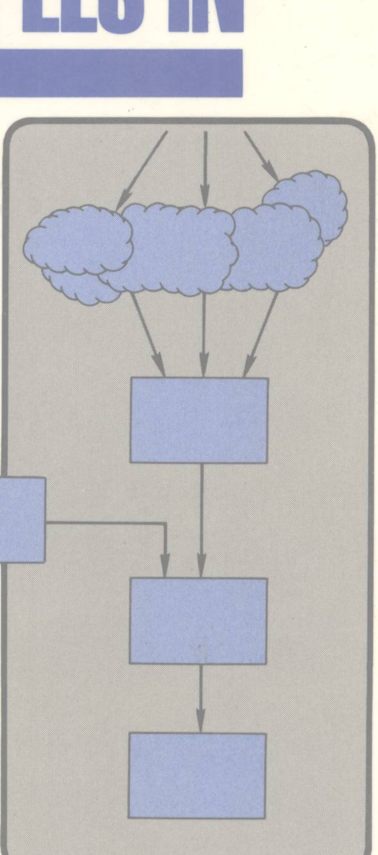
PROBLEM SOLVING METHODS WITH EXAMPLES IN Ada®

Nico Lomuto



PRENTICE-HALL SOFTWARE SERIES

PROBLEM SOLVING METHODS WITH EXAMPLES IN Ada®

Nico Lomuto

Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging-in-Publication Data

Lomuto, Nico (date)
Problem solving methods with examples in Ada.

Includes bibliographies.
1. Ada (Computer program language) 2. Problem solving—Data processing. I. Title.
QA76.73.A35L66 1986 005.13′3 86-12402
ISBN 0-13-721325-5

Editorial/production supervision: *Lisa Schulz*
Interior design: *Christine Wolf*
Cover design: *Lundgren Graphics, Ltd.*
Manufacturing buyer: *Gordon Osbourne*

Prentice-Hall Software Series
Brian W. Kernighan, Advisor

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-721325-5 025

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Prentice-Hall of Southeast Asia Pte. Ltd., *Singapore*

To Annie

PREFACE

Where do programming ideas come from? Why is it that, confronted with the same problem, one programmer will have to struggle to get started while another will effortlessly produce a solution so simple and elegant that its correctness is obvious and its efficiency leaves little to be desired? Is it knowledge of methodologies? Yes—to some extent. Mastery of special techniques? Possession of a special gift? Perhaps. But—whatever it is—can it be learned?

A few years ago, full of hope and optimism, I taught my first course in programming methodology to an audience of experienced programmers and engineers. At the end of the course a group of students, whose programs had shown a marked improvement, surprised me with the statement that what they had learned was intellectually interesting but of little practical consequence. What they had hoped to learn, it turned out, was not how to develop correctly an initial idea, but how to generate such an idea to begin with! For them, the overall programming activity was totally dominated by the invention phase, the creative part of the process. Not knowing how to get started was their bottleneck.

At first the request struck me as rather unreasonable. For complex problems there are specific techniques that one can teach, but for simple day-to-day exercises the idea materializes spontaneously, without the use of any explicit method! Or does it? As the discussion turned to a recent exercise, I managed to reconstruct in slow motion, so to speak, the seemingly instantaneous appearance of the initial idea. On that note, time ran out, and the course ended. But I was intrigued.

For months after that I kept observing how skilled problem solvers attack programming problems. To my surprise, I found that those solutions of baffling simplicity tend to be the outcome of a small set of very specific patterns. But I also became convinced that those patterns are of little use if not put in the context of good problem-solving habits. In other words, the *thinking patterns* used by skilled problem solvers are at least as important as the *solution patterns* that yield their programs.

About This Book

What you will find in this book is a collection of hints—mental tricks that can be used to produce a solution to a programming problem or to make progress toward it. The structure of the list is inspired by Polya's classic, *How To Solve It* (Princeton, NJ: Princeton University Press, 1957). The list in its present form is the result of several years of experience and refinement. All but the most obvious hints are phrased and illustrated specifically in programming terms; I feel that learning is much easier when each concept can be immediately applied!

