Thinking Recursively

ERIC ROBERTS

Thinking Recursively

ERIC ROBERTS
·partment of Computer Science
·llesley College, Wellesley, Mass.

JOHN WILEY & SONS, INC.
New York · Chichester · Brisbane · Toronto · Singapore

Library of Congress Cataloging in Publication Data:

Roberts, Eric.
Thinking recursively.

Bibliography: p.
Includes index.
1. Recursion theory. I. Title.
QA9.6.R63 1985 5113 85-20365
ISBN 0-471-81652-3

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

Preface

In my experience, teaching students to use recursion has always been a difficult task. When it is first presented, students often react with a certain suspicion to the entire idea, as if they had just been exposed to some conjurer's trick rather than a new programming methodology. Given that reaction, many students never learn to apply recursive techniques and proceed to more advanced courses unable to write programs which depend on the use of recursive strategies. This book is intended to demystify this material and encourage the student to "think recursively."

This book is intended for use as a supplementary text in an intermediate course in data structures, but it could equally well be used with many other courses at this level. The only prerequisite for using this text is an introductory programming course. Since Pascal is used in the programming examples, the student must also become familiar with Pascal programming, although this can easily be included as part of the same course. To support the concurrent presentation of Pascal and the material on recursion, the programming examples in the early chapters require only the most basic features of Pascal.

In order to develop a more complete understanding of the topic, it is important for the student to examine recursion from several different perspectives. Chapter 1 provides an informal overview which examines the use of recursion outside the context of programming. Chapter 2 examines the underlying mathematical concepts and helps the student develop an appropriate conceptual model. In particular, this chapter covers mathematical induction and computational complexity in considerable detail. This discussion is designed to be nonthreatening to the math-anxious student and, at the same time, include enough formal structure to emphasize the extent to which computer science depends on mathematics for its theoretical foundations.

Chapter 3 applies the technique of recursive decomposition to various mathematical functions and begins to show how recursion is represented in Pascal. Chapter 4 continues this discussion in the context of recursive procedures, emphasizing the parallel between recursive decomposition and the more familiar technique of stepwise refinement.

Chapters 5 through 9 present several examples of the use of recursion to solve increasingly sophisticated problems. Chapter 7 is of particular importance and covers recursive sorting techniques, illustrating the applicability of the recursion methodology to practical problem domains. Chapter 9 contains many delightful examples, which make excellent exercises and demonstrations if graphical hardware is available.

Chapter 10 examines the use of recursive procedures in the context of recursive data structures and contains several important examples. Structurally, this chapter appears late in the text primarily to avoid introducing pointers prematurely. For courses in which the students have been introduced to pointers relatively early, it may be useful to cover the material in Chapter 10 immediately after Chapter 7.

Finally, Chapter 11 examines the underlying implementation of recursion and provides the final link in removing the mystery. On the other hand, this material is not essential to the presentation and may interfere with the student's conceptual understanding if presented too early.

I am deeply grateful for the assistance of many people who have helped to shape the final form of the text. I want to express a special note of appreciation to Jennifer Friedman, whose advice has been invaluable in matters of both substance and style. I would also like to thank my colleagues on the Wellesley faculty, Douglas Long, K. Wojtek Przytula, Eleanor Lonske, James Finn, Randy Shull, and Don Wolitzer for their support. Joe Buhler at Reed College, Richard Pattis at the University of Washington, Gary Ford at the University of Colorado at Colorado Springs, Steve Berlin at M.I.T., and Suzanne Rodday (Wellesley '85) have all made important suggestions that have dramatically improved the final result.

