5. Divide-and-conquer and recursion

Learning objectives:

- The algorithmic principle of divide-and-conquer leads directly to recursive procedures.
- Examples: Merge sort, tree traversal. Recursion and iteration.
- My friend liked to claim "I'm 2/3 Cherokee." Until someone would challenge him "Two- thirds? You mean 1/2 , or, or maybe 3/8, how on earth can you be 2/3 of anything?" "It's easy," said Jim, "both my parents are 2/3."

An algorithmic principle

Let A(D) denote the application of an algorithm A to a set of data D, producing a result R. An important class of algorithms, of a type called divide-and-conquer, processes data in two distinct ways, according to whether the data is small or large:

- If the set D is small, and/or of simple structure, we invoke a simple algorithm A_0 whose application $A_0(D)$ yields R.
- If the set D is large, and/or of complex structure, we partition it into smaller subsets $D_1, \ldots, D_k$. For each i, apply $A(D_i)$ to yield a result R_i. Combine the results $R_1, \ldots, R_k$ to yield R.

This algorithmic principle of divide-and-conquer leads naturally to the notion of recursive procedures. The following example outlines the concept in a high-level notation, highlighting the role of parameters and local variables.

```
procedure A(D: data; var R: result);
var  D1, … , Dk: data;  R1, … , Rk: result;
begin
  if  simple(D)  then R := A0(D)
    else {  D1, … , Dk := partition(D);
         R1 := A(D1); … ; Rk := A(Dk);
         R := combine(R1, … , Rk) }
end;
```

Notice how an initial data set D spawns sets $D_1, \ldots, D_k$ which, in turn, spawn children of their own. Thus the collection of all data sets generated by the partitioning scheme is a tree with root D. In order for the recursive procedure A(D) to terminate in all cases, the partitioning function must meet the following condition: Each branch of the partitioning tree, starting from the root D, eventually terminates with a data set D_o that satisfies the predicate 'simple(D_o)', to which we can apply the algorithm.

Divide-and-conquer reduces a problem on data set D to k instances of the same problem on new sets $D_1, \ldots, D_k$ that are "simpler" than the original set D. Simpler often means "has fewer elements", but any measure of

"simplicity" that monotonically heads for the predicate 'simple' will do, when algorithm A0 will finish the job. "D is simple" may mean "D has no elements", in which case A_o may have to do nothing at all; or it may mean "D has exactly one element", and A0 may just mark this element as having been visited.

The following sections show examples of divide-and-conquer algorithms. As we will see, the actual workload is sometimes distributed unequally among different parts of the algorithm. In the sorting example, the step 'R:=combine(R_1, ... , R_k)' requires most of the work; in the "Tower of Hanoi" problem, the application of algorithm A0 takes the most effort.

Divide-and-conquer expressed as a diagram: merge sort

Suppose that we wish to sort a sequence of names alphabetically, as shown in Exhibit 5.1. We make use of the divide-and-conquer strategy by partitioning a "large" sequence D into two subsequences D_1 and D_2, sorting each subsequence, and then merging them back together into sorted order. This is our algorithm A(D). If D contains at most one element, we do nothing at all. A_o is the identity algorithm, $A_o(D) = D$.

