanoi' puzzle. The puzzle has been used as an illustration of recursion in the User Guide, but without explanation. The puzzle consists of three pegs mounted on a base together with a number of disks, all of different diameter. The disks have holes in them which allow them to e slipped on and off the pegs. The initial state is:



Towers of Hanoi puzzle

The problem is to find a sequence of moves that transfers the piles of disks from PEG1 to PEG2 subject to the following rules.

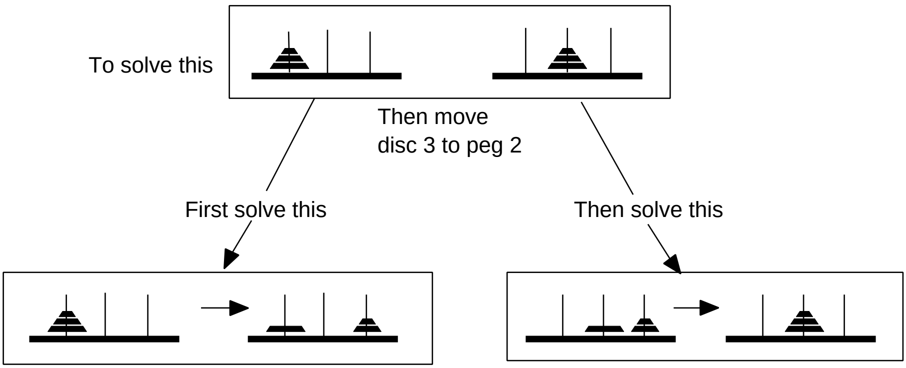
(1) Only one disk can be moved at a time.

(2) No disk can ever rest on a disk that is smaller than itself

PEG3 can be used during the transfer as temporary resting place for disks. Here is a solution to the three disk problem.

```
Move DISK1 from PEG1 to PEG2
Move DISK2 from PEG1 to PEG3
Move DISK1 from PEG2 to PEG3
Move DISK3 tram PEG1 to PEG2
Move DISK1 from PEG3 to PEG1
Move DISK2 from PEG3 to PEG2
Move DISK1 from PEG1 to PEG2
```

In order to produce a recursive procedure for printing a solution to the problem, we can reason as follows. At some stage during the solution, we must move DISK3 (the largest) from PEG1 to PEG2. In order to do this, all the other disks must be out of the way on PEG3 Thus, we must first solve the easier problem of transferring 2 disks to PEG3 (using PEG2 as the spare peg if necessary). While this subproblem is being solved, DISK3 can be treated as part of the fixed base. After this subproblem has been solved, and DISK3 has been moved to PEG2, we need to transfer the 2 disks on to PEG2, DISK3 being treated as part of the base.



This breakdown can be generalised to the n-disk problem:

To transfer a tower of n disks from one peg to another peg given a spare peg:

First transfer a tower of n-1 disks from the 'from peg' to the spare peg using the 'to peg' as a spare.

Then move disk n to the 'to peg'.

Then transfer the tower of n-1 disks from the spare peg to the 'to peg' using the 'from peg' as a spare.

This can be implemented directly as a BASIC procedure.

```
  100  DEF PROCtransfer(n,frompeg,topeg,sparepeg)
  110    IF n=0 THEN ENDPROC
  120    PROCtransfer(n-1,frompeg,sparepeg,topeg)
  130    PRINT "Move DISK " ;n; " from PEG " ;frompeg;
           " to  PEG ";topeg
  140    PROCtransfer(n-1,sparepeg,topeg,frompeg)
  150  ENDPROC
```

which can te called by:

```
   10  INPUT"Number of disks" ,noofdisks
   20  PROCtransfer(noofdisks,1,2,3)
   30  END
```

We leave it as an exercise for the reader to draw the complete tree of procedure calls that takes place in the cases for n = 3 and n = 4.

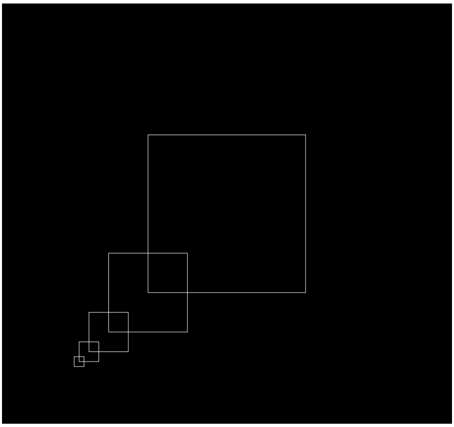
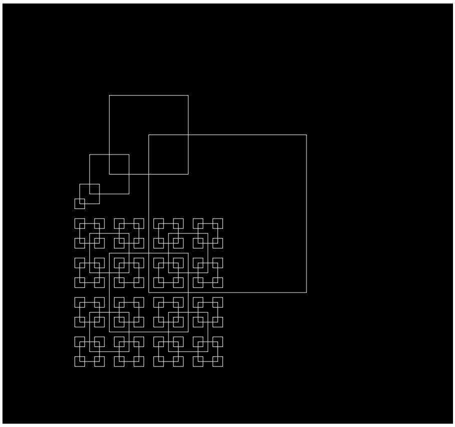
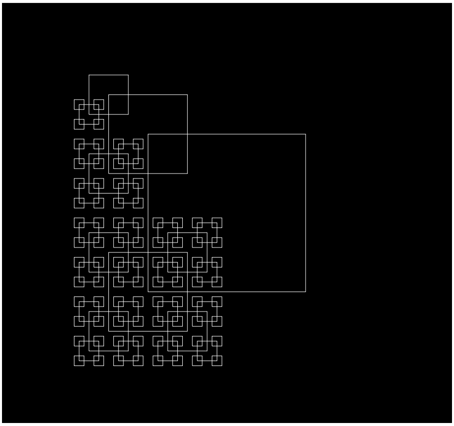
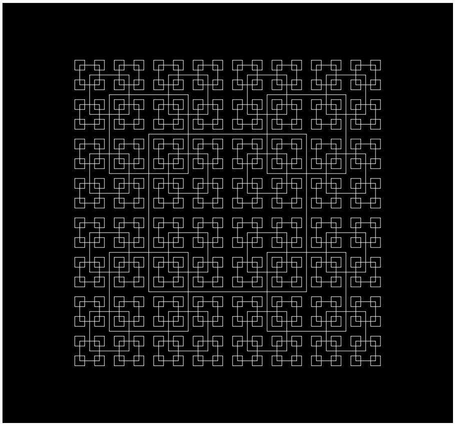
7.4 Recursive patterns and curves

