University of Wollongong

Research Online

Department of Computing Science Working Paper Series

Faculty of Engineering and Information Sciences

1981

Before programming - on teaching introductory computing

R. Geoff Dromey
University of Wollongong - Dubai Campus

Follow this and additional works at: https://ro.uow.edu.au/compsciwp

Recommended Citation

Dromey, R. Geoff, Before programming - on teaching introductory computing, Department of Computing Science, University of Wollongong, Working Paper 81-6, 1981, 11p.
https://ro.uow.edu.au/compsciwp/17

Research Online is the open access institutional repository for the University of Wollongong. For further information contact the UOW Library: research-pubs@uow.edu.au

THE UNIVERSITY OF WOLLONGONG

BEFORE PROGRAMMING - ON TEACHING INTRODUCTORY COMPUTING

By

R. Geoff Dromey

Preprint 81/6

DEPARTMENT OF COMPUTING SCIENCE

P.O. BOX 1144, WOLLONGONG, N.S.W. 2500, AUSTRALIA

BEFORE PROGRAMMING – ON TEACHING INTRODUCTORY COMPUTING

R. Geoff Dromey
Department of Computing Science,
The University of Wollongong,
Post Office Box 1144,
Wollongong, N.S.W. 2500
Australia.

ABSTRACT

In comparison with most other human intellectual activities, computing is in its infancy despite the progress we seem to have made in such a short time. Consequently, there has been insufficient time for the evolution of "best ways" to transmit computing concepts and skills. It is therefore prudent to look to more mature disciplines for some guidelines on effective ways to introduce computing to beginners. In this respect the discipline of teaching people to read and write in a natural language is highly relevant. A fundamental characteristic of this latter discipline is that a substantial amount of time is devoted to teaching people to read **long before** they are asked to write stories, essays, etc. In teaching computing we seem to have overlooked or neglected what corresponds to the reading stage in the process of learning to read and write. In the discussion which follows we will look at ways of economically giving students the "*computer-reading experience*" and preparing them for the more difficult tasks of algorithm design and computer problem-solving.

Keywords: Visible program execution, introductory computing, problem solving, program reading, computer-aided-learning.

CR Categories: 1.50, 1.51, 1.52

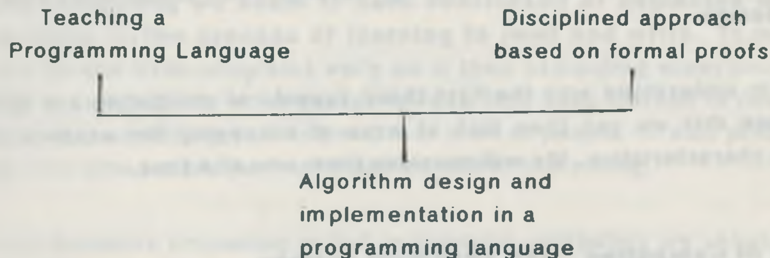
BEFORE PROGRAMMING –
ON TEACHING INTRODUCTORY COMPUTING*

R. Geoff Dromey

Department of Computing Science,
The University of Wollongong,
Post Office Box 1144,
Wollongong, N.S.W. 2500
Australia.

1. Introduction

Introductory computing science courses employ a wide spectrum of different emphases which in turn meet with an equally wide range of degrees of success. It is reasonable to expect that the degree of success we have in teaching computing will be considerably influenced by the way the subject is introduced. The limits and middle ground of the spectrum of approaches taken to teach introductory computing are outlined below:



It is clear that the best students survive and flourish under any of these schemes. However, we are now in a situation where it is highly desirable that computing skills be imparted to a larger pool of students with a considerably wider range of backgrounds and abilities. If we are to be successful in teaching computing to this latter group, we need to take a lot more care in how we introduce them to the discipline. At present we seem to have neither a well-defined philosophy nor a strategy for teaching introductory computing that recognizes these needs.

* The words "programming" and "computing" are used synonomously to describe algorithm design and implementation.

2. The Nature of Computing

Any philosophy or teaching strategy we propose must recognize a number of aspects inherent in computing that are most likely to be new to the beginning student. These characteristics are:

(1) That computing is an active rather than a passive discipline.

(2) That algorithms and programs possess a dynamic nature as well as a static one.

(3) That the process of solving computing problems generally requires a creative step at the outset.

(4) That even simple computing problems have an apparent logical complexity.

