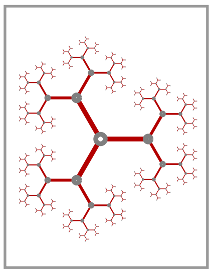
CHAPTER 9

Recursion in PostScript



For various technical reasons, recursion in PostScript is a tricky business. It is possible, however, and in some circumstances nearly indispensable. It is extremely useful to have some idea of how to deal with it, since standard algorithms in the literature, for example those traversing lists of various kinds, are often laid out in terms of recursion.

1. The perils of recursion

The factorial $n!$ of a positive integer n is defined informally by the rule that it is the product of all positive integers less than or equal to n. Thus $4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$. If we want to get a computer to calculate it, we follow these rules: (1) if $n = 1$ then $n! = 1$; (2) if $n > 1$ then $n! = n \cdot (n-1)!$. This sort of formula, where a function is evaluated directly by reduction to a simpler case, is called **recursion**. It could be argued that recursion encapsulates the essence of mathematics—which trys to reduce every assertion either to an axiom or to one that has already been proven, or in broad terms to reduce every problem to a simpler one that has already been solved.

At any rate, we can write a procedure in PostScript that follows these two last rules exactly:

```
/factorial { 1 dict begin
  /n exch def
  n 1 eq {
    1
  }{
    n n 1 sub factorial mul
  } ifelse
end } def
```

This code is correct and will run, but probably only if n is small. Why is that? The reason for failure is somewhat technical. When you `begin` a dictionary in a procedure, that dictionary is put on top of a stack of dictionaries, and when you `end` that dictionary it is taken off this stack. Unlike a few other stacks used by PostScript, the dictionary stack is usually severely limited in size. If you call a procedure recursively which uses a local dictionary, the size of the dictionary stack will build up with every call, quite possibly beyond the maximum size allowed. If this occurs you will get a `dictstackoverflow` error message. So this procedure might very well work for small values of n but fail for large ones.

dictstackoverflow:1

Therefore

- *You should never introduce a dictionary to handle local variables in a recursive procedure exactly as you do in others.*

There is nothing wrong with using dictionaries in recursive procedures, but *they shouldn't remain open across the recursive procedure calls.* That is to say, you should `begin` and `end` the dictionary without making any recursive calls in between. You might very well want to do this so as to do some complicated calculations before setting up the recursion calls. We'll see an example later on. The simplest rule to follow with recursive procedures in PostScript is to use variables as little as possible, and to resort to stack manipulations instead. In effect, you should use data on the stack to serve as a substitute for the execution stack PostScript doesn't want you to use. The drawback, of course, is that such manipulations are very likely to make the procedure much less readable than it should ideally be. The technique is usually painful and bug-prone, and hence best avoided. Nonetheless, here is a simple example—a correct recursive procedure to calculate $n!$ correctly (except for problems with floating point accuracy):

```
 % args:  a single positive integer n
 % effect:  returns n!
/factorial {
   % stack:  n
  1 gt {
     % stack:  n > 1
     % recall that n = n(n-1)!
    dup 1 sub
     % stack:  n n-1
    factorial
     % stack:  n (n-1)!
    mul
  } ifelse
   % stack:  n!
} def
```

The comments here trace what is on the stack, which is an especially good idea in programs where complicated stack manipulations are made.

Of course this version of `factorial` is hardly efficient. It is simple enough to write a `factorial` routine that just uses a `for` loop. But this version illustrates nicely the perils of recursion in PostScript.

Exercise 1. *The gcd (greatest common divisor) of two non-negative integers m and n is n if $m = 0$ or the gcd of m and $n \bmod m$ if $m \neq 0$. Construct a recursive procedure in PostScript to find it.*

Overflow on the dictionary stack occurs in my current implementation only when the stack reaches a size of about 500. So much of what I am saying here is silly in many situations. But there is another reason not to allow the dictionary stack to grow large—when looking up the value of a variable, the PostScript interpreter looks through all of the dictionaries on the stack, starting at the top, until it finds the variable's name as a key. If the name is inserted in one of the bottom dictionaries, a lot of searching has to be done. The efficiency of a recursive procedure can thus be cut down dramatically as the recursion builds up.

2. Sorting

A common and extremely useful example of a procedure which uses recursion is a sorting routine called **quick sort**. I am going to present it below without much explanation, but I shall make a few preliminary remarks.