There are many complex patterns and curves that can easily be drawn recursively and recursion is a useful tool in computer graphics and computer generated art.

Recursive squares

The simplest recursive pattern is one in which a basic shape is drawn together with recursive copies of smaller versions of the ccanplete patttern. For example, the next program creates a pattern of recursive squares. The pattern consists of a square, together with a recursive half-size copy of the complete pattern centered on each corner of the main square.

```
   10  INPUT "radius" ,r
   20  MODE 1
   30  PROCsquare(640,512,r)
   40  k=GET:MODE7
   50  END

   100  DEFPROCsquare(xc,yc,r)
   110  IF r<10 THEN ENDPROC
   120  LOCAL x1,x2,y1,y2
   130   x1=xc-r:x2=xc+r
   140   y1=yc-r:y2=yc+r
   150   MOVE x1,y1
   160   DRAW x1,y2 : DRAW x2,y2
   170   DRAW x2,y1 : DRAW x1,y1
   180   PROCsquare(x1,y1,r/2)
   190   PROCsquare(x1,y2,r/2)
   200   PROCsquare(x2,y2,r/2)
   210   PROCsquare(x2,y1,r/2)
   220  ENDPROC
```

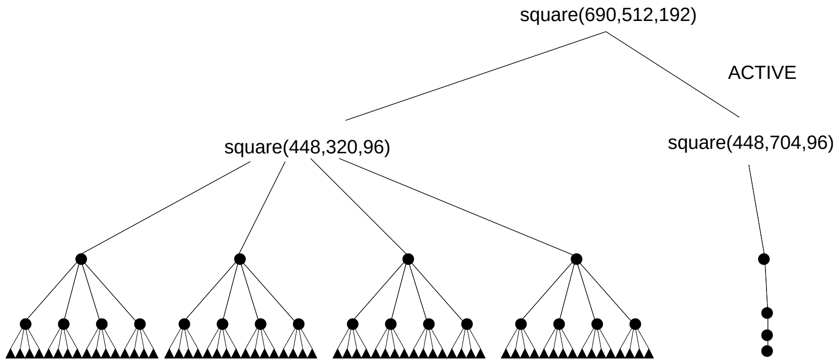





The photographs show the three stages in the build-up for r = 192, together with the complete pattern. For example, the first photograph illustrates the situation when the following procedure calls have been activated.

square (640,512,192)
↓
square (448,320,96)
↓
square (352,224,48)
↓
square (304,176,24)
↓
square (280,152,12)
↓
square (268,140,6)

The last procedure call triggers the stopping condition (r < 10) und terminates without drawing a square.

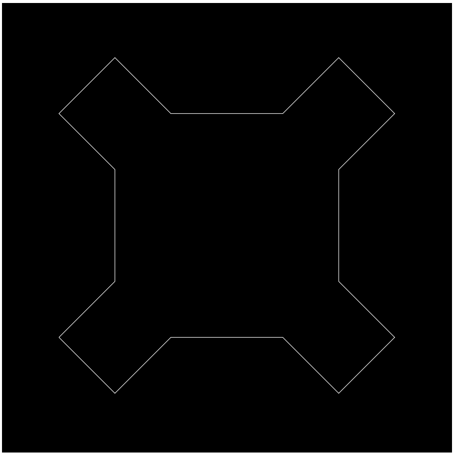
At the stage reached in the second photograph, the tree procedure calls that have been obeyed and terminated, together with the procedure calls that are still active, has the following shape. (The active procedure calls are down the right hand branch.)