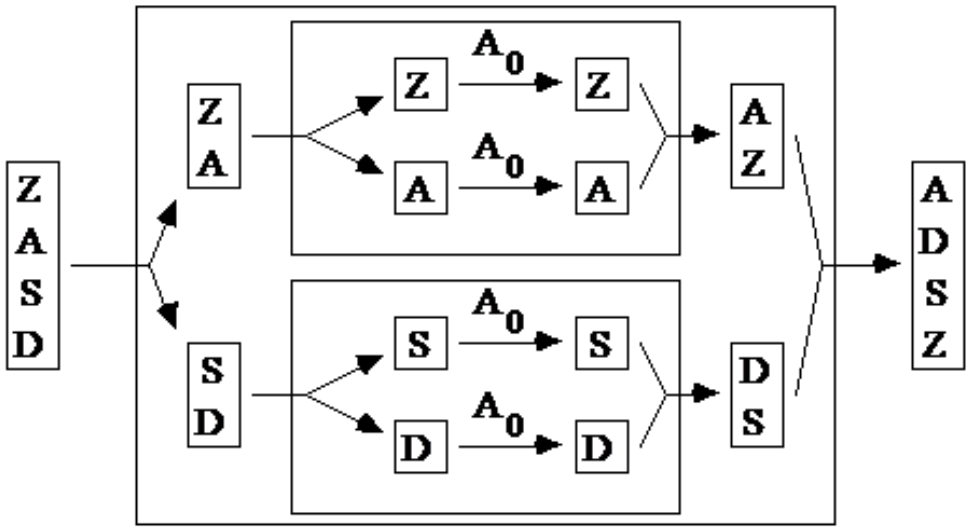

Exhibit 5.1: Sorting the sequence {Z, A, S, D} by using a divide-and-conquer scheme

```
procedure sort(var D: sequence);
var  D1, D2: sequence;

   function combine(D1, D2: sequence): sequence;
   begin  { combine }
      merge the two sorted sequences D1 and D2
      into a single sorted sequence D';
      return(D')
   end;  { combine }

begin  { sort}
   if  |D| > 1  then  { split D into two sequences D1 and D2 of
equal size;
         sort(D1);  sort(D2);  D := combine(D1, D2) }
   { if |D| ≤ 1, D is trivially sorted, do nothing }
end;  { sort }
```

In the chapter on “sorting and its complexity”, under the section “merging and merge sorts” we turn this divide-and-conquer scheme into a program.

Recursively defined trees

A *tree*, more precisely, a rooted, ordered tree, is a data type used primarily to model any type of hierarchical organization. Its primitive parts are *nodes* and *leaves*. It has a distinguished node called the *root*, which, in violation of nature, is typically drawn at the top of the page, with the tree growing downward. Each node has a certain number of children, either leaves or nodes; leaves have no children. The exact definition of such trees can differ slightly with respect to details and terminology. We may define a *binary tree*, for example, by the condition that each node has either exactly, or at most, two children.

The pictorial grammar shown in Exhibit 5.2 captures this recursive definition of 'binary tree' and fixes the details left unspecified by the verbal description above. It uses an *alphabet* of three symbols: the nonterminal 'tree symbol', which is also the start symbol; and two terminal symbols, for 'node' and for 'leaf'.

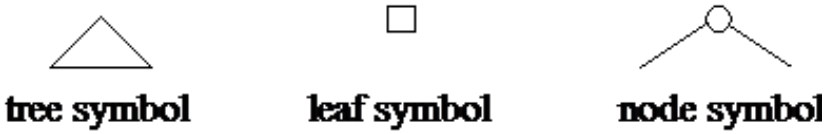

Exhibit 5.2: The three symbols of the alphabet of a tree grammar

There are two production or rewriting rules, p1 and p2 (Exhibit 5.3). The *derivation* shown in Exhibit 5.4 illustrates the application of the production rules to generate a tree from the nonterminal start symbol.

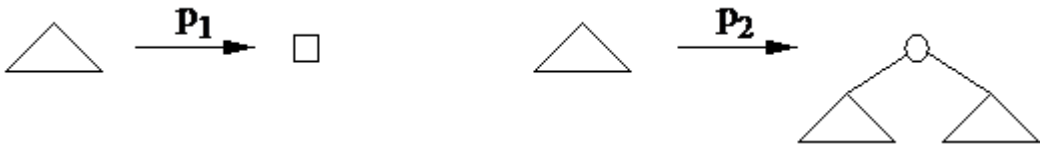

Exhibit 5.3: Rule p_1 generates a leaf, rule p_2 generates a node and two new trees

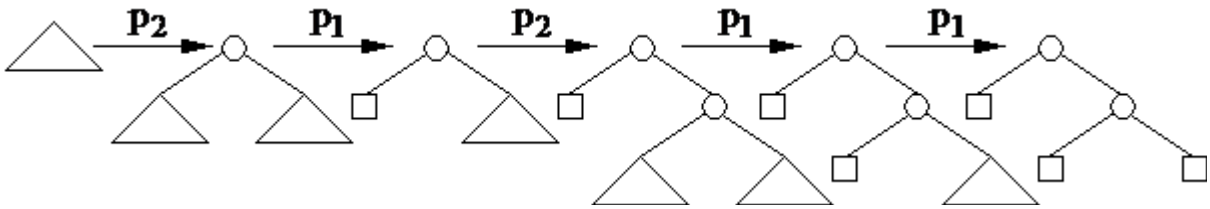

Exhibit 5.4: One way to derive the tree at right

We may make the production rules more detailed by explicitly naming the coordinates associated with each symbol. On a display device such as a computer screen, the x- and y-values of a point are typically Cartesian coordinates with the origin in the upper-left corner. The x-values increase toward the bottom and the y-values increase toward the right of the display. Let (x, y) denote the screen position associated with a particular symbol, and let d denote the depth of a node in the tree. The root has depth 0, and the children of a node with depth d have depth d+1. The different levels of the tree are separated by some constant distance s. The separation between siblings is determined by a (rapidly decreasing) function t(d) which takes as argument the depth of the siblings and depends on the drawing size of the symbols and the resolution of the screen. These more detailed productions are shown in Exhibit 5.5.