Eric Roberts

Contents

The Idea of Recursion

1

Of all ideas I have introduced to children, recursion stands out as the one idea that is particularly able to evoke an excited response.
—*Seymour Papert,* Mindstorms

At its essence, computer science is the study of problems and their solutions. More specifically, computer science is concerned with finding systematic procedures that guarantee a correct solution to a given problem. Such procedures are called *algorithms.*

This book is about a particular class of algorithms, called *recursive algorithms,* which turn out to be quite important in computer science. For many problems, the use of recursion makes it possible to solve complex problems using programs that are surprisingly concise, easily understood, and algorithmically efficient. For the student seeing this material for the first time, however, recursion appears to be obscure, difficult, and mystical. Unlike other problem-solving techniques which have closely related counterparts in everyday life, recursion is an unfamiliar idea and often requires thinking about problems in a new and different way. This book is designed to provide the conceptual tools necessary to approach problems from this recursive point of view.

Informally, *recursion* is the process of solving a large problem by reducing it to one or more *subproblems* which are (1) identical in structure to the original problem and (2) somewhat simpler to solve. Once that original subdivision has been made, the same decompositional technique is used to divide each of these subproblems into new ones which are even less complex. Eventually, the subproblems become so simple that they can be then solved without further subdivision, and the complete solution is obtained by reassembling the solved components.

1-1 An Illustration of the Recursive Approach

Imagine that you have recently accepted the position of funding coordinator for a local election campaign and must raise $1000 from the party faithful. In this age of political action committees and direct mail appeals, the easiest approach is to find a single donor who will contribute the entire amount. On the other hand, the senior campaign strategists (fearing that this might be interpreted as a lack of commitment to democratic values) have insisted that the entire amount be raised in contributions of exactly \$1. How would you proceed?

Certainly, one solution to this problem is to go out into the community, find 1000 supporters, and solicit \$1 from each. In programming terms, such a solution has the following general structure.

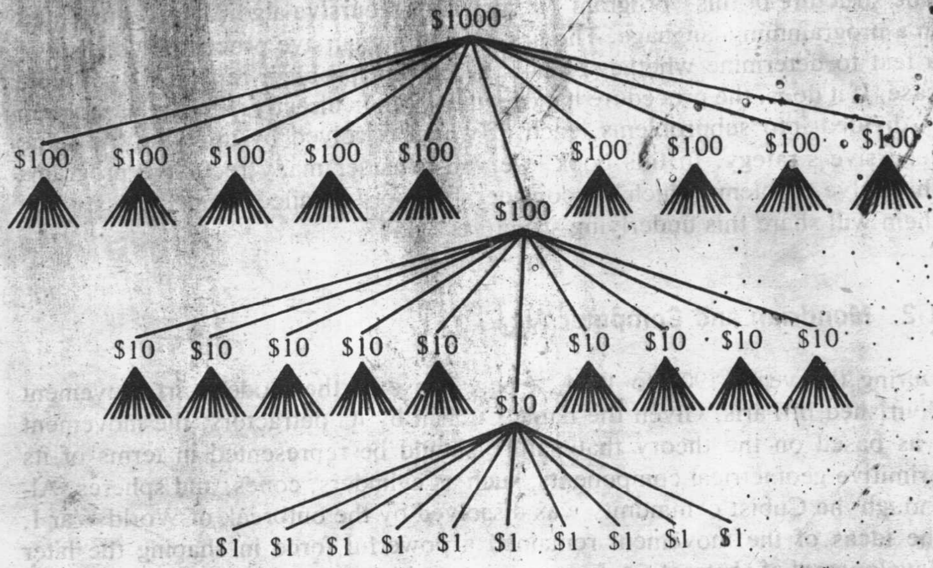
```
PROCEDURE COLLECT1000;
  BEGIN FOR I := 1 TO 1000 DO
    Collect one dollar from person I
  END;
```

Since this is based on an explicit loop construction, it is called an *iterative* solution.

Assuming that you can find a thousand people entirely on your own, this solution would be effective, but it is not likely to be easy. The entire process would be considerably less exhausting if it were possible to divide the task into smaller components, which can then be delegated to other volunteers. For example, you might enlist the aid of ten people and charge each of them with the task of raising \$100. From the perspective of each volunteer, the new problem has exactly the same form as the original task. The only thing which has changed is the dimension of the problem. Instead of collecting \$1000, each volunteer must now collect only \$100—presumably a simpler task.