Let me first explain what just about any sorting routine will do. It will have one argument, an array of items which can be ranked relative to each other—for example an array of numbers. The routine will rearrange the items in the array in ascending order, from smallest up. For example

```
[4 3 2 1] quicksort
```

will rearrange the items in the array to make it equal to `[1 2 3 4]`. You might think that this isn't a very mathematical activity, and that a mathematician would have no serious interest in such a routine, especially in PostScript where you are only interested in drawing stuff. That is not at all correct. *A good sorting routine should be part of the tool kit of every mathematical illustrator.* For example, we shall see a bit later on a procedure that constructs the convex hull of a given finite collection of points, and which depends upon a sorting routine in a highly non-trivial way. Sorting routines can also play a role in 3D drawing, where objects must be drawn from back to front.

As any introduction to programming will tell you, there are lots of ways to sort items in an array. They vary enormously in efficiency. One very simple one is called the **bubble sort**, because it pushes big items up to the top end of the array as though they were bubbles. It makes several scans of the array, each time bubbling up the largest item. For example, on the first pass it moves the largest item to the end, on the second it moves the second largest item to the place next to the end, and so on.

Here is PostScript code for a bubble sort.

```
 % args:  array a of numbers
 % effect:  sorts the array in order
/bubblesort { 4 dict begin
/a exch def
/n a length 1 sub def
n 0 gt {
   % at this point only the n+1 items in the bottom of a remain to be sorted
   % the largest item in that block is to be moved up into position n
  n {
    0 1 n 1 sub {
      /i exch def
      a i get a i 1 add get gt {
         % if a[i] > a[i+1] swap a[i] and a[i+1]
        a i 1 add
        a i get
        a i a i 1 add get
         % set new a[i] = old a[i+1]
        put
         % set new a[i+1] = old a[i]
        put
      } if
    } for
    /n n 1 sub def
  } repeat
} if
end } def
```

For example, if we sort `[5 4 3 2 1 0]` in this way, we get successively in each bubble:

```
[4 3 2 1 0 5]
[3 2 1 0 4 5]
[2 1 0 3 4 5]
[1 0 2 3 4 5]
[0 1 2 3 4 5]
[0 1 2 3 4 5]
```

Bubble sorting is very easy to put into correct PostScript code. But it is pretty inefficient if the size n of the array is large. On the first bubbling pass it makes $n-1$ comparisons, on the second $n-2$, etc. This makes approximately $n(n-1)/2$ comparisons in all, so the time it takes is proportional to the square of n.

We can do much better. The sorting routine which is generally fastest of all is called **quick sort**. One of the things that makes it unusual is that it tries a **divide and conquer** approach, basically splitting the array into halves and then calling itself recursively on each half. Its running time is nearly always proportional to $n \log n$, a great improvement over `bubblesort`.

Quick sort has three components. The principal routine is `quicksort` itself, with only the array as an argument. It calls a routine `subsort` which has as argument not only the array but also two indices $L < R$ picking out a range inside the array to sort. For the call in `quicksort` these are 0 and $n-1$. But `subsort` calls itself recursively with smaller ranges. It also calls a third routine `partition` which does the real swapping. This procedure has four arguments—the array a, a lower index L and an upper index R, and an integer x. It moves items in the $[L, R]$ range of the array around so the left half $[L, i]$ is made up of items less than or equal to x, the right half $[j, R]$ of items greater than or equal to x. It then returns the pair i and j on the stack. I hide the details. of `partition`, but here is the pseudo-code for `subsort`:

```
subsort(a, L, R) {
  x := a[(L+R)/2];
  [i, j] = partition(a, L, R, x);
  if (L < j) subsort(a, L, j);
  if (i < R) subsort(a, i, R);
}
```

and here in PostScript is the whole package:

```
 % args:  a L R x
 % effect:  effects a partition into two pieces [L j] [i R]
 % leaves i j on stack
/partition { 1 dict begin
...  end } def

 % args:  a L R
 % effect:  sorts a[L ..  R]
/subsort {
  1 dict begin
  /R exch def
  /L exch def
  /a exch def
   % x = a[(L+R)/2]
  /x a L R add 2 idiv get def
  a L R x partition
  /j exch def
  /i exch def
   % put recursion arguments on the stack
   % as well as the arguments for the tests
  a L j
  j L gt
  a i R
  i R lt
   % close dictionary
  end
  { subsort }{
     % get rid of unused arguments
    pop pop pop
  } ifelse
  { subsort }{ pop pop pop } ifelse
} def

 % args:  a
 % effect:  sorts the numerical array a
 % and leaves a copy on the stack
/quicksort { 4 dict begin
  /a exch def
```

```
    /n a length 1 sub def
    n 0 gt {
      a 0 n subsort
    } if
    a
  } def
```

The important thing here is to notice how the recursive routine `subsort` manages the dictionary and stack in a coordinated way. The most interesting thing in the way it does that is how the arguments for the recursion are put on the stack before it is even known if the calls will be made, and removed if not used. The procedure `partition`, on the other hand, does not call itself and is therefore allowed to use a local dictionary in the usual way.