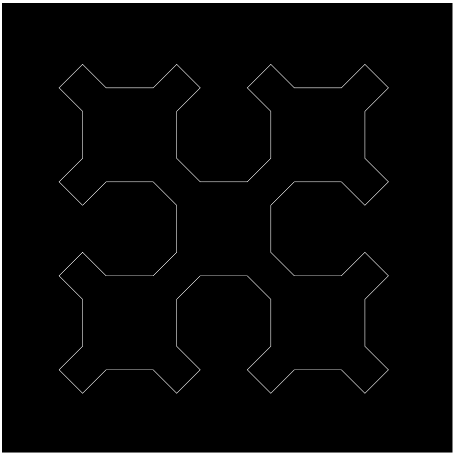


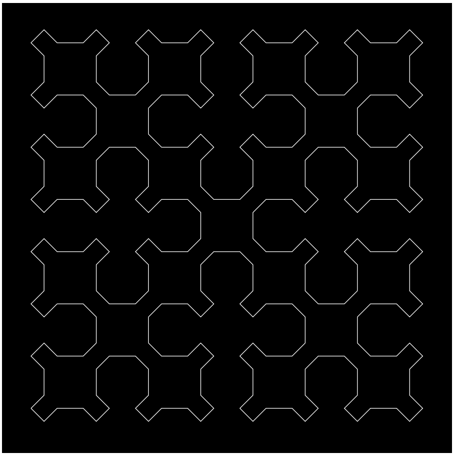
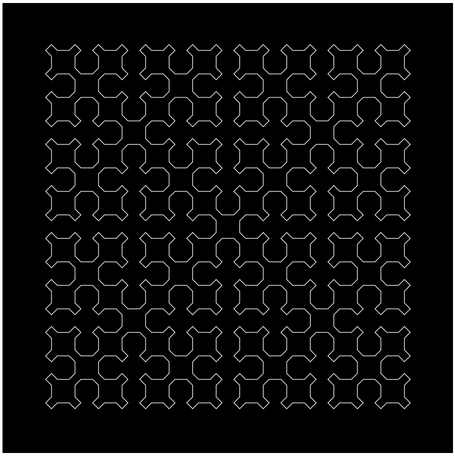
Space-filling curves

There is a large variety of patterns that crane into the category of 'space filling curves'. These curves are such that they can usually be drawn as a single continuous line or curve in some well defined way. We shall illustrate the technique involved by using the so-called 'Sierpinski curves'.

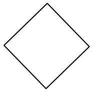
The next set of photographs shows the Sierpinski curves of orders 1 to 4.





It is convenient to define a Sierpinski curve of order 0 which consists of a diamond:

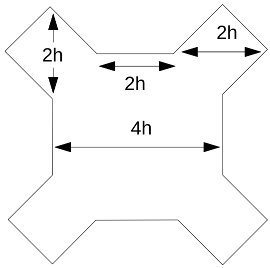


Notice that each of these curves could be drawn as a continuous line, without lifting pencil from paper. We shall look at two ways of drawing these curves, where the second method draws the curve as a continuous line.

The first method is conceptually a little easier and for this approach, we must first recognise that the Sierpinski curve of order 1 consists of four order 0 curves 'joined' at the centre. Similarly the order 2 curve consists of four order I curves joined at the centre. In general, an order n curve consists of four order n-1 curves joined at the centre. Note that when four subcurves are joined, this involves deleting four diagonal lines from the subcurves and joining the subcurves with two horizontal and two vertical lines. This suggests the following outline for a recursive procedure to draw a Sierpinski curve of order n, centred at x, y).

```
100  DEF PROCsierpinski(n, x, y)
110  IF n =0 THEN draw a diamond
120  k = horizontal and vertical distance to
     the centres of the four subcurves
130  PROCsierpinski(n-1, x-k, y-k)
140  PROCsierpinski(n-1, x-k, y+k)
150  PROCsierpinski(n-1, x+k, y+k)
160  PROCsierpinski(n-l, x+k, y-k)
170  ENDPROC
```

In order to fill out thin procedure, we need to examine the geometrical details fairly carefully. Any curve of order 1 or more consists of repeated coppies of the same basic shape and we shall name the various dimensions of this basic shape as follows:

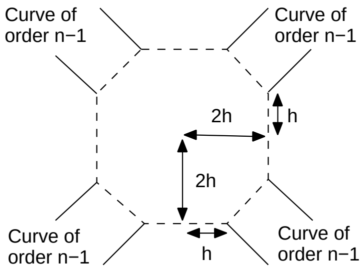


h is the smallest increment that will be required in our DRAW or MOVE statements. Thus the statements needed to draw a curve of order 0 (a diamond) centred at (x, y) are:

```
MOVE x-h, y
DRAW x, y+h : DRAW x+h, y
DRAW x, y-h : DRAW x-h, y
```

The distance from the centre of a curve of order n to the centre of one of its subcurves of order n-1 is 2^n*h. To convince yourself of this, you should mark the various distances on curves of different orders.

Finally the situation at the centre of a curve of order n, when the four subcurves of order n-1 have been drawn, can be illustrated as:



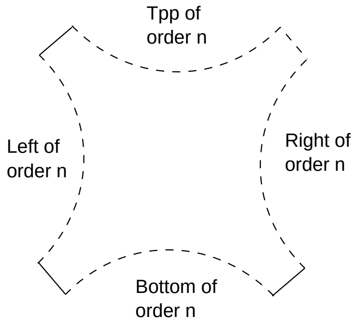
We need to delete the four dotted diagonal lines and draw the dotted vertical and horizontal lines. This can be easily accomplished by drawing round the dotted polygon using alternate PLOT 9 and PLOT 11 commands. These are relative plots, in the foreground and background colour respectively, which do not affect the last point visited on the line. Here is the complete program.

```
 10  INPUT"Order" ,order
 20  size=(2^order-1)*4+2
 30  h=INT(600/size)
 40  h2=h*2
 50  MODE 0

100  PROCsierpinski(order,640,512)
110  key=GET:MODE 7
120  END
130  DEF PROCsierpinski(n,x,y)
140  LOCAL k
150    IF n=0 THEN MOVE x-h,y:DRAW x,y+h:DRAW x+h,y:
        DRAW x,y-h:DRAW x-h,y:ENDPROC
160    k=2^n*h
170    PROCsierpinski(n-1,x-k,y-k)
180    PROCsierpinski(n-1,x-k,y+k)
190    PROCsierpinski(n-1,x+k,y+k)
200    PROCsierpinski(n-1,x+k,y-k)
210    MOVE x-h2,y-h
220    PLOT 9,0,h2 : PLOT 11,h,h
230    PLOT 9,h2,0 : PLOT 11,h,-h
240    PLOT 9,0,-h2 : PLOT 11,-h,-h
250    PLOT 9,-h2,0 : PLOT 11,-h,h
260  ENDPROC
```