(5) That the goal in computing is designing and implementing demonstrably correct, reliable software, that is economical in human and computing resources.

(6) That the way information is represented plays an important role in solving computer problems.

There are certainly other issues that may be included in our list but they are not so central to our primary goal of achieving an understanding of computing for a wide spectrum of people.

Studying our list we see that contemporary teaching methods tend to emphasize the last three aspects as these are seen as the meat-of-the-matter in introductory computing [1-7]. In this paper we will take the stand that it is crucial to place a much greater emphasis on the first three aspects, particularly in the early stages of teaching introductory computing. If the foundations are not laid by careful and detailed study of the fundamental aspects of computing then we are likely to condemn many to the confused computer wilderness rather than unlocking doors to the broad and exciting horizons of computing.

The goal of the first three stages is to provide students with the skills and understanding that will enable them to embark on the demanding course of non-trivial computer problem-solving and algorithm design.

Let us now seek to understand why the first three aspects of computing are so important and when we have done this we can then look at ways of conveying the essence of what is involved in these three characteristics. We will consider them one at a time.

3. The Active Aspect of Computing – the swimming analogy

The most basic lesson to learn about computing is that it is a highly active rather than a passive discipline. In this respect learning computing is very much like learning to swim. Most of us would accept that no matter how many books we studied, nor how much advice or instructions we were given about swimming, we would progress very little until we actually got in the water and started trying to swim. The same is true of computing. We can listen to lectures and read books on computing endlessly but we will not begin to make real progress with our computing until we actively start to do things using a computer.

What we do in our initial encounters with a computer can, I believe, have a strong influence on whether or not we leap forward and start swimming with the computer tide or retreat backwards into a whirlpool of confusion, ignorance and fear of computers. Let us now therefore address the crucial problem of what should be our initial encounter with a computer to give us the best possible chance of catching the computer wave. This brings us to the dynamic aspect of computing.

4. The Dynamic Aspect of Computing - the reading and writing anology

In comparison with most other human intellectual activities, computing is in its infancy despite the progress we seem to have made in such a short time. Because the demand for computers, and people with computing skills is so great in our rapidly changing world, we have not had time to sit back and reflect on the best way to convey the "computing concept" on a wide scale. In time, computing will mature and evolve as a well-understood discipline with clear and well-defined methods for introducing the subject to beginners - but that is in the future. For the present, because we are not in this happy situation, it is most prudent to look to other mature disciplines for guidance on how we should introduce computing to beginners. Assuming we want to have the highest possible success rate in introducing computing to students (a situation which clearly at present does not prevail) and that we want to teach computing on the widest possible scale, then the most obvious discipline to turn to is that of **learning to read and write.**

Educators have mastered the process of teaching people to read and write to the extend that these skills are able to be transmitted on a very wide scale, efficiently, and with a very high rate of success. Putting aside the various methods within the learning-to-read-and-write discipline we see that there are some fundamental principles acknowledged by all methods that are highly relevant to teaching computing.

In teaching people to read and write a very substantial time (years in fact) is devoted to reading. Only after such preparation is it considered appropriate that they should attempt to write stories or essays, or books, etc. In fact, even before people attempt to learn to read they undergo several years of listening to language, speaking, and being "read-to". Clearly, in learning a language, the ultimate goal is to be able to verbalize and write fluently in that language. Similarly in computing the goal is to be able to program (or design and implement algorithms) effectively. In learning to read and write it has long been recognized that reading is easier and that it must precede writing by a considerable margin of time to allow the assimilation of enough models and constructs of text to make writing possible.

In teaching computing we seem to have overlooked or neglected what corresponds to the reading stage in the process of learning to read and write. To put it strongly, asking people to design and write programs early on in their computing experience is like expecting that they be able to competently write essays **before** they have learned to read or even to write short sentences - it is expecting just too much of a lot of people. It also probably explains why many otherwise very able people just don't get started in computing.

What we are therefore proposing is that in teaching computing we should draw as much as we can from the "learning-to-read-and-write" analogy. This means that we must be able to provide the beginning student with his "reading-experience" in computing before embarking on the more difficult algorithm design and program writing tasks which require considerably more creative effort, discipline, and technical skill.