This will give you an idea of what goes on, but the working version of the `quicksort` routine has an extra feature. We will be interested in sorting arrays of things other than numbers in a moment. Rather than design a different sort routine for each kind of array, we add to this one an argument `/comp`, the name of a comparison test replacing `lt` for numbers. For numbers themselves, we would define

```
  /comp { lt } def
```

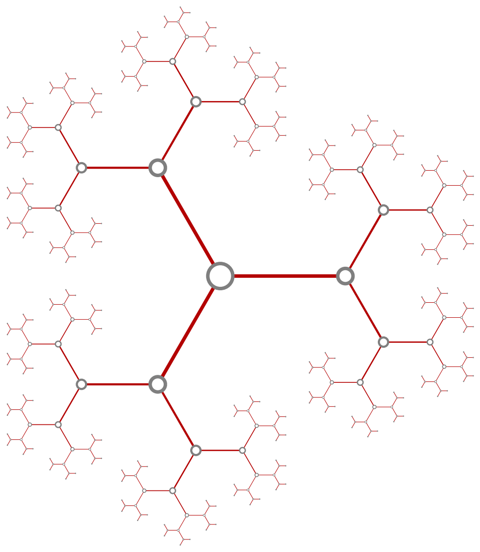
and then write

```
  /comp [5 4 3 2 1 0] quicksort
```

This is an extremely valuable modification.

To get an idea of how much faster `quicksort` is than `bubblesort`, if $n = 1,000$ then $n(n-1)/2 = 495,000$ while $n \log n$ is about $7,000$. Of course each loop in `quicksort` is more complicated. In actual runs, `quicksort` takes 0.07 seconds to sort $1,000$ items, while `bubblesort` takes 0.79 seconds on the same array. The `quicksort` is definitely fast enough to include in drawing programs without noticeable penalty.

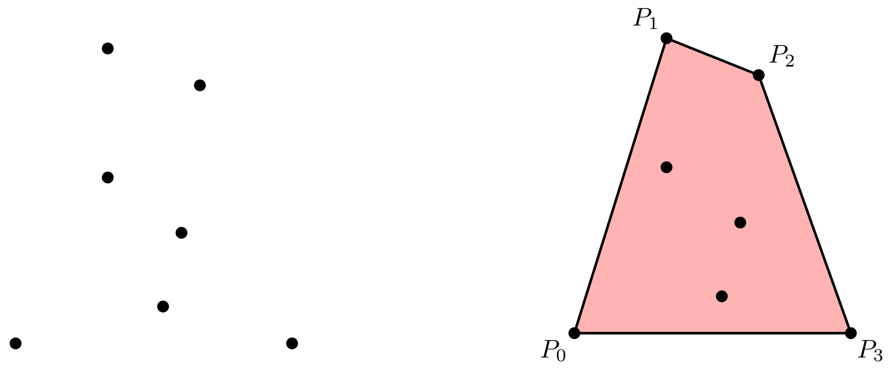
Exercise 2. *Use recursion to draw the following picture, with an optional parameter to determine the depth of the recursion:*



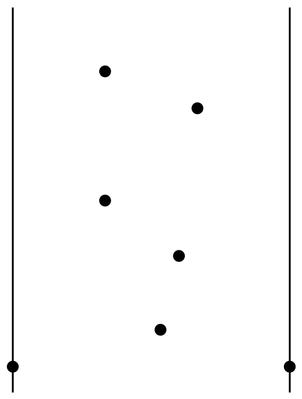
Notice that even the line widths get smaller further out along the branches. If you are a mathematician, you might like to know that this is the Bruhat-Tits building of $SL_2(\mathbb{Q}_2)$. If you are not a mathematician, you can think of it as one non-mathematician of my acquaintance called it, "chicken wire on growth hormones."