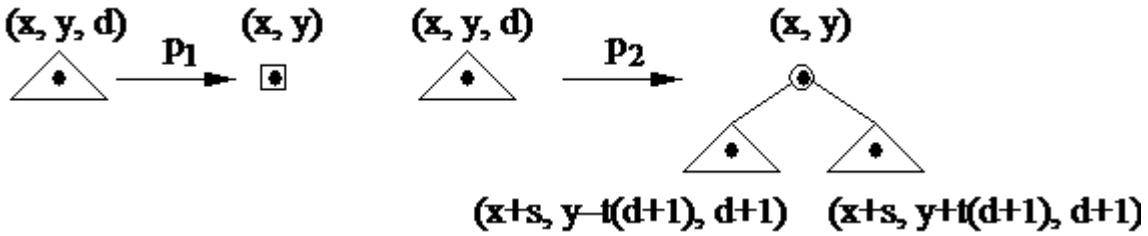

Exhibit 5.5: Adding coordinate information to productions in order to control graphic layout

The translation of these two rules into high-level code is now plain:

```
procedure p1(x, y: coordinate);
begin
   eraseTreeSymbol(x, y);
   drawLeafSymbol(x, y)
end;

procedure p2(x, y: coordinate; d: level);
begin
   eraseTreeSymbol(x, y);
   drawNodeSymbol(x, y);
   drawTreeSymbol(x + s, y - t(d + 1));
   drawTreeSymbol(x + s, y + t(d + 1))
end;
```

If we choose $t(d) = c \cdot 2^{-d}$, these two procedures produce the display shown in Exhibit 5.6 of the tree generated in Exhibit 5.4.

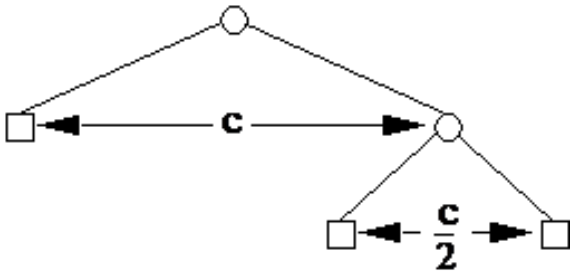

Exhibit 5.6: Sample layout obtained by halving horizontal displacement at each successive level

Technical remark about the details of defining binary trees: Our grammar forces every node to have exactly two children: A child may be a node or a leaf. This lets us subsume two frequently occurring classes of binary trees under one common definition.

1. *0-2 (binary) trees.* We may identify leaves and nodes, making no distinction between them (replace the squares by circles in Exhibit 5.3 and Exhibit 5.4). Every node in the new tree now has either zero or two children, but not one. The smallest tree has a single node, the root.
2. *(Arbitrary) Binary trees.* Ignore the leaves (drop the squares in Exhibit 5.3 and Exhibit 5.4 and the branches leading into a square). Every node in the new tree now has either zero, one, or two children. The smallest tree (which consisted of a single leaf) now has no node at all; it is empty.

For clarity's sake, the following examples use the terminology of nodes and leaves introduced in the defining grammar. In some instances we point out what happens under the interpretation that leaves are dropped.

Recursive tree traversal

Recursion is a powerful tool for programming divide-and-conquer algorithms in a straightforward manner. In particular, when the data to be processed is defined recursively, a recursive processing algorithm that mirrors the structure of the data is most natural. The recursive tree traversal procedure below illustrates this point.

Traversing a tree (in general: a graph, a data structure) means visiting every node and every leaf in an orderly sequence, beginning and ending at the root. What needs to be done at each node and each leaf is of no concern to the traversal algorithm, so we merely designate that by a call to a 'procedure visit( )'. You may think of inspecting the contents of all nodes and/or leaves, and writing them to a file.