The essence of the recursive approach lies in applying this same decomposition repeatedly at each stage of the solution. Thus, each volunteer who must collect \$100 finds ten people who will raise \$10 each. Each of these, in turn, finds ten others who agree to raise \$1. At this point, however, we can adopt a new strategy. Since the campaign can accept \$1 contributions, the problem need not be subdivided further into dimes and pennies, and the volunteer can simply contribute the necessary dollar. In the parlance of recursion, \$1 represents a *simple case* for the fund-raising problem, which means that it can be solved directly without further decomposition.

Solutions which operate in this way are often referred to as "divide-and-conquer" strategies, since they depend on splitting a problem into more manageable components. The original problem divides to form several simpler subproblems, which, in turn, "branch" into a set of simpler ones, and so on, until the simple cases are reached. If we represent this process diagrammatically, we obtain a *solution tree* for the problem:



In order to represent this algorithm in a form more suggestive of a programming language, it is important to notice that there are several different *instances* of a remarkably similar problem. In the specific case shown here, we have the independent tasks "collect $1000", "collect $100", "collect $10", and "collect $1", corresponding to the different levels of the hierarchy. Although we could represent each of these as a separate procedure, such an approach would fail to take advantage of the structural similarity of each problem. To exploit that similarity, we must first generalize the problem to the task of collecting, not some specific amount, but an undetermined sum of money, represented by the symbol N.

The task of collecting N dollars can then be broken down into two cases. First, if N is $1, we simply contribute the money ourselves. Alternatively, we find ten volunteers and assign each the task of collecting one-tenth the total revenue. This structure is illustrated by the procedure skeleton shown below:

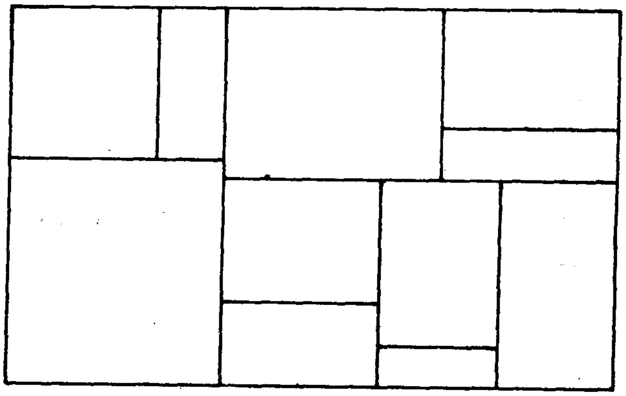
```
PROCEDURE COLLECT(N);
  BEGIN
    IF N is $1 THEN
      Contribute the dollar directly
    ELSE
      BEGIN
        Find 10 people;
        Have each collect N/10 dollars;
        Return the money to your superior
      END
  END;
```

The structure of this "program" is typical of recursive algorithms represented in a programming language. The first step in a recursive procedure consists of a test to determine whether or not the current problem represents a simple case. If it does, the procedure handles the solution directly. If not, the problem is divided into subproblems, each of which is solved by applying the same recursive strategy. In this book, we will consider many recursive programs that solve problems which are considerably more detailed. Nonetheless, all of them will share this underlying structure.