At this point it is important to recognize that the "reading-experience" cannot be gained by sitting down with a book of programs and attempting to "read" them. The problem with this approach is that program instructions as written on a piece of paper are a **static** representation of a computer algorithm. As such they do not very fluently convey anything of the dynamic aspect of computer algorithms and programs. Nievergelt's work with XS0 attempts to convey something of the dynamic aspect of computing. [8]

5. Visible Program Execution - a picture is worth a thousand words

It can be argued that it is important to have a clear and in-depth understanding of the dynamic character of programs before attempting algorithm design and program implementation. What is needed is a practical and economical method of giving students their "reading-experience" in computing. To gain this experience the students need to "see" programs written in a high level language executing. Traditionally, all the student sees is the output printed on the terminal or line printer and prompts by the program when it requires input from the terminal. This level of "seeing" a program execute is unsatisfactory for beginners because it conveys very little about the program's flow of control or about the effects of individual and groups of program statements. What we need is much more explicit demonstrations of what is happening in a program that causes it to produce the outputs that it does. **A far more transparent way to do this is to place before the student on a terminal the text of the program being executed and then to trace the execution on the displayed program while at the same time dynamically updating on the screen the changes in program variables as they are made and related to steps in the displayed program.** This dynamic method of studying algorithms can greatly assist the student in acquiring a level of understanding necessary to design and implement his own algorithms.

The facilities and software tools needed to adequately demonstrate execution of a program in this way are relatively economical to provide. All that is needed is a terminal with cursor addressing facilities and a **small set of software tools** (several hundred lines of code) that can be inserted into the program being studied to make its execution "visible" on the screen. Only one procedure call per statement of the program being studied is needed to operate it in a visible mode (this software is available from the author). The display software is transparent to the user. The appropriate tools allow us to see a program execute in single-step mode while monitoring the changes in variables as they are dynamically displayed on the screen. The student can cause the program to execute the next statement at each stage by simply pressing a single key (e.g. the RETURN) on the terminal.

As an example, the screen layout for "visible" execution of a selection sorting procedure in Pascal is as illustrated in figure 1. The next statement to be executed at each stage is tracked by having an arrow that can be moved from statement to statement on the procedure text displayed on the screen. A more complete description of the software is given in the next section.

```
procedure selectionsort(a:nelements; n:integer);
var i {index for sorted part}, j {index for unsorted part},
   p {position of minimum}, min {minimum in unsorted part}:integer;

begin                                                ------          ------
   for i := 1 to n-1 do                      i  :  |  1 |    a[i] :  | 12 |
     begin                                           ------          ------
       min := a[j];                                  ------          ------
       p := i;                               j  :  |  2 |    a[j] :  |  4 |
       for j := i+1 to n do                          ------          ------
         if a[j] < min then                          ------          ------
           begin                             p  :  |  1 |    a[p] :  | 12 |
-----        min := a[j];                            ------          ------
| 7 |-->     p := j
-----      end;                                      ------          ------
         a[p] := a[i];                       n  :  |  8 |    min :   |  4 |
         a[i] := min                                 ------          ------
     end
end

array:  12    4    56    67    9    23    45
         i    j
                          --------
condition:  a[j] < min  :| true |
                          --------
```

Figure 1

The cause-and-effect nature of what is displayed when executing programs in this manner (automatic-single-step-mode) can accomplish a number of things.

* Most importantly, it provides a good understanding of the dynamic nature of algorithms - an essential pre-requisite if students are to later design and implement their own algorithms.
* It gives a deep insight into the basic laws of composition of computer algorithms (sequence, selection, iteration, and modularity) and how these concepts relate to the flow of control in programs).
* It is a useful vehicle for helping students to learn various programming constructs. Actually observing constructs being used within different programming contexts and linking these constructs to changes in the values of variables and conditions gives the student the necessary concrete demonstrations that will enable him to master the use of these constructs.
* It conveys in a very vivid fashion the workings of a particular algorithm. For example, the selection sort algorithm when visibly executed conveys the difficult concept of how we can have one loop executing within another loop. It also highlights the difference between subscripted variables and subscripts.
* It also provides an ideal tool for teaching students debugging techniques. That is, programs with logical bugs can be implemented and the student asked to diagnose the problem after studying the program in visible execution mode.

Visible program execution has the added advantage of being ideally suited for use in the lecture room, classroom, or library. Most terminals have the capability of video output which can be connected to a video recorder. Using these facilities it is very easy and cheap to monitor and record the visible program execution of a variety of programs for later use on video monitors in the classroom or library. This gives us a teaching aid far superior to handwritten blackboard examples when lecturing about algorithms in the classroom. We have recorded and used a number of programs in this way.