Recursive tree traversals use divide-and-conquer to decompose a tree into its subtrees: At each node visited along the way, the two subtrees L and R to the left and right of this node must be traversed. There are three natural ways to sequence the node visit and the subtree traversals:

```
1. node; L; R   { preorder, or prefix }
2. L; node; R   { inorder or infix }
3. L; R; node   { postorder or suffix }
```

The following example translates this traversal algorithm into a recursive procedure:

```
procedure traverse(T: tree);
   { preorder, inorder, or postorder traversal of tree T with
leaves }
  begin
    if  leaf(T) then visitleaf(T)
      else { T is composite }
        { visit1(root(T));
          traverse(leftsubtree(T));
          visit2(root(T));
          traverse(rightsubtree(T);
          visit3(root(T)) }
  end;
```

When leaves are ignored (i.e. a tree consisting of a single leaf is considered to be empty), the procedure body becomes slightly simpler:

```
if  not empty(T)  then  { … }
```

To accomplish the k-th traversal scheme (k = 1, 2, 3), '$visit_k$' performs the desired operation on the node, while the other two visits do nothing. If all three visits print out the name of the node, we obtain a sequence of node names called 'triple tree traversal', shown in Exhibit 5.7 along with the three traversal orders of which it is composed. During the traversal the nodes are visited in the following sequence:

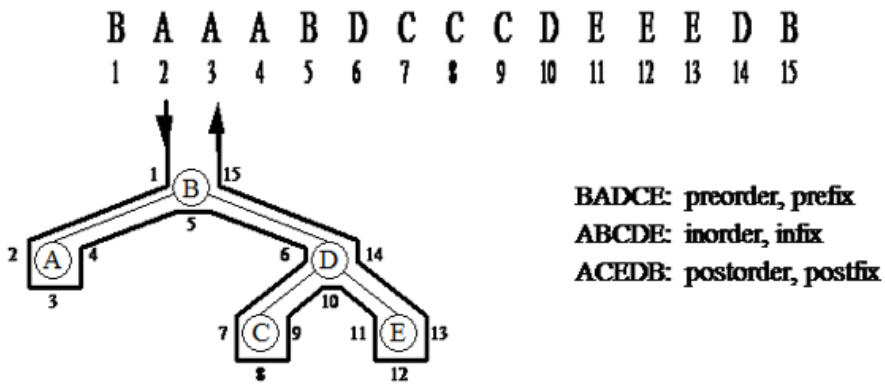

Exhibit 5.7: Three standard orders merged into a triple tree traversal

Recursion versus iteration: the Tower of Hanoi

The "Tower of Hanoi" is a stack of n disks of different sizes, held in place by a tall peg (Exhibit 5.8). The task is to transfer the tower from source peg S to a target peg T via an intermediate peg I, one disk at a time, without ever placing a larger disk on a smaller one. In this case the data set D is a tower of n disks, and the divide-and-conquer algorithm A partitions D asymmetrically into a small "tower" consisting of a single disk (the largest, at the bottom of the pile) and another tower D' (usually larger, but conceivably empty) consisting of the n – 1 topmost disks. The puzzle is solved recursively in three steps:

1. Transfer D' to the intermediate peg I.
2. Move the largest disk to the target peg T.
3. Transfer D' on top of the largest disk at the target peg T.

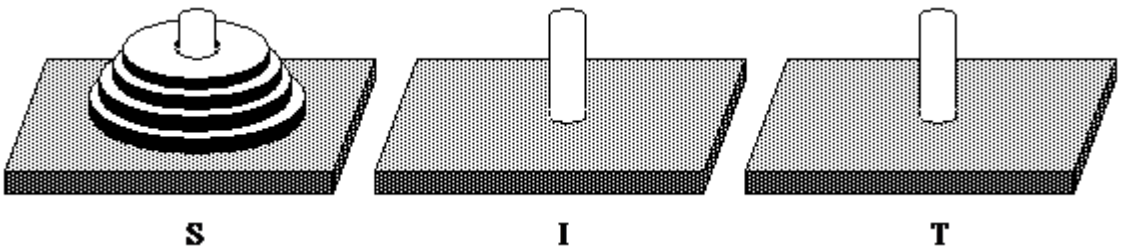

Exhibit 5.8: Initial configuration of the Tower of Hanoi.

Step 1 deserves more explanation. How do we transfer the n – 1 topmost disks from one peg to another? Notice that they themselves constitute a tower, to which we may apply the same three-step algorithm. Thus we are presented with successively simpler problems to solve, namely, transferring the n – 1 topmost disks from one peg to another, for decreasing n, until finally, for n = 0, we do nothing.

```
procedure Hanoi(n: integer; x, y, z: peg);
   { transfer a tower with n disks from peg x, via y, to z }
begin
   if  n > 0  then  { Hanoi(n - 1, x, z, y);  move(x, z);  Hanoi(n -
1, y, x, z) }
end;
```

Recursion has the advantage of intuitive clarity. Elegant and efficient as this solution may be, there is some complexity hidden in the bookkeeping implied by recursion.