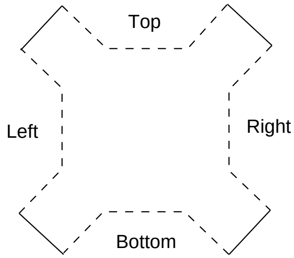
Note the use of INT at line 30 which ensures that increment, h, used in all the PLOTs is an integer. In programs that involve sequences of relative plots, it always advisable to ensure that the increments usexd are integers as the graphics 'current point' is recorded internally as a pair of integer coordinates. Use of real increments in relative plots can result in an accumulation of errors that cause misalignments in the display produced. You can see this effect by reviewing INT at line 30. An even better alternative would be to use an integer variable (with a %) throughout the program.

It is interesting to look at an alternative approach to drawing the Sierpinski curves by drawing the curve as a continuous line. This is the approach that would have to be used if the curve were to be drawn on a hard copy device (where lines cannot be deleted). This method is based on an algorithm described by Wirth (the inventor of the programming language PASCAL).

We first observe that a curve of order n consists of four components connected at the corners - a left component, a top component, a right component and a bottom canponent:



For example, in the case of the order 1 curve, we have:



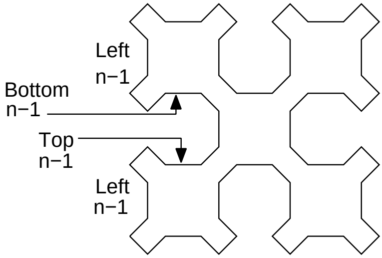
The procedure for drawing a Sierpinski curve of order n will te defined in terms of procedures for drawing its four components :

```
 90  DEF PROCsierpinski(n)
100    PROCleft(n):PLOT 1,h,h
110    PROCtop(n) :PLOT 1,h,-h
120    PPROCright(n):PLOT 1,-h,-h
130    PROCbottom(n) :PLOT 1,-h,h
140  ENDPROC
```

Now an order n component is made up of a sequence of order n-1 ccanponents joined in a well-defined way. For example, a left component of order n consists of:

a left component of order n-1
a diagonal line
a top component of order n-1
a vertical line
a component of order n-1
a diagonal line
a left component order n-1

For example, with n = 2,



If n = 0, the components are empty - joining four empty components diagonally at the corners gives a diamond shape. This gives the following procedure for drawing a left component of order n.

```
 150   DEF PROCleft(n)
 160     IF n=0 THEN ENDROC
 170     PROCleft(n-1):PLOT 1,h,h
 180     PROCtop(n-1):PLOT 1,0,h2
 190     PROCbottom(n-1):PLOT 1,-h,h
 200     PROCeft(n-1)
 210   ENDPROC
```

A similar breakdown can be achieved for the top, right and bottom components and this gives the following complete program

```
  10   MODE 0
  20   INPUT "Order ",order
  30   size=(2^order-l)*4+2
  40   h=INT(600/size) :h2=h*2
  50   MOVE 300,200+h
  60   PRROCsierpinski(order)
  70   K=GET:MODE7
  80   END

  90   DEF PROCsierpinski(n)
 100     PROCeft(n):PLOT 1,h,h
 110     PRROCtop(n) :PLOT 1,h,-h
 120     PRROCright(n):PLOT 1,-h,-h
 130     PROCbottom(n):PLOTT 1,-h,h
 140     ENDPROC
```

```
150  DEF PROCleft(n)
160    IF n=0 THEN ENDPROC
170    PROCleft(n-1):PLOT 1,h,h
180    PROCtop(n-1):PLOT 1,0,h2
190    PROCbottom(n-1):PLOT 1,-h,h
200    PROCle£t(n-1)
210  ENDPROC

220  DEF PROCtop(n)
230    IF n=0 THEN ENDPROC
240    PROCtop(n-1):PLOT 1,h,-h
250    PROCright(n-1):PLOT 1,h2,0
260    PROCleft(n-1):PLOT l,h,h
270    PROCtop(n-1)
280  ENDPROC

290  DEF PROCright(n)
300    IF n=0 THEN ENDPROC
310    PROCright(n-1):PLOT 1,-h,-h
320    PROCbottom(n-1):PLOT 1,0,-h2
330    PROCtop(n-1):PLOT 1,h,-h
340    PROCright(n-1)
350  ENDPROC

360  DEF PROCbottom(n)
370    IF n=0 THEN ENDPROC
380    PROCbottom(n-1):PLOT 1, -h, h
390    PROCleft(n-1):PLOT 1,-h2,0
400    PROCright(n-1):PLOT 1,-h,-h
410    PROCbottom(n-1)
420  ENDPROC
```

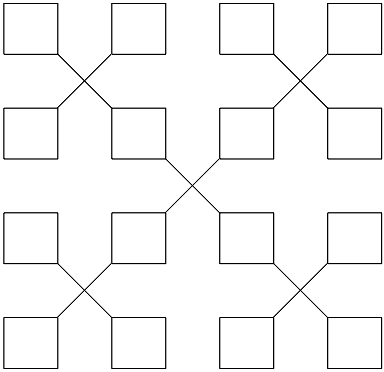
You should notice that there are two types of recursion involved in the last program. There is straightforward recursion where, for example, PROCeft calls PROCleft. There is also 'hidden' or 'mutual' recursion where, for example, PROCeft calls PROCtop which in turn calls PROCleft.