1·2 Mondrian and Computer Art

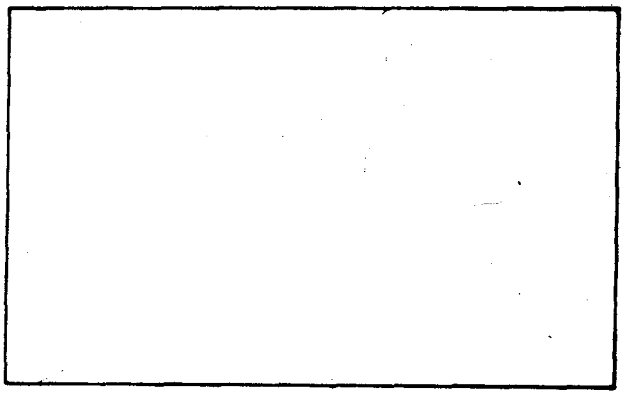
During the years 1907 to 1914, a new phase of the modern art movement flourished in Paris. Given the name Cubism by its detractors, the movement was based on the theory that nature should be represented in terms of its primitive geometrical components, such as cylinders, cones, and spheres. Although the Cubist community was dissolved by the outbreak of World War I, the ideas of the movement remained a powerful force in shaping the later development of abstract art. In particular, Cubism strongly influenced the work of the Dutch painter Piet Mondrian, whose work is characterized by rigidly geometrical patterns of horizontal and vertical lines.

The tendency in Mondrian's work toward simple geometrical structure makes it particularly appropriate for computer simulation. Many of the early attempts to generate "computer art" were based on this style. Consider, for example the following abstract design:

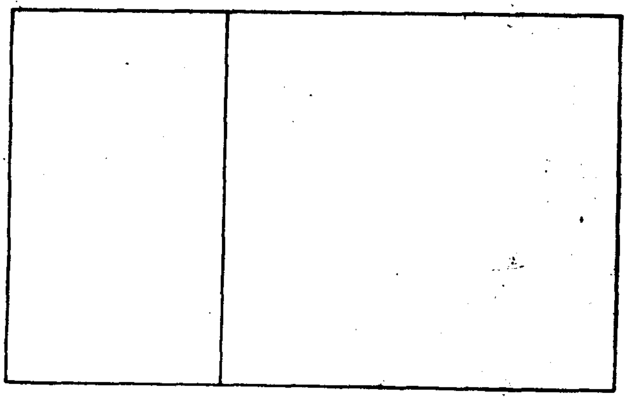


In this example, the design consists of a large rectangle broken up into smaller rectangles by a sequence of horizontal and vertical lines.

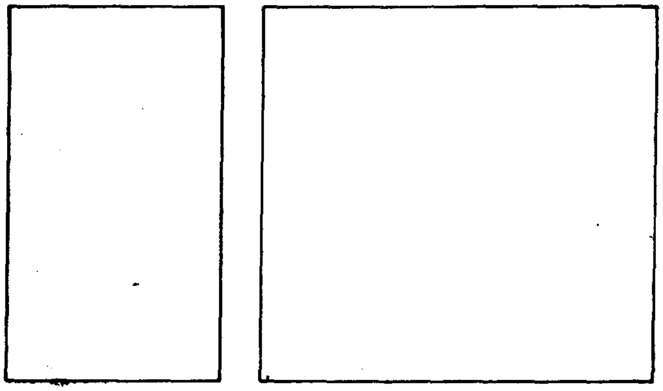
For now, we will limit our concern to finding a general strategy for generating a design of this sort and will defer the details of the actual program until Chapter 9, where this example is included as an exercise. To discover this strategy and understand how recursion is involved, it helps to go back to the beginning and follow this design through its evolutionary history. As with any work of art (however loosely the term is applied in this case), our design started as an empty rectangular "canvas":



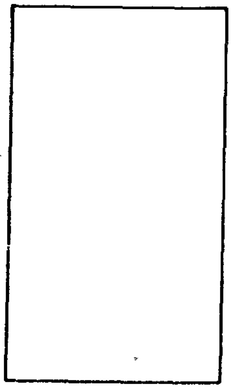
The first step in the process was to divide this canvas into two smaller rectangles with a single vertical line. From the finished drawing, we can see that there is only one line which cuts across the entire canvas. Thus, at some point early on in the history of this drawing, it must have appeared as follows:



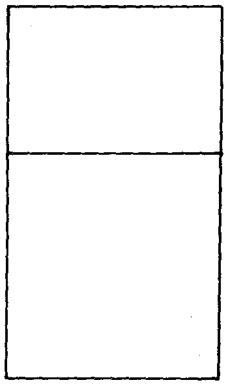
But now what? From here the simplest way to proceed is to consider each of the two remaining rectangles as a new empty canvas, admittedly somewhat smaller in size. Thus, as part of the process of generating a "large" Mondrian drawing, we have reduced our task to that of generating two "medium-sized" drawings, which, at least insofar as the recursive process is concerned, is a somewhat simpler task.