The following procedure is an equally elegant and more efficient *iterative* solution to this problem. It assumes that the pegs are cyclically ordered, and the target peg where the disks will first come to rest depends on this order and on the parity of n (Exhibit 5.9). For odd values of n, 'IterativeHanoi' moves the tower to peg I, for even values of n, to peg T.

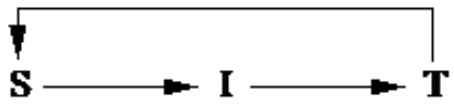

Exhibit 5.9: Cyclic order of the pegs.

```
procedure IterativeHanoi(n: integer);
var  odd: boolean;  { odd represents the parity of the move }
begin
   odd := true;
   repeat
      case  odd  of
        true:  transfer smallest disk cyclically to next peg;
        false: make the only legal move leaving the smallest in place
      end;
      odd := not odd
   until  entire tower is on target peg
end;
```

Exercise: recursive or iterative pictures?

Chapter 4 presented some beautiful examples of recursive pictures, which would be hard to program without recursion. But for simple recursive pictures iteration is just as natural. Specify a convenient set of graphics primitives and use them to write an iterative procedure to draw Exhibit 5.10 to a nesting depth given by a parameter d.

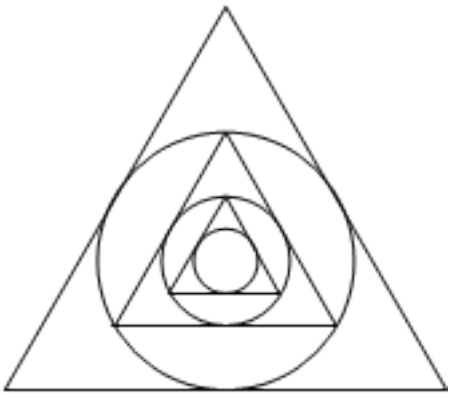

Exhibit 5.10: Interleaved circles and equilateral triangles cause the radius to be exactly halved at each step.

Solution

There are many choices of suitable primitives and many ways to program these pictures. Specifying an equilateral triangle by its center and the radius of its circumscribed circle simplifies the notation. Assume that we may use the procedures:

```
   procedure circle(x, y, r: real); { coordinates of center and
radius }
   procedure equitr(x, y, r: real); { center and radius of
circumscribed circle}
```

```
procedure citr(x, y, r: real; d: integer);
var  vr: real;  { variable radius }
     i: integer;
begin
   vr := r;
   for i := 1 to d do  { equitr(x, y, vr);  vr := vr/2;  circle(x, y,
vr) }
   { show that the radius of consecutively nested circles gets
     exactly halved at each step }
end;
```

The flag of Alfanumerica: an algorithmic novel on iteration and recursion

In the process of automating its flag industry, the United States of Alfanumerica announced a competition for the most elegant program to print its flag:

```
                ********** k blanks followed by k stars
      ********      ******** twice (k/2 blanks followed by k/2 stars)
  ****    ****    ****    **** …
 **  **  **  **  **  **  **  ** continue doubling and halving
* * * * * * * * * * * * * * * down to runs length of 1.
```

All solutions submitted to the prize committee fell into one of two classes, the iterative and recursive programs. The proponents of these two algorithm design principles could not agree on a winner, and the selection process sparked a civil war that split the nation into two: the Iterative States of Alfanumerica (ISA) and the Recursive States of Alfanumerica (RSA). Both nations fly the same flag but use entirely different production algorithms.

1. Write a

   `procedure ISA(k: integer);`

   to print the ISA flag, using an iterative algorithm, of course. Assume that k is a power of 2 and k ≤ (half the line length of the printer).
2. Explain why the printer industry in RSA is much more innovative than the one in ISA. All modern RSA printers include operations for positioning the writing head anywhere within a line, and line feed works both forward and backward.
3. Specify the precise operations for some RSA printer of your design. Using these operations, write a recursive

   `procedure RSA(k: integer);`

   to print the RSA flag.
4. Explain an unforeseen consequence of this drive to automate the flag industry of Alfanumerica: In both ISA and RSA, a growing number of flags can be seen fluttering in the breeze turned around by 90°.

Exercises

1. Whereas divide-and-conquer algorithms usually attempt to divide the data in equal halves, the recursive Tower of Hanoi procedure presented in the section 'Recursion versus iteration: The Tower of Hanoi" divides the data in a very asymmetric manner: a single disk versus n – 1 disks. Why?
2. Prove by induction on n that the iterative program 'IterativeHanoi' solves the problem in 2^n-1 iterations.