Virtually all examples are based on real-life experiences, but many of them are presented simply as puzzles. Why? In real life, stripping a difficult problem of the smoke screen of detail and going straight to the heart of the difficulty is a crucial problem-solving step (Chapter 5). Abstracting from the real-life situation is critical not only to solving a particular problem, but also to learning from the experience (Chapter 4). Becoming comfortable with "toy" problems taken out of context is therefore a major goal in itself (Chapter 1).

Required Background

The reader is expected to have some programming knowledge, but not much in the way of experience or formal computer science education.

My ideal reader is either an experienced programmer with no formal computer science background or a novice who has just completed a first course in programming and is wondering what the "real world" looks like and how it relates to the well-behaved world of academic exercises.

Use of this Book

This book can be used as a supplement to the standard texts on systematic programming or by itself in a workshop. But most of all I hope it can be simply read for fun. I have deliberately kept the presentation informal in an attempt to convey the vividness and the sheer enjoyment of the intuitive process that precedes rigorous development. In a one-semester course, I like to demonstrate the various principles (rather than *talk* about them) throughout the course; toward the end I dedicate three or four two-hour lectures specifically to this topic.

Acknowledgments

It is impossible to remember all the talented people who, through their programming habits or remarks made in conversation, have unknowingly supplied material for the book.

Among those who have been close to the actual writing, I am indebted to Clem McGowan, who for years has been encouraging me to put my incoherent thoughts in the form of a book. I also owe an immense debt to Brian Kernighan, Charles Wetherell, and Jon Bentley for their insightful comments on an early draft of the manuscript.

After the completion of the first draft, just as I was ready to go to work on a revision, I had the unique distinction of being essentially dead. For having turned that nightmare into a bad memory I must thank the talent and dedication of numerous doctors, therapists, and nurses; the rapid reflexes of my friend Bobbie Hayes; but most of all the perseverance, dedication, and enormous intelligence of my wife, Annie. To her this book is dedicated.

Nico Lomuto

NOTES FOR THE READER

The discovery of a solution to a programming problem is a complex, iterative process in which devising the "logic" of the solution is usually a far more demanding task than expressing that logic in a particular programming language. In this book the programming language aspect is of very little interest, but a vehicle is needed to convey the essence of the solutions! In a few cases the solution is presented in several programming languages, but as a general approach this would be unsatisfactory. As a notation to express algorithms it is common to use a pseudolanguage, looking somewhat like most modern algorithmic languages, but with greater expressive power. This is essentially the book's approach, except that instead of going ahead and inventing yet another pseudolanguage I have chosen a real programming language—after all, this is the age of reusability! For the class of problems addressed in the book, Ada has the desired characteristics: it is sufficiently close to all other algorithmic languages to be understandable with a minimum of assistance, but it is sufficiently more powerful to prevent trivial details from obscuring the main points. It goes without saying that no knowledge of Ada is assumed, and enough explanations are given along the way.

Following a well-established tradition, I have marked with an asterisk those exercises and sections that can be safely skipped at first reading. The asterisk is not a warning of a highly esoteric topic included only to impress the author's colleagues, but rather an invitation to postpone a subject if its length or difficulty causes a considerable loss of pace (or interest). And that brings me to the last point:

Read this book quickly rather than intensely. It's been designed expressly to be read quickly. BUT—don't skip the little puzzles proposed in the narrative!

CONTENTS

A SAMPLER

chapter 0

Programming is problem solving. In the course of a program development, one invariably stumbles onto little problems such as the following.

(1) A list of numbers is stored in nondecreasing order in A(1), A(2), ..., A(N), where A is an integer array and N is an integer variable whose value is positive or zero. The problem is to eliminate duplicate entries (if any) from the list.

The problem has a straightforward solution (if you don't find the solution self-explanatory, see Fig. 0.1, or the Ada Notes at the back of the book):

(2)
```
if N > 1 then
   LAST_UNIQUE := 1;
   for I in 2..N
   loop
      if A(I) /= A(LAST_UNIQUE) then
         LAST_UNIQUE := LAST_UNIQUE + 1;
         A(LAST_UNIQUE) := A(I);
      end if;
   end loop;
   N := LAST_UNIQUE;
end if;
```

The list is compacted incrementally, in a loop. For each array component A(I) we check whether its value already appears in the already compacted region of the array; if not, we simply append it there. The literature is full of

ALGORITHM (2) IN SEVERAL LANGUAGES

(Only the algorithm is shown; unrelated parts of the program have been omitted.)