You should run both Sierpinski programs and observe the differences in their behaviour. Other well-known space filling curves are the 'C-curve' and the 'dragon curve'. Programs drawing these curves are presented in 'Creative Graphics' published by Acornsoft.

Exercises

1 Animate a program for solving the 'Towers of Hanoi' puzzle. The program should display a picture of the pegs and disks, and, instead of printing a move, should move the appropriate disk in the display.

2 The family of patterns, of which the following is an example, can be described recursively.



Write a program that generates patterns like this.

3 Write a program that generates patterns like those of the last exercise, but using diamonds instead of squares.

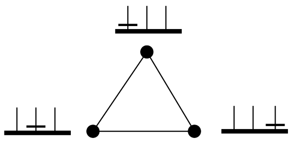
4 The next photographs show a family of curves (due ta Wirth) called W-curves.

Write a program that draws a W-curve of order n as a continuous line.

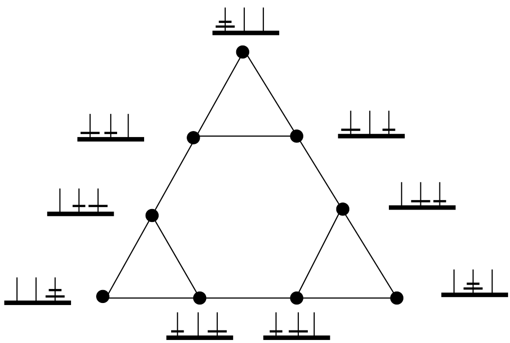
7.5 Towers of Hanoi revisited - state space representation

Many non-numerical problems can be represented by a large (possibly infinite) set of 'problem states' together with a set of moves, or operators, each of which transforms one state into another. The definition of an operator may include restrictions on the states to which it can be applied.

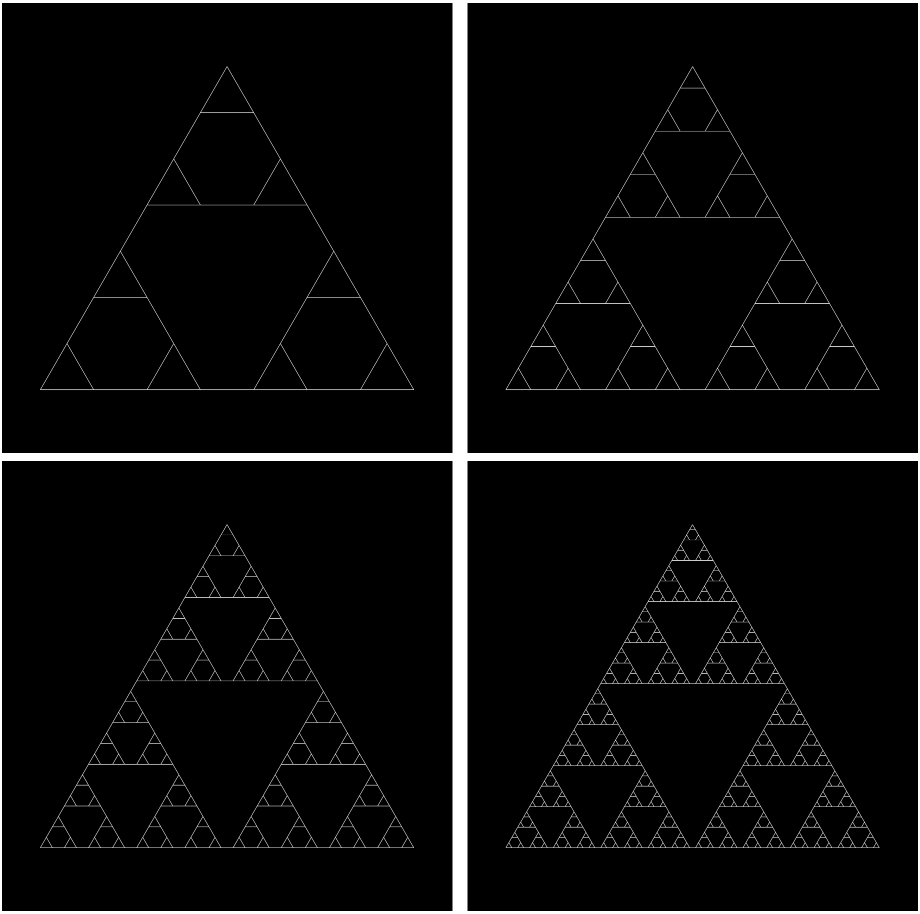
For example, we could represent the complete set of 1-disk Tower of Hanoi states with a single triangle. There are three states in the 1 disk problem - the disk can be on one of three pegs. Each vertex of the triangle represents a state. The lines connecting vertices represent a possible move from one state to another.



In the state space for the 2—disk problem we have three such triangles, one for each possible position of the larger disk. The three triangles are joined together by lines representing the three different ways of moving the larger disk from one peg to another.



Similarly the 3-disk state-space diagram contains three 2-disk state-space diagrams. The block of photographs show the state space diagrams for the 3-, 4-, 5- and 6-disk problem.