We can take the visible mode of program execution a step further by making it more **active** with respect to the student. At each step in the program's execution we can ask the student to supply the appropriate value of the variable or condition etc. How this works can be best illustrated by referring back to figure 1. The arrow is indicating that the 7th statement (i.e. p := j) is the next one to be executed. What the software can do is move the cursor to the "p" box on the screen and indicate to the student that he must supply the appropriate value for **p** (in this case 2) before the program will continue. If the user enters the wrong value the word ERROR will flash in the p-box. This will be followed by a flashing of the INPUT sign to prompt the user to again try to enter the proper **p** value. If the user gets it wrong twice the software flashes VALUE in the p-box and then supplies the user with the correct value, switches from interactive to automatic single step mode, and moves to the next program statement to be executed.

Using a visible program in interactive-single-step mode can in a very direct way reinforce the student's comprehension of how a program really works and what individual program statements accomplish. At the same time it can give the student very positive and direct feedback when he gets something wrong.

What we are therefore advocating is that students be given **considerable** exposure to visible "program-reading" in both the modes described as a preparatory step to considering the problem-solving aspect of computer algorithm design. Exposure to visible-program-execution (VPE) should be accompanied by a study of the basic laws of composition of computer algorithms (sequence, selection, iteration and modularity) including how these laws of form relate to the way programs are executed. Concurrently, a study should also be made of the syntax of the programming language being read.

6. Software for Visible Program Execution

Contrary to the needs of most Computer-Aided-Learning systems, the development needed to provide the software tools for visible program execution is relatively small (several hundred lines of code).

A straightforward way to design such a system is to work on the basis of inserting additional software into the program whose execution we wish to make visible without altering its basic structure. The role of the inserted software is to trace on the displayed program text the statement by statement progress in the program's execution and to dynamically update on the screen the changes in program variables as they are made and related to statements in the program.

Consideration of these requirements suggests that we can design the software in such a way that we only need to add essentially **one procedure call per original program statement to achieve visible program execution.** Since there are only a relatively small number of different classes of program statements (e.g. loops, conditional branches, input, output, and assignments, etc) in simple programs the corresponding number of different procedures needed for insertion into the program being displayed is also relatively small.

The role of the first set of these display procedures is relatively straightforward and usually involves some or all of the following steps:

* halt execution of the displayed program and wait for a response from the user.
* handle any errors in user's responses
* delete the displayed "arrow" at current statement and display a new arrow at the next statement on the displayed program text to be executed
* update any variables or indices affected by execution of the current displayed program statement.

Two other sets of procedures are needed to make up the complete suite of software for visible program execution. One set of these routines is used to clear the terminal screen and write the program that is to be executed in a visible mode onto the screen. This program must be stored in a separate file. The other set of routines must look after the screen layout for dynamic display of variables, arrays, and conditions. These routines must include a routine that can be used for positioning the cursor anywhere on the screen. (The present software has been written for a 24 line x 80 character screen.) Presently, screen co-ordinates must be supplied for all displayed variables, arrays and conditions, etc.

An example will best illustrate how the software described above can be used to make a program execute in a visible mode. We will consider the changes needed to make the selection sort displayed in figure 1 execute in a visible mode. The corresponding procedure is shown in figure 2.

```
procedure selectionsort (var a:nelements; n:integer);*
var i,j,p,min:integer;
   istart, jstart:integer;
begin
  istart := 1;
  for i := 1 to n-1 do begin
    waitsetarrayindex(pstart+4,progcol,irow,icol,i,airow,aicol,a[i],indexrow,istart,'i');
    min := a[i];
    getsetinteger(pstart+5,progcol,minrow,mincol,min,qmode,false);
    p := i;
    getsetarrayindex(pstart+6,progcol,prow,pcol,p,aprow,apcol,a[p],qmode,false);
    jstart := i + 1
    for j := i+1 to n do begin
      waitsetarrayindex(pstart+7,progcol,jrow,jcol,j,ajrow,ajcol,a[j],indexrow,jstart,'j');
      getsetcondition(pstart+8,progcol,ajlxrow,ajlxcol,truerow,truecol,falserow,falsecol,a[j]<min,qmode,true);
      if a[j] < min then begin
        min := a[j];
        getsetinteger(pstart+9,progcol,minrow,mincol,min,qmode,false);
        p := j;
        getsetarrayindex(pstart+10,progcol,prow,pcol,p,aprow,apcol,a[p],qmode,true);
      end;
      loopend(j,n,pstart+7,progcol,pstart+12,progcol,indexrow,jstart);
    end;
    a[p] := a[i];
    getsetarray(pstart+12,progcol,aprow,apcol,a[p],p,arrayrow,qmode,false);
    a[i] := min
    getsetarray(pstart+13,progcol,airow,aicol,a[i],i,arrayrow,qmode,true);
    loopend(i,n-1,pstart+4,progcol,pstart+14,progcol,indexrow,istart)
  end
end
```