BASIC

```
50  IF N <= 1 THEN 130
60      LET U = 1
70      FOR I = 2 TO N
80          IF A(U) = A(I) THEN 110
90              LET U = U + 1
100             LET A(U) = A(I)
110     NEXT I
120     LET N = U
130 REM
```

C

```
if (n > 1) {
    last_unique = 1;
    for (i = 2; i <= n; i++)
        if (a[i] != a[last_unique])
            a[++last_unique] = a[i];
    n = last_unique;
}
```

FORTRAN IV

```
1000 IF (N.LE.1) GO TO 2000
         LAST = 1
1100     DO 1199 I = 2,N
             IF ( A(LAST).EQ.A(I) ) GO TO 1199
                 LAST = LAST + 1
                 A(LAST) = A(I)
1199     CONTINUE
         N = LAST
2000 CONTINUE
```

FORTRAN 77

```
1000 IF (N.GT.1) THEN
         LAST = 1
1100     DO 1199 I = 2,N
             If ( A(LAST).NE.A(I) ) THEN
                 LAST = LAST + 1
                 A(LAST) = A(I)
             END IF
1199     CONTINUE
         N = LAST
     END IF
```

Figure 0.1

Pascal

```
if N> 1 then
  begin
    LastUnique := 1;
    for I := 2 to N do
      if A[LastUnique] <> A[I] then
        begin
          LastUnique := LastUnique + 1;
          A[LastUnique] := A[I]
        end;
    N := LastUnique
  end
```

Figure 0.1 (continued)

hints on how such a simple idea could be correctly refined into a bug-free program, into a highly readable program, or into a superefficient program. But where does such an idea come from?

The rest of this section is a slow-motion walk through the invention of algorithm (2). Phrases in italics are specific problem-solving patterns ("heuristics") that will be discussed in detail in the body of the book.

The first question, of course, is: Where do you start? Systematic problem solving always starts in the same way: *understand the problem.* Before we attack, we want to "size up" the opponent. And for that we use certain specific tricks. First, we *enumerate the inputs, disregarding what has to be done with them:*

- The variable N.
- The array A.

And we do the same for the outputs:

- The variable N, but with a new value in it. Let's call the new value N'.
- The array A, also with a new value, say A'.

Now we relate the inputs to the outputs:

- A' has the same set of values as A, but each value is represented only once. N' is equal to N minus the number of redundant values.

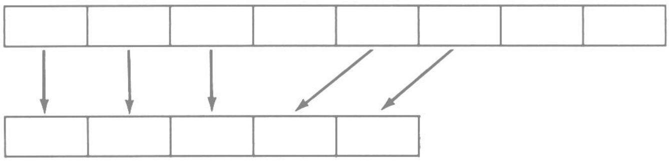
This last point is not completely straightforward, so we *work out an example.* Let's pick a possible set of initial values. Let's say that N has value 8, and the first eight components of A are:

1	3	7	7	9	37	37	37

On exit we want the list to contain:

1	3	7	9	37

Therefore we want N to have value 5. Next we will *draw a picture* of all this. We *draw the inputs and the outputs as separate* (although they are physically the same), and we *show the relationship between the two,* using arrows to indicate data movement:

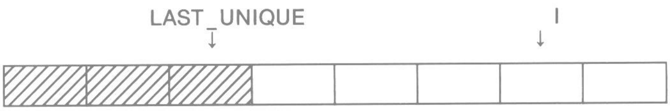


At this point a germ of an idea may begin to appear (although we haven't even started to consider possible solutions!). One thing that we notice immediately is that the arrows in the picture don't cross, and with a little thought we can convince ourselves that that is not a coincidence. Every array component will either remain in place (if it is not preceded by any duplicates) or move "to the left," that is, to a position having a smaller index; no array component will ever "move up." This matches perfectly a specific pattern (presented in Chapter 5 as *the wishful thinking strategy*) and tells us a very interesting thing:

> The compaction can be performed incrementally, "from left to right."

Picture yourself walking on the array from left to right. At each step, you look at the component that you are standing on. If it is a duplicate, you just move on; otherwise you append it to the region that you have already compacted.