We can write a recursive program to generate these diagrams:

```
 10  base = 800
 30  xleft=(1280-base)/2 : xright = 1280-xleft
 40  xtop=xleft+base/2
 50  root3=SQR(3)
 60  height= base*root3/2
 65  ybottesn= (1024-height)/2
 66  ytop = 1024-ybottom
 70  INPUT "No. of disks",n
 80  arclength=base/(2^n-1)
 90  MODE 0
100  VDU 5
110  PROCdrawgraph(n,xleft,xright,ybottom,xtop,ytop)
120  k=GET:MODE 7:END
```

```
140  DEF PROCdrawgraph(n,x1,x2,y12,x3,y3)
150  LOCAL subside, subheight
160    IF n=0 THEN ENDPROC
170    subside = (2^(n-1)-1)*arc1ength
180    subheight = root3*subside/2
190    PROCdrawgraph(n-1,x1,x1+subside,y12,x1+subside/2,
         y12+subheight)
200    PROCdrawgraph(n-1,x2-subside,x2,y12,x2-subside/2,
         y12+subheight)
210    PROCdrawgraph(n-1,x3-subside/2,x3+subside/2,
         y3-subheight,x3;y3)
220    MOVE x1+subside,y12
230    DRAW x2-subside,y12
240    MOVE x1+subside/2,y12+subheight
250    DRAW x3-subside/2,y3-subheight
260    MOVE x2-subside/2,y12+subheight
270    DRAW x3+subside/2,y3-subheight
280  ENDPROC
```

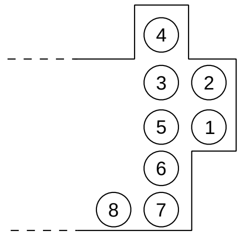
7.6 Problems with recursion

In this section, we shall illustrate two problems that can arise when using recursion. To do this, we revisit the problem of colour filling a region that is defined by boundaries that have already teen drawn on the screen. A non-recursive algorithm for accomplishing this was presented in Chapter 2.

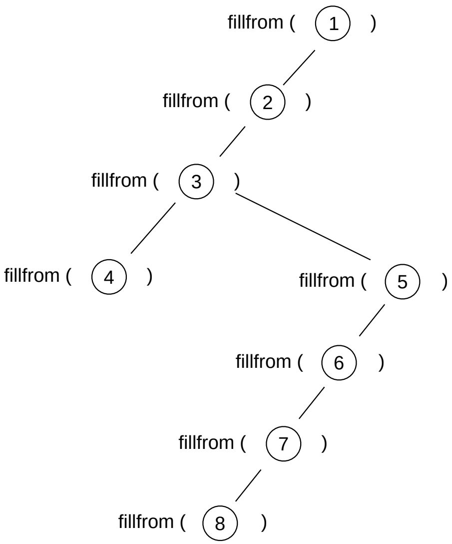
Simple recursive colour till - excessive recursive depth
Recall that the colour fill algorithm of Chapter 2 started from an arbitrary point in the region and worked outwards from that point to adjacent points, eventually visiting the whole region. We can very easily describe a recursive procedure for colour filling a 4-connected region:

```
200  DEF PROCfillfrom(x,y)
210    IF POINT(x,y)>0 THEN ENDPROC
220    PLOT 69 ,x,y
230    PROCfillfrom(x,y+4)
240    PROCfillfrom(x,y-4)
250    PROCfillfrom(x+4,y)
260    PROCfillfrom(x-4,y)
270  ENDPROC
```

This is certainly much shorter than the equivalent procedures in the program in Chapter 2. Now that we are familiar with recursion, the recursive version is also conceptually easier. If, however, you insert the above procedure in a program and run it, you will find that it will work only for very small regions. For larger regions the program will terminate with the error message 'No room' This is because a long sequence of recursive procedure calls has been entered and not yet terminated. To see how this happens, look at the following configuration of pixels.



If we start the fill process by calling PROCfillfrom with parameters that specify pixel 1, then the following tree of procedure activations is created.



This process will continue, and as the pixels in the region are visited, the tree of procedure activations will get deeper and deeper. A procedure call will be terminated only when a dead end is encountered, for example at pixel 4. Each time a procedure is activated, storage space is used up for holding parameters, local variables and a record of where to return to when the procedure is terminated. This space is freed only when the procedure terminates. There is thus a limit to the depth to which the recursion can be extended, and for a region of any size the limit will soon be encountered. Notice also that, in this example, when a long chain in of recursive calls is eventually terminated, most of the other recursive calls that then take place will be unnecessary and will terminate immediately. This redundancy is, however, necessary if thv algorithm is to cater for a convoluted region. On a large processor with virtually unlimited storage space, the simple recursive algorithm might be usable, but on a micro, it is rather unsatisfactory.

The general point illustrated by this example is that recursion must not be allowed to proceed to any great depth.

Using horizontal fill - hidden loop nesting

As we have already seen in the last section the simple recursive approach to colour-fill leads to problems involving the depth of the recursion and the queue method introduced in Chapter 2 is obviously preferable. In this section an alternative recursive approach is examined. Although this also involves a common problem with recursion, this new problem can be easily overcome.

A common provision in graphics systems that operate with a raster scan display is a horizontal fill facility. Such a facility will typically be given the (x,y) coordinates of a point and will colour-fill pixels to the left and right of the given pixel as long as these pixels are in the background colour.

On the BBC micro, the first issue of the operating system (OS 0.1) did not provide such a facility, but this omission was rectified in later versions with a new set of PLOT instructions.