* the variables *true* and *false* are used to indicate whether or not the current statement is at the end of the loop.

Figure 2

The parameters passed to the display routines mostly are the various co-ordinates relating to the program statements, displayed variables, and the actual variables being displayed.

The example in figure 2 illustrates that with the appropriate software tools any simple procedure can be made visible in a relatively straightforward and mechanical way without a large software development overhead. Software for visible program execution is therefore economical to produce once a little practice has been gained in using it. Clearly, it is possible to make a number of more sophisticated variations on the basic system without much additional effort. Presently a project is underway to develop a preprocessor that replaces the handcrafting into the program of the display routines. The present display software has been written in Pascal. It could be used equally well to make simple programs in other languages (e.g. BASIC, Fortran) appear to execute in a visible mode.

7. The Problem-Solving Aspect of Computing - we learn most when we have to invent

It is widely believed that problem-solving is a creative process that largely defies systematization and mechanization. Against this background where can we begin to say something useful about computer problem-solving? The answer to this question is most aptly summed up in a remark made by Richard Hamming who said, "computers are about **understanding** not numbers". Success in solving any problem can only come after we have come to understand what it is we are trying to solve. The role of this **problem definition phase** is to establish **what must be done** rather than **how to do it.**

Much of what Polya had to say in his classic works [9-11] on problem-solving are relevant to the beginning student in computing science. The most crucial thing of all in developing problem-solving skills is practice. Piaget summed this up very nicely with this claim that "we learn most when we have to invent".

In formulating computer solutions to problems one of the most important skills we need to develop is the ability to phrase problems in a way that will allow either an iterative or a recursive solution. Unfortunately, most people prior to their first encounter with a computer have had little experience (at least at the conscious level) in formulating iterative or recursive solutions to problems. Time and again in their early computing days, people are confronted with simple problems which they can solve very easily (e.g. like picking the maximum number in a set) but for which they have great difficulty in formulating an iterative computer solution. This happens because we can get the solution to many problems without explicitly formulating or understanding the method used to obtain it. When dealing with computers we are not granted such a luxury. An extended study of **visible** program execution (VPE) with well chosen examples can go a long way to removing the mystery of formulating iterative solutions to problems.

Before students are asked to formulate their own solutions it is useful for them to see a number of computer solutions to simple problems presented by way of example in what we may call a **discovery** mode. That is, rather than saying here is the computer implementation of Euclid's greatest common divisor algorithm, the problem should be stated and then an effort be made to ask the sort of questions that would enable us to "discover" it anew for ourselves [12]. In considering a number of problems in this way, we can learn a lot about the sort of questions and heuristics that can be useful in tackling any new problem. Too often in introductory courses, and texts in computing this method of treatment is neglected or ignored in favour of a discussion of top-down design as a problem-solving tool. Top-down design is certainly a very powerful tool for dealing with the complexity and detail of computer problems but it can only be employed **after** we have evolved or discovered the basic approach that is to be used in solving the problem.

At this point it may have been seen as appropriate to provide a long list of heuristics that can be useful in solving computing problems. We have refrained from doing so for two reasons. Quoting a problem-solving heuristic out of the context of solving a particular problem is not very convincing and may sometimes even be misleading. Secondly, it is always useful to keep in mind that there are no universal recipes for problem-solving. Problem-solving is a very personal thing. Different strategies appear to work for different people. The important point to remember is that students in their introduction to computing be given ample opportunity to cultivate and develop their own problem-solving skills, first by seeing algorithms "discovered" and then later by discovering their own solutions to problems. [12]

8. For and Against

Much of our present crisis in both teaching computing, and in computing practice at large, stems from the immaturity of the discipline, a factor that will unfortunately only be cured by time. We must recognize there is a mechanism at work whereby advances that are made by the researchers of one generation are taught to the children of succeeding generations. In the case of computing, this fundamental method of transfer of knowledge seems to have happened a little too quickly. To turn around Papert's maxim [13] learning is not easy if we do not have the proper models to assimilate new knowledge to. As he so aptly puts it

"Our culture is relatively poor in models of systematic procedures."