As a practical matter, we must choose one of these two subproblems first and work on it before returning to the other. Here, for instance, we might choose to work on the left-hand "subcanvas" first and, when we finish with that, return to finish the right-hand one. For the moment, however, we can forget about the right-hand part entirely and focus our attention on the left-hand side. Conceptually, this is a new problem of precisely the original form. The only difference is that our new canvas is smaller in size.



Once again, we start by dividing this into two subproblems. Here, since the figure is taller than it is wide, a horizontal division seems more appropriate, which gives us:



Just as before, we take one of these smaller figures and leave the other aside for later. Note that, as we proceed, we continue to set aside subproblems for future solution. Accumulating a list of unfinished tasks is characteristic of recursive processes and requires a certain amount of bookkeeping to ensure that all of these tasks do get done at some point in the process. Ordinarily, the programmer need not worry about this bookkeeping explicitly, since it is performed automatically by the program. The details of this process are discussed in Chapter 5 and again in Chapter 11.

Eventually, however, as the rectangles become smaller and smaller, we reach a point at which our aesthetic sense indicates that no further subdivision is required. This constitutes the "simple case" for the algorithm—when a rectangle drops below a certain size, we are finished with that subproblem and must return to take care of any uncompleted work. When this occurs, we simply consult our list of unfinished tasks and return to the one we most recently set aside, picking up exactly where we left off. Assuming that our recursive solution operates correctly, we will eventually complete the entire list of unfinished tasks and obtain the final solution.

1-3 Characteristics of Recursive Algorithms

In each of the examples given above, finding simpler subproblems within the context of a larger problem was a reasonably easy task. These problems are naturally suited to the divide-and-conquer strategy, making recursive solutions particularly appropriate.

In most cases, the decision to use recursion is suggested by the nature of the problem itself.* To be an appropriate candidate for recursive solution, a problem must have three distinct properties:

*At the same time, it is important to recognize that "recursiveness" is a property of the *solution* to a problem and not an attribute of the problem itself. In many cases, we can take a problem which seems recursive in its structure and choose to employ an iterative solution. Similarly, recursive techniques can be used to solve problems for which iteration appears more suitable.

1. It must be possible to decompose the original problem into simpler instances of the same problem.
2. Once each of these simpler subproblems has been solved, it must be possible to combine these solutions to produce a solution to the original problem.
3. As the large problem is broken down into successively less complex ones, those subproblems must eventually become so simple that they can be solved without further subdivision.

For a problem with these characteristics, the recursive solution follows in a reasonably straightforward way. The first step consists of checking to see if the problem fits into the "simple case" category. If it does, the problem is solved directly. If not, the entire problem is broken down into new subsidiary problems, each of which is solved by a recursive application of the algorithm. Finally, each of these solutions is then reassembled to form the solution to the original problem.

Representing this structure in a Pascal-like form gives rise to the following *template* for recursive programs:

```
PROCEDURE SOLVE(instance);
  BEGIN
    IF instance is easy THEN
      Solve problem directly
    ELSE
      BEGIN
        Break this into new instances I1, I2, etc.;
        SOLVE(I1); SOLVE(I2); . . . and so forth . . .;
        Reassemble the solutions
      END
  END;
```

1-4 Nonterminating Recursion

In practice, the process of ensuring that a particular decomposition of a problem will eventually result in the appropriate simple cases requires a certain amount of care. If this is not done correctly, recursive processes may get locked into cycles in which the simple cases are never reached. When this occurs, a recursive algorithm fails to *terminate*, and a program which is written in this way will continue to run until it exhausts the available resources of the computer.