First of all, we present a BASIC procedure that implements a horizontal fill. At first, we shall not use the new PLOT commands and will implement the horizontal fill without them. Anyone who still has the first issue of the operating system will need to do it this way. The method can also be seen as an explanation of the version using the new PLOT facilities which are presented later.

```
300  DEF PROCfillalong(x,y)
310  LOCAL nextx
320    PROCdirectionfill(x,y,xstep)
330    rightx=nextx-xstep
340    PROCdirectionfill(x,y,-xstep)
350    leftx=nextx+xstep
360  ENDPROC

370  DEF PROCdirectionfill(x,y,dir)
380    nextx = x
390    REPEAT
400      PLOT 69,nextx,y
410      nextx=nextx+dir
420    UNTIL POINT(nextx,y)>0
430  ENDPROC
```

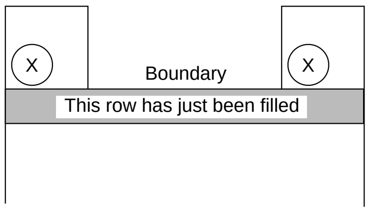
A call of PROCfillalong will first colour-fill to the right of the given pixel until a non-background point is encountered. The same thing is done to the left. Both scans are carried out using the subsidiary procedure PROCdirectionfill whose parameter 'dir' indicates the direction of the scan. As a result of calling PROCfillalong, the two non-local variables 'leftx' and 'rightx' are set to values indicating the extent of the strip that was filled The value of 'xstep' will indicate the width of a pixel and this will depend on the mode being used. For example, MODE 1, 'xstep=4'. We now present a recursive version of PROCfillfrom which could be used in place of previous versions of the same procedure, but which makes use of horizontal fill. The procedure will be given a point, and starts by filling the horizontal strips in which the point specified by its parameters lies. It then calls itself recursively to fill from each pixel above and below the strip that has just been filled. A first attempt at this procedure is:

```
200  DEFPROCfillfrom(x,y)
210  LOCAL leftx,rightx,scanx
220    IF POINT(x,y)>0 THEN ENDPROC
230    PROCfillalong(x,y)
240    FOR scanx = leitx TO rightx STEP xstep
250      PROCfillfrom(scanx,y+ystep)
260      PROCfillfrom(scanx,y-ystep)
270    NEXT scanx
280  ENDPROC
```

Note that 'yetep' is the height of a pixel. In MODE 1, 'ystep=4'. It we run our colour filling program with this. version of PROCfillfrom, we again find that it will only work for small regions. With larger regions the program terminates with the message - 'Too many FORs'. This usually means that too many FOR statements have teen nested inside each other. A common cause of this error message is the omission of a NEXT statement. In our case, there are no explicitly nested FOR statements, but because the recursive procedure calls appear inside a FOR statement, any FOR statement entered during a recursive call behaves as if it were inside the outer FOR statement. The limit on the number of nested FOR statements is 10 and if the recursive depth goes beyond 10, as it will for a larger region, then the program will terminate. Unfortunately the only satisfactory solution is to replace the FOR loop with an equivalent GOTO loop:

```
200  DEF PROCfillfrom(x,y)
210  LOCAL leftx,rightx,scanx
220    IF POINT(x,y)>0 THEN ENDPROC
230    PROCfillalong(x,y)
240    scanx=leftx
250      PROCfillfrom(scanx,y+ystep)
260      PROCfi1lfrom(scanx,y-ystep)
270      scanx=scanx+xstep
280    IF scanx<=rightx GOTO 250
290  ENDPROC
```

The algorithm described above could be considerably improved. Notice that many of the recursive calls will be completely unnecessary. For example, once a line of pixels has been filled, the first recursive call of PROCfillfrom on the adjacent row will in most cases be sufficient and the remaining recursive calls will terminate immediately. Any additional recursive call from the adjacent row will only occasionally be necessary. For example, in the following situation:



At least two recursive calls at pixels marked X are necessary on the row above the one that has just been filled, so as to initiate filling of the two concavities opening off the lower row. Similarly many recursive calls involve looking back at pixels in rows already visited and again this is only occasionally necessary.

An interesting adjustment to the program which will allow you to see which points are being visited for a second or third time, is to replace the line that recognises points that need not te filled by:

```
220  IF POINT(x, y) > 0 THEN PLOT 70, x, y : ENDPROC
```

This will invert the colour of any pixel that is already colour-filled and you will be able to observe the progress of the algorithm not only as it fills the background region, but also as it makes unnecessary recursive calls in areas that have already been filled. The improvements required to avoid this unnecessary work are fairly tricky and we will not go into them here.

Finally, here is an alternative version of PROCfillalong that uses the PLOT 77 command for horizontal fill.

```
300  DEF PROCfillalong(x,y)
310    PLOT 77 ,x,y
320    X%=CPblock : Y%=CPblock DIV 256
330    A%=&0D : CALL &FFF1
340    leftx=(!CPblock AND 65535)
350    rightx=(!(CPblock+4) AND 65535)
360  ENDPROC
```

The PLOT 77 statement at line 310 scans left and right fom the pixel specified by x and y until it reaches the last background point in both directions. A line is drawn be the two points reached. The rightmost point becomes the 'current graphics point' and the leftmost point becomes the previous graphics point. We now need to set 'leftx' to x-coordinate of the previous graphics point and 'rightx' the x-coordinate of the current graphics point. To do this we use the OSWORD call at lines 320 to 330. This uses block of store declared at the start of the program by:

```
  5  DIM CPblock 8
```

You do not need to understand the details of how OSWORD calls work in order to use this 'recipe'. The above version of PROCfillalong is exactly equivalent to that described earlier. It is of course much faster.

It is worth mentioning briefly another PLOT command that could be used to speed up execution of the loop in PROCfillfrom (lines 250 to 280). The statement

```
PLOT 92, x, y
```

searches pixels to the right of (x,y) for a background point and sets the last non-background point reached as current graphics position. We leave the reader to think about how this could be used.

Finally, note that all recursive colour filling algorithms can run out of room for large or highly convoluted regions. The horizontal fill methods described here could all be reorganised to use a queue similar to that used in Chapter 2.