This observation provides a very telling explanation of why the current generation if having such a struggle in mastering computing.

One response to the present situation is to take the view that the only cure for our current problems is to confront students of computing science with discipline and mathematical rigor right from the outset. The idea being that rigor is always a very effective tool for deepening our understanding of a discipline. And from a solid base of understanding, it might then be hoped that people would be in a better position to make more effective use of the tools and concepts of the computing discipline.

The author has much respect in hindsight for an attack on the problem based on rigor. However, faced with the responsibility of having to introduce large numbers of students to computing, it does not appear to be practical, not at this time at least. Maybe later when computing enjoys the same status in the school education system as mathematics, will we be able to apply a similar degree of sophistication in our introductory treatment of the subject. Rigor should be applied in computing but only after students have had a chance to grasp the fundamental concepts.

We would suggest that the tools and philosophy that we have outlined represent a more palatable and efficient alternative in the present climate for increasing the student's understanding of the fundamentals of computing. Of course what we have suggested can be interpreted and expanded in a variety of new and different ways as we gather more experience.

We would not claim that these suggestions by any means can solve all our problems in teaching introductory computing. There are other strong social factors working against us. In particular, we must be on guard against a growing malaise in our society which is pushing quality and professionalism out the window. This is a deep-seated problem that partly reflects the frustration and alienation that is currently grasping at the very fabric of our society. Many people no longer take pride or seek satisfaction in the work they do. This inevitably shows up as poor quality workmanship with the obvious consequences for computer software. Adding to this problem is the real or imagined pace felt by most in our technological society which creates a pressure to get things done that overides the need to do things properly.

9. Conclusions

Reflection on the learning-to-read-and-write analogy can give us some useful guidelines on formulating a more effective way of teaching introductory computing. Exposing students to visible program execution, because of its strong cause-and-effect link, is a useful way of giving insight into the dynamic nature of computer algorithms. It is also an effective tool for teaching the use of programming constructs. Only after the dynamic nature of algorithms has been firmly established are beginners ready to face up to the problem solving aspects of computer algorithm design. Once this point is reached, it is then necessary for the student to discover his own set of heuristics by being exposed to a wide range of problems discussed in a "discovery mode". Only then is the student ready to embark on the more demanding task of trying to solve his own problems.

Acknowledgements

The author would like to thank Richard Miller, Ross Nealon and Ian Pirie for helpful discussions and advice.

Throughout this paper the male gender has been used only for the sake of convenience.

References

1. Maly, K., Hanson, A.R., (1978) "Fundamentals of the Computing Science", Prentice-Hall, N.J.
2. Guttman, A.J., (1977) "Programming and Algorithms", Heinemann, London.
3. Goodman, S.E., Hedetniemi, S.T., (1977) "Introduction to the Design and Analysis of Algorithms", McGraw-Hill, N.Y.
4. Wirth, N. (1973) "Systematic Programming: An Introduction", Prentice-Hall, Englewood Cliffs, N.J.
5. Dijkstra, E.W., (1976) "A Discipline of Programming", Prentice-Hall, N.J.
6. Alagic, S., Arbib, M.A. (1978) "The Design of Well-Structured and Correct Programs", Springer-Verlag, N.Y.
7. Wirth, N. (1971) "Program Development by Stepwise Refinement", Comm. ACM 14, 221
8. Nievergelt, J., "XS-0: A Self-Explanatory School Computer Laboratory Manual"
9. Polya, G. (1971) "How to Solve It", Princeton University Press, Princeton, N.J.
10. Polya, G. (1962) "Mathematical Discovery: On Understanding, Learning and Teaching Problem-Solving", Vols I and II, Wiley, N.Y.
11. Polya, G. (1954) "Mathematics and Plausible Reasoning", Vols I and II, Princeton University Press, Princeton, N.J.
12. Dromey, R.G., (1982) "How to Solve it by Computer", Prentice-Hall, London (in press)
13. Papert, S., (1980) "Mindstorms", Basic Books, N.Y.