Now we have an idea that we think will work, so we *develop the idea,* using certain systematic methods. First we once again *draw a picture,* this time of the solution (or at least of however much of it we understand at this point). To mark where we are on the array, we use an arrow (which, of course, will be implemented as a variable, say I). We also need to mark where the compacted region "behind us" ends; here we introduce another arrow, LAST_UNIQUE:



To reinforce our understanding, let us repeat (in slightly greater detail than before) what we intend to do as soon as we land on a new "I": if A(I) is a duplicate, we move to the next I (that is, I+1); otherwise, we increment LAST_UNIQUE, and move A(I) back there. (Note how at this stage we are barely beginning to introduce programming details into our thinking, which remains largely visual and informal).

For brevity I will not describe how the loop structure and the initialization are derived (that is the most systematic part of the development and will be discussed in Chapter 3). Having written down algorithm (2), we will *look back,* partly to verify the correctness of our solution, but mostly to learn from our experience. This last step would also be guided by specific heuristics.

Exercise 1

Looking back is the most important practice in becoming a better and better problem solver. Try to "digest" aligorithm (2). First, put the book aside and *try to reinvent* the algorithm. You will not necessarily succeed the first time, but that's normal. To become better acquainted with the algorithm, *desk-check some special cases.* Try the empty list (N=O, that is) and a list with only one element (N=1); does the algorithm do anything strange? Now try a list with no duplicates; what happens? Next, try a list that consists of nothing but duplicates—for example:

5	5	5	5	5	5	5

(with N=7); does the algorithm reduce N to 1 and leave A(1) set to 5? Now try again: Set the book aside and reinvent the algorithm.

Exercise 2*

The well-known quicksort algorithm [1] generates the following subproblem. Given two integer variables M and N, with M<N, and a sequence of numbers stored in random order in A(M), A(M+1), ..., A(N), rearrange the sequence so that all values less than or equal to a certain X precede those greater than X. This problem has a solution very similar in spirit to algorithm (2). A recipe for that class of solutions will be given in Chapter 5, but you might enjoy trying your hand at it now.

I hope that this brief example has given you an idea of what problem-solving methods can do for you. By definition, problem-solving methods are concerned with situations where no infallible "cookbook" techniques can be applied to the achievement of a certain goal, and all that one has to go by is trial-and-error, experience, optimism and—why not—luck. What a method can give you is a systematic way to conduct your search, and a number of "heuristics" that tend to point you in the right direction. As you may have noticed in the example, problem solving is largely a matter of "stirring up the waters" to generate ideas, and knowing how to grab a good idea when it presents itself. When an idea fails to surface, the reason is almost invariably that either the stirring has not been sufficiently vigorous or the grabbing has not been sufficiently quick. In this book you will find a catalog of ideas (solution patterns), but also a great deal of material about the stirring and the grabbing. For example. . . .

0.1 A Toy Problem?

Problem (1) is clearly a "toy" example. In real life, problems are not always so simple, and almost never so tidy. Here is an example. The context is an interactive simulation program, originally written in FORTRAN in the early 1960s and continually modified ever since. The user describes a transmission system and requests that certain calculations be performed. The system to be simulated is described as a block diagram. The program recognizes a large number of block types; the system layout is defined with reference to a 10 × 50 grid (Fig. 0.2). A typical user input might look as follows:

```
BLOCK 14: AMPLIFIER, 2.3DB
```

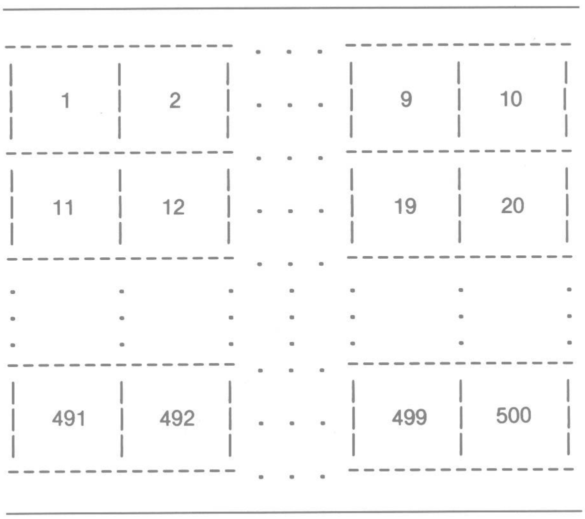


Figure 0.2