For example, suppose that the campaign fund raiser had adopted an even lazier attitude and decided to collect the $1000 using the following strategy:

Find a single volunteer who will collect $1000.

If this volunteer adopts the same strategy, and every other volunteer follows in like fashion, the process will continue until we exhaust the available pool of volunteers, even though no money at all will be raised. A more fanciful example of this type of failure is shown in the following song.

THERE'S A HOLE IN THE BUCKET

Traditional

There's a hole in the bucket, dear Liza, dear Liza
There's a hole in the bucket, dear Liza, a hole
Then fix it, dear Charlie, dear Charlie
Then fix it, dear Charlie, dear Charlie, fix it

With what shall I fix it, dear Liza, dear Liza
With a straw, dear Charlie, dear Charlie

But the straw is too long, dear Liza, dear Liza
Then cut it, dear Charlie, dear Charlie

With what shall I cut it, dear Liza, dear Liza
With a knife, dear Charlie, dear Charlie

But the knife is too dull, dear Liza, dear Liza
Then sharpen it, dear Charlie, dear Charlie

With what shall I sharpen it, dear Liza, dear Liza
With a stone, dear Charlie, dear Charlie

But the stone is too dry, dear Liza, dear Liza
Then wet it, dear Charlie, dear Charlie

With what shall I wet it, dear Liza, dear Liza
With water, dear Charlie, dear Charlie

But how shall I fetch it, dear Liza, dear Liza
In a bucket, dear Charlie, dear Charlie

There's a hole in the bucket, dear Liza, dear Liza,
There's a hole in the bucket, dear Liza, a hole

1-5 Thinking about Recursion—Two Perspectives

The principal advantage of recursion as a solution technique is that it provides an excellent mechanism for managing complexity. No matter how difficult a problem at first appears, if we can determine a way to break that problem down into simpler problems of the same form, we can define a strategy for producing a complete solution. As programmers, all we need to specify is (1) how to simplify a problem by recursive subdivision, (2) how to solve the simple cases, and (3) how to reassemble the partial solutions.

For someone who is just learning about recursion, it is very hard to believe that this general strategy is powerful enough to solve a complex problem. Given a particular problem, it is tempting to insist on seeing the solution in all its gory detail. Unfortunately, this has the effect of reintroducing all the complexity that the recursive definition was designed to conceal. By giving in to skepticism, the usual result is that one takes a hard problem made simpler through recursion and proceeds to make it difficult again. Clearly, this is not the optimal approach and requires finding a new way to think about recursion.

The difference between the perspective of the programmer with considerable experience in recursion and that of the novice is perhaps best defined in terms of the philosophical contrast between "holism" and "reductionism." In *Godel, Escher, Bach,* Douglas Hofstadter defines these concepts by means of the following dialogue:

Achilles I will be glad to indulge both of you, if you will first oblige me, by telling me the meaning of these strange expressions, "holism" and "reductionism."

Crab Holism is the most natural thing in the world to grasp. It's simply the belief that "the whole is greater than the sum of its parts." No one in his right mind could reject holism.

Anteater Reductionism is the most natural thing in the world to grasp. It's simply the belief that "a whole can be understood completely if you understand its parts, and the nature of their 'sum'." No one in her left brain could reject reductionism.

Even though recursion acts as a reductionistic process in the sense that each problem is reduced to a sum of its parts, writing recursive programs tends to require a holistic view of the process. It is the big picture which is important, not the details. In developing a "recursive instinct," one must learn to stop analyzing the process after the first decomposition. The rest of the problem will take care of itself, and the details tend only to confuse the issue. When one cannot see the forest for the trees, it is of very little use to examine the branches, twigs, and leaves.

For beginners, however, this holistic perspective is usually difficult to maintain. The temptation to look at each level of the process is quite strong, particularly when there is doubt about the correctness of the algorithm. Overcoming that temptation requires considerable confidence in the general mechanism of recursion, and the novice has little basis for that confidence.