3. Convex hulls

Mostly to convince you that sorting is not an empty exercise, but also to give you a procedure you might find very useful, I shall describe here a well known and relatively efficient way to find the convex hull of a finite set of points in the plane. It will have a single argument, an array of points, which we may as well take to be arrays `[x y]`. It will return an array made up of a subset of the given set of points, listed so that traversing them in order will go clockwise around the convex hull of the point set. The output, in other words, consists of the **extreme** members of the original set, those which are in some sense on its outside. Finding the outside points, and in the correct order, is not exactly a difficult task, but before you start the discussion you might think of how you would do it yourself. Extra complications are caused by possible errors in the way PostScript handles floating point numbers.

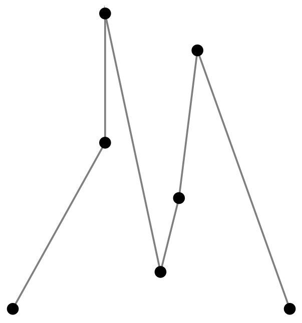


The algorithm to be described is not in fact terribly complicated, but certain aspects of it might not seem so well motivated without a few preliminary remarks. First of all, what does sorting have to do with the problem? The following picture illustrates that points of the original data set furthest to the left and furthest to the right will be part of the output.



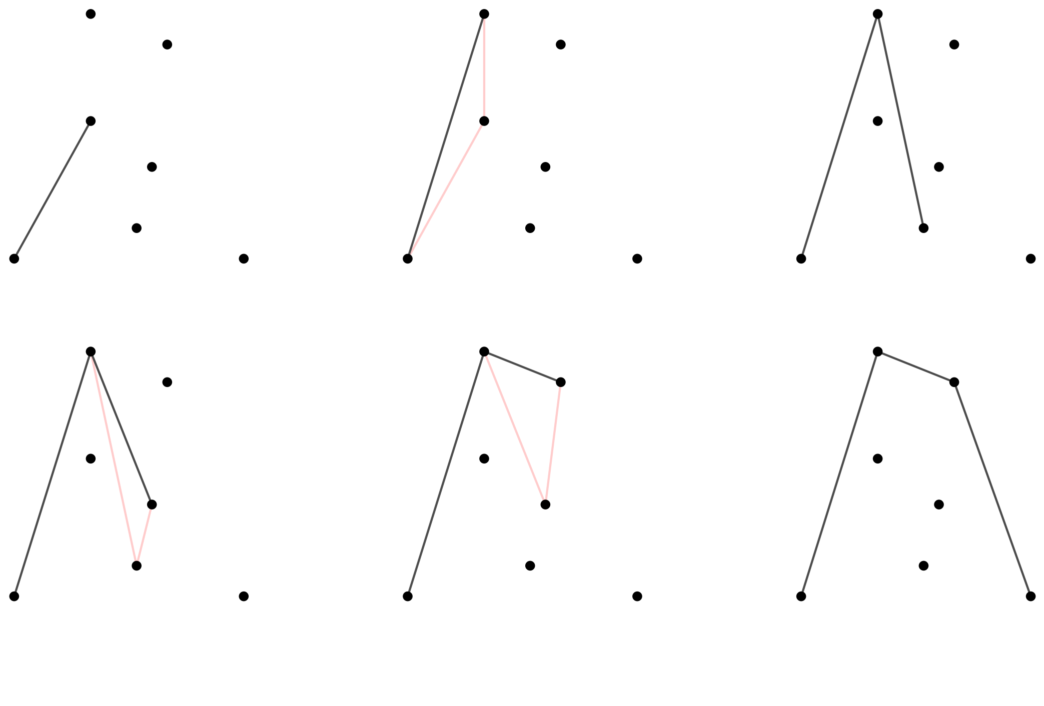
But finding these points will involve sorting the original point set according to values of the x coordinate. We have to be a bit more careful in fact. It might very well be the case that several points in the set have the same x coordinate, so there might not be a unique point furthest left or right. We will therefore be more particular: we say that (x_0, y_0) is less than the point (x_1, y_1) if $x_0 < x_1$ (the first point is further left) or $x_0 = x_1$ but $y_0 < y_1$ (in case they have equal x coordinates, the one on the bottom will be called smaller). When we sort the data, this is the ranking procedure we use. It is still true that the largest and smallest points in the set must be part of the output.

In fact, sorting the original points is one of two key steps in the procedure. We may in effect assume that the original set of points is an array in which the points are listed so that points which are 'smaller' in the sense described above come first. This immediately reduces the difficulty of the problem somewhat, since we can take the first point of the (sorted) input to be the first point in the output. In the following figure the points are linearly ordered along a polygon 'from left to right' in the slightly generalized sense.