7.7 Divide and conquer - merge sorting

Yet another approach to sorting (a number of algorithms were introduced in the last chapter) is the merge sort algorithm, one of the most efficient sort algorithms available. This algorithm is tricky to implement without recursion. It illustrates one of the most important recursive approaches to a problem - that of divide and conquer. We sort the list by sorting each half and merging the two halves - merging is a fast process. Each half is sorted by dividing it into two and sorting each quarter. Each quarter is sorted by dividing it into two - in other word divide and conquer.

A program implementing a recursive merge sort, PROCmergesort, is now given.

```
 10   DIM number(100)
 20   INPUT "No. of items",noofitems
 30   FOR i=1 TO noofitems
 40     INPUT number(i)
 50   NEXT i
 60   PROCmergesort(noofitems)
 70   FOR i=1 TO noofitems
 80     PRINT number(i)
 90   NEXT i
 95   END

100   DEF PROCmergesort(n)
110     DIM aux(n)
120     PROCsubsort(1,n)
130   ENDPROC

150   DEF PROCsubsort(i,j)
160   LOCAL mid
170   IF i>=j THEN ENDPROC
190     mid=(i+j)DIV 2
200     PROCsubsort(i,mid)
210     PROCsubsort(mid+1,j)
220     PROCmerge(i,mid,mid+1,j)
230   ENDPROC

250   DEF PROCmerge(begin1,end1,begin2,end2)
260   LOCAL i,next,firstfinished,secondfinished
270     FOR i=begin1 TO end1:aux(i)=number(i):NEXT i
280     next = begin1
290     firstfinished=FALSE : secondfinished=FALSE
300     REPEAT
310       IF aux(begin1)<number(begin2)
      THEN PROCtake1fromfirsthalf
      ELSE PROCtake1fromsecondhalf
320     next=next+1
330   UNTIL firstfinished OR secondfinished
340     IF secondfinished THEN
          FOR i=next TO end2 : number(i)=aux(begin1) :
          begin1 = begin1+1 : NEXT i
350   ENDPROC
```

```
370  DEF PROCtake1fromfirsthalf
380    number(next)=aux(begin1)
390    IF begin1=end1 THEN
         firstfinished=TRUE
       ELSE begin1=begin1+1
400  ENDPROC

420  DEF PROCtake1fromsecondha1f
430    number(next)=number(begin2)
440    IF begin2=end2 THEN
         secondfinished=TRUE
       ELSE begin2=begin2+1
450  ENDPROC
```

PROCsubsort(i,j) sorts the list of elements from number(i) to number(j). Note that the recursion terminates on a call of subsort(i,j) where i = j, a list of one item is already sorted. In order to ccanplete the above procedure, we need to define the procedure 'merge' which is used to combine the two separate <u>sorted</u> sequences in

number(i) to number(mid)
and number(mid+1) to number(j)

into one sorted sequence in number(i) to number(j).

Merging two sublists that are already sorted is a fast operation although it is not easy to do in situ as required in the present context. We have defined:

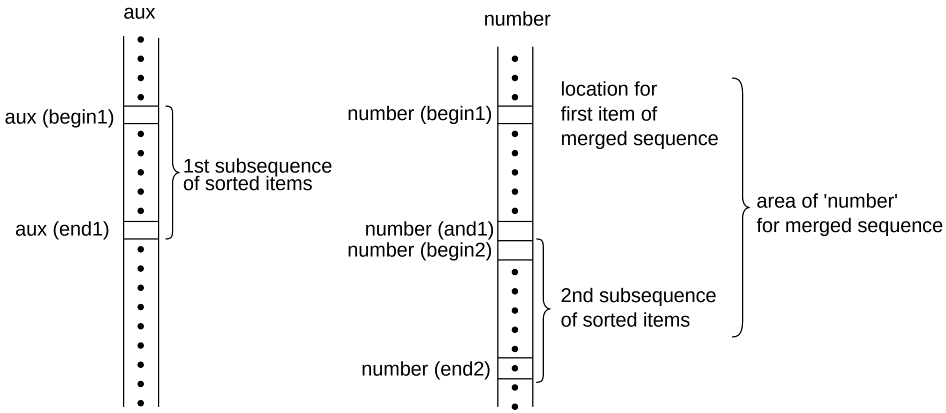
PROCmerge(begin1, end1, begin2, end2)

which first copies the contents of number(begin1) to number(end1) into an auxiliary array 'aux' in locations aux(begin1) to aux(end1).

The procedure will then repeatedly select an item from one of, our two subsequences and insert it into the next available location of the area that is to contain the merged sequence.

If the items of the first subsequence are exhausted then any remaining items in the second subsequence are in their correct positions. If the items in the second subsequence are exhausted, then any remaining items in the first subsequence must be copied from 'aux' into their new locations in 'number'. (Any items from the second subsequence that were previously there must already have teen copied up into their new positions.)

The situation after one of the subsequences has been copied into the auxiliary array, but before merging starts, is illustrated in the next diagram.



Exercises

1 Use the text animation procedures defined in Chapter 4, Section 4.1, to animate a merge sort. You will need to organise the display differently from the way it was organised for the other sort methods - display the two arrays involved, 'number' and 'aux' at either side of the screen.

2 The process of binary search presented in Chapter 6 (Section 6.4) can te described recursively. Do this, and write a recursive procedure to carry out a binary search on a suitable table.

3 The sorting method known as 'quicksort' can be described as follows:

To sort a table of n items:
  Select a random entry (the first say)
  Split the table into two sub-tables - the entries
    that should come before the selected entry and
    the entries that should come after the selected
    entry.
  Sort each of the two sub-tables (recursively).
  The two sub-tables with the selected entry in the
    middle give the final sorted table.

Write a recursive procedure that implements 'quicksort' on a table of numbers.