Another idea is that we shall do a left-to-right scan of the input, producing eventually all the points in the output which lie along the top of the convex hull. Then we shall do a right to left scan which produces the bottom. Again, the preliminary sorting guarantees that the last point in the array will end the top construction and begin the bottom one.

What is it exactly that we are going to do as we move right? At any moment we shall be looking at a new point in the input array. It will either have the largest x-value among all the points looked at so far, or it will at least have the largest y-value among those which have the largest x-value. In either event, let its x-value be x_r. Then we shall try to construct the top half of the convex hull of the set of all input points with x-value at most x_r. Certainly the current point P we are looking at will be among these. But as we add it, we might very well render invalid a few of the points on the list we have already constructed. We therefore go backwards among those, deleting from the list the ones which lie distinctly inside the top hull of the whole list. A point P_k will be one of those precisely when the polygon $P_{k-1}P_kP$ has an upward bend, so that the polygon $P_{k-1}P_kP$ is below the segment $P_{k-1}P$. So we go backwards, throwing away those in such a bend, until we reach a triple where the bend is to the right. Then we move on to the next point to the right. The following figures show the progression across the top in a particular example, including the 'left turns' that are eliminated as we go along:



I reproduce below the principal routine for finding the convex hull. It uses routines comp for sorting, vsub which subtracts two vectors, dot to calculate the dot product of two vectors, and vperp which rotates the difference of two vectors counter-clockwise by $90°$. The routine /comp tests whether $(x_0, y_0) < (x_1, y_1)$. All these are defined internally in hull.inc.

```
% args:  an array of points C
 % effect:  returns the array of points on the boundary of
 % the convex hull of C, in clockwise order
/hull { 16 dict begin
  /C exch def
   % sort the array
  /comp C quicksort
  /n C length def
   % Q circles around to the start
   % make it a big empty array
  /Q n 1 add array def
  Q 0 C 0 get put
  Q 1 C 1 get put
  /i 2 def
  /k 2 def
   % i is next point in C to be looked at
   % k is next point in Q to be added
   % [ Q[0] Q[1] ...  ]
   % scan the points to make the top hull
  n 2 sub {
     % P is the current point at right
    /P C i get def
    /i i 1 add def
    {
       % if k = 1 then just add P
      k 2 lt { exit } if
       % now k is 2 or more
       % look at Q[k-2] Q[k-1] P: a left turn (or in a line)?
       % yes if (P - Q[k-1])*(Q[k-1] - Q[k-2])perp >= 0
      P Q k 1 sub get vsub
      Q k 1 sub get Q k 2 sub get vperp
      dot 0 lt {
         % not a left turn
        exit
      } if
       % it is a left turn; we must replace Q[k-1]
      /k k 1 sub def
    } loop
    Q k P put
    /k k 1 add def
  } repeat
   % done with top half
   % K is where the right hand point is
  /K k 1 sub def
  /i n 2 sub def
  Q k C i get put
```

```
    /i i 1 sub def
    /k k 1 add def
    n 2 sub {
      % P is the current point at right
      /P C i get def
      /i i 1 sub def
      {
        % in this pass k is always 2 or more
        k K 2 add lt { exit } if
        % look at Q[k-2] Q[k-1] P: a left turn (or in a line)?
        % yes if (P - Q[k-1])*(Q[k-1] - Q[k-2])perp >= 0
        P Q k 1 sub get vsub
        Q k 1 sub get Q k 2 sub get vperp
        dot 0 lt {
          % not a left turn
          exit
        } if
        /k k 1 sub def
      } loop
      Q k P put
      /k k 1 add def
    } repeat

    % strip Q down to [ Q[0] Q[1] ...  Q[k-2] ]
    % excluding the doubled initial point
    [ 0 1 k 2 sub {
      Q exch get
    } for ]
  end } def
```

Code

Procedures discussed here are found in `sort.inc` and `hull.inc`.

References

1. M. de Berg, M. van Creveld, M. Overmars, and O. Schwarzkopf, **Computational Geometry - algorithms and applications**, Springer, 1997. The convex hull algorithm is taken from Chapter 1 of this book. Like many books on the subject of computational geometry, the routines it explains are often more beautiful than practical. Among them, however, are a few useful gems. And the illustrations are, rarely for books on computer graphics, absolutely first rate.

2. R. Sedgewick, **Algorithms**, Addison-Wesley. There are several editions of this, setting out code in various programming languages. The latest one (2003) uses Java. Chapter Five has a nice discussion of recursion and tree structures, while Chapters Six and Seven are about sorting.